"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes."

Marcel Proust
Get the Most from JMP®

Whether you are a first-time or a long-time user, there is always something to learn about JMP.

Visit JMP.com to find the following:

• live and recorded Webcasts about how to get started with JMP
• video demos and Webcasts of new features and advanced techniques
• schedules for seminars being held in your area
• success stories showing how others use JMP
• a blog with tips, tricks, and stories from JMP staff
• a forum to discuss JMP with other users

http://www.jmp.com/getstarted/
# Table of Contents

## JMP Scripting Guide

## 1 Introducing JSL

### Tutorials and Demonstrations

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello, World</td>
<td>3</td>
</tr>
<tr>
<td>Modify the Script</td>
<td>4</td>
</tr>
<tr>
<td>Save a Script</td>
<td>6</td>
</tr>
<tr>
<td>Save the Log</td>
<td>6</td>
</tr>
<tr>
<td>Saving and Sharing Your Work</td>
<td>6</td>
</tr>
<tr>
<td>Capturing Scripts for Data Tables</td>
<td>7</td>
</tr>
<tr>
<td>Capturing Scripts for Analyses</td>
<td>8</td>
</tr>
<tr>
<td>A General Method for Creating Scripts</td>
<td>9</td>
</tr>
<tr>
<td>Use JMP Interactively to Learn Scripting</td>
<td>11</td>
</tr>
</tbody>
</table>

### Using the Script Editor and Debugger

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Script Editor</td>
<td>12</td>
</tr>
<tr>
<td>The JSL Debugger (Windows Only)</td>
<td>17</td>
</tr>
</tbody>
</table>

### Help with JSL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSL Browsers</td>
<td>19</td>
</tr>
<tr>
<td>Show Commands</td>
<td>20</td>
</tr>
<tr>
<td>Show Properties</td>
<td>20</td>
</tr>
</tbody>
</table>

## 2 JSL Building Blocks

### Learn the Basic Components of JSL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>First JSL Script</td>
<td>25</td>
</tr>
<tr>
<td>The JSL Language</td>
<td>25</td>
</tr>
<tr>
<td>Lexical Rules of the Language</td>
<td>26</td>
</tr>
<tr>
<td>Data Elements</td>
<td>32</td>
</tr>
<tr>
<td>Operators</td>
<td>33</td>
</tr>
<tr>
<td>Context: Meaning is Local</td>
<td>36</td>
</tr>
<tr>
<td>Programming Versus Scripting</td>
<td>36</td>
</tr>
<tr>
<td>Data Table Context</td>
<td>37</td>
</tr>
<tr>
<td>Scoping Operators</td>
<td>38</td>
</tr>
</tbody>
</table>
Name Resolution ................................................. 38
Name-Binding Rules ............................................. 38
Frequently-Asked Questions ................................. 41
Function Resolution Rules ................................. 42
Gluing Expressions Together .............................. 42
Variables .......................................................... 43
Watch .............................................................. 43
Show Symbols, Clear Symbols, and Delete Symbols .... 44
Locking and Unlocking Symbols ......................... 45
Local .............................................................. 45
Datetime Operators .............................................. 46
Constructing Dates .............................................. 46
Extracting Parts of Dates ................................. 46
Arithmetic on Dates ............................................. 47
Converting Datetime Units .............................. 47
Y2K-Ready Date Handling .......................... 47
Datetime Notation ............................................... 48
Currency .......................................................... 53
Inquiry Functions ............................................... 53
Functions that communicate with users ............... 55
Writing to the log ................................................. 55
Send information to the user ............................ 56

3 Programming Functions
Script Control .................................................. 59
Programming Example ...................................... 61
Iterating .......................................................... 61
For ............................................................... 61
While ............................................................. 63
Summation ........................................................ 63
Product ........................................................... 63
Break and Continue .......................................... 63
Conditional functions ........................................ 64
If ................................................................. 64
Match ............................................................. 65
Choose .......................................................... 66
Interpolate ........................................................ 66
4 Data Tables
Create, Open, and Manipulate Data Tables ................................................................. 85

Data Table Basics ................................................................. 87
Getting a Data Table Object ................................................... 87
Objects and Messages .............................................................. 90
Messages for Data Tables .......................................................... 94
Naming, Saving, and Printing ....................................................... 95
Subscribing to a Data Table ....................................................... 95
Journal and Layout ................................................................. 96
Creating and Deleting Columns ................................................... 97
Accessing Values ................................................................. 98
Summarize .................................................................. 102
Working with Metadata ............................................................ 105
Commands from the Rows menu ............................................... 110
Commands from the Tables Menu .............................................. 116
Manipulating columns ............................................................ 121
Obtaining Column Names ....................................................... 122
Select Columns .................................................................. 122
Moving Columns .................................................................. 123
6 Display Trees
Create and Use Windows .............................................. 175
Manipulating Displays ................................................. 177
  Introduction to Display Boxes ..................................... 177
  DisplayBox Object References ..................................... 180
  Sending Messages .................................................... 182
  Using the << Operator ............................................... 184
  Platform Example .................................................... 185
How to Access Built-in Windows ..................................... 189
  Using the Pick Windows ............................................ 189
Constructing Display Trees .......................................... 190
  Basics ............................................................. 190
  Updating an Existing Display ...................................... 193
  Controlling Text Wrap ............................................. 195
  Interactive Display Elements ...................................... 195
  Send Messages to Constructed Displays ......................... 209
Build Your Own Displays from Scratch ............................. 209
Construct Display Boxes Containing Platforms ................... 210
Construct a Custom Platform ......................................... 212
Sheets ............................................................. 215
Journals ............................................................. 217
Picture Display Type ................................................ 217
Modal Windows ......................................................... 218
  Constructing Modal Windows ...................................... 218
  General-Purpose Modal Window .................................. 219
Data Column Dialog Boxes ........................................... 222
  Constructing Dialogs and Column Dialogs ....................... 225
Scripting the Script Editor .......................................... 227
Syntax Reference ..................................................... 228
7 Scripting Graphs
Create and Edit 2-Dimensional Plots ............................... 237
Adding Scripts to Graphs ............................................ 239
  Ordering Graphics Elements Using JSL ............................ 240
  Adding a Legend to a Graph ....................................... 240
Creating New Graphs From Scratch .................................................. 241
Making Changes to Graphs .............................................................. 242
Graphing Elements ................................................................. 243
Plotting Functions ................................................................. 243
Getting the Properties of a Graphics Frame ..................................... 249
Adding a Legend ................................................................. 249
Drawing Lines, Arrows, Points, and Shapes ...................................... 249
Lines ................................................................. 249
Arrows ................................................................. 251
Markers ................................................................. 252
Pies and Arcs ................................................................. 254
Regular Shapes: Circles, Rectangles, and Ovals .................... 255
Irregular Shapes: Polygons and Contours .................................. 258
Adding text ................................................................. 260
Colors ................................................................. 261
Transparency ................................................................. 263
Fill patterns ................................................................. 264
Line types ................................................................. 264
Drawing With Pixels ............................................................... 265
Interactive graphs ............................................................... 266
Handle ................................................................. 266
MouseTrap ................................................................. 269
Drag Functions ................................................................. 270
Troubleshooting ................................................................. 272

8 Three-Dimensional Scenes
Scripting in Three Dimensions ................................................... 273

About JSL 3-D Scenes ............................................................... 275
JSL 3-D Scene Boxes ............................................................... 275
Setting the Viewing Space ..................................................... 278
Setting Up a Perspective Scene .................................................... 279
Setting up an Orthographic Scene ............................................. 280
Changing the View ................................................................. 281
The Translate Command .......................................................... 281
The Rotate Command ............................................................. 281
The Look At Command ........................................................... 283
The ArcBall ................................................................. 284
9 Matrices

Matrix Algebra in JMP

Basics ......................................................... 315
Definitions .................................................. 315
Constructing Matrices ................................. 315
Numeric Operators ...................................... 317
Manipulating Values .................................. 319
Subscripts ................................................... 321
Matrix Operators ......................................... 324
Solving Linear Systems ............................... 325
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transcendental Functions</td>
<td>572</td>
</tr>
<tr>
<td>Trigonometric Functions</td>
<td>575</td>
</tr>
<tr>
<td>Utility Functions</td>
<td>578</td>
</tr>
<tr>
<td>Platform Scripting Notes</td>
<td>623</td>
</tr>
<tr>
<td>Scripting Options for All Platforms</td>
<td>623</td>
</tr>
<tr>
<td>Individual Platform Options</td>
<td>623</td>
</tr>
<tr>
<td>Syntax Reference</td>
<td>629</td>
</tr>
<tr>
<td>Summary of Object Messages</td>
<td>639</td>
</tr>
<tr>
<td>Associative Arrays</td>
<td>641</td>
</tr>
<tr>
<td>Data Tables</td>
<td>642</td>
</tr>
<tr>
<td>Columns</td>
<td>653</td>
</tr>
<tr>
<td>Rows</td>
<td>657</td>
</tr>
<tr>
<td>Data Filter</td>
<td>658</td>
</tr>
<tr>
<td>Databases</td>
<td>660</td>
</tr>
<tr>
<td>Datafeed</td>
<td>660</td>
</tr>
<tr>
<td>Display Boxes</td>
<td>661</td>
</tr>
<tr>
<td>Windows</td>
<td>661</td>
</tr>
<tr>
<td>All Display Boxes</td>
<td>663</td>
</tr>
<tr>
<td>Outline Boxes</td>
<td>667</td>
</tr>
<tr>
<td>Frame Boxes</td>
<td>667</td>
</tr>
<tr>
<td>Table Boxes</td>
<td>668</td>
</tr>
<tr>
<td>Tab Boxes</td>
<td>669</td>
</tr>
<tr>
<td>Text Boxes</td>
<td>669</td>
</tr>
<tr>
<td>Axis Boxes</td>
<td>670</td>
</tr>
<tr>
<td>Number Col Boxes</td>
<td>672</td>
</tr>
<tr>
<td>Slider Boxes</td>
<td>673</td>
</tr>
<tr>
<td>String Col Boxes</td>
<td>673</td>
</tr>
<tr>
<td>Matrix Boxes</td>
<td>674</td>
</tr>
<tr>
<td>Nom Axis Boxes</td>
<td>674</td>
</tr>
<tr>
<td>Panel Boxes</td>
<td>674</td>
</tr>
<tr>
<td>Border Boxes</td>
<td>674</td>
</tr>
<tr>
<td>Dynamic Link Libraries (DLLs)</td>
<td>675</td>
</tr>
<tr>
<td>Platforms</td>
<td>676</td>
</tr>
</tbody>
</table>
C Compatibility Notes
Changes in Scripting from JMP 8 to JMP 9

D Glossary
Terms, Concepts, and Placeholders

Index
JMP Scripting Guide
Credits and Acknowledgments

Origin

JMP was developed by SAS Institute Inc., Cary, NC. JMP is not a part of the SAS System, though portions of JMP were adapted from routines in the SAS System, particularly for linear algebra and probability calculations. Version 1 of JMP went into production in October 1989.

Credits

JMP was conceived and started by John Sall. Design and development were done by John Sall, Chung-Wei Ng, Michael Hecht, Richard Potter, Brian Corcoran, Annie Dudley Zangi, Bradley Jones, Craig Hales, Chris Gotwalt, Paul Nelson, Xan Gregg, Jianfeng Ding, Eric Hill, John Schroedl, Laura Lancaster, Scott McQuiggan, Melinda Thielbar, Clay Barker, Peng Liu, Dave Barbour, Jeff Polzin, John Ponte, and Steve Amerige.

In the SAS Institute Technical Support division, Duane Hayes, Wendy Murphrey, Rosemary Lucas, Win LeDinh, Bobby Riggs, Glen Grimme, Sue Walsh, Mike Stockstill, Kathleen Kiernan, and Liz Edwards provide technical support.

Nicole Jones, Kyoko Keener, Hui Di, Joseph Morgan, Wenjun Bao, Fang Chen, Susan Shao, Michael Crotty, Jong-Seok Lee, Tonya Mauldin, Audrey Ventura, Ani Eloyan, Bo Meng, and Sequola McNeill provide ongoing quality assurance. Additional testing and technical support are provided by Noriki Inoue, Kyoko Takenaka, Yusuke Ono, Masakazu Okada, and Naohiro Masukawa from SAS Japan.

Bob Hickey and Jim Borek are the release engineers.

The JMP books were written by Ann Lehman, Lee Creighton, John Sall, Bradley Jones, Erin Vang, Melanie Drake, Meredith Blackwelder, Diane Perhac, Jonathan Gatlin, Susan Conaghan, and Sheila Loring, with contributions from Annie Dudley Zangi and Brian Corcoran. Creative services and production was done by SAS Publications. Melanie Drake implemented the Help system.

Jon Weisz and Jeff Perkinson provided project management. Also thanks to Lou Valente, Ian Cox, Mark Bailey, and Malcolm Moore for technical advice.

Thanks also to Georges Guirguis, Warren Sarle, Gordon Johnston, Duane Hayes, Russell Wolfinger, Randall Tobias, Robert N. Rodriguez, Ying So, Warren Kuhfeld, George MacKensie, Bob Lucas, Warren Kuhfeld, Mike Leonard, and Padraic Neville for statistical R&D support. Thanks are also due to Doug Melzer, Bryan Wolfe, Vincent DelGobbo, Biff Beers, Russell Gonsalves, Mitchel Soltys, Dave Mackie, and Stephanie Smith, who helped us get started with SAS Foundation Services from JMP.

Acknowledgments

We owe special gratitude to the people that encouraged us to start JMP, to the alpha and beta testers of JMP, and to the reviewers of the documentation. In particular we thank Michael Benson, Howard Yetter (d),
Andy Mauromoustakos, Al Best, Stan Young, Robert Muenchen, Lenore Herzenberg, Ramon Leon, Tom Lange, Homer Hegedus, Skip Weed, Michael Emptage, Pat Spagan, Paul Wenz, Mike Bowen, Lori Gates, Georgia Morgan, David Tanaka, Zoe Jewell, Sky Alibhai, David Coleman, Linda Blazek, Michael Friendly, Joe Hockman, Frank Shen, J.H. Goodman, David Iklé, Barry Hembree, Dan Obermiller, Jeff Sweeney, Lynn Vanatta, and Kris Ghosh.

Also, we thank Dick DeVeaux, Gray McQuarrie, Robert Stine, George Fraction, Avigdor Cahaner, José Ramirez, Gudmundur Axelsson, Al Fulmer, Cary Tuckfield, Ron Thisted, Nancy McDermott, Veronica Czitrom, Tom Johnson, Cy Wegman, Paul Dwyer, DaRon Huffaker, Kevin Norwood, Mike Thompson, Jack Reese, Francois Mainville, and John Wass.

We also thank the following individuals for expert advice in their statistical specialties: R. Hocking and P. Spector for advice on effective hypotheses; Robert Mee for screening design generators; Roselinde Kessels for advice on choice experiments; Greg Piepel, Peter Goos, J. Stuart Hunter, Dennis Lin, Doug Montgomery, and Chris Nachtsheim for advice on design of experiments; Jason Hsu for advice on multiple comparisons methods (not all of which we were able to incorporate in JMP); Ralph O’Brien for advice on homogeneity of variance tests; Ralph O’Brien and S. Paul Wright for advice on statistical power; Keith Muller for advice in multivariate methods, Harry Martz, Wayne Nelson, Ramon Leon, Dave Trindade, Paul Tobias, and William Q. Meeker for advice on reliability plots; Lijian Yang and J.S. Marron for bivariate smoothing design; George Milliken and Yuri Bulavski for development of mixed models; Will Potts and Cathy Maahs-Fladung for data mining; Clay Thompson for advice on contour plotting algorithms; and Tom Little, Damon Stoddard, Blanton Godfrey, Tim Clapp, and Joe Ficalora for advice in the area of Six Sigma; and Josef Schmee and Alan Bowman for advice on simulation and tolerance design.

For sample data, thanks to Patrice Strahle for Pareto examples, the Texas air control board for the pollution data, and David Coleman for the pollen (eureka) data.

**Translations**

Trish O’Grady coordinates localization. Special thanks to Noriki Inoue, Kyoko Takenaka, Masakazu Okada, Naohiro Masukawa, and Yusuke Ono (SAS Japan); and Professor Toshiro Haga (retired, Tokyo University of Science) and Professor Hirohiko Asano (Tokyo Metropolitan University) for reviewing our Japanese translation; François Bergeret for reviewing the French translation; Bertram Schäfer and David Meintrup (consultants, StatCon) for reviewing the German translation; Patrizia Omodei, Maria Scaccabarozzi, and Letizia Bazzani (SAS Italy) for reviewing the Italian translation; RuiQi Qiao, Rula Li, and Molly Li for reviewing Simplified Chinese translation (SAS R&D Beijing); Finally, thanks to all the members of our outstanding translation and engineering teams.

**Past Support**

Many people were important in the evolution of JMP. Special thanks to David DeLong, Mary Cole, Kristin Nauta, Aaron Walker, Ike Walker, Eric Gjertsen, Dave Tilley, Ruth Lee, Annette Sanders, Tim Christensen, Eric Wasserman, Charles Soper, Wenjie Bao, and Junji Kishimoto. Thanks to SAS Institute quality assurance by Jeanne Martin, Fouad Younan, and Frank Lassiter. Additional testing for Versions 3 and 4 was done by Li Yang, Brenda Sun, Katrina Hauser, and Andrea Ritter.

Also thanks to Jenny Kendall, John Hansen, Eddie Routten, David Schlotzhauer, and James Mulherin. Thanks to Steve Shack, Greg Weier, and Maura Stokes for testing JMP Version 1.
Thanks for support from Charles Shipp, Harold Gugel (d), Jim Winters, Matthew Lay, Tim Rey, Rubin Gabriel, Brian Ruff, William Lisowski, David Morganstein, Tom Esposito, Susan West, Chris Fehily, Dan Chilko, Jim Shook, Ken Bodner, Rick Blahunka, Dana C. Aultman, and William Fehlner.

Technology License Notices

Scintilla is Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>. NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

XRender is Copyright © 2002 Keith Packard. KEITH PACKARD DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL KEITH PACKARD BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

ImageMagick software is Copyright © 1999-2010 ImageMagick Studio LLC, a non-profit organization dedicated to making software imaging solutions freely available.

bzlib software is Copyright © 1991-2009, Thomas G. Lane, Guido Vollbeding. All Rights Reserved.

FreeType software is Copyright © 1996-2002 The FreeType Project (www.freetype.org). All rights reserved.
Chapter 1

Introducing JSL
Tutorials and Demonstrations

This introduction shows you the basics of JMP Scripting Language (JSL). The chapter starts with a simple, progressive tutorial example to show you where to type a script, how to submit it, and how to modify and save it. The purpose of this tutorial is to give you the basic techniques for working with any script, whether it's one you write or one you get from someone else. Next is a showcase of examples to demonstrate how scripting might be useful in a variety of settings. For example, classroom simulations, advanced data manipulations, custom statistics, production lines, and so forth.

Confusion alert! Throughout this book, special shaded “confusion alerts” like this one call your attention to important concepts that could be unfamiliar or more complicated than you might expect, or where JMP might be a little different from other applications. These alerts appear whenever a particularly good example of a potential problem arises in the text, and although you will find them under topics that might not apply to your immediate needs, the ideas presented are always general and important. Please be sure to take a look even when you are skipping pages and looking for something else.

You can quickly locate these pointers by looking up “confusion alert” in the “Index,” p. 713.
Contents

Hello, World ................................................................. 3
Modify the Script ............................................................ 4
Save a Script .................................................................... 6
Save the Log .................................................................. 6

Saving and Sharing Your Work ........................................... 6
Capturing Scripts for Data Tables ........................................ 7
Capturing Scripts for Analyses .......................................... 8
A General Method for Creating Scripts ................................. 9
Use JMP Interactively to Learn Scripting ............................... 11

Using the Script Editor and Debugger ................................. 12
The Script Editor ............................................................... 12
The JSL Debugger (Windows Only) ...................................... 17

Help with JSL .................................................................. 19
JSL Browsers ................................................................. 19
Show Commands ............................................................ 20
Show Properties .............................................................. 20
Hello, World

This exercise is simple and hearkens back to a classic. Perhaps you recognize it.

1. Start JMP.
2. If the log window is not open, open it by selecting View > Log on Windows or Window > Log on Macintosh.
3. In the JMP Starter window, click New Script.
   New scripts can also be created from the menus. On Windows, select File > New > Script. On Macintosh, select File > New > New Script.
4. In the resulting script editor, type these lines:
   A = "Hello, World";
   Show( A );
5. From the Edit menu, select Run Script.
   The keyboard shortcut is to press CTRL-R on Windows or COMMAND-R on Macintosh.

The result is shown in the Log window. Besides showing results and errors, this window is also a script editor.

Figure 1.1 The Script Window (left) and the Log Window (right)

This is how you enter and submit JSL. You have just created a global variable named A, assigned a value to it, and shown its value. Notice that the log echoes your script first, and then echoes any results. In this case, the result is the output for the Show(A) command.

Go To Line

As your scripts grow in size, it is handy to jump to specific lines in the script. Use the Go To Line command from the Edit menu to jump to a specific line.
Hello, World

This command is also useful during the debugging process, since error messages frequently mention the line of the script where the error occurred.

Show Line Numbers

The script window does not show line numbers by default. To see the line numbers, right-click anywhere in the script window, and then select **Show Line Numbers**.

Modify the Script

Now try making this script do a little more work by adding a For-loop to greet the world not just once, but four times. Also, use **Print** this time instead of **Show**, and make a few other changes to see how the For-loop works.

1. In the script window, change the script to this:
   ```javascript
   For( i = 1, i < 5, i++,
       X = i;
       A = "Hello, World";
       Print( X, A );
   );
   Print( "done" );
   ``

2. Submit the script as before.
   (To submit part of the text in a window instead of all text, select the text first and then press Control-R on Windows or COMMAND-R on Macintosh)

Output

```
1  "Hello, World"
2  "Hello, World"
3  "Hello, World"
4  "Hello, World"
    "done"
```

**Print** is similar to **Show** except that it does not label the results with the variable name. It just prints the value.

Stopping a Script

When you run a script, the **Run Script** command in the **Edit** menu changes to **Stop Script**. While a script is running, you can select this option to stop it. You can also press ESC on Windows or COMMAND-PERIOD on Macintosh. Many scripts execute more quickly than you can stop them.
Chapter 1

Introducing JSL

Hello, World

Punctuation and Spaces

Notice the indented lines inside the For-loop. This is not necessary for JMP's sake; it just makes it a little easier to read. You could write this script like this if you want:

```julia
for R(i=1, i<5, i++;
    X=i; A="Hello, World"; PRINT(X,A)); print("done");
```

Words in JMP's scripting language are separated by parentheses, commas, semicolons, and various operators such as +, -, and so on. As far as JMP is concerned, spaces, tabs, returns, and blank lines inside or between operators or within JSL words do not exist. This is because most JSL words come from JMP's usual graphical user interface (GUI), and most of those commands have spaces in them. For example, you will see later on that to do Fit Model, the JSL would be `Fit Model(and some arguments inside parentheses)`. Most people would rather see "Fit Model" than "fitmodel."

JSL is not case-sensitive, so you do not have to worry about reaching for the Shift key all the time.

You do have to be a little careful inside a text string. In “Hello, World” extra spaces would affect the output, because text strings inside double quotation marks are taken literally, exactly as you type them. Also, you would get errors if you put spaces between the two plus signs in `i++`, `(i+ +)`, or in numbers (`43` is not the same as `43`).

Generally, JSL uses:

- **commas** `,` between arguments inside the parentheses for a command.
- **parentheses** `()` to surround all the arguments of a command. Many JSL words have parentheses after them even if they do not take arguments; for example `pi()` has the parentheses even though `π` is just the number 3.14... `Pi` does not expect an argument and will complain about any argument that you do give it. Therefore, `Pi(mincemeat)` would be considered an error (although it seems heretical to say so). But the parentheses are still required.
- **semicolons** `;` to separate commands but also to glue them together. In other words, you use a semicolon to separate one complete message from the next. For example, `a=1; b=2`. What the semicolon really does is tell JMP to continue and do more things. For example, the For-loop example showed how to put several statements in the place of one statement for the fourth argument, because the semicolon effectively turned all three statements into one argument.

Trailing semicolons (extras at the end of a script or at the end of a list of arguments) are harmless, so you can also think of semicolons as terminating characters. In fact, terminating each complete JSL statement with a semicolon is a good habit to adopt.

- **quotation marks** " " to enclose text strings. Anything inside quotation marks is taken literally, exactly as is, including spaces and upper- or lower-case. Nothing that could be evaluated is evaluated. If you have `pi()^2` inside quotation marks, it is just a sequence of six characters, not a value close to ten. See “Quoted Strings,” p. 28, for ways to include special characters and quotation marks in strings.

Introducing JSL
Chapter 1
Saving and Sharing Your Work

Save a Script

To save your script, just follow these instructions:

1. Make the script window active (click the “Untitled” window to make it the front-most window).
2. From the File menu, select Save or Save As.
3. Specify a filename, including the extension .jsl. For example, hello.jsl.
4. Click Save.

Scripts are saved as text files, and you can edit them with any text editor. However, if you do edit scripts with applications other than JMP, be careful to save them as plain text files. If you preserve the .jsl extension, you can double-click a script file to launch JMP.

To reuse a script, use Open from JMP’s File menu, double-click a .jsl file, or drag and drop the file onto JMP’s application icon.

When opening a JSL file, the actual script is always opened in its own script window. However, it might be distracting to some users to see this window. To keep a script from opening in a script window, put this command on the first line of the script.

```javascript
//!
```

If it is not on the very first line by itself, this command does nothing.

You can over-ride this comment when opening the file. Select File > Open. Hold the CTRL key while you select the JSL file and click Open. The script opens into a script window instead of being executed.

Save the Log

You can also save logs as text files, which can be viewed with any text editor. Double-clicking a log file does not launch JMP.

1. Make the log window active (click the Log window to make it the front-most window).
2. From the File menu, select Save or Save As.
3. Specify a filename, including the extension .txt on Windows. For example, hello.txt.
4. Click Save

Saving and Sharing Your Work

Here is something just about everybody will find useful sooner or later: JMP can create scripts to duplicate your data tables and analyses. For example:

- Suppose you need to describe an analysis process in detail, from beginning to end, such as to create an audit trail for a governing agency or for peers reviewing your journal article.
- Suppose you have a set of analysis steps that should be followed routinely by your lab technicians.
- Suppose you fit the same model to new data every day, and you are tired of clicking the same buttons over and over again.
• Suppose you are working with somebody in another city who cannot simply look over your shoulder to see how you put something together.

You can use JMP interactively as usual, save scripts to reproduce your work, and in the future just run those scripts. Next are some examples showing how this works.

**Capturing Scripts for Data Tables**

1. Open a data table and make all types of changes. For example, add rows and columns, change values, rearrange columns, sort the rows, make a formula column, make a row state column, and so on.
2. When you are finished, open a script window and type this:
   ```
   current data table() << get script;
   ```
3. In the Script window, click and drag to select (highlight) the script.
4. Run the script and look at the output shown in the Log window:
   From the **Edit** menu, select **Run Script**.

**Figure 1.2 The Get Script Command and the Log**

Now try running the script in the log window:

1. In the Log window, click and drag to select (highlight) the script, starting with **New Table** and ending with the last line in the log.
2. From the **Edit** menu, select **Run Script**.
Capturing Scripts for Analyses

Launch a platform, such as Fit Model. Look at the default results and then go exploring. Try options to see related tests and graphs. Work with the red triangle menus in the report surface to get exactly the report that you want.

When you are finished, get a script to recreate your results. There are two methods of getting a script for your results. One is to use the Script menu located at the bottom of each platform’s red triangle menu. This method is detailed in “Use JMP Interactively to Learn Scripting,” p. 11.

1. First you need to figure out what JMP calls your analysis. This can be tricky in some cases, as discussed in the “Scripting Platforms” chapter, but usually you can read it from the title of the analysis window, after the name of the data table. For example, the platform for a window titled “Big Class: Fit Least Squares” would be called Fit Least Squares.

2. Now you have to specify a part of the report. You might have fit several models before getting the one that you want to keep. You need to tell JMP which one you want by supplying a subscript, which is just a number inside brackets after the name. If the third model that you fit is the one that you want, you would specify it as Fit Least Squares[3].

3. You are ready to get the script from the object:

   Fit Least Squares[3] << get script;

The results might look something like this, depending on which steps you performed:

   Fit Model(
      Y( :weight ),
      Effects( :sex, :height, :sex * :height ),
      Personality( Standard Least Squares ),
      Run Model(
         Profiler(  
            Confidence Intervals( 1 ),  
            Desirability Functions( 1 )
         ),  
         Contour Profiler(  
            Surface Plot( 1 )
         )
      )
   )

Try running the script:

1. In the Log window, click and drag to select (highlight) the script

2. From the Edit menu, select Run Script.

The script produces a perfect clone of your analysis. If you want a journal, use this command:

   Fit Least Squares[3] << journal window;
A General Method for Creating Scripts

1. Did you use an existing data table as is? Write an Open statement for it:
   ```
   dt = Open( "SAMPLE_DATA/Big Class.JMP" );
   ```

2. Did you create a new table or make changes to an existing data table? Did you work with row states,
   such as to color and label points in your plots? Did you exclude some rows? Did you fix errors? If so, you
   should get a script to recreate your data table.
   ```
   Current Data Table() << get script;
   ```

3. Which platforms did you launch, work with, and keep for your final results? Get scripts to recreate
   them. You will learn the details in the “Scripting Platforms” chapter; here you will just try some
   examples.
   ```
   Bivariate[1] << get script;
   Fit Least Squares[1] << get script;
   ```

4. Now edit the log into a complete script. Be sure to put semicolons (;) in between statements. You
   might have something like this:
   ```
   New Table( "Big Class",
   Add Rows( 40 ),
   New Column( "name",
       Character,
       Nominal,
       Set Property( Notes, "...usually used as a label variable in plots" ),
       Values(
           {"KATIE", "LOUISE", "JANE", "JACLYN", "LILLIE", "TIM", "JAMES",
            "ROBERT", "BARBARA", "ALICE", "SUSAN", "JOHN", "JOE", "MICHAEL",
            "DAVID", "JUDY", "ELIZABETH", "LESLIE", "CAROL", "PATTY",
            "FREDRICK", "ALFRED", "HENRY", "LEWIS", "EDWARD", "CHRIS",
            "JEFFERY", "MARY", "AMY", "ROBERT", "WILLIAM", "CLAY", "MARK",
            "DANNY", "MARTHA", "MARIAN", "PHILLIP", "LINDA", "KIRK",
            "LAWRENCE"}
       ),
   ),
   New Column( "age",
       Numeric,
       Ordinal,
       Set Property( Notes, "Explore data adventurously" ),
       Values(
           [12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 14,
            14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
            15, 16, 16, 16, 17, 17, 17]
       ),
   ),
   New Column( "sex",
       Character,
       Nominal,
       Set Property( Notes, "Explore data adventurously" ),
       Values(
Introducing JSL

Chapter 1

Saving and Sharing Your Work


),
New Column( "height",
  Numeric,
  Continuous,
  Set Property( Notes, "Explore data adventurously" ),
  Values( [59, 61, 55, 66, 52, 60, 61, 51, 60, 61, 56, 65, 63, 58, 59, 61, 62, 65, 63, 62, 63, 64, 65, 64, 68, 64, 69, 62, 64, 67, 65, 66, 62, 66, 65, 60, 68, 62, 68, 70] )
)
),
New Column( "weight",
  Numeric,
  Continuous,
  Set Property( Notes, "Explore data adventurously" ),
)
)
);

Bivariate( Y( weight ),
  X( height ),
  Fit Line( {Confid Curves Fit( 1 )} ),
  Where( :sex == "F" )
);

Bivariate( Y( weight ),
  X( height ),
  Fit Line( {Confid Curves Fit( 1 )} ),
  Where( :sex == "M" )
);

Fit Model( Y( :weight ),
  Effects( :sex, :height, :sex * :height ),
  Personality( Standard Least Squares ),
  Run Model( Profiler( Confidence Intervals( 1 ), Desirability Functions( 1 ) ), :weight << {Plot Actual by Predicted( 1 ), Plot Residual by Predicted( 1 ), Plot Effect Leverage( 1 )} )
)

); Scatterplot 3D( Y( age, weight, height ));

5. Save the script.
6. If you share your JSL file with your colleagues, be sure to include any data tables or additional files along with the script.

**Use JMP Interactively to Learn Scripting**

One of the simplest ways to accomplish anything is to get somebody else to do it for you, and writing JSL is no exception. The best JSL writer is JMP itself. This example shows how to work in JMP interactively and then save the results as a script to reuse later on. With simple modifications, this script can serve as a template for speeding up routine tasks.

**What JMP Can and Cannot do to Help You Write Scripts**

JMP can automatically save scripts to reproduce any data table or analysis in its current state. You can pause any time in your analysis to save a script to a script window, in a data table, or in an analysis report. You can then modify the automatically generated script as needed for future projects.

JMP cannot record scripts while you are working. While script-recording is a useful feature in some other scripting languages, it is less important for a program like JMP, where the important thing is the results. You cannot use script-recording to observe how a sequence of interactive steps is performed.

However, remember that you can save a script when you are finished, and that script reproduces everything that you have accomplished.

JMP's scripting language is not intended to be an alternative command-line interface for using the program. JSL is intended for recreating results and for extending JMP's capabilities beyond JMP's intended use in data discovery.

**There is More Than One Way**

Since JSL is a very flexible language, you can accomplish things many different ways.

Typically the script that JMP saves for you specifies every detail of your analysis, even if most of the details happen automatically by default. Does that mean that the scripts that you write have to be just as complete and detailed? Not at all. You usually just need to specify the details that you would specify when using the graphical user interface (GUI). For example, if you open *Big Class* and want to launch Distribution for *height*, *weight*, and *sex*, you would need only to do this in JSL:

```
Distribution( Y( :height, :weight, :sex ) );
```

But if you select **Script > Save Script to Script Window** from the red triangle menu for the report, you would see this:

```
Distribution( 
    Continuous Distribution( Column( :height ) ),
    Continuous Distribution( Column( :weight ) ),
    Nominal Distribution( Column( :sex ) ),
);
```

Both methods give the same result. Feel free to experiment with JSL. If you think something ought to be possible, it probably is. Give it a try, and see what happens.
Using the Script Editor and Debugger

JMP provides both an editor and a debugger for writing and troubleshooting your JSL scripts.

The Script Editor

The Script Editor (as shown in Figure 1.3) provides a friendly environment for writing and reading JSL scripts.

Figure 1.3 The Script Editor

The script editor has several useful features:

- color-coding for JSL and SAS code
- autocompletion for JSL functions (press CONTROL-SPACE on Windows or OPTION-ESC on Macintosh)
- tooltips when hovering over JSL functions
- value-reporting when hovering over a global symbol
- live brace matching
- highlighting matching braces
- find and replace using regular expressions
- automatic formatting

The script editor is also used in the log window and anywhere else you can edit or write a script.
Chapter 1

Introducing JSL

13

Saving and Sharing Your Work

Color-Coding

The script window uses the following colors for JSL:

- green for comments
- blue for JSL functions
- dark magenta for string values
- dark cyan and bold for scalar values
- black for everything else

Colors can be customized in Preferences. See “Setting Preferences for the Script Editor,” p. 15.

Auto-Completion

If you do not remember the exact name of a function, you can type part of the name and then press CONTROL-SPACE or CONTROL-ENTER on Windows, or OPTION-ESC on Macintosh to see a list of functions that match what you have typed so far.

For example, if you want to clear your JSL variables, but do not remember the command, you can type clear, then CONTROL-SPACE, to see a list of possible clear commands.

Figure 1.4 Autocomplete Example

Select the command that you want to use.

Toolips

If you are using a function and do not remember its syntax or what it does exactly, you can hover over it to get a brief explanation. This only works with JSL function names, not platform commands, messages, or user-created functions. JSL function names are colored blue in the script editor.

Figure 1.5 Tooltip for a JSL Function

The tooltip shows the grammar, any arguments, and a brief explanation of the function.
You can also hover over variable names to see their current value. If you hover over a variable before running the script, no tip appears, because JMP does not yet know about it.

**Example of a Tooltip for a JSL Variable**

1. Enter and run the following line in a script window:
   ```jsl
   my_variable = 8;
   ```
2. Hover over the variable name after you run the line.
   A tooltip shows the name of the variable and its value: 8.
3. Enter and run the following line:
   ```jsl
   my_variable = "eight";
   ```
4. Hover over the variable name after you run the line.
   A tooltip shows the name of the variable and its value: “eight”.

**Brace-Matching**

The script editor helps you match parentheses, square brackets, and curly braces in the following ways:

- The matching closing brace is added when you type an opening brace.
- When you place your cursor next to either an opening or closing brace, it and its match are highlighted in blue. If it does not have a match, it is highlighted in red.
- If you double-click a brace, everything between the matching braces is selected (including the braces).
- If you put your cursor within a statement and press CONTROL-] on Windows, or COMMAND-B on Macintosh, the entire statement is selected (including the braces that define it).

When you type an opening brace, add code in between, and then type the closing brace, the script editor skips the cursor over the brace it that added automatically for you. This prevents you from accidentally adding an additional closing brace.

You can turn on and off the autocompletion of braces in the Preferences window. See “Setting Preferences for the Script Editor,” p. 15 for details.

**Find and Replace**

The **Edit > Search** menu options are now available in scripts. Searching and replacing is the same as for data tables.

1. From the **Edit** menu, select **Search > Find**.
2. In the **Find what** field, enter the text that you want to search for.
3. If you want to find text and replace it, enter the replacement text in the **Replace with** field.
4. Select any other check boxes that you want to use.
5. Click **Find** to find the next occurrence, or click **Replace** to replace the current occurrence and find the next.

See the Using JMP book for details about the Search window.
**Automatic Formatting**

The script editor can format a script for easier reading. Any generated script (for example, by saving a platform script) is automatically formatted with tabs and returns in appropriate places.

If you open or write a script that is poorly formatted (for example, older saved JMP scripts that might have all commands strung together with no whitespace characters), you can have the script editor format it for you.

From the **Edit** menu, select **Reformat Script**.

**Tip:** This command alerts you if your script is badly formed (for example, if your parentheses are not matched).

**Setting Preferences for the Script Editor**

You can customize several parts of the script editor. Open the Preference Settings window by selecting **File > Preferences**.

**Setting the Fonts**

1. Select the Fonts group.
2. Click **Mono** to set the font for the script editor.

For more details about font preferences, see the *Using JMP* book.

![Figure 1.6 Changing the Font for Script Windows](image)
Introducing JSL
Saving and Sharing Your Work

Setting editor preferences

On the Script Editor page, you can make many other customizations:

- **Use tabs**: Check this option to enable tabs in your scripts. If it is unchecked, any tab you type is replaced by spaces. This is on by default.
- **Tab width**: Enter how many spaces a tab should indent. If you have disabled tabs, any tab you type is replaced with the number of spaces specified. The default value is 4.
- **Extra space at bottom of document**: Check this option to enable scrolling the last line of a script to the top of the script editor. This is on by default.
- **Auto-complete parentheses and braces**: Check this option to enable the script editor to automatically add closing parentheses, square brackets, and curly braces when you type an opening one. This is on by default.
- **Show line numbers**: Check this option to show the line numbers on the left side of the script editor. This is off by default.
- **Show operator tips**: Check this option to see tooltips for JSL functions. This is on by default.
- **Show indentation guides**: Check this option to see faint vertical lines that mark indentation. This is on by default.
- **Show variable value tips**: Check this option to see tooltips for variable values. This is on by default.
- **Spaces inside parentheses**: Check this option to cause the script editor to add spaces between parentheses, brackets, and braces and their contents for automatically formatted scripts. This is on by default.
- **Spaces in operator names**: Check this option to cause the script editor to add spaces between words within function names. For example, turning on this option results in `New Window` instead of `NewWindow`. This is on by default.

Setting colors used in the editor

To set your own color for any of the listed types, click the color box and select your color. (See Figure 1.7.)

<table>
<thead>
<tr>
<th>Color selection</th>
<th>Text color</th>
<th>Black color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment color</td>
<td>Gray color</td>
<td>Gray color</td>
</tr>
<tr>
<td>Number color</td>
<td>Cyan color</td>
<td>Cyan color</td>
</tr>
<tr>
<td>Operator name color</td>
<td>Blue color</td>
<td>Blue color</td>
</tr>
<tr>
<td>Operator symbol color</td>
<td>Green color</td>
<td>Green color</td>
</tr>
<tr>
<td>Bracer color</td>
<td>Green color</td>
<td>Green color</td>
</tr>
<tr>
<td>Keyword2 color</td>
<td>Blue color</td>
<td>Blue color</td>
</tr>
<tr>
<td>Keyword3 color</td>
<td>Blue color</td>
<td>Blue color</td>
</tr>
<tr>
<td>Macro color</td>
<td>Blue color</td>
<td>Blue color</td>
</tr>
<tr>
<td>Guide color</td>
<td>Gray color</td>
<td>Gray color</td>
</tr>
</tbody>
</table>
Scripting the Script Editor

The editor is also scriptable, meaning you can write a script to write, change, or get information from another script. First, you need to make a reference to your script window. For example, here is code that creates a new script window and assigns a reference to it to the variable \texttt{ww}:

\begin{verbatim}
ww = New Window( "Script Test", <<script, "initial contents" );
\end{verbatim}

The initial contents can be left off, so that you create a blank script window. Next, you need to get a reference to the display box portion of the script window:

\begin{verbatim}
ed = ww[Script Box( 1 )];
\end{verbatim}

Although there is only one script box object, you still must call it by number.

There is a variety of messages that you can send your script box object.

Table 1.1  Messages for a Script Box Object

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ed &lt;&lt; get text();}</td>
<td>Places all the text in the script window in a single string.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; set text( &quot;string&quot; );}</td>
<td>Removes all the text currently in the script window and replaces it with the string argument.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; append text( &quot;string&quot; );}</td>
<td>Adds the string argument to the end of the script window.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; get line text( 2 );}</td>
<td>Places only the text from the designated line in a string.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; set line text( 2, &quot;string&quot; );}</td>
<td>Removes the text currently in the designated line and replaces it with the string argument.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; get line count();}</td>
<td>Returns the number of lines in the script window as an integer.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; get lines();}</td>
<td>Returns all the text in the script window as a list of strings, one line per string.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; reformat();}</td>
<td>Reformats the script.</td>
</tr>
<tr>
<td>\texttt{ed &lt;&lt; run();}</td>
<td>Runs the entire script in the script window.</td>
</tr>
</tbody>
</table>

The JSL Debugger (Windows Only)

In the JSL debugger, you can run the entire script, or step through the script one line at the time. This feature is available only Windows. If you run a script with the debug command, the script runs normally.

To use the debugger, the first line of the JSL script should be as follows:

\begin{verbatim}
/*debug step*/
\end{verbatim}
Be sure to type it exactly as shown above, in the first line of the script, with no extra blanks (including extra blanks within the comment). All letters must be lower case. Run the script to see the debugger.

The following tasks are possible in the debugger.

- **Run Without Debugger**  Closes the debugger windows and runs the script normally.
- **Step**  Steps through the script one command at a time. A green arrow shows the current command.
- **Run**  Runs the script from the current command.
- **Add Watch**  Adds a variable to the watch list, located at the bottom of the debugger window. Watched variables display their values as the script runs.
- **Remove Watch**  Removes a watched variable.
You can add a breakpoint to the script by clicking in the area to the left of a JSL line (the same area where
the green arrow circulates). A red circle appears at that line, and the script pauses when the line is executed.
Click the circle to remove the breakpoint.

Help with JSL

There are several places within JMP to get help writing or understanding a JSL script.

JSL Browsers

The Help menu contains indexes for browsing functions, objects, and display boxes in the scripting language:

- **JSL Functions**  Shows information about all functions.
- **Object Scripting**  Shows scriptable objects and what messages each can interpret.
- **DisplayBox Scripting**  Shows the elements in results windows and their messages.

An entry in the JSL Functions Index includes the syntax and a brief explanation. Many entries also have example code. The Topic Help button for each entry opens the Help system and shows you the entry for the item in the "JSL Syntax Reference," p. 439. See Figure 1.9 for the entry for `If()`.

**Figure 1.9 JSL Functions Index**
Show Commands

Show Commands lists all the scriptable objects and functions in the log window, producing a text report that is equivalent to the JSL Functions browser. The default argument (if none is given) is Builtins, which shows all JSL commands:

```javascript
Show Commands();
```

`Built-in Commands`

- Add 7 Internal
- Subtract 7 Internal
- Multiply 8 Internal
- Divide 8 Internal
- ...

Other arguments are:

- `Scriptables` (The platforms and their messages.)
- `DisplayBoxes` (For display tree objects and messages that you can send to them. Note that some objects are not scriptable.)
- `ScriptableObjects` (Any objects that currently have instances.)
- `StatTerms` (Places all the information in the Statistics Index into a JMP data table.)
- `All` for all of the above.

Try these commands and look in the log for the output.

Showing Translations

If you are running JMP in another language, you can get a table that shows all objects' JSL commands in English and the localized language.

```javascript
Show Commands( translations );
```

This command creates a data table that lists commands, the English name, and the localized name. Argument translations are in the column `Enum Local`.

Show Properties

Show Properties lists the messages that can be sent to a scriptable object and produces a text report that is equivalent to the Object Scripting browser. Scriptable objects might include a data table, an analysis platform, or a display.

```javascript
Open( "$SAMPLE_DATA/Big Class.jmp" );
Show Properties( Current Data Table() );
```

- `Begin Data Update [Action] [Scripting Only]`
- `Clear Column Selection [Action] [Scripting Only]`
- `Compress Selected Columns [Action] [Scripting Only] (Compresses each column into the most compact form.)`
- `Character data will be 1-byte if there are fewer than 255 levels. Numeric data will be 1-byte if the data is between -127 and 127.)`
- `Delete Columns [Action] [Scripting Only]`
- ...

...
Bivariate( Y( :weight ), X( :height ) );
Show Properties( Bivariate[1] ); //the analysis platform
  Show Points [Boolean] [Default On]
  Fit Mean [Action](Fits a flat line at the mean.)
  Fit Line [Action](Fits a regression line to the data.)
 ...

Show Properties( Report( Bivariate[1] ) ); //the platform's display tree
  Close [Boolean]
  Horizontal [Boolean]
  Open All Below [Action]
  Close All Below [Action]
 ...

JMP is scripted by a very simple language called JMP Scripting Language, or JSL. You might not need to ever learn JSL, because almost every feature in the product is accessible through a direct user interface, as well as through the scripting language. Even if you use JSL, you can usually get JMP to write the scripts for you rather than typing them in yourself. JSL is most useful to power users who want to extend JMP past its normal operating realm, or in production settings to automate a regularly scheduled analysis.

JSL is used in many places in JMP internally.

- Column Formulas are implemented internally in JSL.
- Platforms are launched using JSL internally.
- Platforms are interactively modified using JSL internally.
- Some graphics are performed through JSL.

**Confusion alert!** As described in the section on logical operators, a single pipe symbol (|) represents a logical OR. In the interests of brevity, programming and scripting manuals commonly use a | to represent the word *or* when discussing alternative values.

For example, a filepath can be either absolute or relative. When you see an argument to a filepath function as `absolute|relative`, this means that you enter *either*:

- `absolute` to indicate an absolute filepath *or*
- `relative` to indicate a relative filepath.

More than two options can be strung together with an *or* pipe in this way.

So, when you see words separated with a |, read it as *or*. 

---

**Chapter 2**

**JSL Building Blocks**

**Learn the Basic Components of JSL**
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>First JSL Script</td>
<td>25</td>
</tr>
<tr>
<td>The JSL Language</td>
<td>25</td>
</tr>
<tr>
<td>Lexical Rules of the Language</td>
<td>26</td>
</tr>
<tr>
<td>Data Elements</td>
<td>32</td>
</tr>
<tr>
<td>Operators</td>
<td>33</td>
</tr>
<tr>
<td>Context: Meaning is Local</td>
<td>36</td>
</tr>
<tr>
<td>Programming Versus Scripting</td>
<td>36</td>
</tr>
<tr>
<td>Data Table Context</td>
<td>37</td>
</tr>
<tr>
<td>Scoping Operators</td>
<td>38</td>
</tr>
<tr>
<td>Name Resolution</td>
<td>38</td>
</tr>
<tr>
<td>Name-Binding Rules</td>
<td>38</td>
</tr>
<tr>
<td>Frequently-Asked Questions</td>
<td>41</td>
</tr>
<tr>
<td>Function Resolution Rules</td>
<td>42</td>
</tr>
<tr>
<td>Gluing Expressions Together</td>
<td>42</td>
</tr>
<tr>
<td>Variables</td>
<td>43</td>
</tr>
<tr>
<td>Watch</td>
<td>43</td>
</tr>
<tr>
<td>Show Symbols, Clear Symbols, and Delete Symbols</td>
<td>44</td>
</tr>
<tr>
<td>Locking and Unlocking Symbols</td>
<td>45</td>
</tr>
<tr>
<td>Local</td>
<td>45</td>
</tr>
<tr>
<td>Datetime Operators</td>
<td>46</td>
</tr>
<tr>
<td>Constructing Dates</td>
<td>46</td>
</tr>
<tr>
<td>Extracting Parts of Dates</td>
<td>46</td>
</tr>
<tr>
<td>Arithmetic on Dates</td>
<td>47</td>
</tr>
<tr>
<td>Converting Datetime Units</td>
<td>47</td>
</tr>
<tr>
<td>Y2K-Ready Date Handling</td>
<td>47</td>
</tr>
<tr>
<td>Datetime Notation</td>
<td>48</td>
</tr>
<tr>
<td>Currency</td>
<td>53</td>
</tr>
<tr>
<td>Inquiry Functions</td>
<td>53</td>
</tr>
<tr>
<td>Functions that communicate with users</td>
<td>55</td>
</tr>
<tr>
<td>Writing to the log</td>
<td>55</td>
</tr>
<tr>
<td>Send information to the user</td>
<td>56</td>
</tr>
</tbody>
</table>
First JSL Script

This example shows you how to enter a script in the script editor, run it, and see the output.

2. Select View > Log on Windows, or Window > Log on Macintosh.
3. In the script window that appears, enter the following text:
   \[ X = 12 + 8; \\
   A = "Hello, World"; \\
   Show( X, A ); \]
4. Select Edit > Run Script.
   You can also press CONTROL-R on Windows, or COMMAND-R on Macintosh.

The result appears in the Log window:

\[ X = 20; \\
A = "Hello, World"; \]

You have just created two global variables, \(X\) and \(A\), and assigned values to them.

The JSL Language

JSL consists entirely of nested message names, with message contents enclosed in parentheses:

Message Name ( argument 1, argument 2, ... )

JSL expressions hold data, manipulate data, and send commands to objects.

The meaning of JSL phrases is dependent on who the message is sent to. The same name might mean one thing in one context and something entirely different in another context.

Almost anything that obeys certain punctuation rules, such as matching parentheses, is a valid JSL expression, though it might not be understood by the object it is sent to.

Here is a valid JSL expression:

\[
\text{New Window( "A Window",} \\
\text{<modal,} \\
\text{Text Box( "Hello, World" ),} \\
\text{Text Box( "-----" ),} \\
\text{Button Box( "OK" )} \\
\); \\
\]

Notice the following:

- Names can have embedded blanks.
- Message contents are enclosed in parentheses, which must be balanced.
- Items are separated by commas.
- You can use upper-case and lower-case characters interchangeably.
• Messages are commonly nested inside other messages.

**Lexical Rules of the Language**

The language consists of the following types of tokens. Each are discussed briefly in subsections below.

<table>
<thead>
<tr>
<th>Table 2.1 Types of Tokens in JSL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token</strong></td>
</tr>
<tr>
<td>Commas and Parentheses</td>
</tr>
<tr>
<td>Names</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Numbers and Dates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Quoted strings</td>
</tr>
<tr>
<td>Matrices</td>
</tr>
<tr>
<td>Lists</td>
</tr>
<tr>
<td>Associative Arrays</td>
</tr>
<tr>
<td>Operators</td>
</tr>
<tr>
<td>File Paths</td>
</tr>
<tr>
<td>Comments</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Commas**

Commas (,) separate items, such as items in lists, rows in matrices, or named arguments to a function.

```js
mylist = {1, 2, 3};
yourlist = List( 4, 5, 6 );
mymatrix = [ 3 2 1, 0 -1 -2 ];
```

**Note:** When you have a sequence of commands to execute, you do not separate them with commas. Rather, you glue them together with semicolons, as discussed under “Gluing Expressions Together,” p. 42.
Parentheses

Parentheses ( ) are used to group operations in an expression and to delimit arguments to a function, such as in root(49). Parentheses also delimit the end of a function name even when arguments are not needed. Consider e( ) below, where the empty argument “( )” after “e” is what distinguishes JSL’s function for e from the name “e.”

```
For( i = 0, i < 10, i++, Show( i, e() ∧ i ) );
```

Be careful that parentheses match. Every ( needs an ), or else errors result.

The script editor can match fences (parentheses, brackets, and braces). Press CTRL-] (COMMAND-B on Macintosh) with your cursor in any part of a script. The editor searches for fences, highlighting the text between the first set of opening and closing fences that it finds. Repeat this process to highlight the next-higher fence.

Names

A name is exactly what you think: something to call a thing. For example, when you assign the numeric value 3 to a global variable in the statement a = 3, “a” is a name. Commands and functions have names, too. In the statement Log( 4 ), the word “Log” is the name of the logarithm function. A name is any language token that is not a number or a string or a special symbol operator (such as + or – or ++ or ^).

Names have a few rules:

- Names must start with an alphabetic character or underscore (a-z A-Z _), and can continue with the following:
  - alphabetic characters (a-z A-Z)
  - numeric digits (0-9)
  - whitespace characters (spaces, tabs, and line and page delimiters)
  - double-byte characters
  - a few punctuation or special characters (apostrophes (‘), percent signs (%), periods (.), backslashes (\), underscore (_))

- When comparing names, the whitespace characters (like spaces, tabs, and newlines) are ignored, and upper and lower case is not distinguished. For example, the names Forage and for age are equivalent.

**Note:** Why does the language allow whitespace characters inside names? For one reason, it is good for the names of commands and options to match the equivalent commands in menus and windows. Another reason is because people think in terms of multi-word names. The price of allowing whitespace characters is that JSL needs more delimiters, specifically commas and parentheses.

- Actually, you can still have a name that is any other sequence of characters. If it does not obey the rules above, it needs to be quoted and placed inside a special parser directive called Name( ). For example, to use a global variable with the name –)(*%(*&$%A, you must use Name( ) every time it appears in JSL:

  ```
  Name( "–)(*%(*&$%A") = 42;
  foo = 4;
  ```
Print( foo + Name( "-")(*%(*&%%A" ) );

Name is harmless when it is not needed. For example, foo and Name( "foo" ) are exactly the same thing.

**Numbers**

Numbers can be written as integers, decimal numbers, in scientific notation with an E preceding the power of ten, and as dates, times, or date/time values. A single period by itself is the missing numeric value (sometimes called NAN for “not a number”).

For example, these are all numbers:

.   1   12   1.234  3E3  0.314159265E+1 1E-20

If you precede a number with a minus sign, the minus sign is usually interpreted as a prefix operator, not part of the number. You can follow a number immediately with an E followed by a power of ten you want to scale the number by. For example 3E2 means 3 times 10 to the power 2, or 300. If you need a negative exponent of ten in E notation, that minus sign is part of the number.

**Dates and Times**

JMP supports *datetime* values in a variety of common date/time notation formats. Datetime values are stored and computed as a number of seconds since midnight, January 1, 1904. However, you can enter a datetime value literally, using the format `ddMonyyyy:hh:mm:ss.ddd`. For example:

```plaintext
x = 14Feb2002:19:15:00;
```

Several shorter forms can also be used for datetime literals:

```plaintext
x = 14Feb2002:19:15;
x = 14Feb2002;
y = 19:15:00;
```

JMP has numerous operators for converting datetime values to strings in common notations, for example:

```plaintext
invitation = "Would you like to dine with me on " || Long Date( x ) || "?";
"Would you like to dine with me on Thursday, February 14, 2002?"
```

These and other datetime subjects are detailed in “Datetime Operators,” p. 46.

**Quoted Strings**

Strings are put in double quotation marks. Be careful to include the end quotation mark, or your string includes unintended text until it finds the next double quotation mark.

How do you put a double quotation mark inside a quoted string? Inside quoted strings, you can use a special escape sequence `\!` (backslash-bang) to precede a code for special characters. For example, run the following script and look at the title of the window:

```plaintext
New Window( "For example,\!"Hello\!" is a quoted string",
    Text Box( Char( Repeat( "*", 70 ) ) )
);
```
Warning: The null character is dangerous to use, because it is normally treated as the end of the string.

Table 2.2 Escape Sequences for Quoted Strings

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>blank</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return only</td>
</tr>
<tr>
<td>\n</td>
<td>linefeed (newline) only</td>
</tr>
<tr>
<td>\N</td>
<td>inserts line breaking characters appropriate for the host environment.</td>
</tr>
<tr>
<td>\f</td>
<td>formfeed (page break)</td>
</tr>
<tr>
<td>\0</td>
<td>null character. (See warning.) Type the number zero, not the letter O.</td>
</tr>
<tr>
<td>\</td>
<td>backlash</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quotation mark</td>
</tr>
</tbody>
</table>

a. On Macintosh, this is CR (carriage return character, hexadecimal '0D'). On Windows, this is CR LF (carriage return followed by a linefeed, hexadecimal '0D0A').

Sometimes, long passages require a lot of escaped characters. In these cases, use the notation `\[...\]` and everything between the brackets does not need or support escape sequences. Here is an example where `\[...\]` is used inside a double-quoted string.

```js
jslPhrase = "The JSL to do this is :\[
   a = "hello";
   b = a|| " world.";
   show(b);
]\ and you use the Submit command to run it.;"
```

**Matrices**

Matrices can be specified as signed numbers inside brackets, with values separated by blanks or other whitespace characters, and rows separated by commas or semicolons. Brackets are also used to represent subscripting, when brackets occur after another expression. You can represent an empty matrix of no rows and no columns by `[]`.

```
A = [1 2, 3 4];
two = A[1, 2]; // subscript picks the first row, second column element from A
Empty = [];
```

Matrices are discussed at greater length in the “Matrices” chapter.
Lists

Like matrices, lists are compound data structures, but they are more flexible than matrices. Lists are a way to store numerous items of different type: numbers, text strings, expressions, matrices, and even other lists. Lists can be expressed as arguments for the function `List( )`, or simply between `{` and `}` curly braces:

```julia
A = List( 1, 2, B, Sqrt( 3 ), List( a, b, 3 ) );
A = {1, 2, B, Sqrt( 3 ), {a, b, 3}};
```

Lists are discussed at greater length under "Lists," p. 68 in the "Programming Functions" chapter.

Associative Arrays

An associative array maps unique keys to values (possibly non-unique). Other languages call this type of data structure a dictionary a map, a hash map, or a hash table. A key is a quoted string, while the value associated with that key can be strings, numbers, dates, matrices, lists, and so forth. For example:

```julia
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 127640;
cary["gps"] = [78.7812 35.7873];
cary["weather"] = {"sunny", "warm"};
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
```

Associative arrays are discussed at greater length under "Associative Arrays," p. 72 in the "Programming Functions" chapter.

Operators

Operators are one- and two-character symbols for common arithmetic actions. Operators come in several varieties: infix (with arguments on either side, such as + in 3+4, or = in a=7), prefix (with one argument on its right side, such as !a for logical negation), or postfix (with one argument on its left side, such as a++ for incrementing a).

JSL operators all have function equivalents. Operators are discussed at greater length under “Operators,” p. 33.

File Paths

In JMP, the preferred file path format is the POSIX (or UNIX) format, with forward slashes as separators. Each host still accepts its native format for compatibility. This, along with path variables, often eliminates the need for `if(host is(...))...` logic to open files in a portable script.

Path variables are supported at the beginning of a POSIX path. JMP recognizes `HOME`, `DOCUMENTS`, `SAMPLE_DATA`, `ENGLISH_SAMPLE_DATA`, `SAMPLE_IMPORT_DATA`, `SAMPLE_SCRIPTS`, `SAMPLE_IMAGES`, `JMP_HOME`, and `TEMP` as path variables. If you also have installed JMP Genomics, `GENOMICS_HOME` is also available. They are used with a dollar sign at the beginning of a path.

```julia
Open("$$SAMPLE_DATA/Big Class.jmp")
```

Users can also add their own path variables, or override some of the built-in ones with the JSL functions
The JSL Language

posixPath = Set Path Variable(varName, posixPath);
posixPath = Get Path Variable(varName);

varName is case-sensitive and does not include the dollar sign.

Confusion alert! You cannot override HOME, DOCUMENTS, or TEMP, and Get Path Variable does not retrieve the settings for them. Instead, use this JSL function:

posixPath = Convert File Path("$HOME");

There are JSL functions for accessing the default directory:

posixPath = Set Default Directory(posixPath);
// returns the new directory path
posixPath = Get Default Directory();

The default directory is used for resolving relative paths. The file search list is also used for resolving relative paths for opening files (not for saving).

You can convert among file paths using the Convert File Path command.

Convert File Path (path, <absolute|relative>, <POSIX|windows>, <base(dir_path)>);

For the <optional> arguments above, the defaults are absolute, POSIX, and a base path of the default directory. The input path can be in Windows or POSIX style.

Path variables have the following characteristics:

- When changing the language that JMP runs in, their definitions are changed to match the language automatically.
- When changing from running one version of JMP to another (for example, JMP 8 and then JMP 9), their definitions are changed to match the language automatically.
- The path variable $ENGLISH_SAMPLE_DATA always holds the definition of the English Sample Data folder.

The path variable $ALL_HOME points to a folder that can be accessed by all users on a machine. These folders are different according to the operating system:

Table 2.3 $ALL_HOME Path Variable

<table>
<thead>
<tr>
<th>Operating System</th>
<th>All Users Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 7 or Windows Vista</td>
<td>C:\ProgramData\SAS\JMP\9</td>
</tr>
<tr>
<td>Windows XP</td>
<td>C:\Documents and Settings\All Users\Application Data\SAS\JMP\9\</td>
</tr>
<tr>
<td>Macintosh</td>
<td>/Library/Application Support/JMP/9/</td>
</tr>
</tbody>
</table>
Comments

Comments are notations that you add to your code that are ignored by the parser. Comments can be started by either // or /*. The // form continues until the end of a line. The */ form continues until a closing */. Comments cannot be inside quoted strings.

Table 2.4 Comments in JSL Scripts

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>// comment</td>
<td>Start a comment line; does not have to be at beginning of line, but everything following to the end of the line is a comment.</td>
</tr>
</tbody>
</table>
| /* */  | /* comment */ | A comment that can appear in the middle of a line of script. Script text before and after the comment is undisturbed. The /*...*/ punctuation is a convenient way to remove portions of script temporarily. These two statements are equivalent:  
  \[ c=1+ /* comment */ 2; \]  
  \[ c=1+2; \] |

Note: Add //! to the beginning of a script to prevent it from being seen by users of the script.

Data Elements

Generally the language tokens translate into JSL data elements and expressions. JSL uses the following basic elements in the language:

- integers
- numbers (floating point)
- character strings
- names
- lists
  - a list holds a number of other values, including nested lists and expressions
  \[ \text{Doe family} = \text{List( John Doe, Jane Doe, Baby Doe );} \]
  \[ \text{Doe family} = \{ \text{John Doe, Jane Doe, Baby Doe } \}; \]
  - associative arrays
    - an associative array maps keys to values, which can be just about any other data element
  - matrices
    - a matrix is a row-by-column table of numbers
    \[ \text{Design Matrix} = [1 1, 1 -1]; \]
  - expressions
    - expressions can be treated as complex data and manipulated by certain operators
  - references to scriptable objects:
Chapter 2

The JSL Language

33

JSL Building Blocks

1. data table reference obtained from the Open() or Current Data Table() function (represented by \( dt \) in syntax summaries)
2. data column reference obtained from the Column() function (represented by \( col \) in syntax summaries)
3. scriptable object reference obtained from platform launches or Scriptable class subscripting (represented by \( obj \) in syntax summaries)
4. displayBox reference reports or parts of reports in windows (represented by \( db \) in syntax summaries)

- data table columns

Operators

In order to make writing algebraic expressions natural, JSL has adopted certain special character operators. These operators have the same meaning as if the phrase had been written as a message or function. For example, the following two statements are equivalent:

\[
\text{Net Income After Taxes} = \text{Net Income} - \text{Taxes};
\]

\[
\text{Assign( Net Income After Taxes, Subtract( Net Income, Taxes ) )};
\]

The assignment operation can be written either as a function Assign or as an infix operator \( = \). Similarly, subtraction can be done with the Subtract function, or the infix minus sign; they are equivalent inside JMP.

Another common operator is the semicolon (\( ; \)). The semicolon is a gluing operator that is used to both separate yet join one expression to another in a programming sequence. The function equivalent of this is Glue, so \( \text{a;b} \) is the same as \( \text{Glue(a,b)} \). The semicolon or Glue operator returns the result of its last argument. It is also legal to end an expression with a semicolon. This might lead you to think of it as a statement terminator like some other languages, but it is designed as an infix operator. Terminating semicolons are allowed at the end of a script stream and before a closing parenthesis or closing brace: \( ) \) or \( } \).

Table 2.5 shows operators and their function equivalents. The operators are grouped in their order of precedence, where the binding priority decreases with each group. For example, in \( \text{a*b+c} \), the multiplication \( \text{a*b} \) is done before the addition of \( \text{c} \).

Table 2.5 Operators and Function Equivalents In Precedence Order

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>List</td>
<td>{a,b}</td>
</tr>
<tr>
<td></td>
<td>List(a,b)</td>
<td></td>
</tr>
</tbody>
</table>

Note: Usually white space is ignored in JSL, so “netincomeaftertaxes” and “net income after taxes” are the same thing. There is one exception: the two-character operators must not have a space between characters, or they will be misunderstood. Always type these without spaces in between:

\[
|, *, <>, >=, !=, ==, +=, -=, *=, /=, ++, --, :=, |/, <<, ::, :*, :/, :^, /, /*, */
\]

The assignment operation can be written either as a function Assign or as an infix operator \( = \). Similarly, subtraction can be done with the Subtract function, or the infix minus sign; they are equivalent inside JMP.

Another common operator is the semicolon (\( ; \)). The semicolon is a gluing operator that is used to both separate yet join one expression to another in a programming sequence. The function equivalent of this is Glue, so \( \text{a;b} \) is the same as \( \text{Glue(a,b)} \). The semicolon or Glue operator returns the result of its last argument. It is also legal to end an expression with a semicolon. This might lead you to think of it as a statement terminator like some other languages, but it is designed as an infix operator. Terminating semicolons are allowed at the end of a script stream and before a closing parenthesis or closing brace: \( ) \) or \( } \).

Table 2.5 shows operators and their function equivalents. The operators are grouped in their order of precedence, where the binding priority decreases with each group. For example, in \( \text{a*b+c} \), the multiplication \( \text{a*b} \) is done before the addition of \( \text{c} \).

Table 2.5 Operators and Function Equivalents In Precedence Order

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>List</td>
<td>{a,b}</td>
</tr>
<tr>
<td></td>
<td>List(a,b)</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.5  Operators and Function Equivalents In Precedence Order  (Continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Subscript</td>
<td>Subscripts identify specific elements within a data element ( a ), where ( a ) could be a list, a matrix, a data column, a platform object, a display box, and so forth.</td>
</tr>
<tr>
<td>++</td>
<td>Post Increment</td>
<td>Adds one (1) to ( a ), in place.</td>
</tr>
<tr>
<td>--</td>
<td>Post Decrement</td>
<td>Subtracts one (1) from ( a ), in place.</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
<td>Raise ( a ) to exponent power ( b ). With only one argument, 2 is assumed as the power, so Power(( x )) computes ( x^2 ).</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>Reverses sign of ( a ).</td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
<td>Maps nonzero values to 0, maps 0 values to 1.</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>Multiplies ( a ) by ( b ).</td>
</tr>
<tr>
<td>:*</td>
<td>EMult</td>
<td>Elementwise multiplication for matrices ( a ) and ( b ).</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>Divide(( a, b )) divides ( a ) by ( b ). Divide(( x )) interprets the argument as a denominator and implies 1 as the numerator, yielding the reciprocal ( 1/x ).</td>
</tr>
<tr>
<td>:/</td>
<td>EDiv</td>
<td>Elementwise division for matrices ( a ) and ( b ).</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>Adds ( a ) and ( b ).</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>Subtracts ( b ) from ( a ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>Syntax</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| matrix1 || matrix2  
Concat(matrix1, matrix2) | Horizontally concatenate matrices.                                           |
| | matrix1/matrix2  
V Concat(matrix1, matrix2) | Vertically concatenate matrices.                                             |
| a::b        | Index(a,b)                                  | For matrices, generates the integers from \(a\) to \(b\).                     |
|             |                                             | (Colons are also used as infix operators for scoping, where \(:\) means data table column \(a\), and \(::\) means JSL global variable \(a\). See “Global Scoping Operator,” p. 40.) |
| object << message |                                              | Send \(message\) to \(object\).                                             |
| a==b        | Equal(a,b)                                 | Boolean values for comparisons. They all return 1 if true, 0 if false.        |
|             |                                             | Missing values in either \(a\) or \(b\) causes a return value of missing, which evaluates as neither true nor false. |
| a!=b        | Not Equal(a,b)                             |                                                                            |
| a<b         | Less(a,b)                                  |                                                                            |
| a<=b        | Less or Equal(a,b)                         |                                                                            |
| a>b         | Greater(a,b)                               |                                                                            |
| a>=b        | Greater or Equal(a,b)                     |                                                                            |
| a<=b<c      | Less Equal Less(a,b,c)                    | Range check. Return 1 if true, 0 if false.                                    |
| a<b<=c      | Less Less Equal(a,b,c)                    | Missing values in either \(a\) or \(b\) propagate missing values.          |
| a&&b        | And(a,b)                                   | Logical And. Returns true if both are true. See “Missing Values,” p. 82 in the “Programming Functions” chapter, for treatment of missing values. |
The JSL language is used both for scripting and programming. Programming emphasizes results, which build from the inside out. Scripting emphasizes context, which builds from the outside in. Understanding this distinction is crucial to understanding how JMP executes a script. For producing more complex JMP programs, we recommend that you use namespaces, as described in “Advanced Scoping and Namespaces,” p. 390 in the “Advanced Concepts” chapter.

### Programming Versus Scripting

For example, consider how a typical programming statement evaluates:

\[
x = \text{Log}( a \times b ^ (c + d) )
\]

The goal of this program is to find a result to assign to \( x \). According to the rules of precedence for arithmetic operators, this starts on the inside by evaluating \( c \) and \( d \) and adding them together. Next \( b \) is evaluated and then raised to that power \( (c + d) \), and that result in turn is multiplied by the evaluation of \( a \). That result is next passed to the \text{Log} function, which finds the logarithm, and finally that result is assigned to \( x \).

Evaluation started on the inside with \( c + d \) and worked its way out.

Now consider a typical scripting statement:
Fit Model( Effects( a, b, a * b ), Y( c ), Run Model );

The goal of this script is to launch an analysis platform and return a report. Evaluation starts on the outside with Fit Model, which calls the platform for fitting models. This platform in turn knows that Effects is specifying the independent terms of a model, and in that context, the phrase a*b takes on a whole new meaning. In the Log statement above, it meant to multiply. In the Fit Model statement, a*b means the interaction term of a and b. In turn, a and b are evaluated as the data columns assigned to the global variables a and b. The platform also knows that Y means that the column assigned to c is to be the response term. Finally the platform sees Run Model and goes to work. Evaluation started on the outside with Fit Model and worked its way in.

Data Table Context

A formula column is evaluated down the rows of the data table. If a script needs to refer to a specific cell each time, then you give a subscript to a specific row number. For example, this formula column gives the ratio within each row, where height and weight are columns such as in the sample data table Big Class:

New Column( "Ratio", formula( Height / Weight ) );

This gives the ratio of each row’s Height to the Weight in row 1:

New Column( "Ratio B", formula( Height / Weight[1] ) );

There is a current row number that determines the cells of the data table, which can be set or displayed with the function Row(), for example.

Row();     // returns current row number
Row() = 3; // makes the third row the current row

By default, the current row is always 0, which means that the following statement on its own returns a missing value and be generally useless.

ratio = height / weight;

However, if you include something in the script to set the current row to a specific row number (for example, the third), the result is assigned to the global variable ratio.

Row() = 3;
ratio = height / weight;

Another possibility is to ignore the current row and instead use subscripts to ask for a specific row number. This version divides the height in row 3 by the weight in row 4:


Still another possibility is to create a column ratio and iterate the division a row at a time down the whole column:

New Column( "ratio" );
For Each Row( ::ratio = height / weight );

If you are working with global variables, pre-evaluated statistics, and so forth, you do not need to consider the current row. (Pre-evaluated statistics are single number values computed from the data table, as discussed in “Pre-Evaluated Statistics,” p. 150 in the “Data Tables” chapter.)
::myCalc = Col Mean( height ) / Col Mean( weight );

Scoping Operators

The scripts in the previous section used colons : or double colons :: in front of some names. These are *scoping operators*. They tell JMP how to interpret the names that follow in cases that could be ambiguous. For example, when a data table column and a JSL global variable have the same name. Scoping operators and the rules for name resolution are discussed in detail under “Name Resolution,” p. 38, later in this chapter.

:x; // data table column
::x; // JSL global variable
x; // depends on state when first used

Name Resolution

The following types of objects can be identified by name:

- Columns and table variables in a data table
- Global variables, which exist for a session
- Scriptable object types
- Parameters and *Locals* inside formulas

If a name is just part of a script for an object, then its meaning is usually the name of an option or method in the object. Otherwise it is considered by the evaluator using the rules in this section. If the name is followed by parentheses, then it is regarded as a function. Otherwise, the name is looked up as some object.

*Global variables* are names that exist for the remainder of a session to hold values. Globals can contain many types of values, including numbers, strings, lists, and references to objects. They are called globals because they can be referred to almost anywhere, not just in some specific context.

Most of the time, you can just use an object’s name directly to refer to the object.

\[
\begin{align*}
\text{Ratio} &= \text{Height} / \text{Weight}; \\
N &= N + 1; \\
\text{Distribution( ... )};
\end{align*}
\]

Now the trick is to learn the rules as to when you are referring to a column, when to a global, and when to a type. This knowledge helps you to know when you can use the name directly and when you need to qualify it somehow.

Name-Binding Rules

From a programming point of view, name resolution is important when getting and setting values in a script. Names that are referred to for name-reference, rather than to get or set a value, are not resolved. Rather they are just passed along, as with a script to launch a platform specifying certain columns to be analyzed.

There are six possible resolutions for any name in a script. JMP tries each of the following in order:
1. If followed by a pair of parentheses ( ), look it up as a function; see "Function Resolution Rules," p. 42, for details.
2. If not prefixed by ;, look it up as a global variable.
3. If not prefixed by ::, look it up as a data table column or table variable.
4. Look it up as a local variable.
5. Look it up as a platform launch name (for example, Distribution or Bivariate).
6. If used as the target of an assignment (as an L-value), create and use a global variable.

A name in a script is resolved the first time it is used to get or set a value, and its resolution persists thereafter. The operators : and :: are always resolved respectively as \texttt{As Column()} and \texttt{As Global()}.

**Exceptions**

If a name is resolved to a column in a data table that is closed, then it re-resolves the next time it gets or sets in case the data table has been opened in the meantime. Other exceptions are function definitions, column formulas, and nonlinear formulas.

**When an Unscoped Name Refers to a Table Column**

When an unscoped name is resolved to get or set a value, then it refers to a column in a data table, rather than a global variable under these circumstances:

- if no global, local, or parameter of that name already exists,
- and the data table in context has a column of that name,
- and
  - either the current row is set to a positive value

**Warning:** The current default row is now 0 instead of 1, as in JMP 3 and earlier.

- or the name is subscripted. For example, \texttt{A[i]}.

If the data table has a table variable by that name, then the table variable takes precedence. In all other cases, it binds to a global, local, or parameter.

**Note:** When an unqualified name in JSL is resolved to a data column in a data table, it must be associated with the current data table. This is different from the rule in JMP 4 and earlier.

**Exceptions**

Column formulas and nonlinear formulas.

**Column Scoping Operators**

Column scoping operators can be used to force names to be resolved as columns.
1. The prefix colon (:) means that the name refers to a table column or table variable only, never a global. The prefix : refers to the current data table context. The current data table is the data table that Current Data Table() either returns or is assigned.

: colName

2. The infix colon (: ) operator extends this notion to specify which data table has the column by using a data table reference (data table references are discussed in the chapter “Data Tables,” p. 85). The function equivalent is As Column.

dataTableRef:name;
As Column(dataTableRef, name);

Therefore, these are equivalent:

:name;
Current Data Table():name;
As Column( Current Data Table(), name );

The Column function can also be used. However, it always evaluates its arguments, so you need to specify a row to refer to a cell rather than the whole column.

Column( "X" ); // Refers to column X.
Column( dt, 2 ); // Refers to the second column of the specified data table.
Column( "X" )[12]; // Refers to the cell of the twelfth row of column X.
Column( a ); // Evaluates a and looks up the column of the result.

Global Scoping Operator

A global scoping operator can be used to force names to be resolved as globals. The prefix double-colon (::) means that the name refers to a global only, never a table column. The function equivalent is As Global.

:: globalName;
As Global( globalName );

Table 2.6 Scoping operators

<table>
<thead>
<tr>
<th>Operator and Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>: &amp; As Column</td>
<td>: name dt:name As Column(dt, name)</td>
<td>Scoping operator. Forces name to be evaluated as a data table column in the table given by the optional data table reference argument, dt, rather than as a global variable.</td>
</tr>
<tr>
<td>:: &amp; As Global</td>
<td>:: name As Global(name)</td>
<td>Scoping operator. Forces name to be evaluated as a global variable, not a data table column. (Notice that double-colon is also used as an infix operator to represent ranges.)</td>
</tr>
</tbody>
</table>
Frequently-Asked Questions

Should you scope?

Yes. When in doubt, scope. Scoping is especially important if you are writing scripts that define globals with the same names as data table columns that you are not aware of. For example, scripts that might be used by many people on a variety of data tables. Since global names always take precedence, you have to work to ensure that the column name references map to a column and that globals do not map to a column. If you are writing such scripts, consider using explicit scoping and namespaces. See “Advanced Scoping and Namespaces,” p. 390 in the “Advanced Concepts” chapter.

Prefix scope operators do not take run-time overhead after the first resolution. Infix scope operators always take run-time overhead.

What is the difference between a column referred to by name, and a column reference? If I have a column reference in a global, how do I assign a value to a cell in the column?

Try submitting the following examples one line at a time. These scripts assume that you have a data table with a column named “A” and you do not have a global variable named “A.” The first few examples begin with a Row()=3 assignment to set the current row to a valid row, in this case the third row. The current row is irrelevant in the last few examples.

```
// assumes a data table with a column "A" and sufficient rows
Row() = 3;  A = 1;           //sets the current row of A to 1
Row() = 3;  :A = 2;          //sets the current row of A to 2
Row() = 3;  :"A" = 3;        //sets the current row of A to 3
ColA = Column( "A" );        //sets the global ColA to refer to column A
ColA[3] = 4;                 //sets the third row of A to 4
ColA << Formula( 6 );        //sets the formula for A to be 6
```

Confusion alert!
The current row for scripting is not related to rows being selected (highlighted) in the data table or to the current cursor position in the data table window. The current row for scripting is defined to be zero (no row) by default. You can set a current row with Row (for example, Row() = 3). Please note that such a setting only lasts for the duration of that script. Then, Row() reverts to its default value, zero. This behavior means that submitting a script all at once can produce different results than submitting a script a few lines at a time.

Another way to establish a current row for a script is to enclose it in For Each Row, which executes the script once for each row of the current data table. See “What is the Current Row?,” p. 101 in the “Data Tables” chapter, for more information.

Will a scoping operator “stick” to its name?

Yes. Once you use a scoping operator with a name, that name continues to be resolved accordingly for all subsequent occurrences of the name. For example, to protect against overwriting data tables that happen to contain a column named “i” when you use i as an index inside a For() loop, declaring ::i at the head of the script is all the protection that you need for the whole script.

::i = 1;
Gluing Expressions Together

The semicolon operator evaluates each argument in succession, returning the result of the last one. For example:

```plaintext
i=0; j=2
```

The semicolon in other languages is used as a statement terminator character. In JSL it is just a binary operator for gluing together expressions. The result returned is the result of the right-most expression. The function version of the semicolon is `Glue`, which makes the following two statements the same:

```plaintext
Glue(a=2,b=3)
a=2; b=3
```

You can also add a semicolon at the end of an expression as a terminator character, but it is not required. The semicolon is ignored if nothing follows it. Trailing semicolons are allowed at the end of a script stream, and before a closing parenthesis or closing brace: `)` or `}`. It is useful to get in the habit of using a final semicolon to avoid errors when copying and pasting small scripts into larger ones.

The `First` function works just like `Glue( ; )` except that it returns the first argument's result instead of the last's. For example, this script increments a counter global `i` after getting the `i`th value of a variable, but enclosing the two steps inside `First( ; )` enables the script to return the datum, not the new value of the counter.

Function Resolution Rules

A name followed by a list of arguments enclosed in parentheses is the syntax for a number of contexts in JSL:

- a call to a built-in function
- a call to a user-defined function
- a named argument
- a nesting in a script for an object: either an option with an argument, or a sub-object with messages, or any other use recognized by the object itself
Variables

Assigning a variable this way:

\[ x = 2; \]

makes a global variable. JMP has a single global variable space that all scripts use. You can also create local variables.

We recommend that you begin each script with `Names Default To Here(1)`. This mode makes all your variables local to your script, instead of using the global space. This strategy ensures that scripts cannot “collide” by accidentally using the same global variable names and over-writing their values.

If you turn the `Names Default To Here` mode on, then assigning a variable as shown above makes that variable accessible throughout your script, but does not affect the global variable space.

The following commands can help you manage variables.

Watch

`Watch` lets you keep track of the value in a variable.

`Watch` draws a window to report the values of variables you list. Use the named argument `All` to watch the values for all variables. The window updates whenever values change while the script is executing, or you can...
manually change values yourself by simply editing the value shown in the Watch window. This is a useful debugging tool.

Watch(a, b);
Watch(all);

Show Symbols, Clear Symbols, and Delete Symbols

Show Symbols() lists all variables and namespaces that are defined both globally and in the local script, along with their current values. Clear Symbols erases the values set for variables that are defined both globally and in the local script.

Note: The older Show Globals() and Clear Globals() functions are aliases of the newer Show Symbols() and Clear Symbols() functions.

The function Delete Symbols() actually removes all global variables and namespaces, while Clear Globals() removes only their values.

c = "Global Variable"; // this is in the global space
Names Default To Here(1);
a = 5; // this is local to this script
b = 6; // this is local to this script

Show Symbols();
// Locals
a = 5;
b = 6;
// 2 Locals

// Globals
c = "Global Variable";
// 1 Globals

Clear Symbols();
Show Symbols();
// Locals
a = Empty;
b = Empty;
// 2 Locals

// Globals
c = Empty;
// 1 Globals

Delete Symbols();
Show Symbols();

Nothing is displayed in the log after the last Show Symbols(), because they have all been completely removed from memory.
Note: None of these three functions affect namespaces, although they do affect variables that hold references to namespaces. To show namespaces, use Show Namespaces(). To delete a namespace, use ns << Delete.

Locking and Unlocking Symbols

If you want to lock a symbol to prevent it from being changed, use Lock Symbols(). (Lock Globals() is an alias.)

   Lock Symbols (name1, name2, ...)

To release the lock and enable the global to be changed, use Unlock Symbols(). (Unlock Globals() is an alias.)

   Unlock Symbols (name1, name2, ...)

The primary use of these two commands is to prevent inadvertent changes to variables. For example, locking a variable prevents a Clear Symbols() command from clearing a variable that is being used by another script.

Note: You cannot use Lock Symbols() to lock a namespace. Instead, use ns << Lock().

Local

Local lets you deal with local variables without affecting the name space around you. For example, suppose you want to call something x, but there could already be an x global in another script that is calling your script. Just put x inside a Local block. This differs from using Names To Default To Here(1) because you can set portions of a script local, thus insulating a portion from the rest of the script.

   y= Local({x},
   x = sqrt(b^2-4*a*c);
   {(-b+x)/(2*a), (-b-x)/(2*a)});

The general form is:

   Local( {list_of_variables_or_assignments}, body );

All occurrences of names in the body expression that match names listed for the first argument are resolved locally. Locals can be nested in Functions and other Locals.

Another example: in the following expression, mn and mx are never seen outside the scope of the Local function:

   y = local( {mn=min(x,y),mx=max(x,y)}, mx*(mx-1)+mn );

This example manipulates an expression temporarily without changing the permanent contents of the expression:

   polynomial = Expr( a * x ^ 2 + b * x + c );
   Show( polynomial );
   polynomial = a * x ^ 2 + b * x + c;
Local( {polynomial = Insert( Name Expr( polynomial ), Expr( d * x ^ 3 ), 1 )},
         Show( polynomial )
     );
  polynomial = d * x ^ 3 + a * x ^ 2 + b * x + c;

Show( polynomial );
  polynomial = a * x ^ 2 + b * x + c;

Datetime Operators

Constructing Dates

You can construct dates with the following operators. If it were the stroke of midnight, 1 December 2003, all of the following statements would be equivalent. Each would return 3153081600, and you could pass any of them to a date notation operator such as Long Date to get a recognizable format:

Date DMY(1,12,2003);
Date MDY(12,1,2003);
Today();
3153081600

long date(today());
"Monday, December 1, 2003"

These operators are handy if you are setting up a date column to label rows, or any other case where you can describe a series of date values generally:

new column(Entry date, format("m/d/y"),formula(Date DMY(row(),3,2001)));
// be sure the format is available on your machine

Extracting Parts of Dates

You can extract parts of date values using the operators Month, Day, Year, Day Of Week, Day Of Year, Week Of Year, and Time Of Day, which all return integers. For example:

day of year(today());
335
You can perform the usual arithmetic operations with date/time data as with any other numeric data. For example, you could find the days elapsed between entries:

```
new column("Days elapsed",
  formula(
    if(row()==1,., //to avoid error for row 1
     //else for rows after 1:
     (:Entry date[row()]-:Entry date[row()-1])/in days()));
```

**Converting Datetime Units**

The In Minutes, In Hours, In Days, In Weeks, and In Years operators are useful for re-expressing a time interval in units other than seconds. For example, this returns the number of fortnights between now and the next Leap Day.

```
(Date DMY(29,2,2004)-Today())/InWeeks(2);
```

Usually $n$ is 1, but you could use In Years(10) (for example, to re-express an interval in decades).

**Y2K-Ready Date Handling**

JMP uses its own Y2K-ready algorithms for interpreting and displaying date/time strings rather than supporting operating system-specific date/time formats. JMP respects the date/time separator-characters chosen in the Regional Settings control panel (Windows) or the Date&Time control panel (Macintosh) for interpreting and displaying dates.

JMP always displays four-digit years regardless of Regional Settings. If for some reason it should be necessary to show two-digit years, use JMP’s operators for manipulating character strings.

Two-digit years are interpreted according to the current system clock year, according to the rules below. To avoid ambiguity or override the default interpretation, type a four-digit year value.

---

Confusion alert! Because Sunday is the first day of the week, the first Sunday of each year *always* starts week 2. Days before the first Sunday (if there are any) are in week 1. This means that in a year in which January 1 is on Sunday, the first week is counted as the second week.

Example:

```
Week of Year(Date DMY(1,1,2006));
2
```

In years in which Sunday is the first day of the year, subtract 1 from Week Of Year() to obtain the correct week number.
Note that three locale-specific formats (Locale Date, Locale Date Time h:m, Locale Date Time h:m:s) always display the date time according to your regional settings.

### Datetime Notation

Several operators convert numbers directly to common formats.

- **Short Date**(2998080000);
  - "01/02/1999"
- **Long Date**(2998080000);
  - "Saturday, January 2, 1999"
- **Abbrev Date**(2998080000);
  - "Jan 2, 1999"
- **mdyhms**(2998080000);
  - "01/02/1999 12:00:00 AM"

You can also use the **Format** and **InFormat** operators for displaying and entering date/time values in one of many notation formats.

- **Format**(3096558900,":day:hr:m");
  - "35839:19:15"
- **InFormat**("02Jan1999 13:01:55","ddMon yyyy h:m:s");
  - 02Jan1999:13:01:55

Dates that are the results of expressions that are to be displayed in a text window (like the log window) are given as a date attribute using **As Date**. For example:

\[
x = \text{As Date}(8\text{Dec2000} + \text{inDays}(2))
\]

shows as

10Dec2000

---

**Table 2.7 How JMP interprets two-digit years**

<table>
<thead>
<tr>
<th>What you type</th>
<th>When it is evaluated</th>
<th>Result</th>
<th>Examples</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>00–10</td>
<td>before 1990 (Win)</td>
<td>19__</td>
<td>type 5 in year 1979</td>
<td>1905</td>
</tr>
<tr>
<td></td>
<td>before or during 1990 (Mac)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>during or after 1990 (Win)</td>
<td>20__</td>
<td>type 5 in year 1991</td>
<td>2005</td>
</tr>
<tr>
<td></td>
<td>after 1990 (Mac)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11–89 (Win)</td>
<td>any time</td>
<td>current century</td>
<td>type 13 in year 1988</td>
<td>1913</td>
</tr>
<tr>
<td>11–90 (Mac)</td>
<td></td>
<td></td>
<td>type 13 in year 2024</td>
<td>2013</td>
</tr>
<tr>
<td>90–99 (Win)</td>
<td>before 2011</td>
<td>19__</td>
<td>type 99 in year 1999</td>
<td>1999</td>
</tr>
<tr>
<td>91–99 (Mac)</td>
<td>during or after 2011</td>
<td>20__</td>
<td>type 99 in year 2015</td>
<td>2099</td>
</tr>
</tbody>
</table>
The formats JMP currently supports are shown in Table 2.8. You can also use them for the `format` argument to a `Format` message to a data column, as discussed in “Display Formats,” p. 127 in the “Data Tables” chapter. Note that the date-separator character on your machine might differ from the / character shown here.

You can enter time values in 24-hour format (military time) or with AM/PM designators.

### Table 2.8 Date/time formats supported in JMP

<table>
<thead>
<tr>
<th>Type</th>
<th>Format string</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date only</td>
<td>&quot;m/d/y&quot;</td>
<td>&quot;01/02/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;mmddyyyy&quot;</td>
<td>&quot;01021999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;m/y&quot;</td>
<td>&quot;01/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y&quot;</td>
<td>&quot;02/01/1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddmmyyyy&quot;</td>
<td>&quot;02011999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy&quot;</td>
<td>&quot;02Jan1999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Monddyyyy&quot;</td>
<td>&quot;Jan021999&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d&quot;</td>
<td>&quot;1999/01/02&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyymmdd&quot;</td>
<td>&quot;19990102&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyy-mm-dd&quot;</td>
<td>&quot;1999-01-02&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyyQq&quot;</td>
<td>1999Q1</td>
</tr>
<tr>
<td>Date and time</td>
<td>&quot;m/d/y h:m&quot;</td>
<td>&quot;01/02/1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;01/02/1999 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;m/d/y h:m:s&quot;</td>
<td>&quot;01/02/1999 13:01:55&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;01/02/1999 1:01:55 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y h:m&quot;</td>
<td>&quot;02/01/1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02/01/1999 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;d/m/y h:m:s&quot;</td>
<td>&quot;02/01/1999 13:01:55&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02/01/1999 1:01:55 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d h:m&quot;</td>
<td>&quot;1999/01/02 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1999/01/02 1:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;y/m/d h:m:s&quot;</td>
<td>&quot;1999/01/02 13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1999/01/02 1:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyy h:m&quot;</td>
<td>&quot;02Jan1999 13:01&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;02Jan1999 1:01 PM&quot;</td>
</tr>
</tbody>
</table>
Table 2.8 Date/time formats supported in JMP (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Format string</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day number and time</td>
<td>&quot;ddMonyyyy h:m:s&quot;</td>
<td>&quot;02Jan1999:13:01:02&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyyh:m&quot;</td>
<td>&quot;02Jan1999:13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;ddMonyyyyh:m:s&quot;</td>
<td>&quot;02Jan1999:13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Monddyyyy h:m&quot;</td>
<td>&quot;Jan021999 13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Monddyyyy h:m:s&quot;</td>
<td>&quot;Jan021999 13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;:day:hr:m&quot;</td>
<td>&quot;34700:13:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;:day:hr:m:s&quot;</td>
<td>&quot;34700:13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;h:m:s&quot;</td>
<td>&quot;13:01:02 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;h:m&quot;</td>
<td>&quot;13:01 PM&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyy-mm-ddThh:mm&quot;</td>
<td>1999-01-02T13:01</td>
</tr>
<tr>
<td></td>
<td>&quot;yyyy-mm-ddThh:mm:ss&quot;</td>
<td>1999-01-02T13:01:02</td>
</tr>
<tr>
<td>Duration</td>
<td>&quot;:day:hr:m&quot;</td>
<td>&quot;52:03:01&quot;, which reads fifty-two days, three hours, and one minute.</td>
</tr>
<tr>
<td></td>
<td>&quot;:day:hr:m:s&quot;</td>
<td>&quot;52:03:01:30&quot;, which reads fifty-two days, three hours, one minute, and thirty seconds.</td>
</tr>
<tr>
<td></td>
<td>&quot;hr:m&quot;</td>
<td>&quot;17:37&quot;, which reads seventeen hours and thirty-seven minutes.</td>
</tr>
<tr>
<td></td>
<td>&quot;hr:m:s&quot;</td>
<td>&quot;17:37:04&quot;, which reads seventeen hours, thirty-seven minutes, and 4 seconds.</td>
</tr>
<tr>
<td></td>
<td>&quot;min:s&quot;</td>
<td>&quot;37:04&quot;, which reads thirty-seven minutes and 4 seconds.</td>
</tr>
<tr>
<td>Long date</td>
<td>&quot;Date Long&quot;</td>
<td>(Display only) &quot;Saturday, January 2, 1999&quot;</td>
</tr>
<tr>
<td>Abbreviated date</td>
<td>&quot;Date Abbrev&quot;</td>
<td>(Display only) &quot;Jan 2, 1999&quot;</td>
</tr>
</tbody>
</table>
Note that the last five date/time formats are not available for date input.

Table 2.9 Date/time operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Minutes(n)</td>
<td>These operators return the number of seconds per n minutes, hours, days,</td>
</tr>
<tr>
<td></td>
<td>weeks, or years. Divide by these to re-express an interval in seconds as an</td>
</tr>
<tr>
<td></td>
<td>interval in other units.</td>
</tr>
<tr>
<td>In Hours(n)</td>
<td></td>
</tr>
<tr>
<td>In Days(n)</td>
<td></td>
</tr>
<tr>
<td>In Weeks(n)</td>
<td></td>
</tr>
<tr>
<td>In Years(n)</td>
<td></td>
</tr>
<tr>
<td>Date DMY(day, month, year)</td>
<td>Returns the specified date, expressed as the number of seconds since</td>
</tr>
<tr>
<td></td>
<td>midnight, 1 January 1904. For example, the second Leap Day of the</td>
</tr>
<tr>
<td></td>
<td>second millennium is DateDMY(29, 2, 2004), which returns</td>
</tr>
<tr>
<td></td>
<td>3160857600.</td>
</tr>
<tr>
<td>Date MDY(month, day, year)</td>
<td>Returns the specified date, expressed as the number of seconds since</td>
</tr>
<tr>
<td></td>
<td>midnight, 1 January 1904. For example, the second Leap Day of the</td>
</tr>
<tr>
<td></td>
<td>second millennium is DateMDY(2, 29, 2004), which returns</td>
</tr>
<tr>
<td></td>
<td>3160857600.</td>
</tr>
<tr>
<td>Today()</td>
<td>Returns the current date and time expressed as the number of seconds</td>
</tr>
<tr>
<td></td>
<td>since midnight, 1 January 1904. No arguments are accepted, but the</td>
</tr>
<tr>
<td></td>
<td>parentheses are still needed.</td>
</tr>
<tr>
<td>Second(datetime)</td>
<td>Returns an integer representation for the second part of the date-time</td>
</tr>
<tr>
<td></td>
<td>value supplied.</td>
</tr>
<tr>
<td>Minute(datetime)</td>
<td>Returns an integer representation for the minute part of the date-time</td>
</tr>
<tr>
<td></td>
<td>value supplied.</td>
</tr>
</tbody>
</table>
Table 2.9 Date/time operators (Continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour(datetime)</td>
<td>Returns an integer representation for the hour part of the date-time value supplied.</td>
</tr>
<tr>
<td>Day(date)</td>
<td>Returns an integer representation for the day of the month of the date supplied.</td>
</tr>
<tr>
<td>Month(date)</td>
<td>Returns an integer representation for the month of the date supplied.</td>
</tr>
<tr>
<td>Year(date)</td>
<td>Returns an integer representation for the year of the date supplied.</td>
</tr>
<tr>
<td>Day Of Week(date)</td>
<td>Returns an integer representation for the day of the week of the date supplied. Weeks are Sunday–Saturday.</td>
</tr>
<tr>
<td>Day Of Year(date)</td>
<td>Returns an integer representation for the day of the year of the date supplied.</td>
</tr>
<tr>
<td>Week Of Year(date)</td>
<td>Returns an integer representation for the week of the year of the date supplied. Weeks are Sunday–Saturday. Note that the first Sunday of the year begins week 2.</td>
</tr>
<tr>
<td>Time Of Day(date)</td>
<td>Returns an integer representation for the time of day of the date/time supplied.</td>
</tr>
<tr>
<td>Short Date(date)</td>
<td>Returns a string representation for the date supplied, in the format mm/dd/yyyy. For example, &quot;02/29/2004&quot;.</td>
</tr>
<tr>
<td>Long Date(date)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;Sunday, February 29, 2004&quot;.</td>
</tr>
<tr>
<td>Abbrev Date(date)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;Feb 29, 2004&quot;.</td>
</tr>
<tr>
<td>MDYHMS(date)</td>
<td>Returns a string representation for the date supplied, formatted like &quot;2/29/2004 00:02:20 AM&quot;.</td>
</tr>
<tr>
<td>Format(date, &quot;format&quot;)</td>
<td>Returns the value in the format specified in the second argument. Most typically used for formatting date/time values from a number of seconds to a formatted date. Format choices are those shown in the Column Info dialog box; also see Table 2.8 “Date/time formats supported in JMP,” p. 49.</td>
</tr>
<tr>
<td>In Format(string, &quot;format&quot;)</td>
<td>Parses a string of a given format and returns date/time value expressed as if surrounded by As Date(), returning the date in ddMonyyyy format.</td>
</tr>
<tr>
<td>As Date(expression)</td>
<td>Formats a number or expression so that it shows as a date when streamed out to a text window.</td>
</tr>
</tbody>
</table>
Currency

JMP supports the use of currency symbols. To set a specific currency, use the `Format()` function. The syntax is:

```
Format(x, "format", <"currency code">, <decimal>)
```

where `x` is a column, or a number. (For datetime formats, see “Datetime Operators,” p. 46).

To designate a number or a column as a currency, the format argument must be set to “Currency”. Then add the ISO 4217 code for a specific currency as a quoted string. This third argument is invalid unless the format is set to “Currency”. For example:

```
Format(12345.6, "Currency", "GBP", 3)
```

"£12,345.600"

Inquiry Functions

How can you tell what type of data element you have? JSL has inquiry operators to test whether something is or is not a specific type of data element.

JSL also has an inquiry operator to test whether a global variable, a data table, or a data column has a value assigned or not: `Is Empty`. You can get errors if you try to refer to something that has not been created or assigned a value yet. Programmers call this an “uninitialized variable.” To check whether a global variable, data table, or data column has a value yet, use `Is Empty` with the name of the item.

```
Is Empty(myGlobal);
Is Empty(dt);
Is Empty(col);
```

In addition, JSL provides a general `Type` function that returns a string naming the type of the resulting value. The possible type names are `Unknown`, `List`, `DisplayBox`, `Picture`, `Column`, `TableVar`, `Table`, `Empty`, `Pattern`, `Date`, `Integer`, `Number`, `String`, `Name`, `Matrix`, `RowState`, `Expression`, `Associative Array`, or `Blob`.

For example:

```
Show(Type(1), Type("hi"), Type({"a",2}));
```

results in

```
Type(1):"Integer"
Type("hi"):"String"
Type({"a",2}):"List"
```

`Is Scriptable()` is especially useful when parts of a script depend on previous-generated results. Test that the previous results are scriptable and only continue with a script after `IsScriptable()` returns `true`. 
54

JSL Building Blocks
Inquiry Functions

Chapter 2

Table 2.10 Inquiry functions to identify object types
Syntax

Explanation

Is Number(x)

Returns 1 if the evaluated argument is a number or missing numeric value, or 0
otherwise.

Is String(x)

Returns 1 if the evaluated argument is a string, or 0 otherwise.

Is Matrix(x)

Returns 1 if the evaluated argument is a matrix, or 0 otherwise.

Is Expr(x)

Returns 1 if the evaluated argument is an expression, or 0 otherwise.

Is Name(x)

Returns 1 if the evaluated argument is a name-only expression, or 0 otherwise.

Is List(x)

Returns 1 if the evaluated argument is a list, or 0 otherwise.

Is Associative Array(x)

Returns 1 if the evaluated argument is an associative array, or 0 otherwise.

Is Scriptable(x)

Returns 1 if the evaluated argument is a scriptable object, or 0 otherwise.

Is Empty(global)

Returns 1 if the global variable, data table, or data column does not have a value
(is uninitialized), or 0 otherwise.

Is Empty(dt)
Is Empty(col)
Type(x)

Returns a string naming the type of X.

Version and Host Information
Host Is is an inquiry operator. Host Is returns a Boolean result, either 1 if true or 0 if false. This is useful
for identifying the current operating system in scripts intended for use on Windows and Macintosh
platforms. Although now not necessary, Host Is was frequently used to open data tables. For example:
dt=if(
host is(Windows),
open("..\sample data\Big Class.JMP"),
host is(Macintosh),
open("::sample data:big class"));

Another use is to specify different text sizes in reports for different operating systems. If you commonly
write your scripts on Windows and share them with people who use them on Macintosh, the results can
look different from what you intended. For example, this line sets the text to a larger size on Macintosh and
a smaller size on Windows:
txtsize = if(host is(Mac),12,10);

The JMP Version() function returns the JMP version as a string. The function puts a leading blank on
the version if it is less than 10 to make it easier to compare versions 7, 8, or 9 with version 10 in the future.
This function is not available in releases before 6.0.
JMP Version();
" 9.0"


Functions that communicate with users

Show, Print, and Write put messages in the log window. Speak, Caption, Beep, and StatusMsg provide ways to say something to a viewer. Mail can send an e-mail alert to a process operator.

JMP’s scripting language has methods for constructing dialog boxes to ask for data column choices and other types of information. See “Modal Windows,” p. 218 in the “Display Trees” chapter.

Writing to the log

Show

Show displays the items that you specify in the log. Notice that when you show variables, the resulting message is the variable name, a colon :, and its current value.

```julia
X = 1; A = "Hello, World";
show(X, A, "foo");
```

```
x = 1
a = "Hello, World"
"foo"
```

Print

Print sends the message that you specify to the log. Print is the same as Show except that it prints only the value of each variable without the variable name and colon.

```julia
X = 1; A = "Hello, World";
print(X, A, "foo");
```

```
1
"Hello, World"
"foo"
```

Write

Write sends the message that you specify to the log. Write is the same as Print except that it suppresses the quotation marks around the text string, and it does not start on a new line unless you include a return character yourself with the \!N escape sequence.

```julia
write("Here is a message.");
```

```
Here is a message.
```

```julia
myText = "Here is a message.";
write(myText || " Do not forget to buy milk."); // use || to concatenate
write("\!NAnd bread."); // use \!N for return
```

```
Here is a message. Do not forget to buy milk.
And bread.
```

The sequence \!N inserts the line breaking characters that are appropriate for the host environment. For an explanation of the three line breaking escape sequences, see “Quoted Strings,” p. 28 in the “JSL Building Blocks” chapter.
Send information to the user

**Beep**

Beep causes the user's computer to make an alert sound.

**Speak**

Speak reads text aloud. On Macintosh, Speak has one Boolean option, Wait, to specify whether JMP should wait for speaking to finish before proceeding with the next step. The default is not to wait, and you need to issue Wait(1) each time. For example, here is a script certain to drive anybody crazy. With Wait(1), you probably want to interrupt execution before too long. If you change it to Wait(0), the iterations proceed faster than the speaking possibly can and the result sounds strange. On Windows, you can use a Wait(n) command to accomplish the same effect.

```julia
for(i=99,i>0,i--,
  speak(wait(1),char(i)||" bottles of beer on the wall, "
    ||char(i)||" bottles of beer; "
    ||"if one of those bottles should happen to fall, "
    ||char(i-1)||" bottles of beer on the wall. ")
```

A more practical example has JMP announce the time every sixty seconds:

```julia
// Time Announcer
script = expr(
  tod = mod(today(),indays(1));
  hr  = floor(tod/inHours(1));
  min = floor(mod(tod,inHours(1))/60);
  timeText = "time, " || char(hr) || " " || char(min);
  text = Long Date(today()) || ", " || timeText;
  speak(text);
  show(text);
  schedule(60,script);    // seconds before next script
); script;
```

You might use a similar technique to have JMP alert an operator that a process has gone out of control.

**Caption**

Caption brings up a small window with a message to the viewer. Captions are a way to annotate demonstrations without adding superfluous objects to results windows. The first argument is an optional \{x,y\} screen location given in pixels from the upper left; the second argument is the text for the window. If the location argument is omitted, windows appear in the upper left corner.

The Spoken option causes captions to be read aloud by the operating system's speech system (if available). Spoken takes a Boolean argument, and the current setting (on or off) remains in effect until switched by another Caption statement that includes a Spoken setting.

You can include pauses in the playback by including the named argument Delayed and a time in seconds. Such a setting causes that caption and all subsequent caption windows to be delayed by that number of
seconds, until a different Delay setting is issued in a Caption statement. Use Delay(0) to stop delaying altogether.

This script turns speaking on and leaves it on until the last caption.

```javascript
caption({10,30},"A tour of the JMP Analyses", spoken(1), Delayed(5));
caption("Open a data Table");
bigClass=open("$SAMPLE_DATA/Big Class.JMP");
caption("A JMP Data Table consists of rows and columns of data");
caption("The rows are numbered and the columns are named");
caption({250,50},"The data itself is in the grid on the right");
caption({5,30},spoken(0),"A panel along the left side shows columns and other attributes");
```

Each new Caption hides the previous one. In other words, there is only one caption window available at a time. To close a caption without displaying a new one, use the named argument Remove.

```javascript
caption(remove);
```

### StatusMsg

This command sends a message to the status bar.

```javascript
StatusMsg("string")
```

### Mail

(Windows only) Mail sends an e-mail message to alert a user about a condition in JMP. For example, a process control manager might include a test alert script in a control chart to trigger an e-mail warning to her pager:

```javascript
mail("JaneDoe@company.com", "out of control", "Process 12A out of control at "||Format(today(), "d/m/y h:m:s"));
```

Mail can also send an attachment with the e-mail. An optional fourth argument specifies the attachment. The attachment is transferred in binary format after its existence on the disk is verified. For example, to attach the Big Class.jmp data table, submit

```javascript
mail("JohnDoe@company.com", "Interesting Data Set", "Have a look at this class data.", "C:\myJMPData\Big Class.jmp");
```
functions that communicate with users
Sometimes you might need to program repetitive operations, or perhaps implement some statistical test that are not part of JMP. Then you need to learn some of the functions that control flow.

The following functions take on the roles that in other programming languages have statements with special syntactic structure. They are just functions in JSL, like any other functions, with arguments separated by commas.

This chapter covers basic programming functions and variables. The chapter “Advanced Concepts,” p. 385, covers more advanced techniques, such as throwing and catching exceptions and working with expressions.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Example</td>
<td>61</td>
</tr>
<tr>
<td>Iterating</td>
<td>61</td>
</tr>
<tr>
<td>For</td>
<td>61</td>
</tr>
<tr>
<td>While</td>
<td>61</td>
</tr>
<tr>
<td>Summation</td>
<td>63</td>
</tr>
<tr>
<td>Product</td>
<td>63</td>
</tr>
<tr>
<td>Break and Continue</td>
<td>63</td>
</tr>
<tr>
<td>Conditional functions</td>
<td>64</td>
</tr>
<tr>
<td>If</td>
<td>64</td>
</tr>
<tr>
<td>Match</td>
<td>65</td>
</tr>
<tr>
<td>Choose</td>
<td>66</td>
</tr>
<tr>
<td>Interpolate</td>
<td>66</td>
</tr>
<tr>
<td>Step</td>
<td>67</td>
</tr>
<tr>
<td>Controlling script execution</td>
<td>68</td>
</tr>
<tr>
<td>Wait</td>
<td>68</td>
</tr>
<tr>
<td>Schedule</td>
<td>68</td>
</tr>
<tr>
<td>JSL Data Structures</td>
<td>68</td>
</tr>
<tr>
<td>Lists</td>
<td>68</td>
</tr>
<tr>
<td>Matrices</td>
<td>72</td>
</tr>
<tr>
<td>Associate Arrays</td>
<td>72</td>
</tr>
<tr>
<td>Character Functions</td>
<td>77</td>
</tr>
<tr>
<td>Concat</td>
<td>77</td>
</tr>
<tr>
<td>Munger</td>
<td>78</td>
</tr>
<tr>
<td>Repeat</td>
<td>79</td>
</tr>
<tr>
<td>Sequence</td>
<td>79</td>
</tr>
<tr>
<td>Comparison and Logical Operators</td>
<td>81</td>
</tr>
<tr>
<td>Missing Values</td>
<td>82</td>
</tr>
<tr>
<td>Missing Character Values</td>
<td>83</td>
</tr>
<tr>
<td>Short-Circuiting Behavior</td>
<td>83</td>
</tr>
</tbody>
</table>
Programming Example

Here is an example script using JSL to implement a statistical method, a type of runs test. A flow-control function, Summation, is also used to iterate computations over many rows at a time.

```jsl
Open("$SAMPLE_DATA/Big Class.jmp");
Summarize(mean=mean(Height),n=count(Height));
nr = nrow();
n1 = summation(i=1, nr, Height[i]>mean);
n2 = n-n1;
r = summation(i=2, nr, (Height[i]>mean)!=(Height[i-1]>mean));
muR = 2*n1*n2/(n1+n2);
sigmaR = sqrt(2*n1*n2*(2*n1*n2-n1-n2)/(n^2*(n-1)));
zTest = (r-muR)/sigmaR;
pValue = (1-normal distribution(abs(zTest)))*2;
show(r, muR, sigmaR, zTest, p Value);
```

The first statement opens a JMP data table. The `Summarize` function obtains the mean and count for the column height. `NRow` gets the number of rows, which is the same as the count `n`, since there are no missing values. Then it finds `n1`, the number of heights greater than the mean; `n2`, the remaining number; and `r`, the number of adjacent heights that are not both above or both below the mean. The `muR` and `sigmaR` are the mean and standard deviation of the test statistic under the hypothesis that there is no run pattern, `zTest` is the normal test statistic, and `p Value` is the significance level of the test. If there is a pattern across time, then neighboring values would be more likely to be alike in whether they were above or below the mean.

Iterating

JSL provides `For`, `While`, `Summation`, and `Product` to iterate actions according to the conditions that you specify.

To iterate actions over rows of a data table, use `For Each Row`, which is discussed in the “Data Tables” chapter.

For

The `For` function first evaluates `init` expression, but just one time. Then it tests the `while` expression, breaking if not true. As long as the `while` expression is true, it keeps executing the `body` and `iter` expressions.

```jsl
For(init, while, iter, body);
```

`For` loops just like it does in the C (and C++) programming language, although the punctuation is different.

Confusion alert! If you know C, watch out for the difference between JSL and C in the use of semicolons and commas. In JSL `For` is a function where commas separate arguments and semicolons join them; in C `for` is a special clause where semicolons separate arguments and commas join them.
For example, you can do a Distribution on the first three columns of the current data table:

```plaintext
for(i=0, i<3, i++, Distribution(column(i+1))); 
```

This loop does the following:
1. Sets \( i \) to 0.
2. Evaluates \( 0<3 \) as true.
3. Launches the Distribution platform for the first column.
4. Increments \( i \) to 1.
5. Evaluates \( 1<3 \) as true.
6. Launches the Distribution platform for the second column.
7. Increments \( i \) to 2.
8. Evaluates \( 2<3 \) as true.
9. Launches the Distribution platform for the third column.
10. Increments \( i \) to 3.
11. Evaluates \( 3<3 \) as false.
12. Breaks out of the loop.

Of course, it is not hard to write scripts that get into infinite loops. When this happens, you can use the keyboard cancel sequence to get out of the loop (press ESC on Windows, or press COMMAND-PERIOD on Macintosh).

The following expression sums the numbers from 0 to 20:

```plaintext
s=0; For(i=0, i<=20, i++, s=s+i);
```

The `For` loop works like this. It expects four arguments separated with commas ( , ). The first three arguments are rules for how many times to repeat the loop, and the fourth is what to do repeatedly. This is where the action takes place.

1. First it sets an initial condition or state. In this case, it assigns the value 1 to a variable \( i \). The first argument of a for loop is always a starting condition like this one, but the variable and the value could be anything.

2. Next it tests whether the second argument, \( i<=20 \), is true. If it is true, then it does the action script. If false, it ends the for loop and continues with the rest of the script (if there is more waiting after the last parenthesis of the for loop). The second argument is always a condition, like a little `If` test. Usually the condition involves the same variable initialized in the first argument, but that is not always the case. Be careful not to create an infinite loop. (If you make a mistake, you can press ESC on Windows or COMMAND-PERIOD on Macintosh to stop the looping.)

3. The third argument is what to do after completing the actions; typically it increments \( i \) (adds one to its current value). The actions are everything that comes next inside the for's parentheses. You come back to this in step 5.

4. The fourth argument does the work of the for loop. In this case, it replaces \( s \) with \( s+i \). Only one command statement is expected in the fourth argument. If you want to do several things on each step,
glue the statements together with semicolons (;). Semicolons let numerous things stand in place of one thing.

5. Now the third argument kicks in. After the action of the fourth argument is done, the counter variable $i$ is incremented, and the execution starts over at the second argument, testing again whether $i \leq 20$ is still true.

**While**

A related function is `While`, which repeatedly tests the condition and executes its body script as long as the condition is true.

```
While(condition, body);
```

For example, here are two different programs to find the least power of 2 that is greater than or equal to $x$:

```
x=287;
y=1; while(y<x, y*=2); show(y);
k=0; while(2^k<x, k++); show(2^k);
```

**Summation**

The `Summation` function adds up the body results over all $i$ values. For example:

```
s = Summation(i=0,10,i); // 0+1+2+....+9+10
```

This is the same idea as $\sum$ in the calculator except that with `Summation`, you do not need to restrict yourself to creating formula variables.

```
Summation(i=1, NRow(), x^2);
```

This is the JSL equivalent in the formula calculator:

```
\sum_{i=1}^{\text{NRow}} x^2
```

**Product**

The `Product` function works just like `Summation`, except that it multiplies the body results rather than adding them.

```
p = Product(i=1,5,i); // 1*2*3*4*5
```

This is the JSL equivalent of this in the formula calculator:

```
\prod_{i=1}^{5} i
```

**Break and Continue**

Two functions give you further control over looping: `Break()` and `Continue()`. These two functions work with `For` and `While` loops, and also with `For Each Row`. 
Programming Functions

Chapter 3

Conditional functions

**Break**

Break immediately stops the loop and proceeds to the next command after the loop. This is usually used inside a conditional statement. For example:

```plaintext
For( i = 1, i <= 10, i++,
    If( i == 5, Break() );
    Print( "i=" || Char( i ) );
);
Show( "Finished!" );
```

Without the If statement that contains Break, the loop would print out the values of \( i \) from 1 to 10, and then “Finished”. However, as soon as \( i \) reaches 5, the loop is immediately halted (before the print statement, but after \( i \) is 5) and the following statement is processed (Show(“Finished”)).

If the If and Break statements were at the end of the loop, the final value of \( i \) printed to the log would be 5 rather than 4.

**Continue**

Continue is a gentler form of Break. It immediately stops the current iteration of the loop, but sends execution back to the beginning of the loop rather than stopping the loop completely. Continue is also usually used inside a conditional statement. For example:

```plaintext
For( i = 1, i <= 10, i++,
    If( i < 5, Continue() );
    Print( "i=" || Char( i ) );
);
```

This loop iterates through \( i \), but until \( i \) is greater than 5, the print statement is not executed. There is no point to placing a Continue at the end of a loop. The entire loop body has already been executed, so there is nothing to skip. The loop continues anyway.

---

**Conditional functions**

JSL provides five functions to execute script conditionally: If, Match, Choose, Interpolate, and Step.

**If**

The If function returns the result statement(s) when condition evaluates as true (nonzero and nonmissing). Otherwise it skips ahead and returns the next result when that condition evaluates as true. The final result is the result if none of the preceding conditions are true. It is the “else” result.

```plaintext
If( \( \text{condition1}, \text{result1}, \text{condition2}, \text{result2}, \ldots, \text{resultLast} \) )
```

If at least one condition evaluates to missing, the function keeps on evaluating and only returns missing (rather than evaluating the else clause) if none of the clauses evaluated to true and at least one conditional returned missing. The way to get a clause to always return despite missing conditionals is to add a conditional at the end that is always true. For example,
If (age<12,"Young", a>12,"Old", 1,"Ageless");

For example, the following code sets X to Y if Y is less than 20. Otherwise, it sets X to 20.

\[ X = \text{If}(Y<20, Y, 20); \]

You can use this to recode ranges, for example:

\[ \text{row}()=1; \ \text{Age} \ \text{Group} = \text{If} (\text{Age}<12,"Child",\text{Age}<18,"Teen","Adult"); \]

You can put actions and assignments in the result clauses, and you do not have to return a result. For example the following two are equivalent:

\[ X = \text{If}(y<20,y,20); \]
\[ \text{If}(y<20,x=y,x=20); \]

Similar conditionals are \texttt{Choose} and \texttt{Match}.

Be careful to use == for equality tests, not =. An If with an argument like name=value would assign the value rather than test it.

\section*{Match}

You can use the \texttt{Match} function to make a sequence of equality comparisons without needing to rewrite the value to be compared. The result of the first operand is compared to each of the succeeding even operands, except the last. If they are equal, the value of the operand immediately after is returned. If nothing matches, then the last operand is returned. For example:

\[ \text{SLabel} = \text{Match}(\text{SCode}, 1,"Male", 2,"Female", "unknown"); \]

This would be equivalent to the following If expression:

\[ \text{SLabel} = \text{If}((\text{SCode}=1,"Male", \text{SCode}=2,"Female", "unknown"); \]

With more groups, the value of \texttt{Match} becomes more apparent:

\[ \text{dt}=\text{open}("\$SAMPLE\_DATA/Big Class.JMP"); \]
\[ \text{for each row(ageword=} \text{match(age, 12, "twelve", 13, "thirteen", 14, "fourteen", 15, "fifteen", 16, "sixteen", "seventeen");}\]
\[ \text{show(age, ageword));} \]

If a has integer values 1, 2, 3, ..., you can save even more typing by using \texttt{Choose}. 

\[ \text{if(a<=12,"Young", a>12,"Old", 1,"Ageless");} \]
Choose

The Choose function can be used to shorten certain recoding even more, provided the arguments are to be tested against integers 1, 2, 3, and so forth. If the first argument evaluates to 1, Choose returns the first replacement value; if it is 2, the second, and so on. If the first argument evaluates to an integer out of range (for example, 7 when only 5 replacement values are listed), Choose returns the last replacement value. For example:

```
SLabel = Choose(SCode, "Male", "Female", "Unknown");
```

Thus, if group is a column with values 1, 2, and 3, the following all accomplish the same task, replacing 1s with “low,” 2s with “medium,” and 3s with “high.”

```
if(group==1, "low", group==2, "medium", group==3, "high");
g2=match(group, 1, "low", 2, "medium", 3, "high");
g3=choose(group, "low", "medium", "high");
```

Interpolate

Interpolate linearly interpolates the \(y\)-value corresponding to a given \(x\)-value between two points \((x_1, y_1)\) and \((x_2, y_2)\).

The points can be specified as a list of points

```
Interpolate(x, x1, y1, x2, y2, ...)
```

or as matrices containing the \(x\)- and \(y\)-values

```
Interpolate(x, xmat, ymat)
```

Suppose we wanted to approximate the exponential function by linear interpolation between integers, 0 to 5. The following would create the two vectors of values:

```
xgrid = (0::5);
yvalues = exp(xgrid);
show(xgrid, yvalues);
```

This produces the output

```
xgrid: [0 1 2 3 4 5]
yvalues: [1 2.718281828459045 7.38905609893065 20.08553692318767 54.59815003314424 148.41315910257666]
```
Figure 3.1 shows a plot of what the interpolation function would look like (dashed line), compared with what the continuous version (solid line), which is what is being approximated.

**Figure 3.1 Example of Interpolate**

The points must be arranged in ascending order. For example, `Interpolate(2,1,1,3,3)` returns 2. However, `Interpolate(2,3,3,1,1)` returns a missing value (..).

**Step**

`Step` is like `Interpolate` except that it finds the corresponding `y` for a given `x` from a step-function fit rather than a linear fit between points. That is, it returns the `y`-value corresponding to the greatest `x`-value that is less than or equal to the `x` specified as the first argument.

```
Step(x, x1, y1, x2, y2, ...)
```

For example:

```
for(i=1;i<7;i++,print(step(i, 1,10, 3,30, 4.5,45, 7,70)));
```

```
10
10
30
30
45
45
```

As with `Interpolate`, the points must be in ascending order.
Controlling script execution

When you run a script, the Run Script command in the Edit menu changes to Stop Script. While a script is executing, you can select this item (or type ESC on Windows or COMMAND-PERIOD on Macintosh) to stop it. Many scripts execute more quickly than you can stop them.

Wait

Stopping a script manually stops the entire script completely. In contrast, the Wait command enables you to pause script execution for a time and then continue running the script. This approach is sometimes necessary for scripts intended for interactive use or demonstrations.

Wait enables you to slow down scripting playback by pausing the specified number of seconds before continuing with the script. For example, this is a 15-second timer:

```
Wait(15);
```

Wait also yields to the system, so that system tasks can be run while waiting. This includes updating the contents of the screen. Issuing a wait time of 0 (Wait(0)) lets JMP to do other pending events, but not wait any longer than needed to do these events. Waiting zero seconds is useful for allowing results to be displayed right away; otherwise JMP waits for the script to end before yielding to the system to update windows.

Schedule

Schedule runs commands or entire scripts at the times that you specify. See the “Extending JMP” chapter.

JSL Data Structures

Besides numbers and strings, JSL provides several other data types variables can hold:

- Lists
- Matrices
- Associative Arrays

Lists

Lists are containers to store numerous items, potentially items of different type: numbers, text strings, expressions, matrices, and even other lists. A list is just the function List with arguments. The shorthand for List is to use curly braces. Thus the following two statements are equivalent:

```
A = List(1,2,B,sqrt(3));
A = {1,2,B,sqrt(3)};
```

When you define or evaluate a list, it just returns a copy of itself. It does not evaluate items inside the list. For example, this step reveals that the list contains the variable B, not its current value:
Use `Eval List` if you want a list with all its items evaluated:

```javascript
C = Eval List(A);
{1, 2, 7, 1.732050807568877}
```

Lists can also hold assignments (`{a=1, b=2}`), or function calls (`{a(1), b(2)}`). Remember, statements such as these just return lists. If you want to evaluate the contents, use `Eval List`.

```javascript
assignList = {a=1, b=3, c=5};
{a=1, b=3, c=5}
show(assignList);

assignList = {a = 1, b = 3, c = 5}
evalList(assignList);
{1, 3, 5}
show(a, b, c);

assignList = {a = 1, b = 3, c = 5}
evalList(assignList);

assignList = {a = 1, b = 3, c = 5}
h = function({x}, x + 2); i = function({x}, x + 3); k = function({x}, x + 4);
fnList = {h(1), i(3), k(5)};
evalList(fnList);
{3, 6, 9}
show(h, i, k);

fnList = {h(1), i(3), k(5)}
show(fnList);
```

### Assignment

List assignment is powerful.

```javascript
{a, b, c} = {1, 2, 3}; // assigns 1 to a, 2 to b, 3 to c
{a, b, c} += 10; // adds 10 to each variable a, b, and c.
{{a}, {b, c}}++; // increments a, b, and c by 1
{a, b, c} --; // decrements a, b, and c
myList = {1, log(2), e()^pi(), height[40]}; // stores the expressions in a list
```

Note that any structure of a list is supported, as long as the arguments are conformable and ultimately numeric. One-element lists are treated like scalars for conformability. For example,

```javascript
a = {{{{1, 2}, {3, 4}}, {5, {6, 7}}}, 8}, {9, 10}};
b = {{{{10, 20}, {30, 40}}, {50, {60, 70}}}, 80}, {90, 100}};
```
c = a + b; show(c);
d = a + 100; show(d);
e = sqrt(a); show(e);
results in the output

\[
\begin{align*}
c &= \{\{\{11, 22\}, \{33, 44\}\}, 55, \{66, 77\}\}, 88, \{99, 110\} \\
d &= \{\{\{101, 102\}, \{103, 104\}\}, \{105, \{106, 107\}\}\}, 108, \{109, 110\} \\
e &= \{\{\{1, 1.414213562373095\}, \{1.732050807568877, 2\}\}, \{2.23606797749979, \\
& \quad \{2.449489742783178, 2.645751311064591\}\}, 2.82842712474619\}, \{3, \\
& \quad 3.16227766016838\}\}
\end{align*}
\]

**How many items?**

To determine the number of items in a list, use the `NItems` function:

\[
N = N \text{Items}(A);
\]

**Subscripts**

Subscripts for lists extract the specified item(s) from a list. Subscripts can in turn be lists. Note that JSL starts counting from 1, so the first element in a list is [1], not [0] as in some other languages.

\[
a = \{"bob", 4, [1,2,3], \{x,y,z\}\};
\]

\[
\begin{align*}
&\text{show}(a[1]); \quad /\text{return } \text{"bob}\text{"} \\
&\text{show}(a[[1,3]]); \quad /\text{return } \{\text{"bob"}, [1,2,3]\}
\end{align*}
\]

\[
a[2] = 5; \quad \text{// changes the 4 to 5}
\]

When you have assignment lists or function lists, you can use a quoted name for the subscript:

\[
X = \{a(1), b(3), c(5)\};
\]

\[
XX = \{a=1, b=3, c=5\};
\]

\[
\begin{align*}
&\text{show}(X[\"a\"], XX[\"a\"]); \quad \quad /\text{return } 1 \\
&X[\"a\"] = 1 \\
&XX[\"a\"] = 1
\end{align*}
\]

You need to put the name in quotation marks, because otherwise JMP would try to evaluate it and use its value, like this:

\[
a = 2; \text{show}(X[a], XX[a]);
\]

\[
\begin{align*}
&X[a] = b(3) \\
&XX[a] = b=3
\end{align*}
\]

Multiple left-side subscripts (for example, \(a[i][j] = \text{value}\) where \(a\) contains a list of things that are subscriptable) are allowed in certain cases.

1. Each level except the outermost must be a list. So, in the above example, \(a\) must be a list but \(a[i]\) can be anything subscriptable.

2. Each subscript except the outermost must be a number. So, in the above example, \(i\) must be a number, but \(j\) could be a matrix or list of indices.

The subscripting can be done to any level of nesting. For example,

\[
a[i][j][k][l][m][n] = 99;
\]
Locating items in a list

To find values in a list, use the `Loc` function.

\[
\text{Loc(list, value)}
\]

This returns a matrix of indices to the values in the \text{list} that are equal to \text{value}, where \text{value} and \text{list} can be strings or numbers. If \text{value} is not found, the resulting matrix has zero rows and zero columns. Therefore, \text{NRow(Loc(list, value))}>0 is equivalent to \text{Contains(list, value)}>0.

\textbf{Note:} Complete details about matrix manipulation are found in the “Matrices” chapter, including the description of the equivalent \text{Loc} command for matrices (which requires only one argument).

For example, given these lists:

\begin{align*}
\text{nameList} &= \{“Katie”, “Louise”, “Jane”, “Jane”\}; \\
\text{numList} &= \{2, 4, 6, 8, 8\}
\end{align*}

\text{Loc(nameList, “Katie“)} returns \[1\]

\text{Loc(nameList, “Erin“)} returns [], an empty matrix, since there are no matches.

\text{Loc(nameList, “Jane“)} returns \[3, 4\]

\text{Loc(nameList, 1)} returns [], Note that the type mismatch is tolerated.

\text{Loc(numList, 1)} returns []

\text{Loc(numList, 6)} returns \[3\]

\text{Loc(numList, 8)} returns \[4, 5\]

Here is a summary of the list operators:

\textbf{Table 3.1 Operators for working with lists}

\begin{table}[h]
\begin{tabular}{|c|c|l|}
\hline
\textbf{Operator} & \textbf{Syntax} & \textbf{Explanation} \\
\hline
\text{List} & \text{List(a, b, c)} & \text{Constructs a list from a set of items. An item can be any expression, including other lists. Items must be separated by commas. Text strings should either be enclosed in double quotation marks (” “) or stored in a variable and called as that variable.} \\
\hline
\text{N Items} & \text{N Items(list)} & \text{Returns the number of elements in the list specified. Can be assigned to a variable.} \\
\hline
\text{Is List} & \text{Is List(arg)} & \text{Returns true (1) if arg is a classical list (in other words, one that would result from the construction by \text{List(items)} or \{items\}) and returns false (0) otherwise. An empty list is still a list, so \text{IsList(\{} \}) returns true. If \text{miss=}, then \text{IsList(miss)} returns false, not missing.} \\
\hline
\end{tabular}
\end{table}
Matrices

Matrices can be used to represent values in a data table as well as matrix algebra. See the “Matrices” chapter for details.

Associative Arrays

An associative array associates unique keys with values, which might not be unique. Though associative arrays are generally not ordered, JMP keys are returned in lexicographical order for the purpose of iteration and serialization.

To create an empty associative array, use one of the following:

```javascript
  cary = [=];
  cary = Associative Array();
```

The keys and values can be any JSL objects. Items can be added and changed with subscripting:

```javascript
  cary["state"] = "NC";
```

### Table 3.1 Operators for working with lists (Continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>list[]</td>
<td>Subscripts for lists extract the $i^{th}$ item from the list. Subscripts can in turn be lists or matrices.</td>
</tr>
<tr>
<td>]</td>
<td>x = list[]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>list[i] = value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a[b,c]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscript(a,b,c)</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>(list) = (list)</td>
<td>If the target of an assignment operator is a list and the value to be assigned is a list, then it assigns item by item. The ultimate values in the left list must be L-values (in other words, names capable of being assigned values).</td>
</tr>
<tr>
<td>+</td>
<td>(list) += (list)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>(list) += value</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>(list) -= value</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(list) *= (list)</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>(list) /= (list)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Eval List</td>
<td>Eval List(list)</td>
<td>Returns a list of the evaluated expressions inside list; see “Lists,” p. 68.</td>
</tr>
</tbody>
</table>
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};

Note that L-values in an associative array are just like any other L-value.

The default value determines the value of map[key] when key is not in the map. Initially, maps have no default value and map[key] causes an error much like list[-1] does. If a default value has been set, then that value is returned. Besides the Set Default Value message, a default value can be set in the literal constructor using =>value without a key. For example:

```javascript
counts = ["a"=>10, "b"=>3, =>0]; // default value of 0
counts["c"] += 1; // ["a" => 10, "b" => 3, "c" => 1, => 0]
```

### Associative Array Constructors

```javascript
map = [=>]; // empty associative array
map = [=>0]; // with default value
map = ["yes" => 0, "no" => 1]; // literal associative array
map = ["yes" => 0, "no" => 1, => 2]; // with default value

map = Associative Array(); // empty associative array
map = Associative Array(0); // with default value
map = Associative Array({"yes", 0},{"no", 1}); // list containing 2-item lists
map = Associative Array({"yes", 0},{"no", 1}, 2); // with default value
map = Associative Array("yes", "no"), {0, 1}); // two lists, keys and values
map = Associative Array("yes", "no"), {0, 1}, 2); // with default value
map = Associative Array(:name, :height); // two column refs
map = Associative Array(:name, :height, .); // with default value

set = Associative Array("yes", "no"); // one list creates a set with default value 0
set = Associative Array(:name); // one column ref creates a set with default value 0
```

### Working with associative arrays

To determine the number of keys that an associative array contains, use N Items. Using the cary associative array from earlier:

```javascript
N Items(cary);
4
```

There are several functions that can be used for associative arrays in addition to lists, strings, and expressions.
Adding and deleting keys and values

Insert, Insert Into, Remove, Remove From, Reverse, and Reverse Into can all be used to add key-value pairs or remove them from the associative array entirely. Insert and Remove return copies of the given associative array with the key-value pair added or removed. Insert Into and Remove From add or remove the key-value pairs directly from the given associative array.

Insert and Insert Into take three arguments: the associative array, a key, and a value.

```javascript
newcary = Insert(cary, "time zone", "Eastern");
show(cary, newcary);
```

```javascript
cary = Associative Array(
    {
        "high schools", {"Cary", "Green Hope", "Panther Creek"},
        {"population", 116244},
        {"state", "NC"},
        {"weather", "sunny"}
    }
)
newcary = Associative Array(
    {
        "high schools", {"Cary", "Green Hope", "Panther Creek"},
        {"population", 116244},
        {"state", "NC"},
        {"time zone", "Eastern"},
        {"weather", "sunny"}
    }
)
```

Insert Into(cary, "county", "Wake");
```javascript
show(cary);
```
```javascript
cary = Associative Array(
    {
        "county", "Wake"},
    {"high schools", {"Cary", "Green Hope", "Panther Creek"},
        {"population", 116244},
        {"state", "NC"},
        {"weather", "sunny"}
    }
)
```

Remove and Remove From take two arguments: the associative array, and a key.

```javascript
newcary = Remove(cary, "high schools");
show(cary, newcary);
```
```javascript
cary = Associative Array(
    {
        "county", "Wake"},
    {"high schools", {"Cary", "Green Hope", "Panther Creek"},
        {"population", 116244},
        {"state", "NC"},
        {"weather", "sunny"}
    }
)
newcary: [{"county" => "Wake", "population" => 116244, "state" => "NC", "weather" => "sunny"]
```
```javascript
Remove From(cary, "weather");
show(cary);
```
```javascript
cary = Associative Array(
    {
        "county", "Wake"},
    {"high schools", {"Cary", "Green Hope", "Panther Creek"},
        {"population", 116244},
```
Insert Into and Remove From also have message equivalents. These two statements are equivalent.

cary << Insert("county", "Wake");
    Insert Into(cary, "county", "Wake");

Likewise, these two statements are equivalent:

cary << Remove("weather");
    Remove From(cary, "weather");

Another message lets you set the default value:

cary << Set Default Value("Cary, NC");

Use <<Get Default Value to determine whether there is a default value set for an associative array, and if so, what its value is. If there is no default value set, Empty() is returned.

Obtaining information about an associative array's contents

Contains can be used to determine whether a certain key is contained inside an associative array:

    Contains(cary, "high schools");
    1
    Contains(cary, "lakes");
    0

Several messages provide information about the associative array's contents.

Use << Get Keys to obtain a list of all keys contained in an associative array.

cary << Get Keys;
    {"county", "high schools", "population", "state"}

Likewise, << Get Values returns a list of all values:

cary << Get Values;
    {"Wake", {"Cary", "Green Hope", "Panther Creek"}, 116244, "NC"}

If you want only the values for certain keys, you can specify them as arguments:

cary << Get Values({"state", "county"});
    {"NC", "Wake"}

Note that the keys must be given as a list.

Similar to << Get Values is << Get Value, which returns the value for a single key:

cary << Get Value("population");
    116244

Note that only one key can be specified, and it must not be in a list.

To obtain a list of all key-value pairs in an associative array, use << Get Contents:

cary << Get Contents;
    {"county", "Wake"},
    {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
    {"state", "NC"}
}
Iterating through an associative array

To iterate through an associative array, use <<First and <<Next. The following removes all key-value pairs from the associative array `cary`, leaving an empty associative array.

```julia
currentkey = cary <<First;
total = N Items(cary);
for (i = 1, i <= total, i++,
    nextkey = cary<<Next(currentkey);
    Remove From (cary, currentkey);
    currentkey = nextkey;
);
```

**Sets**

A set is an application of an associative array in which all the keys have a value of 1 and non-keys have an implicit value of 0.

To facilitate set usage, 1 is the default value for inserted keys if no value is provided:

```julia
map << insert("sample");
map["sample"];
1
```

If you construct an associative array from a list, then the default value is set to 0 for you. Otherwise, you need to set the default value to 0 explicitly.

**Graph Theory**

Associative arrays can be used for graph theory data structures, such as the following directed graph example:

```julia
g = Associative Array();
g[1] = Associative Array({1, 2, 4});
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});
```

The following depth-first search JSL function can be used to traverse this map, or any other directed graph expressed as a map:

```julia
dfs = Function( {ref, node, visited},
    Local( {chnode, tmp},
        Write( "Node: " || Char( node ) || ", " || Char( ref[node] << get contents ) || "\n" );
        visited[node] = 1;
        tmp = ref[node];
        chnode = tmp << first;
        While( !Is Missing( chnode ),
```

```julia
```
Chapter 3
Programming Functions
Character Functions

If( !visited[chnode],
    visited = Recurse( ref, chnode, visited )
);
chnode = tmp << next( chnode );
visited
);

The first parameter is a reference to the map, the second is the node that you want to use as the starting point, and the third is simply a vector that the function uses to keep track of nodes visited. To see how this function works try the following:

dfs( g, 2, J( N Items( g << get keys ), 1, 0 ) );

Character Functions

This section shows how to use some of the more complex character functions that are described in the “JSL Syntax Reference,” p. 439.

Concat

In the Concat function, expressions yielding names are treated like character strings, but globals that have the name values are evaluated. The following example demonstrates that if you have a stored name value, you need to either use Char before storing it in a global, or Name Expr on the global name.

n = { abc }; 
c=n[1] || "def";
show(c);
//result is "abcdef"

m=expr(mno);
c = m || "xyz";
show(c);
//result is an error message that mno is unresolved

m = expr(mno);
c = Name Expr(m) || "xyz";
show(c);
//result is "mnoxyz"

Concat Items() converts a list of string expressions into a single string, with each item separated by a delimiter. If unspecified, the delimiter is a blank. Its syntax is

resultString = Concat Items ({list of strings}, <"delimiter string">);
Programming Functions
Character Functions

For example,

\[ a = \{"ABC", "DEF", "HIJ"\}; \]
\[ result = \text{Concat Items}(a, \"/\"); \]

returns

"ABC/DEF/HIJ"

Alternatively,

\[ result = \text{Concat Items}(a); \]

returns

"ABC DEF HIJ"

Munger

Munger works many different ways, depending on what you specify for its arguments:

\[ \text{Munger}(\text{string}, \text{offset}, \text{find} | \text{length}, \langle \text{replace} \rangle); \]

<table>
<thead>
<tr>
<th>Find, length, and replace arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you specify a string as the find and specify no replace string, Munger returns the position (after offset) of the first occurrence find string.</td>
<td>Munger(&quot;the quick brown fox&quot;, 1, &quot;quick&quot;); 5</td>
</tr>
<tr>
<td>If you specify a positive integer as the length and specify no replace string, Munger returns the characters from offset to offset + length.</td>
<td>Munger(&quot;the quick brown fox&quot;,1,5); &quot;the q&quot;</td>
</tr>
<tr>
<td>If you specify a string as the find and specify a replace string, Munger replaces the first occurrence after offset of text with replace.</td>
<td>Munger(&quot;the quick brown fox&quot;, 1, &quot;quick&quot;, &quot;fast&quot;); &quot;the fast brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a positive integer as the length and specify a replace string, Munger replaces the characters from offset to offset + length with replace.</td>
<td>Munger(&quot;the quick brown fox&quot;, 1, 5, &quot;fast&quot;); &quot;fast\text{ick brown fox}&quot;</td>
</tr>
</tbody>
</table>
| If you specify a positive integer as the length, and offset + length exceeds the length of text, Munger either returns text from offset to the end or replaces that portion of text with the replace string, if it exists. | Munger("the quick brown fox",5,25); "\text{quick brown fox}"
Munger("the quick brown fox",5,25, "fast"); "the fast" |
| If you specify zero as the length and specify no replace string, Munger returns a blank string. | Munger("the quick brown fox", 1, 0); "" |
Chapter 3
Programming Functions

Character Functions

Repeat

The Repeat function makes copies of its first argument into a result. The second (and sometimes a third) argument is the number of repeats, where 1 means a single copy.

If the first argument evaluates to a character value or list, the result is that many copies.

\[
\text{repeat("abc",2)}
\]

"abcabc"

\[
\text{repeat({"A"},2)}
\]

{"A","A"}

\[
\text{repeat({1,2,3},2)}
\]

{1,2,3,1,2,3}

If the first argument evaluates to a number or matrix, the result is a matrix. The second argument is the number of row repeats, and a third argument can specify the number of column repeats. If only two arguments are specified, the number of column repeats is 1.

\[
\text{repeat([1 2, 3 4],2,3)}
\]

\[
\begin{bmatrix}
1 & 2 & 1 & 2 & 1 & 2, \\
3 & 4 & 3 & 4 & 3 & 4
\end{bmatrix}
\]

\[
\text{repeat(9,2,3)}
\]

\[
\begin{bmatrix}
9 & 9 & 9, \\
9 & 9 & 9
\end{bmatrix}
\]

The Repeat function is compatible with the function of the same name in the SAS/IML language, but is incompatible with the SAS character DATA step function, which repeats one more time than this function.

Sequence

Sequence() corresponds to the Sequence function in the Formula Editor and is used to fill the cells in a data table column. It takes four arguments and the last two are optional:

\[
\text{Sequence()} 
\]

Table 3.2 Munger behaviors for various types of arguments (Continued)

<table>
<thead>
<tr>
<th>Find, length, and replace arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you specify zero as the length and specify a replace string, the string is inserted before the offset position.</td>
<td>Munger(&quot;the quick brown fox&quot;, 1, 0, &quot;see &quot;); &quot;see the quick brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a negative integer as the length value and specify no replace string, Munger returns all characters from the offset to the end of the string.</td>
<td>Munger(&quot;the quick brown fox&quot;, 5, -5); &quot;quick brown fox&quot;</td>
</tr>
<tr>
<td>If you specify a negative integer for length and specify a replace string, Munger replaces all characters from the offset to the end with the replace string.</td>
<td>Munger(&quot;the quick brown fox&quot;, 5, -5, &quot;fast&quot;); &quot;the fast&quot;</td>
</tr>
</tbody>
</table>
Sequence(from, to, stepsize, repeat)

From and to are not optional. They specify the range of values to place into the cells. If from = 4 and to = 8, the cells are filled with the values 4, 5, 6, 7, 8, 4, ...

Stepsize is optional. If you do not specify a stepsize, the default value is 1. Stepsize increments the values in the range. If stepsize = 2 with the above from and to values, the cells are filled with the values 4, 6, 8, 4, 6, ...

Repeat is optional. If you do not specify a Repeat, the default value is 1. Repeat specifies how many times each value is repeated before incrementing to the next value. If repeat = 3 with the above from, to, and stepsize values, the cells are filled with the values 4, 4, 4, 6, 6, 6, 8, 8, 8, 4, ... If you specify a Repeat value, you must also specify a Stepsize value.

The sequence is always repeated until each cell in the column is filled.

Example:

```// Create a new data table
dt = New Table("Sequence Example");
```

```// Add 2 columns and 50 rows
dt << New Column("Count to Five");
dt << New Column("Count to Seventeen by Fours"); dt << Add Rows (50);
```

```/* Fill the first column with the data sequence 1, 2, 3, 4, 5, ...
Fill the second column with the data sequence 1, 1, 5, 5, 9, 9, 13, 13, 17, 17, ...
*/```

```for each row (```
```
column(1)[ ] = Sequence(1,5);
column(2)[ ] = Sequence(1,17, 4, 2);
```
```
```);```

Since Sequence() is a formula function, you can also set a column’s formula to use Sequence to fill the column. This example creates a new column named “Formula Sequence” and adds a formula to it. The formula is a sequence that fills the column with values between 25 and 29, incremented by 1, and repeated twice (25, 25, 26, 26, 27, 27, 28, 28, 29, 29, 25, ...).
```
dt << new column("Formula Sequence", formula(Sequence(25, 29, 1, 2)));
```

Examples of Sequence results:

Sequence(1,5) produces 1,2,3,4,5,1,2,3,4,5,1,...
Sequence(1,5,1,2) produces 1,1,2,2,3,3,4,4,5,5,1,1,...
Sequence(10,50,10) produces 10,20,30,40,50,10,...
10*Sequence(1,5,1) also produces 10,20,30,40,50,10,...
Sequence(1,6,2) produces: 1,3,5,1,3,5,... The limit is never reached exactly.

Note: If you want a matrix of values, then use the Index function, not Sequence.
Comparison and Logical Operators

Here are some examples to show what you should expect for comparisons involving missing values, mismatched types, and matrices.

Table 3.3 Some Special-Case Comparison Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m==.; m==1</td>
<td>.</td>
<td>Equality test with missing returns missing.</td>
</tr>
<tr>
<td>m==.; m!=1</td>
<td>.</td>
<td>Inequality test with missing returns missing.</td>
</tr>
<tr>
<td>m==.; m&lt;1; m&gt;1; and so forth</td>
<td>.</td>
<td>Any comparison with missing returns missing (unless it could not possibly be true, see next).</td>
</tr>
<tr>
<td>m==.; 1&lt;m&lt;0</td>
<td>0</td>
<td>Comparison involving missing value that could not possibly be true returns false, because false takes precedence over missing for comparisons with more than two arguments (as with logical operators).</td>
</tr>
<tr>
<td>{a, b}==list(a, b)</td>
<td>1</td>
<td>Equality test of list arguments returns a single result.</td>
</tr>
<tr>
<td>{a, b}&lt;{a, c}</td>
<td>.</td>
<td>Comparison test of list arguments is not allowed.</td>
</tr>
<tr>
<td>1==&quot;abc&quot;</td>
<td>0</td>
<td>Equality test with mixed types returns false.</td>
</tr>
<tr>
<td>1&lt;=&quot;abc&quot;</td>
<td>.</td>
<td>Comparison with mixed types returns missing.</td>
</tr>
<tr>
<td>IsMissing(m)</td>
<td>1</td>
<td>Returns 1 if missing, 0 otherwise.</td>
</tr>
<tr>
<td>[1 2 3]==[2 2 5]</td>
<td>[0 1 0]</td>
<td>Equality test of matrices returns matrix of elementwise results. When a matrix is compared to a matrix, comparison is done element-by-element and returns a matrix of 1s and 0s.</td>
</tr>
<tr>
<td>[1 2 3]==2</td>
<td>[0 1 0]</td>
<td>Equality test of matrix and a matrix filled with 2s. If a matrix is compared to a number, the number is treated as a matrix filled with that number.</td>
</tr>
<tr>
<td>[1 2 3] &lt; [2 2 5]</td>
<td>[1 0 1]</td>
<td>Comparison of matrices returns matrix of elementwise results.</td>
</tr>
<tr>
<td>[1 2 3] &lt; 2</td>
<td>[1 0 0]</td>
<td>Comparison of matrix and a matrix filled with 2s.</td>
</tr>
<tr>
<td>All([2 2]==[1 2])</td>
<td>0</td>
<td>All summarizes elementwise comparisons; it returns 1 only if all comparisons are true and returns 0 otherwise.</td>
</tr>
<tr>
<td>Any([2 2]==[1 2])</td>
<td>1</td>
<td>Any summarizes elementwise comparisons; it returns 1 if any comparison is true and returns 0 otherwise.</td>
</tr>
</tbody>
</table>
Missing Values

Missing values propagate missing values for most comparisons and logical operations. Two exceptions: if one value is true and another missing, Or returns true; if one value is false and the other missing, And returns false.

Here is an easy way to remember how missing values work. Suppose you have survey data and some respondents did not specify gender, male or female, so you have missing values. Now you compare two missing values. Are they the same gender? They might be, but you do not know for sure, so the result is missing. Are they not the same gender? Again, you do not know, so the result is missing. Likewise, you cannot know whether some value (including a missing value) is greater than or equal to a missing value, so the result is missing.

What about when you compare a missing value with a known value? Sometimes it is possible to determine the result. For example, False AND Missing evaluates to False, because for an AND test to be true, both things must be true. Since one thing is False, it does not matter what the other is. The result is false. Similarly, True OR Missing evaluates to True, because only one thing in an OR test needs to be true for the result to be true.

If you prefer to see truth tables, submit this script:

```plaintext
T = [0, 0, 0, 1, 1, ..., .];
U = [0, 1, 0, 1, 0, ...];
```

You cannot make comparisons directly with explicit missing values, but only with variables, matrices, or other things, which could contain missing values. For direct comparisons, use Is Missing or Zero Or Missing:

```plaintext
a==.; a<=.; // produce errors - you should use IsMissing(a)
IsMissing(a); // returns 1 when a is missing
b==.;a==b; a<>b; // does the comparisons
ZeroOrMissing(a); // returns 1 when a is missing or a is zero
```


**Missing Character Values**

Only numeric missing values are considered to be missing values. Missing character values are considered to be strings of zero length, not missing values, so you would use a test such as `a==""` rather than `IsMissing(a)`.

**Short-Circuiting Behavior**

JMP’s comparison and logical operators are short-circuiting operators. They stop working as soon as they can return a result. Usually this produces the results that you expect, but you should be careful in the following circumstances.

When a later argument depends on an earlier argument

For example, in this expression, `List[50]` is not examined unless the list actually has at least 50 elements:

\[
\text{n items(list)} \geq 50 \text{ & list[50]} == 10;
\]

When lengthy calculations pose performance concerns

Suppose you have an accurate measurement function that is slow, but an approximate version that is fast. You might consider doing a quick, rough comparison first, and then running the lengthy test only when the quick test’s result crosses a certain threshold:

\[
\text{est(x)} < .05 \text{ & acc(x)} < 0.01;
\]

When you depend on later arguments being executed

Be careful about placing assignment statements (for example, in a part of an expression that might never be executed).
JMP has many types of objects that can respond to messages. This chapter shows how to create and manage data objects: data tables, data columns, and rows. JSL’s messages for working with data correspond to the items in the **Tables**, **Rows**, and **Cols** menus in the graphical user interface, and in turn to items in the various panes of a data table window.

This chapter starts with an introduction to data objects and how to send messages to them and continues with the JSL for working with each of the objects.
Data Table Basics

JSL has full access to the values and properties of a data table. This section examines how to open and close data tables, view their properties, and interact with their data.

Once you have a data table open, you accomplish most other data tasks by sending messages to a reference to the data table object. Objects acting on messages is a common theme in this book. As soon as you see how to create or open a data table object, step back and look at how objects and messages work.

Getting a Data Table Object

Any of the following methods of beginning to work with a data table produces a data table object.

Opening a Data Table

Opening a data table returns a reference to the data table. Usually, you assign the reference to a variable for later use. The usual argument for Open is a path (either relative or absolute) to the data table. For information about using POSIX filenames and the six built-in path variables, see “File Paths,” p. 30 in the “JSL Building Blocks” chapter.

```
Open("$SAMPLE_DATA/big class.jmp"); // just open the data table
dt=Open("$SAMPLE_DATA/big class.jmp"); // open and assign a reference
```

The path can be a literal path (absolute or relative) in quotation marks or an expression yielding a pathname. Relative paths are interpreted relative to the location of the .JSL file (if the script has been saved) or the JMP application (for an unsaved script); use ../ to move up one folder level. Example scripts typically use a relative path to open files from the installed Sample Data folder. You can also use the Set Current Directory command.

If no arguments are given, Open presents a window prompting the user to navigate to a file.

If a data table with that path is already open, then JMP does not read it again but instead brings that window in the front. If you have made changes, then the table is not the same as the one on disk that you might expect. Since JMP always keeps the whole table in memory, opening is equivalent to reading the whole file.

Importing Data From a Text File

If you want to import data from a text file, you need to include some additional arguments.

```
Number of Columns(number)
Columns(colName=colType(colWidth),...)
  // colType is Character|Numeric
  // colWidth is an integer specifying the width of the column
End Of Field (Tab|Space|Comma|Semicolon|Other|None)
EOFother("char")
End Of Line (CRLF|CR|LF|Semicolon|Other)
EOLOther("char")
```
Strip Quotes|Strip Enclosing Quotes (Boolean)
Labels|Table Contains Column Headers (Boolean)
Year Rule|Two digit year rule (“10-90”| “19xx”| “20xx”| Custom)
Column Names Start|Column Names are on line (number)
Data Starts|Data starts on line (number)
Flags (number)
Custom Rule (name)
Lines to Read
Use Apostrophe as Quotation Mark

For example,

open("c:\test.txt", End of Field(comma), Labels(0),
columns(studentName=character(12), age=numeric(14)));

or:

open("c:\test2.txt", End of Field(Other), EOFOther ("|"), Labels(1),
columns(studentName=character(12), age=numeric(14)));

The options for Open:

**Column names and lengths**  You can specify column names, types, and field widths with a Columns argument. For example, columns(studentName=character(12), age=numeric(14)) in the example above.

Note that if you specify settings for a column other than the first column in the file, you must also specify settings for all the columns that precede it. For example, You want to open a text file that has four columns, Name, Sex, Age, and ID, in that order, and you want to set Age to numeric(2). You must also set Name and Sex, and you must list them in the same order:

columns(Name=character(15), Sex=character(1), Age=numeric(2))

You are not required to specify the settings for any columns that follow the one that you want to set.

**Strip Quotes**  Boolean. Specifies whether to remove quotation marks from string values.

When saving spreadsheet-style data in text format, most programs, including JMP, enclose string values inside double quotation marks (" "). This ensures that when another program imports the text file, any spaces, tabs, or commas that are meant to be part of a string value are not misinterpreted as field delimiters. For example, a string value such as John Doe would be interpreted as two separate strings, John and Doe, if space were the field delimiter. But "John Doe" inside quotation marks would be interpreted as a single string, because when most programs including JMP read a quotation mark, they disregard any delimiter characters and keep reading until they encounter the second quotation mark.

However, since you do not usually want those double quotation marks to be part of the value (for example, you want John Doe, not "John Doe", in the data table), JMP provides an option Strip Quotes. If you choose this option, JMP respects the quotation mark rules but removes the marks from the values.

Warning: text import in JMP 4 and later is different from JMP 3 and earlier. In version 3, delimiter characters within quoted string were ignored only if they were not the chosen delimiter character. For example, if you chose "space" as the delimiter, you could safely use tabs, commas, and semicolons inside quoted strings, but you could not use a space inside a quoted string. In version 4
and above, text import was improved to ignore even the chosen delimiter, as long as the value is inside double quotation marks.

Another warning: many word processors have a "smart quotes" feature that automatically converts double quotation mark " characters into left and right curled quotation marks " ". These special characters cause problems with text import and also for JSL.

**End of Line** (CR | LF | CRLF | Semicolon | Other)  
Specifies the character(s) used to separate records. The choices are CR for carriage returns (typical for Macintosh text files), LF for linefeeds (typical for UNIX text files), or CRLF for both carriage returns and linefeeds (typical for Windows text files). The default is to interpret all three as line delimiters. Other adds another character of your choice to be interpreted as a record separator. Specify the character with **EOLOther**. Note that specifying Other does not block the default action, but adds to it.

**End of Field** (Tab | Space | Spaces | Comma | Semicolon | Other | None)  
Specifies the character(s) used to separate fields. The default field delimiter is Tab. Other adds another character of your choice to be interpreted as a record separator. Specify the character with **EOFOther**. Note: Space uses a single space as a delimiter, while Spaces uses two or more.

**EOFOther, EOLOther**  
specifies a special character as the end-of-line or end-of field. For example, `EOLOther("*")` specifies the asterisk as the end-of-line character.

**Labels**  
Boolean. Indicates whether the first line of the text file contains column labels or not.

**Year Rule** (Two digit year rule)  
specifies the year rule. Options are 10-90, 19xx, 20xx, or Custom.

**Custom Rule**  
specifies JSL that handles dates in a custom way.

**Column Names Start** (Column Names are on line)  
specify the starting line for column names.

**Data Starts** (Data Starts on line)  
specify the starting line for data.

**Lines to Read**  
Specifies the number of lines to read.

**Use Apostrophe as Quotation Mark**  
Declares apostrophes as quotation marks.

**Importing Other File Types**

JMP can also import other supported file types through the **Open** command.

On Windows, JMP relies on the usual three-letter filename extensions (for example, .XPT, .XLS, and so on.) to identify the type of file and how to interpret its contents. Some examples:

```julia
sasxpt=open("carpoll.xpt");
sassd2=open("class.sd2");
sasdbf=open("bigclass.dbf");
xls=open("bigclass.xls");
csv=open("book1.csv");
```

On Macintosh, JMP relies on the Macintosh Type and Creator codes (if present) and secondarily on three-letter filename extensions. (Type and Creator codes are invisible data that enable the Finder to display a file with the correct icon corresponding to the application that created it; files with generic icons (such as often appear on files obtained from other system) should have the filename extensions.)
Open Database

Open Database opens a database using ODBC and extracts data into a JMP data table. See the “Extending JMP” chapter.

Starting a New Data Table

You can start a new data table, or start a new data table and store its reference in a global. In either case, you can specify a name for the table as an argument.

```julia
New Table();
dt=New Table("myName.jmp");
```

If you want to avoid a window appearing when you create a new data table, use the invisible keyword. For example, to create an invisible table Abc with one column of ten rows,

```julia
dt=newTable("Abc", invisible, newColumn("X"),addRows(10));
```

**Note:** To prevent “lost” data tables created invisibly, you can create an invisible data table only if you assign it a reference. If you do create an invisible data table without a reference, JMP immediately destroys it, so any further uses of the table result in errors.

**Tip:** Once you are finished with an invisible data table, do not forget to close it. Otherwise, it remains in memory until you quit JMP.

More Ways of Opening Data

You can open data tables that are accessible through a Web page:

```julia
open("URL", <extension>)
```

The Web site URL is a quoted string that exactly describes the Internet location of a JMP data table. Add the extension (for example, jmp) as an unquoted named argument if the URL for the data table does not have the .jmp extension.

Note that the URL is case sensitive.

You can import data the same way. For example, specify `text` for a text import using your text import preferences.

```julia
open("http://www.sas.com/", text);
```

If you do not add the text argument, the web page is returned as a string.

Additional formats include `csv`, `script`, and `journal`.

Objects and Messages

Many JMP capabilities are performed by scriptable objects within JMP that know how to execute tasks relevant to themselves. Objects are entities that, once created, know how to do certain things for themselves. A JMP data table is one such object.
Chapter 4

Data Tables
Data Table Basics

91

To work with an object from JSL, you send a message to the object, asking it to perform one of its tasks.
Messages are commands that can be understood only in context by a particular type of object. For example,
messages for data table objects include Save, New Column, Sort, and so on. Most of these would not make
any sense for a platform object, for example.
Each object defines its own set of messages and what those messages mean. Although there might be some
messages that happen to exist for more than one type of object, each message’s effect is determined solely by
that object. This is one way in which JSL is heavily context-dependent.
First, you see how to address a data table object with a reference, then how to send a message to the reference,
and then learn about the messages data table objects can understand. A later section examines another type
of object, data table columns. Subsequent chapters explore analysis and graph platform objects, display
objects, and others.
Storing a Reference to a Data Table
To review, here are the various ways to get a data table open. Notice that each one begins with an assignment
to a name. That is because each of these operators, in addition to creating or opening the data, return a
value that is a reference to the table (a reference to a scriptable object). By assigning the operator to a global
variable, you capture that reference in the variable, and from now on you can use that variable to talk to the
object. (This manual uses the placeholder dt to represent data table references.)
dt=open("..\sample data\big class.jmp"); //Windows
dt=open("::sample data:big class");
//Macintosh
dt=open("$SAMPLE_DATA/Big Class.JMP"); // all operating systems
dt=new table("myTable"); // create a new data table

You can also assign a reference to a table you already have open, in case you did not include an assignment in
your Open or New Table statement (or if you opened it through the menus):
dt=data table("myTable"); // the open table named myTable
dt=current data table(); // the table in the topmost window
dt=data table(3);
// the third open table
Current Data Table can also take a scriptable object reference as an argument. For example, this would

make current the data table in use by the second Bivariate object:
Current Data Table(Bivariate[2]);

The section “Sending Script Commands to a Live Analysis,” p. 160 in the “Scripting Platforms” chapter,
discusses references to analysis platform objects.
Sending a Message to a Reference
Now, having gotten a reference to a table, you just send it a message with the << operator, also known as the
Send function:
dt=Open("$SAMPLE_DATA/Big Class.jmp");
dt<<save();
Send(dt,save());


Most of the rest of this chapter discusses various messages that you can send to data tables, or rows or columns inside data tables. The pattern is always the same:

```
reference << message(arguments);
```

Usually saving a reference in a global once and then sending many messages to the globals is most convenient, but the direct route also works, and is easier if you have only one message to send:

```
current data table() << save();
```

Finally, you can stack up a series of messages in one statement. The result of sending one message to an object is a usually reference to the newly altered object, so then subsequent messages just get sent to the evolving object in turn.

```
dt << set name("myName") << new column("myColumn") << save();
close(dt);
```

### Running a Script Attached to a Data Table

The Run Script command, when sent to a data table, finds a named property (in other words, a script) and runs it as a JSL script.

For example, if you open Big Class.jmp and submit the following, the script named "Distribution" is run.

```
Current Data Table() << Run Script ("Distribution")
```

### Closing a Data Table

To close a data table, use a Close command with the data table’s reference as the argument. If there are unsaved changes, the data table is saved automatically. To close a data table without saving it, use the No Save argument.

```
dt = Open("$SAMPLE_DATA/Big Class.jmp");
   // This command closes the file and saves it if it has been changed:
   close(dt);
   //This command closes the file and saves a copy with a new name:
   close(dt, save("myFile"));
   //This command closes the file and does not save any changes.
   close(dt, No Save);
```

### Reverting to a Saved Data Table

To revert to the most recently saved data table, send the Revert message to the data table. Before reverting, JMP asks whether the current data table should be saved.

### Counting Open Data Tables

```
N Table() returns the number of open data tables.
```
Getting and Using a List of Open Data Tables

If you want to do something to all the data tables that are currently open, you can use N Table to get a list of references to each one:

```
openDTs = List();
For( i = 1, i <= N Table(), i++,
    Insert Into( openDTs, Data Table( i ) );
);
```

`openDTs` now is a list of references to all open data tables. You can send messages to any one by using `openDTs(n)`. You can use a for loop to send messages to all of the open data tables. This loop adds a new column named My Column to each open data table.

```
For( i = 1, i <= N Items(openDTs), i++,
    openDTs[i]<<newcolumn("My Column");
);
```

If you just want a list of table names and not references, use the Get Name message:

```
For( i = 1, i <= N Table(), i++,
    Insert Into( openDTs, Data Table( i )<<Get Name());
);
```

Why are Some Commands Sent to Objects and Others Used Directly?

New Table and Open are commands to create objects that do not exist yet. Once created, these objects know how to manipulate themselves, so you send them messages requesting changes. To close such objects, you must use a command to its container, because the objects cannot delete themselves. Close is a command.
Learning About Data Tables’ Messages

Show Properties lists the messages a data table can receive in the Log window. Show Properties is a command that takes any scriptable object, such as a data table, as its argument.

```julia
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show Properties( dt );
```

The message list is hierarchical, and [Subtable] entries show the menus in which messages are found. For example, one category of messages is Tables, and these are all the messages shown in the main Tables menu.

The messages labeled as [Action]s all result in some action being taken. Some are shown to be available for [Scripting Only] or [Interactive Only].

The Analyze and Graph items are listed here because you can send a platform name as a message to a data table to launch the platform for that data table through the usual launch window. The “Scripting Platforms” chapter discusses platform-launching in more detail.

```julia
dt<<Distribution(Y(height));
```

Data table objects contain column objects, which can handle messages of their own. The later section “Manipulating columns,” p. 121, discusses how to refer to column objects, show their properties, and send them messages.

Show Properties also works with platforms and display boxes; see “Learning the Messages an Object Responds to,” p. 162 in the “Scripting Platforms” chapter and “Learning What You Can do with a Display Box,” p. 183 in the “Display Trees” chapter. Some general tips that also apply to data table objects appear in “How to Interpret the Listing from Show Properties,” p. 163 in the “Scripting Platforms” chapter. Refer to the JSL browsers in the Index pane of the JMP Starter Window for a brief overview of any item.

Messages for Data Tables

This section details the messages that you can send to data tables. Previous sections discussed how to start or open data tables and how to send basic messages to them. The next section discusses data table column objects and their messages.
Naming, Saving, and Printing

Before saving, you might want to give the table a name, or change it to a new name. The Set Name message does this, and its argument is just a filename in quotation marks, or else something that evaluates to a filename.

```
dt << set name("new big class.jmp");
```

There is a command to obtain the name:

```
name = dt << Get Name;  // to retrieve the name
```

To save, you send a Save message. You can also specify the filename (and a complete path) as an argument to a Save message, if you do not want to do a separate Set Name message. The syntax is:

```
dt << Save (<"pathname">, "file type", <JMP(5)>);
```

Some examples:

```
dt << save();  // Save using the current name
dt << save("c:/mydata/new big class.jmp");  // Save a new file
dt << save("MyTable", JMP(5));  // Save as JMP 5 table
```

On Windows, saving with a .txt extension exports according to the current Text Export preferences. On the Macintosh, add Text as a second argument to the save function:

```
current data table() << save("New Big Class.txt", Text);
```

You can print the current data table by sending it a Print Window message.

```
current data table() << Print Window
```

Subscribing to a Data Table

Use the Subscribe message to runs scripts when a certain data table is closed, when columns are added or deleted, and when rows are added or deleted. The basic syntax is as follows:

```
dt << Subscribe("application name", On Close( function ));
```

The first argument is a name for the subscription (or "application"), so that it can also be removed:

```
dt << Unsubscribe("application name", On Close);
```

The application can also subscribe as a client to the data table (for example, most built-in platforms such as Distribution). If a data table has clients when the data table is closed, the user is warned that there are applications open that might need the data table.

Example

This example prints the information to the log when a row or column is either added or deleted.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

fcols = Function( {a, b},
   n = N Items( b );
   dtname = (a << getname());
   ```
Chapter 4

Messages for Data Tables

```plaintext
Print( dtname );
Print( n );
For( i = 1, i <= n, i++,
    colname = (b[i] << getname());
    Print( colname );
);

dt << subscribe( "coldel", ondeletecolumns( fcols ) );
dt << subscribe( "coladd", onaddcolumns( fcols ) );

frows = Function( {a, b},
    dtname = (a << getname());
    Print( dtname );
    Print( b );
 );

dt << subscribe( "rowadd", onAddRows( frows ) );
dt << subscribe( "rowdel", onDeleteRows( frows ) );
```

Journal and Layout

Journal and Layout for a data table are accessible from JSL.

```plaintext
current data table() << journal;
current data table() << layout;
```

You can also create a new journal window and immediately add to it within the `NewWindow()` command. For example:

```plaintext
NewWindow(<<Journal);
```
creates an empty untitled journal.

```plaintext
NewWindow("title",<<Journal);
```
creates an empty journal with the title "title".

```plaintext
New Window( "Test Buttons", <<Journal,
    Button Box( "Test One", New Window( "Hi there1", <<Modal ) ),
    Button Box( "Test Two", New Window( "Hi there2", <<Modal ) )
 );
```
creates a journal titled "Test Buttons" with two buttons in it.

There is an optional parameter in the journal and journal window commands. This parameter “freezes” the current display. That is, it converts the display tree with just a picture.

The four options for parameters are:

```plaintext
<<journal (freeze all)
```
wants the selected display tree in a picture node and takes its picture.
Messages for Data Tables

<<journal(freeze pictures)
takes pictures of each picture node.

<<journal(freeze frames)
takes pictures of each frame.

<<journal(freeze frames with scripts)
takes pictures of each frame that has a script.

Current Journal() returns a reference to the display box at the top of the current journal display window. If no journal is open, one is created. There are no arguments.

You can add to the journal using the Append command

Current Journal()<<Append(<display box reference>);

For example,

Current Journal()<<Append(Text Box("Hello World"));

You can also find items in an existing journal by enclosing the search specification in square brackets

Current Journal()[specification]

For example,

Current Journal()["Parameter Estimates"]<<Append(Text Box("Asterisks show items significant at 0.05"));

Creating and Deleting Columns

New Columns

To add a new column to an existing data table, send a New Column message to a data table reference. The first argument (required) is the column's name, which must either be a name in quotation marks or else an expression evaluating to that. Further arguments are optional and set attributes for data type (Numeric, Character, or RowState) and analysis type (Continuous, Nominal, or Ordinal). You can also include Values, Formula, and other script messages appropriate to a column.

dt<<NewColumn("logHt");
tt="logHt"; dt<<New Column(tt, numeric, ordinal);
dt<<New Column("Ratio", Numeric, Continuous, Formula(Height/Weight));
dt<<New Column("Last Name", Character, Width(16), Values({"Smith", "Jones", "Anderson"}));
dt<<NewColumn("RowNumber",Numeric,Values(1::NRow()), Format("Best",5));
dt<<NewColumn("myMarkers",row states, set formula(marker state(age-12)));}

New Column can also be used as a built-in function, in other words, without the << (send) command applied to the data table reference. When used in this way, it is applied to the current data table. The built-in function and command work identically.

New Column("logHt");
New Column returns a column reference that can be saved for later use.

```js
myCol = dt<<NewColumn("logHt");
```

Further details about column references and how you can send commands to column objects are covered in “Manipulating columns,” p. 121. The commands covered there can all be used as additional arguments to New Column.

You can also add columns with Add Multiple Columns... from the Columns pop-up menu. The JSL equivalent for this is a Add Multiple Columns message. Arguments are, in order, the prefix to use in naming the columns, then how many columns, where to insert them (Before First, After Last, or After(col)), and data type (Numeric, Row State, or Character(width)).

Some examples are:

```js
dt<<add multiple columns("beginning",2,before first, rowstate);
dt<<add multiple columns("middle",3,after(height),numeric);
dt<<add multiple columns("end",4,after last,character(4));
```

### Deleting Columns

To delete columns, send a Delete Columns message and specify which column or columns to delete. To delete more than one column, list the columns as multiple arguments or as a list. Without an argument, Delete Columns deletes the currently selected columns.

```js
dt<<delete columns("weight", "age", "sex"); // specified column(s)
dt<<delete columns({"weight", "height"});
```

To create columns, see “Creating and Deleting Columns,” p. 97.

### Reordering Columns

These messages enable you to rearrange columns in a data table:

```js
dt<<Reorder By Name; // alphanumeric order
dt<<Reverse Order; // reverse current order
dt<<Reorder By Data Type; // character then numeric
dt<<Reorder By Modeling Type; // continuous, ordinal, nominal
dt<<Original Order; // saved order
```

### Accessing Values

The typical way to access values in a data table is to set up a data table as the “current data table,” specify a row to be the “current row,” and then access the value for a particular column on that row through a column name. This section discusses each of these steps separately, puts them together with a few examples, and then also looks at other strategies.
Chapter 4

Data Tables

Messages for Data Tables

99

Current Data Table

A data table becomes the current data table when you Open it, or when you create it with New Table. To switch to another already-open, already-created data table, you can use the Current Data Table command:

Current Data Table(myTable);

Current Row

The current row is determined by the value of a pseudo-variable called Row(). You can either assign a value to Row() or else use the iterative command For Each Row().

Row()=3; ...
For Each Row(...);

“Iterating on Rows of a Table,” p. 100, and “What is the Current Row?,” p. 101 give more details.

Selected Columns

To get a list of selected columns, use the Get Selected Columns message.

dt<<Get Selected Columns

To select a column, use the Set Selected message. See more information in “Manipulating columns,” p. 121.

col<<Set Selected (1);

Selecting Rows

To select certain rows, use the Select Rows message.

Current Data Table()<<Select Rows({1, 3, 5, 7})

Setting or Getting Values by Column Name

The easiest way to refer to a column is by its name. To prevent ambiguity in case you have a global variable that also uses the same name, you can use the : prefix operator to scope it as a column name. It works without the prefix, but we recommend the prefix. To set the cell of the current row of the column name, use the assignment statement with the column name on the left (it is an L-value).

:name = "Archibald"

If you want to get the value, just specify the name, preferably scoped with the colon.

show(:name);
myGlobal = :name;

If you want to set a specific row of the column, rather than the current row, use a subscript with the row number.

:name[rownum] = "Archibald"
rowValue = :name[rownum];
By convention, an empty subscript refers to the current row, so :name[] is the same as :name.

Confusion alert! Be careful that you are subscripting to a row of the table that exists. The default row number is zero, so statements like :name that refer to row zero generate an Invalid Row Number error.

Putting it all Together (Example)

Here, you first open a table to make it the current data table, use For Each Row to use every row in turn as the current row, and refer to a column to get its values.

```julia
dt=open("$SAMPLE_DATA/Big Class.JMP");
for each row( print(:age) );
```

Other Ways

There are other ways to specify all three elements (the data table, the row, and the column). Some of these involve the indirect reference to a column through a column reference, as covered in detail in “Manipulating columns,” p. 121. All the following lines obtain the value of row 2 of column age.

```julia
:age[2];                     // returns value of age in row 2: 12
column("age")[2];           // same but indirectly through column reference
col=column("age"); col[2];   // same but with a stored column reference
Row()=2; col[];              // empty subscript means current row
```

Confusion alert! When referring to columns indirectly (using column references) a subscript is required to refer to the individual values. Without a subscript, the reference is to the column object as a whole.

You can specify all three items directly by using the :infix operator and a subscript:

```julia
dt:Age[2] = 12;              // table, column, and subscript
```

If you want to target multiple rows, you can use subscripts with a list or matrix of row numbers.

```julia
if (age<13,print(row()));
if (age[i]<13,print(i));
age[i] = 3;
age[{3,12,32}] = 14;
list = age[{3,12,32}]; // results in a list
vector=age[1::20];     // results in a matrix
```

Iterating on Rows of a Table

In addition to the programming operators for iterating that are built into JSL (see “Iterating,” p. 61 in the “Programming Functions” chapter), JSL has operators for iterating through data table rows, groups, or conditional selections of rows. This section first introduces basic concepts and some useful operators.

Generally, script executes on the current row of the data table only. Some obvious exceptions are the expressions inside formula columns, Summarize and the pre-evaluated statistics operators, and any use of
data table columns by analysis platforms. Otherwise, though, a script is performed on the current row of the data table.

**What is the Current Row?**

By default, the current row number is 0. The first row in a table is row 1, so row 0 is essentially not a row. In other words, by default, an operation is done on no rows. Unless you take action to set a current row or to specify some set of rows, you get missing values due to the lack of data. For example, a column name returns the value of that column on the current row. Use the prefix : operator to avoid ambiguity (to force the name to be interpreted as a data table column name).

```julia
:sex; //returns ""
:age; //returns .
```

Why? This default protects you from getting a result that might look reasonable for the whole data table but is actually based on only one row. It also protects you from accidentally overwriting data values when making assignments to ambiguous names under most circumstances. (You can have even more complete protection by using the prefix or infix : operator to refer unambiguously to a data column and the prefix :: operator to refer unambiguously to a global script variable. See “Stored expressions,” p. 406 in the “Advanced Concepts” chapter.)

You can use the `Row()` operator to get or set the current row number. `Row( )` is an example of an *L-value* expression in JSL: an operator that returns its value unless you place it before an assignment operator (=, +=, and so on.) to set its value.

```julia
Row();     //returns the number of the current row (0 by default)
x=Row();   //stores the current row number in x
Row()=7;   //makes the 7th row current
Row()=7; :age;   //returns 12
```

Note that the current row setting only lasts for the portion of a script that you select and submit. After the script executes, the current row setting resets to the default (row 0, or no row). Therefore, a script submitted all at once can produce different results from the same script submitted a few lines at a time.

**How Many Rows and Columns?**

The `NRow` and `NCol` operators count the rows and columns in a data table (or matrix; see the “Matrices” chapter).

```julia
NRow(dt); // number of rows
NCol(dt); // number of columns
```

**For Each Row**

To iterate a script on each row of the current data table, put `For Each Row` around the script.

```julia
For Each Row(if(:age>15, show(age)));
```

A typical use is for setting row states without creating a new formula column in the data table. The scripts below are similar except that the first one creates a row state column and the `For Each Row` version simply sets the row state without creating a column.
new column("My Rowstate", rowstate, formula( color state(age-9)));
for each row(color of(rowstate())=age-9);

To iterate a script on each row meeting a specified condition, combine For Each Row and If.

for each row(marker of(rowstate())=if(sex=="F",2,6));

Break and Continue can be used to control execution of a For Each Row loop. For details about using these two functions, see “Break and Continue,” p. 63 in the “Programming Functions” chapter.

Dif and Lag

JMP has two special operators that are useful for statistical computations, particularly when working with time series or cumulative data. Lag returns the value of a column \( n \) rows before the current row. Dif returns the difference between the value in the current row and the value \( n \) rows previous. The following lines are equivalent:

dt<<new column("htDelta");
for each row( :htDelta=height-lag(height,1) );
for each row( :htDelta=dif(height,1) );

Comparing Two Data Tables (Example)

Here is a little program to compare two data tables and show entries that differ:

dt1=open();
dt2=open();
nr = min(nrow(dt1),nrow(dt2));
nc = min(ncol(dt1),ncol(dt2));
for(i=1,i<=nr,i++,
    for(j=1,j<=nc,j++,
        v1 = Column(dt1,j)[i];
        v2 = Column(dt2,j)[i];
        if(v1\!\!=v2,show(i,ColumnName(j),v1,v2));
    ));

You could make a fancier program to compare data tables that would also check whether they have the same numbers of rows and columns, whether the columns share the same names, and whether they have the same row states in effect.

Summarize

The Summarize command gathers summary statistics for a data table and stores them in global variables (as opposed to Summary, which gathers summary statistics and presents them in a new data table). Named arguments are Count, Sum, Mean, Min, Max, StdDev, and Quantile, and each of these take a data column argument. If a name=By(groupvar) statement is included, then a list of subgroup statistics is assigned to each name. Count does not require a column argument, but it is often useful to specify a column to count the number of nonmissing values. Quantile also takes a second argument for specifying which quantile. For example, 0.1 for the 10th percentile.
Note: Excluded rows are excluded from Summarize calculations. If all data are excluded, Summarize returns lists of missing values. If all data have been deleted (there are no rows), Summarize returns empty lists.

For example, using Big Class:

```
Summarize(
    a = by( age ),
    c = count,
    sumHt = Sum( height ),
    meanHt = Mean( height ),
    minHt = Min( height ),
    maxHt = Max( height ),
    sdHt = Std Dev( height ),
    q10Ht = Quantile( height, .10 )
);
```

Since the script included a By group, the results are a list and six matrices:

```
show(a, c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht);
```

You can format the results using TableBox, as discussed in “Build Your Own Displays from Scratch,” p. 209 in the “Display Trees” chapter.

```
New Window( "Summary Results",
    Table Box(  
        String Col Box( "Age", a ),
        Number Col Box( "Count", c ),
        Number Col Box( "Sum", sumHt ),
        Number Col Box( "Mean", meanHt ),
        Number Col Box( "Min", minHt ),
        Number Col Box( "Max", maxHt ),
        Number Col Box( "SD", sdHt ),
        Number Col Box( "Q10", q10Ht )
    )
);
```
Figure 4.1 Results from Summarize

<table>
<thead>
<tr>
<th>Age</th>
<th>Count</th>
<th>Sum</th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>SD</th>
<th>Q10</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>465</td>
<td>59.125</td>
<td>51</td>
<td>68</td>
<td>5.36224</td>
<td>51</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>422</td>
<td>50.2857</td>
<td>56</td>
<td>65</td>
<td>3.03242</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>710</td>
<td>64.1667</td>
<td>61</td>
<td>69</td>
<td>2.36771</td>
<td>61.15</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>452</td>
<td>44.5714</td>
<td>82</td>
<td>87</td>
<td>1.38606</td>
<td>82</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>193</td>
<td>64.3333</td>
<td>60</td>
<td>68</td>
<td>4.94146</td>
<td>60</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>260</td>
<td>66.6667</td>
<td>82</td>
<td>70</td>
<td>4.16333</td>
<td>62</td>
</tr>
</tbody>
</table>

With a little work, you can combine total and grouped results in the window:

```r
Summarize(
  tc = count,
  tsumHt = Sum( height ),
  tmeanHt = Mean( height ),
  tminHt = Min( height ),
  tmaxHt = Max( height ),
  tsdHt = Std Dev( height ),
  tq10Ht = Quantile( height, .10 )
);

Insert Into( a, "Total" );
c = c |/ tc;
sumHt = sumHt |/ tsumHt;
meanHt = meanHt |/ tmeanHt;
minHt = minHt |/ tminHt;
maxHt = maxHt |/ tmaxHt;
sdHt = sdHt |/ tsdHt;
q10Ht = q10Ht |/ tq10Ht;

New Window( "Summary Results",
  Table Box(
    String Col Box( "Age", a ),
    Number Col Box( "Count", c ),
    Number Col Box( "Sum", sumHt ),
    Number Col Box( "Mean", meanHt ),
    Number Col Box( "Min", minHt ),
    Number Col Box( "Max", maxHt ),
    Number Col Box( "SD", sdHt ),
    Number Col Box( "Q10", q10Ht )
  )
);
```
When you do not use a By group, the result in each name is a single value. Compare:

```julia
Summarize( // a=by(age),
    c = count,
    sumHt = Sum( height ),
    meanHt = Mean( height ),
    minHt = Min( height ),
    maxHt = Max( height ),
    sdHt = Std Dev( height ),
    q10Ht = Quantile( height, .10 )
);
Show( c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
c = 40;
sumHt = 2502;
meanHt = 62.55;
minHt = 51;
maxHt = 70;
sdHt = 4.2423849397192;
q10Ht = 56.2;
```

The statistical results of Summarize are always matrices if there is a By clause, or scalars if there is no By clause.

Working with Metadata

The main purpose for a JMP data table is to store observation data, or measurements of various variables on a specific set of subjects. However, JMP data tables can also store metadata, or data about the data. Metadata include information such as the source of the data, comments about each variable, scripts for working with the data, analysis roles for columns, and so on.
Tools for working with metadata are found in the Table menu in the upper left pane of a data table, in the Column Info window, and so on. This section discusses how to work with metadata in JSL.

**Table Variables**

Table variables are for storing a single text string value, such as “Notes”. To see how variables work, first get its existing value and assign that to a global, \texttt{oldvar}, by sending a \texttt{Get Table Variable} message:

\begin{verbatim}
dt=open("$SAMPLE_DATA/Solubility.jmp");
dt<<get table variable("Notes");
"Chemical compounds were measured for solubility in different solvents. This table shows the log of the partition coefficient (logP) of 72 organic solutes in 6 aqueous/nonpolar systems."
\end{verbatim}

Now change the source from the string by using \texttt{Set Table Variable}, and then get the variable to check your work.

\begin{verbatim}
dt<<set table variable("Notes","Solubility of chemical compounds");
dt<<get table variable("Notes");
"Solubility of chemical compounds"
\end{verbatim}

You can create new variables of your own with the \texttt{New Table Variable} message, such as to store the source of the data table in its own table variable:

\begin{verbatim}
dt<<new table variable("Last used", abbrev date(today()));
\end{verbatim}

If you specify too few arguments for any of these messages, JMP presents a window to get the necessary information from you. (JMP often presents windows when your script is incomplete, a behavior that you can use to your advantage when writing scripts that need to query users for their choices.)

**Properties and Scripts**

A \textit{property} is similar to a table variable, but it is used to store an expression (for example, a JSL script). For example, if you do an analysis and then use the \texttt{Save Script to Data Table} command from the report’s pop-up menu, it saves a script named for the analysis platform in the data table. In the data table window, you can double-click the property name to see its script value, or you can select \texttt{Run Script} from its pop-up menu to execute the script. Also, when you run Design of Experiments (DOE), the resulting data tables automatically include properties called Model to launch Fit Model with the appropriate design. You can save a table property named QC Alarm Script or QC Test Alert to set up an alert script for control charts.

JSL has \texttt{New Table Property}, \texttt{Set Property}, and \texttt{Get Property} messages.

\begin{verbatim}
dt<<new table property("Bivariate", Bivariate(Y(weight),X(height)));
dt<<get property("bivariate");
Bivariate(Y(weight),X(height))
dt<<set property("Bivariate",Bivariate(Y(weight),X(height),Fit Line));
dt<<get property("bivariate");
Bivariate(Y(weight),X(height),Fit Line)
\end{verbatim}

If you specify too few arguments for any of these messages, JMP presents the appropriate window to get the necessary information from you.

See also “Setting and Getting Attributes,” p. 125, for ways to store extra information about columns.
Deleting Variables, Properties, and Formulas

Given a data table \( dt \) or column \( col \) with a property or variable \( name \), the following commands delete them.

\[
\begin{align*}
&dt << \text{Delete Table Variable}(name); \\
&dt << \text{Delete Table Property}(name); \\
&col << \text{Delete Formula}; \\
&col << \text{Delete Property}(name); \\
&col << \text{Delete Column property}(name);
\end{align*}
\]

On Open Property

A script named \( \text{On Open} \) that is stored as a property of the data table is automatically executed when the data table opens. Either use the \text{Save Script to Data Table} from pop-up menus to save a script as a property in the current data table and then double-click the property name for a window to change its name to \( \text{On Open} \), or else store the script using a \text{New Table Property} message. In this example, you send the \text{Sort} message to \( \text{Current Data Table()} \) rather than \( dt \), since \( dt \) might not be defined when the data table is opened.

\[
\begin{align*}
&dt<<\text{new table property}("OnOpen", \text{sortedDt=\text{current data table()}<<\text{Sort(\text{By(Name)},\text{output table name} \("\text{Sorted big class}\")})});
\end{align*}
\]

A system preference lets you suppress the automatic execution of \( \text{On Open} \) scripts. As a precaution, you should consider suppressing auto-execution when opening data tables that you receive from others:

\[
\begin{align*}
&\text{preference(suppress \text{On Open script eval(1)})};
\end{align*}
\]

Running a Script at Start Up

If you want to run the same script every time you start JMP, name it \( \text{jmpStart.jsl} \) and place it in one of the following places, as appropriate for your operating system. When JMP starts, JMP looks for the \( \text{jmpStart.jsl} \) script in these folders in the order in which they are listed here. The first one that is found is run, and the search immediately stops.

1. C:\Documents and Settings\<user>\Local Settings\Application Data\SAS\JMP\9 (Windows XP)
   C:\Users\<user>\AppData\Local\SAS\JMP\9 (Windows Vista and Windows 7)
   ~/Library/Application Support/JMP/9 (Macintosh)
2. C:\Documents and Settings\<user>\Local Settings\Application Data\SAS\JMP (Windows XP)
   C:\Users\<user>\AppData\Local\SAS\JMP (Windows Vista and Windows 7)
   ~/Library/Application Support/JMP (Macintosh)

The \( \text{jmpStart.jsl} \) script runs only for a particular user on a computer. You can add a script named \( \text{jmpStartAdmin.jsl} \) in one of the following places, as appropriate for your operating system. This script is run for every user on a computer. JMP searches for the administrator start-up script first, and runs it if found. Then JMP searches for the user start-up script, and runs it if found.

1. C:\Documents and Settings\All Users\Application Data\SAS\JMP\9 (Windows XP)
   C:\ProgramData\SAS\JMP\9 (Windows Vista and Windows 7)
Suppose you want a text representation of a data table, perhaps to e-mail to a colleague or to use as part of a script. You can obtain a script that reconstructs the information in a data table with `Get Script`:

```julia
dt<<get script;
New Table("Class", Add Rows(19), New Table Variable("Notes", "Example data to use for simple demonstrations of each platform. Use Name as a label, and any combination of the variables as Xs and Ys"), New Table Variable("Distribution", ""), New Table Variable("Oneway", ""), New Table Property("Distribution", Distribution(Column(Age, Height))), New Table Property("Oneway", Oneway(Y(Height), X(Age), Quantiles(1), Box Plots(1))), New Column("Name", Character, Nominal, Values("BARBARA", "ALFRED", "ALICE", "CAROL", "HENRY", "JAMES", "JANE", "JEFFREY", "JOHN", "JOYCE", "JUDY", "LOUISE", "MARY", "PHILIP", "ROBERT", "RONALD", "THOMAS", "WILLIAM"))), New Column("Sex", Character, Nominal, Set Property("Notes", "Explore data adventurously"), Values("F", "M", "F", "F", "M", "F", "F", "M", "F", "F", "M", "F", "F", "M", "M", "M", "M")), New Column("Age", Numeric, Nominal, Set Property("Notes", "Explore data adventurously"), Values([13, 14, 13, 14, 12, 12, 15, 13, 12, 11, 14, 12, 15, 16, 12, 15, 11, 15])), New Column("Height", Numeric, Continuous, Set Property("Notes", "Explore data adventurously"), Values([65.3, 69, 56.5, 62.8, 63.5, 57.3, 59.8, 62.5, 62.5, 59, 51.3, 64.3, 56.3, 66.5, 57, 64.8, 67, 57.5, 66.5])), New Column("Weight", Numeric, Continuous, Set Property("Notes", "Explore data adventurously"), Values([98, 112.5, 84, 102.5, 102.5, 83, 84.5, 112.5, 84, 99.5, 50.5, 90, 77, 112, 150, 128, 133, 85, 112])))
```

### Controlling Formula Evaluation

The message `Suppress Formula Eval` takes a Boolean argument to specify whether formula evaluation should be suppressed or not. You might want to suppress evaluation if you plan to make numerous changes to the data table and do not want to wait for formula updates between steps.

```julia
dt<<suppress formula eval(1);
dt<<suppress formula eval(0);
dt<<suppress formula eval;  //toggle current state
```

To accomplish the same effect for all data tables, use the `Suppress Formula Eval` command to turn off formulas globally. This is the same as the message above except that you do not send it to a data table object.

```julia
suppress formula eval(1); // make formulas static globally
suppress formula eval(0); // make formulas dynamic globally
```

Note that formulas are not evaluated when they are installed in the column. Even when you force evaluation, they end up getting evaluated again in the background. This is a problem for scripts if they
depend on the column having the values while the script is running. Some users need a mechanism to control evaluation, provided by the following commands.

To force a single column to evaluate, send an `EvalFormula` command to the column. You can even do this inside the command to create the column, after the formula clause. For example,

```
currentDataTable() << New Column("Ratio", Numeric, Formula(:height/:weight), EvalFormula)
```

`dt` << `Run Formulas` performs all pending formula evaluations: those that are pending, or those that are scheduled as a consequence of evaluating other formulas. It is useful when there are a whole series of columns to run. This method is preferred, since although `EvalFormula` evaluates the formulas, it does not suppress the background task from evaluating them again. The background task takes great care to evaluate the formulas in the right order.

If you also send the `Run Formulas` command to data column, the evaluation is done at the time of the command, but it does not suppress the scheduled evaluations that are pending. Therefore, formulas might end up being evaluated twice if you also send the command to the data table. Being evaluated twice might be desired for formulas that have random function in them, or it might be undesirable if they depend on randomization seeds being set. If you use random numbers and use the `Random Reset(seed)` feature to make a replicable sequence, then use the `Run Formulas` command, since it avoids a second evaluation.

**Note:** Starting in version 6, all platforms send a `Run Formulas` command to the data table to assure that all formulas have finished evaluating before analyses start.

**Setting Values Without a Formula**

```
col << Set Each Value(expression)
```
evaluates the expression for each row of the data table and assigns the result to the column. It does not store the expression as a formula.

**Controlling Display Updating**

Whenever you change values in a data table, messages are sent to the displays to keep them up-to-date. However, if you have thousands of changes in a script, this increases the time it takes to complete the updates.

In order to speed up changes, use `Begin Data Update` before the changes to block these update messages, and `End Data Update` after the changes have been completed to release the messages and update the displays. Be sure to always send the `End Data Update` message, otherwise the display is not updated until forced to do so in some other way.

For example,

```
Current Data Table() << Begin Data Update;
...<many changes>...
Current Data Table() << End Data Update;
```

**Resizing the Data Table**

```
dt << Maximize Display
```
forces the data table to remeasure all its columns and zoom to the best-sized window. For example,
Messages for Data Tables

Current Data Table() << Maximize Display;

Converting Between Matrices and Data Tables

If $dt$ is a variable containing a reference to a data table, then Get As Matrix creates a matrix $A$ with the values from all numeric columns in the data table:

\[
A = dt<<\text{GetAsMatrix};
\]

This also works with column objects, as discussed under “Values,” p. 126. If $col$ is a variable containing a reference to a data column in a data table, then Get As Matrix creates a matrix $A$ as a column vector of values from the column.

\[
A = col<<\text{GetAsMatrix};
\]

In addition, the As Table (matrix) command creates a new data table from a matrix argument. The columns are named Col1, Col2, and so forth.

Get All Columns As Matrix returns the values from all columns of the data table in a matrix. Character columns are numbered according to the levels, starting at 1.

Set Matrix creates a data table (or adds data to an existing data table), making new rows and new columns as needed to store the values of the matrix. The new columns are named col1, col2, and so forth.

For example,

\[
\text{newTable("A", Set Matrix([1 2 3, 4 5 6, 7 8 9])});
\]

creates a new data table named A with three rows and three columns.

\[
dt=\text{New Table("B")};
dt<<\text{Set Matrix([1 2 3 4 5, 6 7 8 9 10])};
\]

creates a new data table with two rows and five columns.

Commands from the Rows menu

This section describes the messages for working with the rows in a data table. Row messages are directed to a data table reference, and most act on the currently selected rows. Many of the row messages would not usually be useful in scripts, unless you were writing demo scripts that simulate interactive use of JMP.

Adding Rows

To add rows, send an Add Rows message and specify how many rows. You can also specify after which row to insert them. The arguments can either be numbers or expressions that evaluate to numbers.

\[
dt=\text{Open("$SAMPLE\_DATA/Big Class.jmp")};
dt<<\text{Add Rows(3)}; \quad // \text{adds 3 rows to bottom of data table}
dt<<\text{Add Rows(3, 10)}; \quad // \text{adds 3 rows after the 10th row, moving the 11th and lower rows farther down *}/
\]

A variation of Add Rows lets you specify an argument yielding a list of assignments. Assignments can be separated with commas or semicolons.
You can send several arguments yielding lists, or even a list of lists. This script creates a data table with Add Rows commands of each variety.

```plaintext
dt << Add Rows({:name="Peter", :age=14, :sex="M", :height=61, :weight=124});
add point = expr( dt << addRows({:xx=x; :yy=y}) );
```

Further details for rows can be specified with messages described under “Row State Operators,” p. 133.

### Deleting Rows

To delete rows, send a **Delete Rows** message and specify which row or rows to delete. To delete more than one row, give a list or matrix as the rownum argument, or combine **Delete Rows** with other commands such as For. The rownum argument can be a number, list of numbers, range of numbers, matrix, or an expression that yields one of these. Without an argument, **Delete Rows** deletes the currently selected rows. With neither an argument nor rows selected, **Delete Rows** does nothing.

```plaintext
dt = new Table("Cities");
dt << NewColumn("xx", Numeric);
dt << NewColumn("cc", Character, width(12));

dt << AddRows({xx=12, cc="Chicago"}); // single list
dt << AddRows({xx=13, cc="New York"}, {xx=14, cc="Newark"}); // several lists
dt << AddRows({{xx=15, cc="San Francisco"}, {xx=sqrt(256), cc="Oakland"}});
    // list of lists

a = {xx=20, cc="Miami"};
dt << AddRows(a); // evaluates as single list

b = {{xx=17, cc="San Antonio"}, {xx=18, cc="Houston"}, {xx=19, cc="Dallas"}};
dt << AddRows(b); // evaluates as list of lists
```

### Selecting Rows

**Select All Rows** selects (highlights) all the rows in a data table.

```plaintext
dx=5;
n=NRow(dt); For(i=n, i>n-x, i--,
    dt << Delete Rows(i));
```

Here is a general way to remove the bottom $x$ rows of a data table of any size. **NRow** (see “For Each Row,” p. 101) counts the rows in the table.

```plaintext
x=5;
n=NRow(dt); For(i=n, i>n-x, i--,
    dt << Delete Rows(i));
```
Data Tables
Messages for Data Tables

\[
\text{dt} \ll \text{Select All Rows};
\]

If all rows are selected, you can deselect them all by using \text{Invert Row Selection}. This command reverses the selection state for each row, so that any selected rows are deselected, and any deselected rows are selected.

\[
\text{dt} \ll \text{Invert Row Selection};
\]

With the exception of \text{Invert Row Selection}, whose result depends on the current selection, any new selection message starts over with a new selection, so if you have certain rows selected and you then send a new message to select rows, an implied "deselect selected rows" happens first.

To select a certain row, use \text{Go To Row}:

\[
\text{dt} \ll \text{Go To Row}(9);
\]

You can select rows that are currently excluded, hidden, or labeled:

\[
\begin{align*}
\text{dt} & \ll \text{Select Excluded;} \\
\text{dt} & \ll \text{Select Hidden;} \\
\text{dt} & \ll \text{Select Labeled;}
\end{align*}
\]

To select rows that are not excluded, hidden, or labeled, stack a select message and an invert selection message together in the same statement, or send the two messages sequentially:

\[
\begin{align*}
\text{dt} & \ll \text{Select Hidden} \ll \text{Invert Row Selection}; \\
\text{dt} & \ll \text{Select Hidden;} \\
\text{dt} & \ll \text{Invert Row Selection;}
\end{align*}
\]

To select rows according to data values, use \text{Select Where}, specifying a logical test inside the parentheses. You can use all the usual functions and operators from the formula calculator, which are summarized in "JSL Syntax Reference" chapter. For example, using \text{Big Class.jmp}, you might use:

\[
\text{dt} \ll \text{Select Where(age}>13); 
\]

To select specific rows in a data table based on their row number, use the \text{Select Rows} command. The parameter to the command is a list of row numbers. For example, to select rows 1, 3, 5, and 7 of a data table,

\[
\text{dt} \ll \text{Select Rows}([1, 3, 5, 7])
\]

To obtain a random selection:

\[
\begin{align*}
\text{dt} & \ll \text{Select Randomly(number)} \\
\text{dt} & \ll \text{Select Randomly(probability)}
\end{align*}
\]

These commands use a conditional probability to obtain the exact count requested.

The row menu command \text{Select Matching Cells} is also implemented in JSL.

\[
\text{Current Data Table()} \ll \text{Select Matching Cells;}
\]

selects matching cells in the current data table.

\[
\text{Current Data Table()} \ll \text{Select All Matching Cells}
\]

selects matching cells in all open data tables.
For more complicated selections, or to store selections permanently as row state data, see the discussion of “Row State Operators,” p. 133.

**Moving Rows**

These commands move the currently selected rows to the indicated destination point.

\[
\begin{align*}
dt<<Move\ Rows(AtStart); \\
dt<<Move\ Rows(AtEnd); \\
dt<<Move\ Rows(After(rowNumber));
\end{align*}
\]

**Finding Rows**

These two commands return a matrix of row numbers:

\[
\begin{align*}
result &= dt<<Get\ Rows\ Where(condition)\; //\ for\ example,\ <<Get\ Rows \\
&\quad \text{Where}(:age>=16) \\
result &= dt<<Get\ Selected\ Rows;
\end{align*}
\]

**Assigning Colors and Markers to Rows**

You can use the Colors and Markers messages to assign (or change) colors and markers used for rows. These settings mostly affect graphs produced from the data table. Both messages expect numeric arguments to pick which color or marker to use; how numbers correspond to colors and markers is summarized in “Colors and Markers,” p. 143.

\[
\begin{align*}
dt<<Colors(3); &\; //\ set\ selected\ rows\ to\ red \\
dt<<Markers(2); &\; //\ pick\ the\ X\ marker\ for\ selected\ rows
\end{align*}
\]

As with other row messages, you can stack selection and other messages together:

\[
\begin{align*}
dt<<Select\ Where(age==13) &\; //\ select\ the\ youngest\ subjects \\
&\quad <<colors(8)<<markers(8); &\; //\ and\ use\ purple\ open\ circles\ for\ them
\end{align*}
\]

Color By Column sets colors according to the values of a column that you specify, and Marker by Column is analogous:

\[
\begin{align*}
dt<<Color\ By\ Column(:age); \\
dt<<Marker\ By\ Column(:age);
\end{align*}
\]

Additional, named arguments are as follows:

- **Continuous Scale** *(Color by Column only)* Assigns colors in a chromatic sequential fashion based on the values in the highlighted column.
- **Reverse Scale** *(Color by Column only)* Reverses the color scheme in use.
- **Make Window with Legend** *(Both Color by Column and Marker by Column)* Creates a separate window with a legend.
- **Excluded Rows** *(Both Color by Column and Marker by Column)* Applies the row states to excluded columns.
Coloring Cells

You can color individual cells in the data grid. For example, this line uses the row state color to color the cells:

```
> dt<<Color Rows by Row State;
```

You can also specify a color theme for either categorical or continuous columns:

```
> :height << Set Property( "Color Gradient", {"White to Blue", Range( 40, 80 )} );
> :height << Color Cell By Value( 1 );

> :age << set property( "Value Colors", 
    {12 = red, 13=yellow, 14=green, 15=blue, 16=magenta, 17=gray} );
> :age << Color Cell By Value( 1 );

> :age << Color Cell By Value( 0 ); // turns off the cell coloring
```

You can also color specific cells:

```
// set rows 1, 5, 8 of the column "name" to red
> :name << color cells(red, {1, 5, 8});
```

To remove the color from specific cells, set the color to black:

```
// remove the color for row 1 of the column "name"
> :name << color cells(black, {1});
```

Toggling Rows Between States

For example, to hide all rows in `Big Class` where age is greater than 13, you could do the following.

```
> dt<<Select Where(age>13);
> dt<<Hide;
```

Since messages to the same object can be stacked together in a single statement, you could equivalently do this:

```
> dt<<Select Where(age>13)<<hide;
```

Similar messages let you exclude or unexclude, label or unlabel:

```
> dt<<Exclude;
> dt<<Label;
```

These messages are all toggles; send the message again to unhide, unexclude, and so on.

Locating Selected Rows

Next Selected and Previous Selected scroll the data table window up or down so that the next selected row that is not already in view moves into view. The table “wraps,” so Next Selected jumps from the bottommost selected row to the topmost and vice versa for Previous Selected.

```
> dt<<Next Selected;
```
Clear Selection

To cancel a selection, leaving no rows selected, use Clear Select:

```
dt<<Clear Select;
```

Working with Row States

Row states are a special type of data. You can work with row states both through the messages to data tables discussed above and through row state operators, which are discussed at length in the section “Row State Operators,” p. 133. Please refer to that discussion for details about the row states mentioned above.

Data Filter

Data Filter provides a variety of ways to identify subsets of data. Using Data Filter commands and options, you can select complex subsets of data, hide these subsets in plots, or exclude them from analyses.

The basic syntax is:

```
dt<<Data Filter( <mode( ... )>, <Add Filter ( ... )>, ... );
```

Options for Data Filter include:

- **Show Columns Selector**, Animation, Animate, Clear, Delete All, Delete, columns, Filter Columns, Filter Column, Filter Group, Add Filter, Add, Match, Mode, Display, Where, Make Subset, display

You can send an empty Data Filter message to a data table, and the initial Data Filter window appears, showing the Add Filter Columns panel that lists all the variables in the data table.

**Mode** takes three arguments, all optional:

- Select(bool), Show(bool), Include(bool)

They either turn on or turn off the corresponding options. The default value for Select is true (1). The default value for Show and Include is false (0).

**Add Filter** adds rows and builds the WHERE clauses that describe a subset of the data table. The basic syntax is:

```
Add Filter( columns( col, ... ), Where( ... ), ... )
```

To add columns to the data filter, list the columns names separated by commas. Note that this is not a list data structure.

You can define one or more WHERE clauses. For example:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

def = dt << Data Filter( 
    Mode( Show( 1 ) ),
    Add Filter( 
        columns( :age, :sex, :height ),
    Where( 
        :sex == "Male", 
    )
    )
```

This example filters the data table to include only male individuals. You can add more WHERE clauses to define more specific subsets.
Data Tables

Chapter 4

Messages for Data Tables

Where( :age == {13, 14, 15} ),
Where( :sex == "M" ),
Where( :height >= 50 & :height <= 65 )
);

You can also send messages to an existing Data Filter object:
Clear(), Display(...), Animate(), Mode(), ...

Clear takes no arguments and clears the data filter.

Commands from the Tables Menu

The messages in this section provide different ways of rearranging data tables and making new tables. If no arguments are supplied to these messages, they produce the usual window for interactive use. If these operations are unfamiliar, see the Using JMP book for more details and examples.

Column arguments also allow list arguments, so the following script fragments are all valid:

Stack(height,weight)
Stack({height,weight}) // equivalent

Most of these commands also generate small scripts that record how the resulting tables were made. The script is stored in the new table as a table property with the name “Source.” Exceptions are Summary when performed by script or Subset when performed by script or the window. Most people do not write data table scripts from scratch, but instead use the Tables menu items and then copy the Source property from the resulting table.

Any options in these windows are also scriptable. The simplest way to discover the syntax for any given option is to perform the table operation interactively using the option, and look at the Source script in the resulting new table.

Summary

Summary creates a new table of summary statistics according to the grouping columns that you specify.

summDt = dt<<Summary(
   Group(groupingColumns),
   Subgroup(subGroupColumn),
   Statistic(columns),//where statistic is Mean, Min, Max, Std Dev, and so on.
   Output Table Name(newName));

For example:

summDt = dt<<Summary(
   Group(Age),
   Mean(Height,Weight), Max(Height), Min(Weight),
   output table name("Height-Weight Chart"));

Output Table name can take a quoted string or a variable that contains a string.
Note: Do not confuse Summary, which creates a data table with summary statistics, with Summarize (p. 102), which stores summary statistics in global variables for later use in custom result displays, and so on.

By default, a summary table is linked to the original data table. If you want to produce a summary that is not linked to the original data table, add this option to your Summary message:

```plaintext
dt << Summary( Group( :age ), Mean( :height ),
                 Link to original data table( 0 )
);
```

Subset

Subset creates a new data table from rows that you specify. If you specify no rows, Subset uses the selected rows. If no rows are selected or specified, it uses all rows. If no columns are specified, it uses all columns.

```plaintext
dt << Subset( Columns(columns),
              Rows(row matrix),
              Linked,
              Output Table Name("name"),
              Copy Formula (1 or 0),
              Sampling rate (n),
              Suppress Formula Evaluation(1 or 0));
```

For example:

```plaintext
//To select all columns for all rows where age is 12:
for each row(Selected(Rowstate())=(age==12));
subdt = dt<<Subset(output table name("subset"));

//To select three columns and all rows:
subDt1 = dt << Subset (Columns(Name,Age,Height), Output Table Name("Big Class NAH"));

//To select specified rows of two columns, linking:
subDt2 = dt << Subset (Columns(Name,Weight),Rows([2,4,6,8]),Linked);
```

Sort

Sort rearranges the rows of a table according to the values of one or more columns, either replacing the current table or creating a new table with the results. Specify ascending or descending sort for each column listed for By.

```plaintext
dt<<Sort(
     By(columns), Order(Descending | Ascending, ...),
     Replace Table | Output table name("name"))
```

For example:

```plaintext
sortedDt = dt<<Sort(
     By(Age,Name),
     Order(Descending, Ascending),
);
Stack (Unsplit)

Stack combines values from several columns into several rows in one column.

```plaintext
dt << Stack(
    Stack(columns),  // the columns to stack together
    ID(columns),  // to identify source columns
    Stacked("name"),  // name for the new stacked column
    Output Table Name("name"),  // name for the new data table
    Columns(columns));  // specify which columns to include in the stacked table
```

For example, where `dt` is a reference to Big Class:

```plaintext
stackedDt = dt << Stack(
    Stack(Height, Weight),
    ID("ID"),
    Stacked("Y"),
    Output table name("Stacked"));
```

The `Columns(columns)` argument can take a list of columns, or an expression that evaluates to a list.

Split (Unstack)

Split breaks a column into several columns, mapping several rows in the first into single rows in other columns.

```plaintext
dt << Split(
    Split(columns),  // the column to split
    Group(columns),  // (optional) column to identify rows uniquely
    ColID(column),  // the grouping variable on which to split
    Remaining Columns( Keep All | Drop All | Drop(columns) | Keep(columns)),
    Output Table Name("name"));
```

The optional Remaining Columns argument specifies which of the other columns from the source data table (those not specified for Split, Group, or ColID) to include in the new data table. The default is to Keep All, or you can explicitly list columns to Keep or Drop.

This example reverses the previous example for Stack, returning essentially the original table, except that the height and weight columns now appear in alphabetic order:

```plaintext
splitDt = stackedDt << Split(
    split(y),
    ColID(ID),
    output table name("Split"));
```

Transpose

Transpose creates a new data table by flipping a data table on its side, interchanging rows for columns and columns for rows. If you specify no rows, Transpose uses the selected rows. If no rows are selected, it uses all rows.
Chapter 4

Data Tables

Messages for Data Tables

```
dt << Transpose(
   Columns(columns),
   Rows(row matrix),
   Output Table Name("name"));
```

For example:
```
tranDt = dt<<Transpose(Columns(Height,Weight),
   output table name("Transposed Columns"));
```

**Note:** In earlier versions of JMP, Transpose did not have a window. Starting with JMP 5.1, a window appears when the transpose command is invoked. The simple transpose command `dt<<Transpose` shows this window. If you are doing a simple transpose (of all selected columns and no By column) and you do not want the window to appear, invoke the transpose as `dt<<Transpose(no option)`.

**Concatenate (Vertical Join)**

Concatenate combines rows of several data tables top to bottom.

```
dt << Concatenate(DataTableReferences,...,Keep Formulas,
   Output Table Name("name"));
```

For example, if you make subsets for males and females, you can stick them back together with Concatenate:
```
dt<<Select Where(sex=="M"); m=dt<<Subset(output table name("M"));
dt<<Invert Row Selection; f=dt<<Subset(output table name("F"));
both=m<<Concatenate(f,output table name("Both"));
```

Instead of creating a new table containing all the concatenated data, you can append all the data to the current data table:
```
dt << Concatenate(DataTableReferences, Append to first table);
```

**Join (Horizontal Concatenate)**

Join combines data tables side to side.

```
dt<<Join(  // message to first table
   With(dataTable), // the other data table
   Select(columns), // optional column selection
   SelectWith(columns), // optional column selection
   By Row Number, // join type; alternatives are Cartesian Join or
   // By Matching Columns(col1==col2,col),
   Update first table with data from second table,
   Merge same name columns,
   Copy formula(0), // on by default; 0 turns it off
   Suppress formula evaluation(0), // on by default; 0 turns it off
   // options for each table:
   Drop Multiples(Boolean, Boolean),
   Include NonMatches(Boolean, Boolean),
   Output Table Name("name")); // the resulting table
```

To try this, first break Big Class into two parts.

```plaintext
part1 = dt << Subset(Columns(Name, Age, Height), Output Table Name("NAH_Big Class"));
part2 = dt << Subset(Columns(Name, Sex, Weight), Output Table Name("NSW_Big Class"));
```

To make it a realistic experiment, rearrange the rows in part 2.

```plaintext
sortedPart2 = part2 << sort(by(name), Output Table Name("SortedNSW_Big Class"));
```

Now you have a data set in two separate chunks, and the rows are not even in the same order, but you can join them together by matching on the column the two chunks have in common.

```plaintext
joinDt = part1 << Join(
    With(sortedPart2),
    By Matching Columns(name == name),
    output table name("Joined Parts"));
```

The resulting table has two copies of the name variable, one from each part, and you can inspect these to see how Join worked. Notice that you now have four Robert rows, because each part had two Robert rows (there were two Roberts in the original table) and Join formed all possible combinations.

**Update**

Update replaces data in one table with data from a second table.

```plaintext
dt << Update( // message to first table
    With(dataTable), // the other data table
    By Row Number, // default join type; alternative is
    //      By Matching Columns(col1 == col2)
    Ignore Missing, // optional, does not replace values with missing values
);
```

To try this, make a subset of Big Class.

```plaintext
NewHt = dt << Subset(Columns(Name, Height), Output Table Name("hts"));
```

Next, add 0-6 inches to each student’s height.

```plaintext
diff = random uniform(6,0);
for each row(height += diff);
```

Finally, update the heights of students in Big Class with the new heights from the subset table.

```plaintext
dt << Update(
    With(NewHt),
    By Matching Columns(name == name),
);
```

**Controlling the Columns Added to an Updated Table**

Your second table might contain more columns than your original table. You can select which columns are included in your updated table using the option Add Column from Update Table().

To add no additional columns:
Manipulating columns

This section discusses ways to set, create, and modify columns. Recall that to add and delete columns, you need to send a message to the data table; see “Messages for Data Tables,” p. 94. To get or set data column values, you specify the current data table, current row, and column, as detailed in “Accessing Values,” p. 98.

Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update Table( None )
);

To add some columns:

Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update table( :col1, :col2, :col3 )
);

Merge Update

Merge Update is an alias for Update.

Tabulate

Tabulate constructs tables of descriptive statistics. The tables are built from grouping columns, analysis columns, and statistics keywords. This example tabulates Big Class.jmp.

dt<<Tabulate( // message to data table
   Add Table( // start a new table
      Column Table( Grouping Columns( :sex ) ),
      // group using the column sex
      Row Table( // add rows to the table
         Analysis Columns( :height, :weight ),
         // use the height and weight columns for the analysis
         Statistics( Std Dev, Mean ) // show the standard deviation and mean
   ));

Missing Data Pattern

If your data table contains missing data, you might want to see whether there is a pattern that the missing data creates.

dt<<Missing Data Pattern( // message to data table
   columns( :miss ), // find missing data in this column
   Output Table( "Missing Data Pattern" ) // name the output table
);
Obtaining Column Names

Column Name(n) returns the name of the n-th column.

```
column name(2); // returns age
```

The returned value is a name value, not a quoted string. What this means is you can use it anywhere you would normally use the actual name in a script. For example, you could subscript it:

```
column name(2)[1];
```

If you want the name as a text string, just quote it with `Char`:

```
char(column name(2));
"age"
```

To retrieve a list of the names of all columns in a data table, submit `Get Column Names`.

```
dt<<Get Column Names(argument)
```

where the optional `argument` controls the output of the `Get Column Names` function as follows:

- Specify `Numeric`, `Character`, or `RowState` to include only those column data types.
- Specify `Continuous`, `Ordinal`, or `Nominal` to include only those modeling types.
- Specify `String` to return a list of strings rather than column names.

For example,

```
names=dt<<Get Column Names (Numeric, Continuous)
```

returns

```
{height, weight}
```

Also,

```
names=dt<<Get Column Names(Numeric, Continuous, String)
```

returns

```
{"height", "weight"}
```

Select Columns

To select a column, use the `Set Selected` message.

```
col<<Set Selected (1);
```

For example, to select all Continuous variables in the Big Class table, use the following script:

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
cc=dt<<Get Column Names(Continuous);
ncols = N Items(cc);
for(i=1,i<=ncols,i++,
```
Manipulating columns

cc[i]<<Set Selected(1);
);

Get Selected Columns

To get a list of currently selected columns, use the Get Selected Columns message.

dt << Get Selected Columns

Moving Columns

These commands move the currently selected columns to the indicated destination point.

dt<<Move Selected Columns(To First)
dt<<Move Selected Columns(To Last)
dt<<Move Selected Columns(After("name"));

You can also move columns without first selecting them in the data table by using the following syntax.

dt<<Move Selected Columns({"name"}, To First)
dt<<Move Selected Columns({"name"}, To Last)
dt<<Move Selected Columns({"name"}, After("name"));

Data Column Objects

Just as you send data table messages to a data table reference, you send column messages to a reference to a data column object. The Column function returns a data column reference. Its argument is either a name in quotation marks (or something that evaluates to a name in quotation marks) or a number.

Column("age"); // a reference to the age column
col = Column(2); // assigns a reference to the second column

This manual uses col to represent data column references. To see the messages that you can send to data column objects, use Show Properties:

show properties(col);

Accessing Cell Values Through Column References

Always use a subscript on a column reference to access the values in the cells of columns. It is common to use an empty subscript to refer to the current row.

x = col[irow]  // specific row
x = col[]      // current row
col[irow] = 2; // as an l-value for assignment
currentDataTable()<<Select Rows Where(col[]<14); // in a WHERE clause

Sending Messages to Columns

You saw above how to store a data column reference in a global variable, such as col. The rest of the ways to manipulate columns are all messages that should be sent to a data column reference.
Sending messages to columns is analogous to sending messages to data tables. Either state the object, a double-angle operator <<, and then the message with its arguments in parentheses; or use the Send() function with the object and then the message. In some cases the messages themselves need no arguments, so the trailing parentheses are optional.

\[
\text{col} \ll \text{message(arg, arg2, ...)}; \\
\text{Send(col, message(arg, arg2, ...))};
\]

As with data tables and other types of objects, you can stack or list messages.

\[
\text{col} \ll \text{message} \ll \text{message2} \ll ... \\
\text{col} \ll \{\text{message, message2, }...\};
\]

Note that to delete a column, you need to send a message to a data table reference, because objects cannot delete themselves, only their containers can.

**Confusion alert!** JMP can now display math symbols and Greek letters (controlled in Preferences in the Font category). This means that if you save a column (such as T square limits), the column name could either be “T Square Limits” (no special characters) or “\(T^2\) Limits” (with special characters). Any reference using the column name must match the name exactly, or it fails.

### Grouping Columns

You can group columns by sending the data table the Group Columns message, which takes a list of columns to group as an argument. For example, the following code opens the Big Class data table and groups the age and sex columns.

\[
\text{dt} = \text{Open("SAMPLE_DATA\Big Class.jmp");} \\
\text{dt} \ll \text{Group Columns( \{:age, :sex\} );}
\]

You can also send a column name followed by the number of columns to include in the group. The group includes the first column named and the \(n-1\) columns that follow it. This line is equivalent to the line above:

\[
\text{dt} \ll \text{Group Columns( :age, 2 );}
\]

If you need to group discontiguous columns, use the list argument.

You can ungroup grouped columns by sending the data table the Ungroup Columns message, which takes a list of columns in a group as an argument. For example, the following line ungroups the two columns grouped in the previous code sample.

\[
\text{dt} \ll \text{Ungroup Columns( \{:age, :sex\} );}
\]

### Notes

Both messages take a single list as an argument. The list must be enclosed in curly braces. You cannot create more than one group in a single message (for example, by giving the Group Columns message two lists of columns). Instead, you must send the data table two separate Group Columns messages.
The **Ungroup Columns** message takes a list of columns to ungroup, not the name of a group of columns. You can remove a partial list of columns from a group. For example, this line creates a group of four columns:

```plaintext
dt << Group Columns( {:age, :sex, :height, :weight} );
```

And this line removes two of the columns, while leaving the other two in the group:

```plaintext
dt << Ungroup Columns( {:age, :sex} );
```

Notice that the grouped columns are now **height** and **weight**, but the group name still contains **age**. Once a group is created, its name does not change, even if you remove the first column that was originally grouped.

### Setting and Getting Attributes

A collection of message pairs for data table columns let you control all the various attributes or characteristics of a column, including its name, its data, and its metadata. The messages come in pairs, one to “set” or assign each attribute and one to “get” or query the current setting of each attribute.

**Hide**, **exclude**, **label**, and **scroll lock** for a column can be activated and deactivated through scripting. Submit a 1 to turn the column attribute on, and a zero to turn it off. For example, the following four lines of code scroll lock a column called **Name**, add it to the list of label columns, unexclude it, and unhide it.

```plaintext
column("name") << scroll lock(1);
column("name") << label(1);
column("name") << exclude(0);
column("name") << hide(0);
```

To clear a column selection, submit a **Clear Column Selection** message to the data table.

```plaintext
dt<<Clear Column Selection;
```

### Names

**Set Name** lets you name or rename a column, and **Get Name** returns the name for a column.

```plaintext
col<<Set Name("ratio");
col<<Get Name;
```
Values

Similarly, Set Values sets values for a column. If the variable is character, the argument should be a list; if numeric, a matrix (vector). If the number of values is greater than the current number of rows, the necessary rows are added. Get Values returns the values in list or matrix form, and Get As Matrix is a synonym.

```julia
col<<Set Values(myMatrix); // for a numeric variable
col<<Set Values(myList);   // for a character variable
col<<Get As Matrix;       // returns a matrix, or list if character
```

For example:

```julia
dt=Open("$SAMPLE_DATA/Big Class.jmp");
column("name")<<values({Fred, Wilma, Fred, Ethel, Fred, Lamont});
myList = :name<<get values;    //returns list
myList = :name<<get as matrix; //equivalent

column("age")<<values([28,27,51,48,60,30]);
myVector = :age<<get values;    //returns one-column matrix
myVector = :age<<get as matrix; //equivalent
```

Value Labels

Complete details about value labels are found in the Using JMP book. Essentially, they provide a method of displaying a descriptive label for abbreviated data. For example, you might have a data column holding values that are 0 or 1, where 0 represents a male and 1 represents a female. Value labels enable you to show the descriptive label (male and female) instead of the numeric label (0 and 1).

You can specify value labels in any of three ways. In each of the following examples, assume M maps to Male, and F maps to Female.

Using two lists

```julia`:sex << Value Labels({"F", "M"}, {"Female", "Male"});`

Using a list of pairs

```julia`:sex << Value Labels({"F", "Female", "M", "Male"});`

Using a list of assignments

```julia`:sex << Value Labels({"F" = "Female", "M" = "Male"})`

In any case, activate value labels by sending Use Value Labels as a message to the column.

```julia`:sex << Use Value Labels(1);`

To revert back to showing the column's actual values,

```julia`:sex << Use Value Labels(0);`

The same message can be used for the data table to turn value labels on and off for all columns.

```julia`:Current Data Table()<<Use Value Labels (1)`
Data and Modeling Types

For example, you can set or get the data type of a column from JSL. Choices are Character, Numeric, and Row State.

```js
dt<<new column("new"); column("new")<<data type(character);
column("new")<<get data type;
```

You can set or get the modeling type of a column.

```js
col<<Set Modeling Type("Continuous");
col<<Modeling Type("Ordinal");
col<<Get Modeling Type; // returns "Continuous", "Nominal", or "Ordinal"
```

You can specify the format of a column when changing its data type. For example,

```js
column ("date") << data type(numeric, format("ddMonYYYY"));
```

Display Formats

The Format message controls numeric and date/time formatting. The first argument is a quoted string from the list of format choices shown in the Column Info window. Subsequent parameters depend on the format choice. You can also set the field width by itself. Examples:

```js
col<<Format("best",5); // width is 5
col<<Format("Fixed Dec",9,3); // width is 9, with 3 decimal places
col<<Format("PValue",6);
col<<Format("d/m/y",10);
col<<set fieldwidth(30);
```

For date formats, the Format message sets the way dates are displayed in a data table column. To set the format that you use for entering data or for displaying the current cell when you have it selected for entry or editing, use the Input Format message. For example:

```js
col<<Format("d/m/y",10); // display the date in day-month-year order
col<<Input Format("m/d/y"); // enter the date in month-day-year order
```

The date/time format choices are further discussed under “Datetime Notation,” p. 48 in the “JSL Building Blocks” chapter.

Confusion alert! Do not confuse the Format message for columns with the Format operator for converting numeric values to strings according to the format specified (typically used for date/time notation as described in “Datetime Operators,” p. 46 in the “JSL Building Blocks” chapter). Sending a message to an object has a very different effect from using a function that might happen to have the same name.

To get the current format of a column, submit a Get Format message.

```js
col<<Get Format;
```
Roles

You can Preselect Roles. Choices are No Role, X, Y, Weight, and Freq. Get Role returns the current setting.

```julia
col<<Preselect Role(X);
col<<Get Role;
```

Formulas

Similarly, you can set, get, and evaluate a formula for a column:

```julia
col=New Column("Ratio"); // creates column and stores its reference
col<<Set Formula(:height/:weight); // sets formula
col<<EvalFormula; // evaluates the formula
col<<Get Formula; // returns the expression :height/ :weight
```

Confusion alert! Be sure to add commands to evaluate the formula in order to use the values from these columns in scripts. Formula evaluation timing can differ between versions of the JMP software. Read the following paragraphs carefully.

When formulas are added, they are scheduled to be evaluated in a background task. This is a problem for scripts if they depend on the column having the values while the script is running.

To force a single column to evaluate, you can send an Eval Formula command to the column. You can even do this inside the command to create the column, right after the Formula clause, for example:

```julia
dt<<NewColumn("Ratio",Numeric,Formula(:height/:weight),EvalFormula);
```

where Formula is an alias for Set Formula.

Actually, it is best to wait until you are through adding a set of formulas, and then use the command Run Formulas to have all the formulas evaluated in their proper order:

```julia
current Data Table()<<Run Formulas;
```

The Run Formulas command is preferred to Eval Formula, since Eval Formula, while it evaluates the formulas, does not suppress the background task from evaluating them again. The formula dependency system background task takes great care to evaluate the formulas in the right order, and RunFormulas just calls this task until all the formulas are finished evaluating.

If you use random numbers and use the Random Reset(seed) feature to make a replicable sequence, then you have another reason to use Run Formulas to avoid a second evaluation in the background.

Range and List Checks

List and range check properties can be manipulated with JSL. The following examples set and clear the list check property in the column Sex.

```julia
column("Sex") << list check({"M", "F"}); // sets it
column("Sex") << list check(); // clears it
```

Range checks require the specification of a range using the following syntax.
Chapter 4

Data Tables

Manipulating columns

So, for example, to specify that the column \textit{age} must be in the range $0 < \text{age} < 120$, use

\begin{verbatim}
  column("age") << range check(LTLT(0, 120));
\end{verbatim}

Note that any of the LXLX operators can be preceded by \texttt{Not}, and that at most one of them can be missing.

So, to specify that the \textit{age} column should be $\geq 12$, use

\begin{verbatim}
  column("age") << range check(not(lt(12)));
\end{verbatim}

Submit an empty \texttt{range check()} to clear a range check state.

\begin{verbatim}
  column("age") << range check();
\end{verbatim}

To retrieve the list or range check assigned to a column, send a \texttt{Get List Check} or \texttt{Get Range Check} message to the column.

\begin{verbatim}
  column("sex") << get list check;
  column("age") << get range check;
\end{verbatim}

For the $0 < \text{age} < 120$ example with a range check above, JMP returns

\begin{verbatim}
  range check(LLT(0, 120))
\end{verbatim}

Note that you can also use \texttt{Set Property}, \texttt{Get Property}, and \texttt{Delete Property} to set, retrieve, and remove List Checks and Range Checks. See “Setting, Retrieving, and Removing Column Properties,” p. 129 for more information.

Operations sent through JSL that involved range check columns show any warnings in the log rather than in interactive windows.

\section*{Setting, Retrieving, and Removing Column Properties}

Data columns have numerous optional metadata attributes that can be set, queried, or cleared using the messages \texttt{Get Property}, \texttt{Set Property}, and \texttt{Delete Property}. The name of the property in question is always the first argument for \texttt{Set Property}, and what is expected for subsequent arguments depends on which property you set. \texttt{Get Property} and \texttt{Delete Property} always take a single argument, which is the name of the property. \texttt{Get Property} returns the property’s settings, while \texttt{Delete Property} completely removes the property from the column.

\begin{verbatim}
  col<<Set Property( "propertyName", {argument list} );
  col<<Get Property( "propertyName" );
  col<<Delete Property( "propertyName" );
\end{verbatim}

\begin{table}[h]
\centering
\caption{Range Check Syntax}
\begin{tabular}{|c|c|}
\hline
To specify this range & Use this function \\
\hline
$a \leq x \leq b$ & LELE(a, b) \\
$a \leq x < b$ & LELT(a, b) \\
$a < x \leq b$ & LTLE(a, b) \\
$a < x < b$ & LTLT(a, b) \\
\hline
\end{tabular}
\end{table}
If you want to set several properties, you need to send several separate \texttt{Set Property} messages. You can stack several message-sends in a single JSL statement if you want:

\begin{verbatim}
  col<<Set Property("Axis",{Min(50), Max(180)})<<Set Property("Notes", "to get proportions");
\end{verbatim}

To get a property's value, send a \texttt{Get Property} message whose argument is the name of the property that you want:

\begin{verbatim}
  column("ratio")<<Get Property("axis"); // returns axis settings
\end{verbatim}

Choices for properties are the same as in the Column Properties menu in a Column Info window, and the arguments for each correspond to the settings seen in the graphical user interface. An easy way to learn the syntax is to establish the property that you want in the Column Info window first, then use \texttt{Get Property} to view the JSL.

\begin{verbatim}
  dt << Set Label Columns(col1, col2, col3);
\end{verbatim}

sets col1, col2, and col3 as label columns.

\begin{verbatim}
  dt << Set Label Columns();
\end{verbatim}

clears all the label columns.

The same syntax works for \texttt{Set Scroll Lock Columns}, and \texttt{Scroll Lock}.

\textbf{Table 4.2} Properties for Data Table Columns

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Arguments Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes</td>
<td>Stores notes about a column.</td>
<td>Quoted text string, for example:</td>
</tr>
</tbody>
</table>
|                    |                                       | \begin{verbatim}
  col<<Set Property("Notes", "Extracted from Fisher iris data");
  col<<getproperty("notes");
\end{verbatim}                                                                 |
| List Check         | Prescribes the possible values that can be entered in a column.                  | List of a parameter and its values, for example:                                 |
| Range Check        |                                       | \begin{verbatim}
  col<<Set Property(List Check, ["F","M"]);
  col<<Set Property(Range Check, LTLT(0, 120));
  col<<Get Property(List Check);
  col<<Delete Property(Range Check);
\end{verbatim}                                                                 |
|                    |                                       | Set the range check warning to appear in the log only:                            |
|                    |                                       | \begin{verbatim}
  col<<Set Property("Range Check", LELE( 80, 140 ), No Warn);
\end{verbatim}                                                                 |
| Missing Value Codes| Specifies column values that should be treated as missing.                       | List of values:                                                                  |
|                    |                                       | \begin{verbatim}
  col<<Set Property("Missing Value Codes", [0, 1])
\end{verbatim}                                                                 |
Table 4.2 Properties for Data Table Columns (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Arguments Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Labels</td>
<td>Specifies labels to be displayed in place of the values.</td>
<td>A list of value-label pairs: col&lt;&lt;Value Labels( {0 = &quot;Male&quot;, 1 = &quot;Female&quot;} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Once the values labels are set, you can turn them on or off:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use Value Labels( 1 )</td>
</tr>
<tr>
<td>Value Ordering</td>
<td>Specifies the order in which you want the data to appear in reports.</td>
<td>Specify the value order: col&lt;&lt;Set Property(&quot;Value Ordering&quot;, {&quot;Spring&quot;, &quot;Summer&quot;, &quot;Fall&quot;, &quot;Winter&quot;})</td>
</tr>
<tr>
<td>Value Colors and Color Gradient</td>
<td>Specifies colors for either categorical or continuous data, respectively.</td>
<td>Specify the colors for Value Colors: col&lt;&lt;Set Property(&quot;Value Colors&quot;, {&quot;Female&quot; = 3, &quot;Male&quot; = 5})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specify the gradient and the range and midpoint for Value Gradient:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Color Gradient&quot;, {&quot;White to Blue&quot;, Range( {18, 60, 25} )})</td>
</tr>
<tr>
<td>Axis</td>
<td>Most platforms use this (if it exists) when constructing axes.</td>
<td>List of parameters and their values, mostly Boolean, for example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Axis&quot;,{Min(50), Max(180), Inc(0), Minor Ticks(10), Show Major Ticks(1), Show Minor Ticks(1), Show Major Grid(0), Show Labels(1), Scale(Linear)});</td>
</tr>
<tr>
<td>Coding</td>
<td>Used for DOE and fitting.</td>
<td>List with low and high values, for example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Coding&quot;,{59,172}); col&lt;&lt;get property(&quot;coding&quot;);</td>
</tr>
<tr>
<td>Mixture</td>
<td>Used for DOE, fitting, and profiling.</td>
<td>Boolean:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Mixture&quot;,1); //on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Mixture&quot;,0); //off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Get Property(&quot;Mixture&quot;); //on</td>
</tr>
<tr>
<td>Row Order Levels</td>
<td>Specify to sort levels by their occurrence in the data instead of by value.</td>
<td>Boolean:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property( &quot;Row Order Levels&quot;, 1 )</td>
</tr>
</tbody>
</table>
### Table 4.2 Properties for Data Table Columns (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Arguments Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec Limits</td>
<td>Used for capability analysis and variability charts.</td>
<td>List of parameters and their values, for example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Spec Limits&quot;, {LSL(-1), USL(1), Target(0)});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Get Property(&quot;spec limits&quot;);</td>
</tr>
<tr>
<td>Control Limits</td>
<td>Used for control charts.</td>
<td>List of parameters and their values, for example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Control Limits&quot;, {XBar(Avg(44), LCL(29), UCL(69))});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;get property(&quot;control limits&quot;);</td>
</tr>
<tr>
<td>Response Limits</td>
<td>Set in DOE and used in desirability profiling.</td>
<td>List of parameters and their values. Choices for Goal are Maximize, Match Target, Minimize, None. Other parameters take numeric value and desirability arguments, for example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Response Limits&quot;, {Goal(Match Target), Lower(1,1), Middle(2,2), Upper(3,3)});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Get Property(&quot;response limits&quot;);</td>
</tr>
<tr>
<td>Design Role</td>
<td>Used for DOE.</td>
<td>Specify a single role for DOE (Design of Experiments). Choices for role are Continuous, Categorical, Blocking, Covariate, Mixture, Constant, Signal, Noise. Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Design Role&quot;,Covariate);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;get property(&quot;design role&quot;);</td>
</tr>
<tr>
<td>Factor Changes</td>
<td>Sets the difficulty of changing a factor.</td>
<td>Set Factor Changes to Easy, Hard, or Very Hard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Factor Changes&quot;, Hard)</td>
</tr>
<tr>
<td>Sigma</td>
<td>Used for control charts.</td>
<td>Specify known sigma value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Sigma&quot;,1.332);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;get property(&quot;sigma&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Each type of chart uses a different sigma calculation. Refer to Quality and Reliability Methods for details.</td>
</tr>
<tr>
<td>Units</td>
<td>Provided for custom uses.</td>
<td>Specify the units of measure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;set property(&quot;units&quot;, grams);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;get property(&quot;units&quot;);</td>
</tr>
<tr>
<td>Distribution</td>
<td>Set the distribution to fit to the column.</td>
<td>Specify the distribution type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>col&lt;&lt;Set Property(&quot;Distribution&quot;, Distribution(GLog))</td>
</tr>
</tbody>
</table>
### Lock

To lock or unlock a column, use `lock` or `set lock` with a Boolean argument. `Get Lock` returns the current setting.

```plaintext
col<<lock(1);     // lock
col<<set lock(0); // unlock
col<<get lock;    // show current state
```

### Scripts

Get Script returns a script to create the column:

```plaintext
New Column("Ratio", SetFormula( :height/ :weight));
column("ratio")<<get script;
    New Column("Ratio", Numeric, Continuous, Formula( :height / :weight))
```

### Row State Operators

There is a special data element type called a row state for storing various attributes in the data table, such as whether a row is selected or not, excluded or not, hidden or not, and labeled or not; which marker to use for graphs, and which color, shade, and hue to use for graphs. Row states can be freely converted to integers and vice versa.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Arguments Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Frequency</td>
<td>Set the type of time frequency.</td>
<td>Specify the time frequency:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>col&lt;&lt;Set Property(&quot;Time Frequency&quot;, Time Frequency( Annual ))</code></td>
</tr>
<tr>
<td>Map Role</td>
<td>Set how the column is used to connect map shape data with name data.</td>
<td>Specify the role and other information as necessary:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>col&lt;&lt;Set Property(&quot;Map Role&quot;, Map Role( Shape Name Use(&quot;filepath to data table&quot;, &quot;column name&quot; )) )</code></td>
</tr>
<tr>
<td>[Custom property]</td>
<td>Provided for custom uses.</td>
<td>Corresponds to Column Properties &gt; Other in the Column Info window. The first argument is a name for the custom property, and the second argument is an expression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>`col&lt;&lt;Set Property(&quot;Date recorded&quot;,12Dec1999);</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>long date(col&lt;&lt;get property(&quot;Date recorded&quot;));</code></td>
</tr>
</tbody>
</table>

---

**Table 4.2 Properties for Data Table Columns (Continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Arguments Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Frequency</td>
<td>Set the type of time frequency.</td>
<td>Specify the time frequency:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>col&lt;&lt;Set Property(&quot;Time Frequency&quot;, Time Frequency( Annual ))</code></td>
</tr>
<tr>
<td>Map Role</td>
<td>Set how the column is used to connect map shape data with name data.</td>
<td>Specify the role and other information as necessary:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>col&lt;&lt;Set Property(&quot;Map Role&quot;, Map Role( Shape Name Use(&quot;filepath to data table&quot;, &quot;column name&quot; )) )</code></td>
</tr>
<tr>
<td>[Custom property]</td>
<td>Provided for custom uses.</td>
<td>Corresponds to Column Properties &gt; Other in the Column Info window. The first argument is a name for the custom property, and the second argument is an expression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>`col&lt;&lt;Set Property(&quot;Date recorded&quot;,12Dec1999);</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>long date(col&lt;&lt;get property(&quot;Date recorded&quot;));</code></td>
</tr>
</tbody>
</table>
Data Tables
Row State Operators

What are Row States?

Row state is a collection of six attributes that all rows in a data table have. These six attributes are actually packed into a single number internally. (How row states are defined numerically is invisible and unimportant for most users, but you can find the formal definitions under AsRowState.) JSL supports row state values as a separate type in expressions. First this section shows what row states do in JMP, and then it examines in greater detail how to work with row states in scripts.

What Row States Do

Once row states are set up, they change the way JMP works with your data. The table below explains each row state separately, but keep in mind that you can also use several row states at once to get the combination of effects that you want.

Table 4.3 Row States and How They Affect JMP Analyses, Charts, and Plots

<table>
<thead>
<tr>
<th>Row states</th>
<th>How they affect results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excluded</td>
<td>If rows are excluded, JMP omits them from calculations for statistical analyses (text reports and charts). Results are the same as if you had never entered those rows in the table. However, points are still included in plots. (To omit points from plots, use Hide. To omit points from all results, use both Exclude and Hide.)</td>
</tr>
<tr>
<td>Hidden</td>
<td>If rows are hidden, JMP does not show them in plots. However, the rows are still included in text reports and charts. (To omit points from reports and charts, use Exclude. To omit points from all results, use both Exclude and Hide.)</td>
</tr>
<tr>
<td>Labeled</td>
<td>If rows are labeled, JMP places row number labels, or the values from a designated Label column, on points in scatterplots.</td>
</tr>
<tr>
<td>Color</td>
<td>If rows have colors, JMP uses those colors to distinguish the points in scatterplots.</td>
</tr>
</tbody>
</table>
There are several ways to set row states in the JMP graphical user interface.

Manually
1. Select one or more rows in the data table.
2. Select row state choices from the \textit{Rows} menu.

The JSL equivalent is similar to these lines, depending on the rows selected:
\[
dt<<\text{select where}(\text{row}()==3|\text{row}()==6|\text{row}()==7|\text{row}()==9); \\
dt<<\text{markers}(2);
\]

Algebraically
1. Set a row state with an expression that evaluates to row state number values.
2. Create a row state column.
3. In the Column panel, right-click on the row state column and select \textit{Copy to Row States} or \textit{Add to Row States}.

The JSL equivalent would be something like this:
\[
\text{for each row}(
\quad \text{rowstate}()=\text{combine states(marker state(row()), color state(row()))});
\]

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Row states & How they affect results \\
\hline
Marker & If rows have markers, JMP uses those markers to distinguish the points in scatterplots. \\
\begin{itemize}
\item 
\item +
\item *
\item ▽
\item △
\item □
\item □
\item □
\item □
\item □
\item □
\item □
\item □
\end{itemize} \\
\hline
Selected & If rows are selected, JMP highlights the corresponding points and bars in plots and charts. \\
\includegraphics[width=1in]{selected_row_states.png} \\
\hline
\end{tabular}
\caption{Row States and How They Affect JMP Analyses, Charts, and Plots (Continued)}
\end{table}
**How Row States Work**

When you give a row state operator a number as its argument, or else something that evaluates to a number, the operator interprets that number as an index to its possible values. Most of the row states (Exclude, Hide, Label, Select) are Boolean, which mean they are either on or off, 1 or 0. Marker can be 0, 1, 2, ..., 15. Color choices are 0, 1, 2, ..., 84; 0–15 for the basic colors, 16–31 for dark shades of the same colors, 32–47 for light shades, 48–63 for very dark shades, 64–79 for very light shades, and 80–84 for shades of gray.

Values outside the range 0 to 84 are undefined. For best results, use Modulo or other operators to enforce valid values.

Missing values result in no row state.

**Row State Columns Versus Row States in Effect**

There are two places row state data can appear in the data table window. One is in a row state column: a column that is not numeric or character but rather row state. Row state columns *store* row state information but do *not* put the information into effect.

**Example**

1. Submit this line to start a new data table:
   ```
   dt = New Table( "Row State Testing" );
   ```
2. Submit these lines to add row states to a column and add 10 rows:
   ```
   dt << New Column( "Row State Data", Row State, Set Formula( Color State( Row() ) ) );
   dt << Add Rows(10);
   ```
3. Submit this line to put the row states into effect:
   ```
   For Each Row( Row State() = :Row State Data );
   ```

---

**Figure 4.3** Table with No Row States (left) and Table with Row States (right)
This replaces the row states in effect with the row state combination from a column. “Some Operators set one Characteristic and Clear Others,” p. 139, discusses techniques for changing selected attributes of a row state without changing or canceling others.

An Overview of Row State Operators

From JSL, you can get or set row states directly using the Row State operator. You can get or set the state for row n with Row State(n). If you do not supply an argument, you set or get the current row. To work with all rows, either use a For Each Row loop or else work with formula columns. Continuing the example with the data table just created:

```
rowstate(1); // returns row state for row 1
Color State(1)

row()=8;rowstate(); // returns row state for current (8th) row
Color State(8)
```

For each row(print(rowstate()));

```
Color State(1)
Color State(2)
Color State(3)
Color State(4)
Color State(5)
Color State(6)
Color State(7)
Color State(8)
Color State(9)
Color State(10)
```

To get a row state and store it in a global, place Row State() on the right side of an assignment:

```
::x=rowstate(1); // puts the row state of row 1 in x, a global
row()=8;::x=rowstate();  // puts the row state of the 8th row in x
show(x);

x = Color State(8)
```

// put row states in rscol, a row state column
dt<<new column("rscol", rowstate);
for each row(:rscol=rowstate());

To set the current state for a row, use Row State() on the left side of an assignment.

```
row state(1)=color state(3);              // makes row 1 red
row()=8; row state()=color state(3);      // makes the 8th row red
for each row(row state()=color state(3)); // makes each row red
```

Be careful whether you set every aspect of Row State() or just one aspect of it, such as Color Of(Row State()). To see how this works, first color and mark all the rows:

```
for each row(
    rowstate()=combine states(color state(row()), marker state(row()));
)
```

And now observe the difference between setting one attribute of a row state:
Row State Operators

- color of(rowstate(1)) = 3; // makes row 1 red without changing marker

And setting every aspect of a row state to a single state:

- row state(1) = color state(5); // makes row 1 blue and loses its marker

To copy all the current row states into a row state column:

- new column("rscol", set formula(rowstate()));
- for each row(rscol = rowstate());

To copy several but not all the current row states into a row state column, you would use a script like the following, commenting out or omitting any states that you do not want.

- new column("rscol2", set formula(CombineStates(
  ColorState( color of() ),
  ExcludedState( excluded() ),
  HiddenState( hidden() ),
  LabeledState( labeled() ),
  MarkerState( marker of() ),
  SelectedState( selected() ))));

Confusion alert! The current row for scripting is not related to rows being selected (highlighted) in the data table or to the current cursor position in the data table window. The current row for scripting is defined to be zero (no row) by default. You can set a current row with Row. For example, Row() = 3, but please note that such a setting only lasts for the duration of that script, and then Row() reverts to its default value, zero, so submitting a script all at once can produce different results than submitting a script a few lines at a time.

Another way to establish a current row for a script is to enclose it in For Each Row, which executes the script once for each row of the current data table.

Throughout this chapter, examples without an explicit current row, such as below, should be assumed to take place within a context that establishes a current row. For example, inside For Each Row( ) or after Row() = 1.

- rowstate() = color state(3);

See “What is the Current Row?,” p. 101, for more information.

Combinations of Characteristics

You can get or set many characteristics at once by combining state settings inside Combine States, or you can get or set each characteristic one at a time, with the ultimate row state being the accumulation of characteristics. For example, with Big Class, you can set green Y markers for males but for now hide them in plots, and set red X markers for females and not hide them in plots:

- for each row(  
  if(sex == "M",
    /*then*/ row state() = combine states(
      color state(4), marker state(6), hidden state(1)),
    /*else*/ row state() = combine states(  
      color state(5), marker state(7), hidden state(0))));

- row state() = color state(3);
color state(3), marker state(2), hidden state(0));

Get the row state for one row, such as the 6th:

row state(6);

Combine States(Hidden State(1), Color State(4), Marker State(6))

Notice that JMP returns a Combine State combination. This is because a row state datum is not just the state of one characteristic, such as color, but the cumulative state of all the characteristics that have been set: exclusion, hiding, labeling, selection, markers, colors, hues, and shades. A list of such characteristics is called a row state combination.

Just as there can be many row state characteristics in effect, a row state column can have multiple-characteristic row states as its values.

**Some Operators set one Characteristic and Clear Others**

In addition to the overall Row State operator for getting or setting all the characteristics of a row state, there are separate operators to get or set one characteristic at a time preemptively. That is, to give a row one characteristic, canceling any other characteristics that might be in effect. These operators are Color State, Combine States, Excluded State, Hidden State, Hue State, Labeled State, Marker State, Selected State, Shade State.

For example, to make row 4 be hidden only:

rowstate(4)=hidden state(1);

**Other Operators set or get one Characteristic at a Time**

As seen above, a row state is not just one characteristic but many. To work with just one thing at a time, use one of the L-value operators with Row State on either side of the equal sign, depending on whether you want to get or set. There is an L-value operator for each of the characteristics: Color Of, Excluded, Hidden, Labeled, Marker Of, Selected.

// hide row 4 without affecting other characteristics:
hidden(rowstate(4))=1;

// store color of row 3 without getting other characteristics:
::color = color of(rowstate(3));

**Get Row States**

To access a row state, place the row state expression on the right side of an assignment:

x = rowstate(i);  // puts row state of row i into x
x = rowstate();   // row state of current row

To access just a component of a row state, place the row state expression on the right side of an assignment and also use one of the L-value operators:

x = selected(rowState());  // selection index of current row selected
**Set Row States**

To set a row state, place the row state expression on the left of an assignment (L-value):

```plaintext
row state() = expression;     // for the current row
row state(i) = expression;    // for a specified row
```

To set a component of a row state,

```plaintext
ColorOf(row state(i)) = 3;    // change color to red for row i
Selected(RowState(i)) = 1;    // select the ith row
```

Confusion alert! As seen above, some of the operators convert numbers into states, and others convert states into numbers. Here are hints for remembering which are which:

**Number-to-state operators have the word “State”**  The operators that take number arguments and either return states or accept state assignments all have the word “State” in their names: `Row State`, `As Row State`, `Color State`, `Combine States`, `Excluded State`, `Hidden State`, `Hue State`, `Labeled State`, `Marker State`, `Selected State`, `Shade State`.

**State-to-number operators are one word or have the word “Of”**  The operators that take row state arguments (and assume that the argument `Row State()` if none is given) and return or get set to numbers are either one word or their second word is “Of”: `Color Of`, `Excluded`, `Hidden`, `Labeled`, `Marker Of`, `Selected`.

Table 4.4 “Operators for converting between numbers and row states,” p. 141, is a helpful comparison chart for these operators.

The following lines are equivalent to their interactive commands.

- `Copy From Row States`
- `Add From Row States`
- `Copy To Row States`
- `Add To Row States`

**Set Row State Combinations**

Use `Combine States` to put together the settings of the various States:

```plaintext
x = CombineStates(SelectedState(1),LabeledState(0),ColorState(3));
for each row(rowstate()=x);
```

**Create Row State Columns**

To create a row state directly in a column, use a formula:

```plaintext
dt<<new column("color formula", row states, set formula(color state(row())));
```

To put row states from a column into effect:

```plaintext
for each row(rowstate()=:color formula);
```
**Make a Row State Handler**

The `MakeRowStateHandler` message (sent to a data table object) obtains a callback when the row states change. For example,

```plaintext
f = Function( {X}, Show( x ) );
obj = Current Data Table() << MakeRowStateHandler( f );
```

Then when you select a group of rows, the row numbers of any row whose row state changed are sent to the log. For example:

```plaintext
x:[3, 4, 28, 40, 41]
```

When a group is highlighted, it might call the handler twice, once for rows whose selection is cleared, then again for the new selection.

**Each of the Row State Operators in Detail**

Here is a comparison chart of the different row state operators, so you can see which to use to convert from row states to indices and vice versa.

**Table 4.4** Operators for converting between numbers and row states

<table>
<thead>
<tr>
<th>Numbers</th>
<th>Convert from numbers to row states</th>
<th>Row states</th>
<th>Convert from row states to numbers</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 0</td>
<td>Excluded State((n))</td>
<td>Excluded</td>
<td>Excluded(rowstate)</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Hidden State((n))</td>
<td>Hidden</td>
<td>Hidden(rowstate)</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Labeled State((n))</td>
<td>Labeled</td>
<td>Labeled(rowstate)</td>
<td>1 or 0</td>
</tr>
<tr>
<td>1 or 0</td>
<td>Selected State((n))</td>
<td>Selected</td>
<td>Selected(rowstate)</td>
<td>1 or 0</td>
</tr>
</tbody>
</table>
### Table 4.4 Operators for converting between numbers and row states (Continued)

<table>
<thead>
<tr>
<th>Numbers</th>
<th>Convert from numbers to row states</th>
<th>Row states</th>
<th>Convert from row states to numbers</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 31</td>
<td>Marker State(n)</td>
<td>Marker</td>
<td>Marker Of(rowstate)</td>
<td>0 to 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• + × ◊</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>◊ △ γ z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>◊ ◊ ◊ ◊ *</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ = ■ ■</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>◊ ◆ ◄ ◄</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▲ ▼ ◄ ◄</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∧ ∨ &lt; &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▷ ◄ ◄ ◄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 84</td>
<td>Color State(n)</td>
<td>Color</td>
<td>Color Of(rowstate)</td>
<td>0 to 84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0–15 basics, 16–31 dark, 32–47 light, 48–63 very dark, 64–79 very light, 80–84 grays</td>
<td></td>
<td></td>
<td>(0–15 basics, 16–31 dark, 32–47 light, 48–63 very dark, 64–79 very light, 80–84 grays)</td>
<td></td>
</tr>
<tr>
<td>0–11</td>
<td>Hue State(n)</td>
<td>Hue</td>
<td></td>
<td>0 to 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–2 to 2</td>
<td>Shade State(n)</td>
<td>Shade</td>
<td></td>
<td>–2 to 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Excluding, Hiding, Labeling, and Selecting**

This section discusses the Boolean states: conditions that are either on or off. The next section looks at the Color, Marker, Hue, and Shade states: conditions that have many choices.

**Excluded** gets or sets an excluded index, which is 1 or 0 (true or false) indicating whether each row is excluded. Similarly, **Hidden** gets or sets a hidden index, which is 1 for hidden or 0 for not hidden; **Labeled** gets or sets a labeled index, which is 1 for labeled or 0 for not labeled; and **Selected** gets or sets a selected index, which is 1 for selected or 0 for not selected. For example:

- `Excluded(Row State()); // returns 1 if current row is excluded, 0 if not`
- `Hidden(); // returns 1 if current row is hidden, 0 if not`
- `Labeled(Row State()); // returns 1 if current row is labeled, 0 if not`
- `Selected(); // returns 1 if current row is selected, 0 if not`

Excluded State, Hidden State, Labeled State, and Selected State do the reverse. That is, they get or set a row state condition as true or false according to the argument. Nonzero values set the row state to true, and zero values set it to false. Missing values result in no change of state.

- `rowstate()=excluded state(1); // changes current row state to excluded`
- `rowstate()=hidden state(0); // changes current row state to not hidden`
- `rowstate()=labeled state(1); // changes current row state to labeled`
- `rowstate()=selected state(0); // changes current row state to not selected`

Notice that the last two commands replace the row state with just the exclusion or just the unhiding, so that any preexisting row state characteristics are lost. More commonly you would issue the State commands for all the characteristics that you want inside a Combine States:

- `rowstate()=combine states(color state(4), marker state(3), hidden state(1)); // changes the current row to hidden and a green square marker`

Another common way to use a -State command would be in a row state data column whose values could be added to the row state (for cumulative characteristics). This example excludes each odd numbered row.

```
dt<<new column("myExcl", rowstate, set formula(Excluded State(Modulo(Row(),2))));
for each row(rowstate()=combine states(:myExcl, rowstate()));
```

**Colors and Markers**

**Color Of** returns or sets the color index, which is a number from the JMP color map that corresponds to the row state, or a missing value if there is no assigned color.

- `ColorOf(rowstate()); // returns color index for current row`
- `ColorOf()=4; // sets current row to Color 4`
Similarly, MarkerOf returns or sets the marker index, which is a number from the JMP marker map that corresponds to the active marker, or a missing value if there is no assigned marker.

```
MarkerOf();                // returns marker index for current row
MarkerOf(rowstate())=4;   // sets current row to Marker 4
```

Both ColorOf and MarkerOf accept any row state expression or column or Row State() as arguments and assume that the argument Row State() if none is given (some examples are shown with and some without).

Color State and Marker State are similar to ColorOf and MarkerOf, except they work in the opposite direction. Where the -Of functions turn actual states into indices, the -State functions turn indices into states.

```
rowstate()=color state(4);  // changes current row to green
rowstate()=marker state(4); // changes current row to the diamond marker
```

Notice that the last two commands replace the row state with just the color or just the marker, so that any preexisting row state characteristics are lost. More commonly you would issue the -State commands for all the characteristics that you want inside a Combine States:

```
rowstate()=combine states(color state(4), marker state(3), hidden state(1)); // changes current row to hidden with green square marker
```

This script (which you learn how to create in the “Display Trees” chapter) shows the standard JMP markers, which are numbered 0–31. Indices outside the range 0–31 have undefined behavior.

```
New Window( "Markers",
   Graph Box( 
      FrameSize( 300, 300 ),
      Y Scale( -1, 16 ),
      X Scale( 0, 7 ),
      For( 
         i = 0;
         jj = 15;;
         i < 16;
         jj >= 0;;
         i++; 
         jj--; // 16 rows, 2 columns
         Marker Size( 3 );
         Marker( i, {1, jj + .2} ); // markers 0-15
         Marker( i + 16, {4, jj + .2} ); // markers 16-31
         Text( {1.5, jj}, "marker ", i ); // marker labels 0-15
         Text( {4.5, jj}, "marker ", i + 16 ); // marker labels 16-31
      )
   )
);
```
JMP colors are numbered 0 through 84, where the first 16 are the basic colors shown below, and higher numbers are darker or lighter shades of those colors. Indices outside the range 0–84 have undefined behavior. This script shows the standard JMP colors. The first 16 colors also have names. The others are shades of those 16. For more information about using JMP colors, see “Colors,” p. 261 in the “Scripting Graphs” chapter.

```
Text Color( 0 );
New Window( "Colors",
   Graph Box(
      FrameSize( 640, 400 ),
      Y Scale( -1, 17 ),
      X Scale( -3, 12 ),
      k = 0;
      For( jj = 1, jj <= 12, jj += 2,
          l = 15;
          For( i = 0, i <= 15 & k < 85, i++,
              thiscolor = Color To RGB( k );
              Fill Color( k );
              thisfill = 1;
              If( thiscolor == {1, 1, 1},
                  Pen Color( 0 );
                  thisfill = 0;
                  ,
                  Pen Color( k )
              );
              Rect( jj, l + .5, jj + .5, l, thisfill );
              Text( {jj - 1, l}, "color ", k );
          k++;
      l--;)
      ));
```
Data Tables
Chapter 4
Row State Operators

```
jj = -2;
"Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan", "Fuschia", "Black"};
For( 
i = 0;
l = 15; i <= 15 & l >= 0,
i++; l--;,
Text( {jj, l}, color[i + 1] )
);
```

Figure 4.5 JMP Colors

If you prefer to use RGB values, each color should be a list with percentages for each color in red, green, blue order.

```
pen color(.38, .84, .67); // a lovely teal
```
**Hue and Shade**

Hue State and Shade State together are an alternative to Color State for picking colors. You cannot select black, white, or the shades of gray when you use Hue State. For these, you must use Shade State alone, or Color State.

The script below demonstrates how hue and shade values relate to colors:

```plaintext
New Window( "Hues and Shades",
  Graph Box(
    FrameSize( 600, 300 ),
    Y Scale( -3, 3 ),
    X Scale( -2, 12 ),
    k = 0;
    For( h = 0, h < 12, h++,
      For( s = -2, s < 3, s++,
        myMk = Combine States( Hue State( h ), Shade State( s ), Marker State( 15 ) );
        Marker Size( 3 );
        Marker( myMk, {h, s} );
      )
    );
    Text( Center Justified, {5, 2.5}, "<--- Hues 0-11 ---> " );
    Text( Right Justified,
      {-.5, -2}, "Shade -2", {-.5, -2.25}, "(Very dark)",
      {-.5, -1}, "Shade -1", {-.5, -1.25}, "(Dark)",
      {-.5, 0}, "Shade 0", {-.5, -.25}, "(Basic hue)",
      {-.5, 1}, "Shade 1", {-.5, .75}, "(Light)",
      {-.5, 2}, "Shade 2", {-.5, 1.75}, "(Very light)"
    );
  ));
```

In the example below (which assumes you have CP and CA data), row state values are prepared ahead and passed to the Marker routine, along with matrices of coordinates.

// assumes CP and CA data such as simulated below
dt = New Table( "Artificial CP and CA data", Add Rows( 26 ), New Column( "cover_cp", Numeric, Continuous, Formula( Random Uniform() / 100 + 0.94 ) ), New Column( "cover_ca", Numeric, Continuous, Formula( Random Uniform() * 0.04 + 0.94 ) ), New Column( "p", Numeric, Continuous, Formula( Random Uniform() ) ) );
dt<<Run Formulas;
greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
New Window( "CP and CA Comparisons", Graph Box(

Note: There are no -Of operators for Hue and Shade. Color Of returns the equivalent Color State index for a color row state that has been set with Hue State or Shade State. The following example gives rows 4 and 5 the same dark red marker:

Row State( 4 ) = Combine States( Hue State( 0 ), Shade State( -1 ), Marker State( 12 ) );
Row State( 5 ) = Combine States( Color State( Color Of( Row State( 4 ) ) ), Marker State( Marker Of( Row State( 4 ) ) ) );

QC Chart Example

In the example below (which assumes you have CP and CA data), row state values are prepared ahead and passed to the Marker routine, along with matrices of coordinates.

// assumes CP and CA data such as simulated below
dt = New Table( "Artificial CP and CA data", Add Rows( 26 ),
    New Column( "cover_cp", Numeric, Continuous, Formula( Random Uniform() / 100 + 0.94 ) ),
    New Column( "cover_ca", Numeric, Continuous, Formula( Random Uniform() * 0.04 + 0.94 ) ),
    New Column( "p", Numeric, Continuous, Formula( Random Uniform() ) ) );
dt<<Run Formulas;
greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
New Window( "CP and CA Comparisons", Graph Box(
Optional: The Numbers Behind Row States

This section is an optional topic for advanced users who are interested in working with row states through their internal numeric codes.

Earlier the chapter mentioned that all six row states are actually stored as a single number in JMP. You can see row states’ internal coding if you want. Simply copy row states to a column, and then change the column’s type to numeric to see the numbers that JMP uses.

It is also possible to assign row states through their internal numeric codes using the *As Row State* operator, which simply converts integers to their equivalent row states. For example, to assign row states according to the row number, you could do:

```julia
for each row(rowstate()=as row state(row()));
```

In addition, the *Set Row States* command enables you to submit a matrix of codes that assign the row states all at once. The matrix should have dimension (number of rows) by 1, and contain one entry for each row. The entries are the row state codes corresponding to the row’s desired state.

Such row states are unlikely to be of any use, however. For practical applications, understanding the way numbers are related to row states is important. Briefly, for some row state \( r \), such as the row state of the 3rd row as shown here, the row state code is computed by this formula:

\[
rscode = selected(r) + 2*excluded(r) + 4*hidden(r) + 8*labeled(r) + 16*marker of(r) + 256*color of(r);
\]

**Example**

Take advantage of this method to develop a compact formula to distinguish females and males with Xs and Ys, while excluding females from calculations, hiding males from plots, and assigning different colors for each age.

Recall that the logical `==` operator is an equality test that returns 1 for true and 0 for false.

```julia
dt=open("$SAMPLE_DATA/Big Class.JMP");
for each row(
    rowstate() = as row state(
```
Calculations

This section discusses functions for pre-evaluated columnwise and rowwise statistics and shows how JSL expressions work behind the scenes in the JMP formula calculator. See the earlier discussion of “Summarize,” p. 102, for a way to gather summary statistics on a data table.

Pre-Evaluated Statistics

JMP has \texttt{Col Maximum}, \texttt{Col Mean}, \texttt{Col Minimum}, \texttt{Col N Missing}, \texttt{Col Number}, \texttt{Col Quantile}, \texttt{CV} (Coefficient of Variation), \texttt{Col Standardize}, \texttt{Col Std Dev}, \texttt{Col Sum}, \texttt{Maximum}, \texttt{Mean}, \texttt{Minimum}, \texttt{NMissing}, \texttt{Number}, \texttt{Std Dev}, and \texttt{Sum} functions, which are special “pre-evaluated” functions.

All the statistics are \textit{pre-evaluated}. That is, JMP calculates them once over the rows or columns specified and thereafter uses the results as constants. Because they are computed once and then used over and over again, they are more efficient to use in calculations than an equivalent formula-calculated results.

When JMP encounters a pre-evaluated function in a script, it immediately evaluates the function and then uses the result as a constant thereafter. Thus, pre-evaluated functions enable you to use columnwise results for rowwise calculations. For example, if you use \texttt{Col Mean} inside a column formula, it first evaluates the mean for the column specified and then uses that result as a constant in evaluating the rest of the formula on each row. For example, a formula might standardize a column using its pre-evaluated mean and standard deviation:

\[
\frac{\text{Height} - \text{Col Mean(Height)}}{\text{Col Std Dev(Height)}}
\]

For the Big Class data set, \texttt{Col Mean(Height)} is 62.55 and \texttt{Col Std Dev(Height)} is 4.24. So for each row, the formula above would subtract 62.55 from that row’s height value and then divide by 4.24.

Note: the pre-evaluated functions disregard the excluded row state, so any excluded rows are included in calculations. For summary statistics that obey row exclusion, use the Distribution platform.
Columnwise Functions

The functions whose names begin with “Col” all work columnwise, or down the values in the specified column, and return a single number. For example, Col Mean(height) finds the mean of the values in all the rows of the column height and returns it as a scalar result. Some examples:

\[
\text{Average Student Height} = \text{Col Mean}(\text{height}); \\
\text{Height Sigma} = \text{Col Std Dev}(\text{height});
\]

Rowwise Functions

The functions without “Col” listed below work rowwise across the values in the variables specified and return a column result. For example, Mean(height, weight) finds the mean of the height and weight for the current row of the data table. The rowwise statistics are valid only when used in an appropriate data table row context. Here are some possibilities:

\[
\text{// scalar result for row 7 assigned to JSL global variable} \\
\text{row()=7; ::scalar = Mean(height, weight);} \\
\text{// formula column created in data table} \\
\text{new column("Scaled Ht-Wt Ratio",} \\
\text{ formula(mean(height, weight)/age));} \\
\text{// vector of results} \\
\text{vector=J(1,40); // create a 1x40 matrix to hold results} \\
\text{for each row(vector[row()]=mean(height,weight)); //fill vector}
\]

Rowwise functions can also take vector (column matrix) or list arguments:

\[
\text{myMu=mean([1 2 3 4]); mySigma=stddev([1, 2, 3]);}
\]

Calculator Formulas

JMP enables you to store formulas in columns that are automatically evaluated to create the values in the cells of the column. If you open the formula, you get a calculator interface to edit the formula structurally, as shown below on the left.

However, the formula is implemented with JSL, and you can obtain the text JSL form of any expression in the calculator by double clicking on it. The text can be edited, and when it is defocused, it is compiled back into the structural form.

There is no difference between a formula column created through the calculator window and one created directly through JSL with commands such as New Column(..., Formula(...)) or Col<<Formula(...).
Platforms can be launched from scripts and subsequently controlled from scripts. If you do an analysis interactively, you can save a script that recreates the analysis.

Learning how to script platforms is no harder than learning how to control platforms interactively. Writing scripts yourself takes some effort, since you have to type them and get the syntax right. Fortunately, you can have the platform write much of a script for you, and the additional work is well worth it if you need to automate repetitive production analyses. Also, scripting presents valuable opportunities to customize and combine analyses.

Be aware that the script-saving feature of platforms currently has limitations. JMP saves analytical commands and most (but not all) presentation customizations. Therefore, if your presentation customizations are important to you, you might occasionally need to learn to program the display interface. JMP records the state of an analysis, but not the sequence of events leading to that state. If you want scripts to play back demonstrations of JMP platforms in action, you have some learning to do.

Platform results exist in two layers: the platform itself, containing the analytical results and responding to analytical commands; and the presentation display, which responds to a different set of commands. A third object is the data table itself, which can be involved in a live analysis.

1. If you want to add to the analysis, such as to fit a new line, you send a command to the platform itself using techniques discussed in this chapter.
2. If you want to make the frame of a graph larger, you send a command to the display. See “Manipulating Displays,” p. 177 in the “Display Trees” chapter.
3. If you want to highlight certain points representing rows in a data table, you send a row state command to the data table. See “Row State Operators,” p. 133 in the “Data Tables” chapter.

If you are interested in building your own custom analysis platforms and reports, see “Constructing Display Trees,” p. 190 in the “Display Trees” chapter. If you would like to build your own custom statistical calculations using JSL's compact matrix notation, see the “Matrices” chapter.
Contents

Scripting Analysis Platforms .......................................................... 155
Launching Platforms Interactively and Obtaining the Equivalent Script .................................................. 155
   Launch a Platform ................................................................. 156
   Save Script ................................................................. 156
   Make Some Changes .......................................................... 157
   Syntax for Platform Scripting .................................................. 157
   BY Group Reports ............................................................. 158
   Saving BY Group Scripts ...................................................... 160
Sending Script Commands to a Live Analysis .......................................................... 160
   Conventions for Commands and Arguments .................................. 161
   Sending Several Messages ...................................................... 162
   Learning the Messages an Object Responds to .............................. 162
   How to Interpret the Listing from Show Properties ......................... 163
Launching Platforms .......................................................... 164
   Specifying Columns ........................................................... 164
   Platform Action Command ...................................................... 164
   Invisible Reports .............................................................. 165
   Titles .............................................................. 165
General Messages for Platform Windows .......................................................... 165
   Options for All Platforms ...................................................... 168
Additional Notes .......................................................... 169
   Spline Fits ........................................................... 169
   Fit Model Effects ........................................................... 170
   Fit Model Send Command ...................................................... 171
   DOE Scripting ........................................................... 171
   Scatterplot Scripting ........................................................ 173
Scripting Analysis Platforms

Scripting platforms is done with the same keywords that you see in the windows and menus, including the choices that you see in red triangle menus and context menus inside a report.

You can use JMP interactively and then obtain the script corresponding to the analysis that you created. You can take advantage of this behavior in several ways:

1. Learn JSL by example. Launch platforms and work with them interactively, and then save and examine the script that matches your work. Make changes through the interface and then study the changes in the script.
2. Save a number of scripts from analyses to a script window, and then save it to a file. You can use that script to recreate the analysis later, or to reproduce the analysis using new data.
3. Save platform scripts, and then edit them to use JSL programming features to make the analyses more general and to customize reports.

Select Help > Object Scripting to see all the scripting commands available for each platform in the Object Scripting Index. This index includes all scriptable platform options, with a description, the syntax, and an example.

Notation in this chapter

This chapter Capitalizes The Names of command words that you need to use exactly as they are shown and shows arguments that are placeholders for actual choices in lowercase. For example, Connect Color is a command that you need to type as is, and color stands for some color choice that you make yourself.

Connect Color(color)

In this case, the argument in parentheses must be some color value. For example, a JMP color number, or a supported color name like "red", "blue", and so on, or an RGB value given as a list, such as {.75, .50, .50}. Sometimes alternatives like these are shown with the pipe (|) character for "or," like this:

Connect Color( number | "color name" | {r,g,b} );

You do not actually have to use upper- and lower-case letters, and you only type the | character in scripts when you want the logical OR operator.

You can use the following terms interchangeably depending on context: command, option, or message. The term argument refers to anything that goes in parentheses after an item, and arguments can also contain options with further arguments.

Launching Platforms Interactively and Obtaining the Equivalent Script

Here is an example to see how the JMP interface and scripting language are related.
Launch a Platform

1. Open the Big Class sample data table.
2. Select Fit Y by X from the Analyze menu.
4. Select age and click X, Factor.
5. Click OK.

Save Script

All analysis platforms have a Script submenu with the commands shown here to create a JSL script that duplicates the analysis in its current state. These commands are choices for where to send that script.

- **Redo Analysis**  
  Launches a new copy of the platform, rebuilding all the analyses as specified. This is useful for updating your results when the state of the data table has changed. For example, after you fix errors in the data table, select a subset, add more data, and so on.

- **Relaunch Analysis**  
  Opens the launch window, already filled in with what is needed to produce the report.

- **Automatic Recalc**  
  Turn on or off Automatic Recalc.

- **Copy Script**  
  Copies the script that reproduces the report onto the clipboard. You can then paste the script into a script window or another application.

- **Save Script to Data Table**  
  Saves the script as a new property in the data table. In the data table window, you see the script named for the platform, and it has a pop-up menu to let you Run Script. Properties such as saved scripts are saved with the data table for later use. Many sample data tables have example scripts.

- **Save Script to Journal**  
  Saves a button to the current journal (or a new one, if a journal is not open) that runs a script to reproduce the report.

- **Save Script to Script Window**  
  Presents the script for the current state of the platform in a script window, where you can view, edit, and submit it.

- **Save Script to Report**  
  Saves the script in a text box at the top of the report. The script to reconstruct an analysis is available in any journals that you might save, as well as in the platform window itself. This also serves as a concise overview, listing all the report's analyses and their parameter settings.

- **Save Script for All Objects**  
  Saves the script for all objects within a multi-platform report in a script window, where you can view, edit, and submit it. An example is Fit Y by X, where various combinations of continuous, nominal, and ordinal columns result in Bivariate, Oneway, Contingency, and Logistic platform objects all within a single report. Save Script to Script Window saves only the script for the single object from which you selected the option. Save Script for All Objects saves the script for all objects within the window, regardless of which object's menu option you used. Another example is that if you have By groups, Save Script for All Objects saves the script for all the BY group objects within the window, whereas Save Script to... captures the script for only the one group. All the BY groups appear in the same window when the report is re-run, because Save Script For All Objects wraps all By-variable analyses.
in a `New Window()` command. For example, a bivariate fit of height by weight grouped by males and females would result in the following script.

```
New Window("Big Class.jmp: Bivariate",
    Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "F"));
    Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "M")));
```

**Save Script to Project**  Creates a script to reproduce the report and saves it as a JSL file in a JMP project.

**Data Table Window**  Displays a data table window with the BY group data associated with the analysis when it has BY groups, and brings the data table window forward otherwise.

### Make Some Changes

Now, continue working in the Oneway report window.

1. From the red triangle menu for Oneway Analysis of height by age, select **Means/Anova**.
2. From the same menu, select **Compare Means > Each Pair, Student’s t**.
3. From the same menu, select **Script > Save Script to Script Window**.

The resulting script is as follows:

```
Oneway(
    Y(:height),
    X(:age),
    Each Pair(1),
    Means(1),
    Mean Diamonds(1),
    Comparison Circles(1)
);
```

The script records the choices that you made in the menu. Since these are all Boolean (on or off) options, they have an argument of 1 to turn them on. In addition to the options selected, three additional display options have been implied from the other commands given. Choices that you make in menus and their corresponding JSL commands have exactly the same effects.

You could submit this script to get exactly this report quickly, without all the interactive steps. To run a script from a script window, select the text and then either select **Edit > Run Script**, or type CONTROL-R (Windows) or COMMAND-R (Macintosh). If you select no text, the entire window is run.

### Syntax for Platform Scripting

Look at the script in more detail.

```
Oneway(
    Y(:height),
    X(:age),
    Each Pair(1),
    Means(1),
    Mean Diamonds(1),
);`
Comparison Circles(1);  

All platform scripts start with a command to call the platform, in this case Oneway. Inside the Oneway command are two types of arguments: arguments like the Y and X column role lists are required at launch, and options like Each Pair(1) are sent to the platform after it is launched.

Most options are simple on/off choices with check marks (or not). The scripting equivalent of a check box is a Boolean argument 1 or 0.

Other commands lead to dialog boxes where you specify values or make choices. In scripts, such specifications are given inside parentheses and separated by commas, usually in the same order as they appear in the window (top to bottom, left to right).

**BY Group Reports**

In most platforms, you can run the platform repeatedly across subgroups of the rows as defined by one or more by columns. To do this in scripts, include a By argument in the launch command, listing each column as the argument to By.

**Example**

1. Open the Big Class sample data table.
2. Run this script to produce a bivariate report by sex:
   
   ```
   biv = Bivariate(Y(weight), X(height), By(sex));
   ```

   This launch message produces a report window with two nodes in the outline, one for the rows of the data table where sex is “F” and a second for rows where sex is “M.” Rather than returning a reference to a platform, Bivariate[], the platform object returns a list of references {Bivariate[], Bivariate[]}:

   ```
   show(biv);
   biv = {Bivariate[], Bivariate[]}
   ```

   You can direct messages either to each reference individually or to the list of references, depending on whether you want to change selected nodes individually or all nodes simultaneously.

3. Run this line to add a regression fit to both nodes:
   
   ```
   biv << fit line;
   ```

4. Run this line to add a cubic fit to the F node:
   
   ```
   Bivariate[1] << fit polynomial(3);
   ```

5. Run this line to add a quartic fit to the M node:
   
   ```
   Bivariate[2] << fit polynomial(4);
   ```
Figure 5.1 Accessing By Group Reports

Multiple columns listed for By() produce nodes for each subgroup. For example, By(sex, age) would produce nodes for females age 12, females age 13, ..., females age 17, males age 12, males age 13, ..., and males age 17.

The following shows how to launch a platform with BY groups and extract results from each group:

```
// open data table Big Class
dt = Open("$SAMPLE_DATA/Big Class.jmp");

// launch Oneway platform
onew = Oneway(x(age), y(height), by(sex), anova);
   // onew is a list of platform object references
r = onew<<report;   // r is a list of display boxes
nBy = nItems(r);    // the number of by groups
vc = j(nBy, 1, 0); // a place to store the variances
```
// now extract results
for(i=1, i<=nBy, i++,
    vc[i] = r[i][
        OutlineBox("Analysis of Variance"),
        ColumnBox("Sum of Squares")][2];
    show(vc);

    summarize(byValues=by(sex));
newTable("Variances")
    << newColumn("Sex",character,width(8),values(byValues))
    << newColumn("Variance",numeric,continuous,values(vc));

### Saving BY Group Scripts

In addition to the Script submenu in an analysis platform, a **Script All By-Groups** submenu appears when appropriate. This submenu lets you save scripts to reproduce a report created with By groups. It contains the following options:

- **Redo Analysis**
- **Relaunch Analysis**
- **Copy Script**
- **Save Script to Data Table**
- **Save Script to Journal**
- **Save Script to Script Window**

All of these work just like their counterparts on the Script submenu, except they reproduce all By groups in a report.

### Sending Script Commands to a Live Analysis

After the platform has been launched, there is a different syntax to send messages to control the live platform, using the `Send` function or its operator equivalent, `<<`.

First you need a way to address the object. The simplest way is to launch the platform from a script that assigns a reference to the object to a global variable. For example, this script saves a reference to a Oneway analysis in the variable `oneObj`:

```plaintext
oneObj = Oneway(Y(height), X(age));
```

Then, you use either the `Send` function or the equivalent `<<` operator to send a message:

```plaintext
Send(oneObj, Unequal Variances(1));
onеObj << Unequal Variances(1);
```
Figure 5.2 Adding Elements to a Report Using a Script

Conventions for Commands and Arguments

1. You can omit the argument for a Boolean option to *toggle* the state: if the state is off, the message turns it on. If it is on, the message turns it off. In the example above, these following two commands would be equivalent. If you resubmit the first command repeatedly, it has no further effect, but if you submit the second command repeatedly, it flips the feature on and off.

   ```markdown
   oneObj<<Unequal Variances(1);
   oneObj<<Unequal Variances;
   ```

2. In cases where the menu gives several options separated by comma or slash, as in the case of *Means/Anova/t Test* above, you can use any one of the commands. In cases where several commands have the same alias, as in the two cases of “Means” above, the first one takes precedence in the scripting language.

3. If you make changes to the display, such as resizing the graph, they are also saved in the saved script.

4. If there are submenus whose items represent commands rather than settings, then the corresponding script is the items themselves without the parent item. For example, above see that Oneway has a menu item *Nonparametric* with three commands in a submenu, including *Wilcoxon Test*. You would use just the subitem name in scripts, for example:

   ```markdown
   oneObj = Oneway(Y(height), X(age), Wilcoxon Test(1));
   oneObj << Wilcoxon Test(1);
   ```

5. If there are submenus whose items are values for a setting rather than independent commands, then in script, you give the parent item with the submenu choice as its argument. For example, Oneway has a submenu for *Set Alpha Level*, whose choices are 0.10, 0.05, 0.01, and Other... To change the value in script, you give your choice as an argument to *Set Alpha Level*:

   ```markdown
   oneObj << SetAlphaLevel(.01);
   ```

6. The script returned from a platform often looks different from a script that you write yourself. For example, you could launch Distribution with this brief script:
dist=Distribution(Y(Height, weight));

But if you then ask Distribution to save its script, you see considerably more detail:

\[
\text{Distribution}(\text{Continuous Distribution}(\text{Column}(\text{ :height}), \text{Axis Settings}(\text{Scale(Linear)}, \text{Format("Fixed Dec", 0)}, \text{Min(50)}, \text{Max(72.5)}, \text{Inc(5)}), \text{Minor Ticks(1)})), \text{Continuous Distribution}(\text{Column}(\text{ :weight}), \text{Axis Settings}(\text{Scale(Linear)}, \text{Format("Fixed Dec", 0)}), \text{Min(70)}, \text{Max(180)}), \text{Inc(10)}));
\]

### Sending Several Messages

To send several messages, you can add more `<<` operators or more `Send` arguments:

\[
\text{dist}<<\text{quantiles(1)}<<\text{moments(1)}<<\text{more moments(1)}<<\text{horizontal layout(1)};
\]

\[
\text{Send(dist, quantiles(1), moments(1), more moments(1), horizontal layout(1))};
\]

Because `<<` is an *eliding operator*, it combines arguments and works differently than if its arguments were grouped. You can stack up multiple messages with extra `<<` symbols to perform them all in order (left to right). You can use grouping parentheses to send a message to the result of sending a message:

\[
(\text{dist}<<\text{stem and leaf(1)}) << \text{horizontal layout(0)};
\]

In this case, the associative grouping is of no consequence, because messages are performed left-to-right anyway. However, a case where it *would* matter is when sending messages to child objects, discussed next.

Another way to stack messages is to send a list of messages:

\[
\text{dist}<<\{\text{quantiles(1), moments(1), more moments(1), horizontal layout(1)}\};
\]

### Learning the Messages an Object Responds to

Now that you have an object, what messages can you send it? There are several ways to learn your options:

1. Try the procedure through the interactive interface first, then study the saved script.
2. Study the interface in the platform window. Items in the pop-up menus and context menus all have JSL equivalents with the same names and arguments, as discussed under “Launching Platforms Interactively and Obtaining the Equivalent Script,” p. 155.
3. Go to the **Help** menu, select **Indexes > Object Scripting**, find the object type that you are interested in, and click the item in the list.
4. **Show Properties**(*objectRef*) lists to the Log window all messages the object can receive:

\[
\text{show properties(oneObj)};
\]

\[
\text{Quantiles [Boolean](Shows or hides a quantile report.)}
\]

\[
\text{Means/Anova [Boolean](Shows or hides both an ANOVA and a means report.)}
\]

\[
\text{Means/Anova/Pooled t [Boolean](Shows or hides a t test, an ANOVA, and a means report.)}
\]

\[
\text{Means and Std Dev [Boolean](Shows or hides a report with both the mean and standard deviation for each level of the X variable.)}
\]

...
Send Script Commands to a Live Analysis

Show Properties also works with data tables and display boxes; see “Learning About Data Tables’ Messages,” p. 94 in the “Data Tables” chapter and “Learning What You Can do with a Display Box,” p. 183 in the “Display Trees” chapter.

How to Interpret the Listing from Show Properties

Notice that most items in the Show Properties output have hints inside brackets [ ] at the end of each line. This section examines the Show Properties for Bivariate as a general example.

```javascript
biv=Bivariate(Y(height), X(age));
show properties(biv);
```

[Column Names] properties are at the bottom. For example, X, Y, Freq, and Weight are things that need to be declared when you create a Bivariate analysis. They cannot be set or changed in an existing analysis.

```
X, Regressor [Column Names] [Scripting Only] (1UNC)
Y, Response [Column Names] [Scripting Only] (1UNC)
Freq [Column Names] [Scripting Only] (01N )
Weight [Column Names] [Scripting Only] (01N )
```

[Subtable]s refer to a set of commands that are put in a submenu. The commands in the subtable are indented, and you use the subitem itself, not the parent item.

```
Script [Subtable]
  Redo Analysis [Action]
  Save Script to Datatable [Action]
  Save Script to Report [Action]
```

[Boolean]s turn an option on or off, and their arguments are usually 1 or 0. If specified without an argument, sending the message flips it to the opposite state. Frequently [Boolean] messages also indicate that they by [Default On].

```
Show Points [Boolean] [Default On]
```

[Action]s and [New Entity]s are general purpose commands, often leading to windows in the user interface. Actions do not have a specific standard for their arguments, so try the item in the interface first and then study the script that the platform saves.

```
Fit Mean [New Entity]
Fit Line [New Entity]
Fit Polynomial [ActionChoice] {,2,quadratic,,3,cubic,,4,quartic,5,6}
```

Confusion alert! Do not confuse a reference to a platform with a reference to a report. They are different types of objects and can receive different types of JSL messages. For example, platforms can do such things as run tests, draw plots, or close entire windows. Reports can do such things as copy pictures, select display boxes, or close outline nodes.

This chapter discusses how to script platforms. To learn how to script reports, see the section “Manipulating Displays,” p. 177 in the “Display Trees” chapter.
Launching Platforms

Specifying Columns

Scripts to launch a platform should generally specify the columns to analyze. If you submit a script that launches a platform without specifying columns and roles, you get the dialog box for launching the platform. After you choose columns and click OK, you get the analysis that you specified by script. In other words, JMP remembers and obeys any other messages in your script after getting the column assignments that it needs.

If you want to use an expression to be evaluated for column arguments, put the names inside an Eval or EvalList function.

\[
\text{Distribution}(Y(\text{Eval}("X"||\text{char}(i)))));
\]

Column arguments for platform launch scripts can also be lists with braces {}, so the following are all valid:

\[
\text{Distribution}(Y(\text{height},\text{weight}));
\]
\[
\text{Distribution}(Y(\{\text{height},\text{weight}\})); \text{ // equivalent}
\]
\[
\text{Distribution}(\text{Weight}(\{\})); \text{ // empty specification}
\]

// (presents the platform's launch window)

Throughout this manual, a col placeholder represents any data table column reference, and a nomCol, ordCol, or contCol placeholder suggest that a nominal, ordinal, or continuous column, respectively, would be most appropriate. In many cases, columns of other modeling types would also be accepted. JMP returns an error if you try to cast a column into a role that is strictly not allowed.

Platform Action Command

The command Action, if sent to a platform, simply evaluates the expression, whatever it is. You can use this command to chain invocations to platforms where you want to use the platform launch window to ask the user to choose columns and then continue the script after the platform is launched.

In the following example, the script first asks you to assign columns for each of the four platforms in turn, and then print Done to the log. The script is effectively stopped four times, each time to prompt for columns for a platform launch. At each step, the user fills in a launch window. Four reports are open at the completion of the script.

This script runs to completion when the first platform launch window is brought up. The other behaviors are run from the stored expressions.

\[
\text{Distribution}(\text{Action}(\text{doit}));
\]
\[
\text{doit} = \text{expr(}
\text{ \hspace{1pt} Bivariate(\text{Action}(\text{doit2}))}
\text{ \hspace{1pt}});
\]
\[
\text{doit2} = \text{expr(}
\text{ \hspace{1pt} OneWay(\text{Action}(\text{doit3}))}
\text{ \hspace{1pt}});
\]
\[
\text{doit3} = \text{expr(}
\text{ \hspace{1pt} Contingency(\text{Action}(\text{doit4}))}
\text{ \hspace{1pt}});
\]
Invisible Reports

Platform launches have an invisible option, which suppresses the showing of the window. Using this option on a Fit Model script suppresses both the model launch window and the results window.

When using this option, be careful to keep track of the window in the script and close it when the script is done with it. The invisible windows use resources that must be manually freed.

The following example extracts the F-Ratio from a bivariate report. Make sure Big Class.jmp is open when running this script.

```julia
biv = bivariate(x(height),y(weight),invisible);
biv<<fit line;
r = biv<<report;
fratio = r[ColumnBox("F Ratio")][1];
r<<close window;
```

The invisible option also works on the Table menu operations, suppressing the creation of the window (rather than just hiding it).

```julia
Current Data Table()<<Select Where(:age==14);
subDt = Current Data Table() << Subset(invisible);
subDt<<bivariate(x(height),y(weight),fit line);
```

Note that the above script could more easily be done by a WHERE clause in the Bivariate command.

```julia
bivariate(x(height),y(weight), Where(:age==14),fit line);
```

Titles

You can specify the title (shown in the title bar of a platform's report) by adding the title command to the launch request. For example, the following replaces the standard bivariate report's title with a user-specified one.

```julia
Bivariate(x(height),y(weight), title("my title"));
```

General Messages for Platform Windows

The Save Script commands are applicable to almost all of the platforms. Some additional commands are shown in Table 5.2 “Messages that can be sent to platform windows,” p. 166. In particular the Report command lets you access the display surface to control appearance details such as window zooming, scrolling, and so on. For example, to close an outline node for a report, you first obtain a reference to the Display Box tree, subscript to navigate to the outlineBox, and then send it the close message:

```julia
r = platformRef << Report;
```
This is discussed in greater detail in “Manipulating Displays,” p. 177 in the “Display Trees” chapter.

Table 5.1 Scripting Analysis Platforms

<table>
<thead>
<tr>
<th>Goal</th>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>See available messages</td>
<td>Show Properties(obj)</td>
<td>Shows the messages that a given object can interpret, along with some basic syntax information. Works with all scriptable objects, not just platforms.</td>
</tr>
<tr>
<td>Send message</td>
<td>obj &lt;&lt; message</td>
<td>Sends a message to a platform object.</td>
</tr>
<tr>
<td>Send several messages</td>
<td>obj &lt;&lt; message &lt;&lt; message...</td>
<td>Sends a series of messages (in order, left to right) to the platform object.</td>
</tr>
<tr>
<td>Send message to child</td>
<td>obj &lt;&lt; (child &lt;&lt; message)</td>
<td>Sends a message to a child object within a platform object.</td>
</tr>
<tr>
<td>Suppressing output</td>
<td>Platform name(...,invisible)</td>
<td>Keeps output from a report from displaying on-screen.</td>
</tr>
<tr>
<td>Changing a report's title</td>
<td>Platform name(...,title(&quot;string&quot;))</td>
<td>Changes the report title to string.</td>
</tr>
<tr>
<td>Changing the automatic recalc setting</td>
<td>Platform name (...automatic recalc(Boolean))</td>
<td>True (1) sets automatic recalc on, so that any changes in the data or in the excluded state are immediately reflected in the report. False (0) sets automatic recalc off, so you must use redo analysis to see the changes. <strong>Note:</strong> If automatic recalc is on, you should use wait(0) commands to let the triggers take effect and do the recalculations.</td>
</tr>
</tbody>
</table>

Table 5.2 Messages that can be sent to platform windows

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redo Analysis</td>
<td>obj&lt;&lt;Redo Analysis</td>
<td>Launches the platform again with the same options.</td>
</tr>
<tr>
<td>Save Script to Data Table</td>
<td>obj&lt;&lt;Save Script to Data Table</td>
<td>Saves script to reproduce analysis as a property in the associated data table.</td>
</tr>
</tbody>
</table>
### Table 5.2 Messages that can be sent to platform windows

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save Script to Report</td>
<td>obj&lt;&lt;Save Script to Report</td>
<td>Saves script to reproduce analysis as a text box at the top of the report.</td>
</tr>
<tr>
<td>Save Script to Script Window</td>
<td>obj&lt;&lt;Save Script to Script Window</td>
<td>Saves script to reproduce analysis in the Script Journal window.</td>
</tr>
<tr>
<td>Save Script for All Objects</td>
<td>obj&lt;&lt;Save Script for All Objects</td>
<td>Saves script to reproduce all analyses found within the object’s window in the Script Journal window.</td>
</tr>
<tr>
<td>Get Script</td>
<td>x = obj&lt;&lt;Get Script</td>
<td>Returns script to reproduce the analysis as an expression.</td>
</tr>
<tr>
<td>Data Table Window</td>
<td>obj&lt;&lt;Data Table Window</td>
<td>Makes the associated data table window active (frontmost).</td>
</tr>
<tr>
<td>Close Window</td>
<td>obj&lt;&lt;Close Window</td>
<td>Closes the window identified by the obj, typically a platform surface.</td>
</tr>
<tr>
<td>Move Window</td>
<td>obj&lt;&lt;Move Window(x, y)</td>
<td>Moves the window to the (x,y) location on your screen.</td>
</tr>
<tr>
<td>Show Window</td>
<td>obj&lt;&lt;Show Window(0</td>
<td>1)</td>
</tr>
<tr>
<td>Zoom Window</td>
<td>obj&lt;&lt;Zoom Window</td>
<td>Resizes the window to the maximum size of its contents.</td>
</tr>
<tr>
<td>Scroll Window</td>
<td>obj&lt;&lt;Scroll Window(x, y)</td>
<td>Scrolls the window x pixels to the left and y pixels down from the current position; negative coordinates go right and up. If the coordinates are a list in braces { }, they are absolute coordinates. The window scrolls to the point x pixels from the left and y pixels from the top.</td>
</tr>
<tr>
<td>Bring Window to Front</td>
<td>obj&lt;&lt;Bring Window To Front</td>
<td>Brings the identified window to the front.</td>
</tr>
<tr>
<td>Size Window</td>
<td>obj&lt;&lt;Size Window(x, y)</td>
<td>Resizes the window to x pixels wide by y pixels high.</td>
</tr>
</tbody>
</table>
Options for All Platforms

Almost all platforms support the BY argument in the launch command, so By is not documented separately for each platform. See “BY Group Reports,” p. 158, for a general discussion.

Table 5.2 Messages that can be sent to platform windows

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximize Window</td>
<td>obj&lt;&lt;Maximize Window</td>
<td>Maximizes the window; equivalent to clicking the maximize button in the upper right corner of the window. This message takes an optional Boolean argument:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// maximize the window: obj&lt;&lt;Maximize Window(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// restore the window: obj&lt;&lt;Maximize Window(0)</td>
</tr>
<tr>
<td>Minimize Window</td>
<td>obj&lt;&lt;Minimize Window</td>
<td>Minimizes the window; equivalent to clicking the minimize button in the upper right corner of the window. This message takes an optional Boolean argument:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// minimize the window: obj&lt;&lt;Minimize Window(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// restore the window: obj&lt;&lt;Minimize Window(0)</td>
</tr>
<tr>
<td>Print Window</td>
<td>obj&lt;&lt;Print Window</td>
<td>Prints the selected window.</td>
</tr>
<tr>
<td>GetWindow Size</td>
<td>obj&lt;&lt;Get Window Size</td>
<td>Gets the size of the selected object. Returns an ordered pair showing width and height.</td>
</tr>
<tr>
<td>GetWindow Position</td>
<td>obj&lt;&lt;Get Window Position</td>
<td>Gets the position of the selected object. Returns an ordered pair.</td>
</tr>
<tr>
<td>Report TopReport</td>
<td>obj&lt;&lt;Report Report(obj)</td>
<td>Returns a display box reference for the report in the platform window. See “DisplayBox Object References,” p. 180 in the “Display Trees” chapter for a discussion. TopReport goes to the top display box, and is useful for BY groups or other cases when several platform reports are in one window.</td>
</tr>
<tr>
<td>Journal Window</td>
<td>obj&lt;&lt;Journal Window</td>
<td>Appends the contents of the window to the journal.</td>
</tr>
</tbody>
</table>
Title

You can use a Title command across all platforms, which replaces the standard title. For example,

\[ \text{Bivariate (X(height), Y(weight), Title("my title"));} \]

Axes

For platforms that include graphs, you can customize the axes within the platform script. Options specific to the graphs in each platform are listed.

Automatic Recalculation

If automatic recalc is on, you should use \texttt{wait(0)} commands to let the triggers take effect and do the recalculation.

Additional Notes

Spline Fits

There are also functions that allow spline fits. In each of the following functions, \( x \) is a vector of regressor variables, \( y \) is the vector of response variables, and \( \lambda \) is the smoothing argument. Larger values for \( \lambda \) result in smoother splines.

\[ \text{coef = Spline Coef(x, y, \lambda);} \]

returns a five column matrix of the form

\[ \text{knots||a||b||c||d} \]

where \( \text{knots} \) is the unique values in \( x \), and the spline calculated using the coefficients in the other columns as described with the Spline Eval function.

\[ \text{yhat=Spline Smooth(x, y, \lambda);} \]

returns the smoothed predicted values from the spline fit.

\[ \text{yhat=Spline Eval(x, coef);} \]

evaluates the spline predictions using the \texttt{coef} matrix in the same form as returned by \texttt{SplineCoef}. In other words, \texttt{knots||a||b||c||d}. The \( x \) argument can be a scalar or a matrix of values to predict. The number of columns of \texttt{coef} can be any number greater than 1 and each is used for the next higher power. The powers of \( x \) are centered at the knot values. For example, the calculation for \texttt{coef} of \texttt{knots||a||b||c||d} is

\[ j \text{ is such that knots}[j] \text{ is the largest knot smaller than } x \]

\[ xx = x-\text{knots}[j] \text{ is the centered } x \text{ value} \]

\[ \text{result} = a[j]+xx^*(b[j]+xx^*(c[j]+xx^*d[j])) \]

or, equivalently,
result = a[j] + b[j]*xx + c[j]*xx^2 + d[j]*xx^3

**Fit Model Effects**

Effects in the model can be more than just a list of columns, and have a specialized syntax:

```
Effect( list of effects, list of effect macros, or both lists)
```

An effect can be a column name, a crossing of several column names with asterisk (*) notation, or nested columns specified with subscript bracket ([ ]) notation. Additional effect options can appear after an ampersand (&) character. Some examples:

- A, // a column name alone is a main effect
- A*B, // a crossed effect, interaction, or polynomial
- A[B], // nested
- A*B[C D], // crossed and nested
- effect&Random, // a random effect
- effect&LogVariance, // a variance model term
- effect&RS, // a response surface term
- effect&Mixture, // for an effect participating in a mixture
- effect&Excluded, // for an effect that with no model parameters
- effect&Knotted, // for a knotted spline effect

Effect macros are:

- Factorial( columns ), // for a full factorial design
- Factorial2( columns ), // for up to 2nd-degree interactions only
- Polynomial( columns ), // for nth degree polynomial

Scripts for the fitting personality are put inside **Run Model**, if you want to run the model immediately. Use **Add Script** if you want to set the options, but not immediately run the model.

**Responses and Effects for MANOVA**

To address an individual response function analysis, use a subscripted **Response**:  

```
manovaObj << (Response[1] << {response options});
manovaObj << (Response["Contrast"] << {response options});
```

Each response function supports this:

```
Custom Test( matrix, <Power Analysis( ... )>, <Label( "..." )> )
```

where each row of the *matrix* specifies coefficients for all the parameters in the model.

To address an individual Effect test, use subscripted **Effect** with a name or number:

```
manovaObj << (Response[1] << (Effect["Whole Model"] << {effect options}));
manovaObj << (Response[1] << (Effect[i] << {effect options}));
```

The effects are numbered as follows:

- 0 for the intercept
- 1, 2, and so on, for regular effects
- \( n+1 \) for the “Whole Model” test, where \( n \) is the number of effects *not* including the intercept
Each effect in each response function supports these, where each row of the *matrix* has coefficients for all the levels in the effect:

- Test Details( 1 ),
- Centroid Plot( 1 ),
- Save Canonical Scores,
- Contrast( matrix, <Power Analysis(...) )

Here is a test script that works on the Dogs data table:

```vbscript
manObj = Fit Model(  
  Y( logHist0, logHist1, logHist3, logHist5 ),  
  Effects( dep1, drug, drug * dep1 ),  
  Personality( MANOVA ),  
  Run Model  
);
manObj << response function( Contrast, Go );
manObj << (response[1] << CustomTest([0 1 0 0, 0 0 1 0, 0 0 0 1]); // same as whole model
manObj << (response["contrast"] << (effect["whole model"] << TestDetails));
manObj << (response["contrast"] << (effect[3] << TestDetails));
```

**Fit Model Send Command**

To send commands to an individual response column's fit, use a syntax like this:

```vbscript
fitObj << (responseName << {options, ...});
```

The `Send` command inside the `Send` command finds the named response and sends the list of commands to it. If you instead send the options directly to the `fitObj` with a single `Send` command, the options are sent to all responses.

To send commands to an individual effect, you nest `Send` commands even further:

```vbscript
fitObj << (responseName << ((effectName) << effectOption));
```

**DOE Scripting**

The DOE platforms (Design of Experiments) are usually used as interactive windows, but are scriptable if you want to drive the platform through scripting. The result of DOE is a data table containing a table property named Model, which is a script to launch Fit Model with the settings appropriate for your design.

The following table lists the available DOE commands.

- Save Responses, Load Responses, Save Factors, Load Factors, Save Constraints, Load Constraints, Set Random Seed, Simulate Responses, Suppress Cotter Designs, Show Diagnostics, Save X Matrix, Optimality Criterion, Number of Starts, Sphere Radius, Disallowed Combinations, Advanced Options (Search Points Per Factor, Mixture Sum, Split Plot Variance Ratio, Number of Monte Carlo Samples, N Monte Carlo Spheres), Add Response, Add Factor, Add Term,
Additional Notes

Add Potential Term, Add Constraint, Make Model, Make Design, Make Table, Set Sample Size, Set N Whole Plots, Set N Subplots, Set Runs Per Random Block, Make Strip Plot Design, Center Points, Replicates, Report, Prior Parameter Variance, Theta (a vector that applies to only Maximum Entropy and IMSE designs, Entropy, Sphere Packing, Latin Hypercube, Uniform, Minimum Potential, IMSE

**Tuning Commands**

There are scripting commands that let you predefine guidelines for the design search by the custom designer. The following commands are described briefly here, and examples also appear in the *Design of Experiments*.

**DOE Mixture Sum** If you want to keep a component of a mixture constant throughout an experiment then the sum of the other mixture components must be less than one. In defining the factors, you enter the constant mixture component as a constant factor. For example, if you have a mixture factor with a constant value of 0.1, then the command

```
DOE Mixture Sum = 0.9;
```

constrains the remaining mixtures factors to sum to 0.9 instead of the default 1.0.

**DOE Starts** is the number of random starts used by the custom designer. In some situations, the default number of starts might not produce the design that you want. You can increase the number of starts with the **DOE Starts** command. For example, submitting the JSL statement

```
DOE Starts = 100;
```

overrides the default number of starts and sets the number of starts to 100.

**DOE Starting Design** For example,

```
DOE Starting Design = matrix;
```

replaces the random starting design with a specified matrix. If a starting design is supplied, the custom designer has only one start using this design.

**DOE Search Points Per Factor** For a linear model, the coordinate exchange algorithm in the custom designer only considers the high and low values by default. Suppose the low and high values for a factor are -1 and 1 respectively. If you submit the JSL command:

```
DOE Search Points Per Factor = 11;
```

then for each row, the coordinate exchange algorithm checks the eleven values, –1, -0.8, -0.6, -0.4, –0.2, 0, 0.2, 0.4, 0.6, 0.8, 1.0.

**DOE K Exchange Value** By default, the coordinate exchange algorithm considers every row of factor settings for possible replacement in every iteration. Some of these rows might never change. For example

```
DOE K Exchange Value = 3;
```

sets this value to a lower number, three in this case, so the algorithm only considers the most likely three rows for exchange in each iteration.

**DOE Bayes Diagonal** For example,
DOE Bayes Diagonal = vector:
This vector is used to modify the diagonal elements of the $XX$ matrix used for finding the $D$-optimal
design. The supplied vector is added to the current diagonal elements of the $XX$ matrix.

**DOE Sphere Radius** constrains the Custom Designer to a sphere instead of a hypercube.

**DOE Sphere Radius** = $n$
where $n$ is the radius of the constraining sphere.

Scatterplot Scripting

Use the `Scene3DHardwareAcceleration` command to set the **Use Hardware Acceleration** in Scatterplot 3D. For example,

```julia
dt = Open( "$SAMPLE_DATA/solubility.jmp" );
Scene3DHardwareAcceleration = 1;
dt << Scatterplot 3D(
   Y( :Name( "1-Octanol" ), :Ether, :Chloroform, :Benzene, :Carbon
   Tetrachloride, :Hexane )
);
```

About Hardware Acceleration

A computer's graphics card takes the graphics instructions away from the CPU so that the CPU can focus
on the core instructions and tasks. The CPU and graphics card both handle graphics requests using
software instructions. If a graphic card has the hardware acceleration feature, then it uses hardware to run
those instructions. Running instructions on hardware usually results in better performance than doing so
using software. In addition, hardware acceleration generally executes instructions concurrently, rather than
sequentially, for even better performance.

If the `Scene3DHardwareAcceleration` option is set to `True` (1), your graphics card uses hardware
acceleration if the card has that feature. Drawing 3-D images using this feature should result in faster
drawing times. However, using this option can affect your PC’s performance in other areas.
Chapter 6

Display Trees
Create and Use Windows

Now comes the chance to do more than program. This chapter shows how to create things in windows and interact with them. These sections discuss different types of display scripting:

- navigating displays to manipulate items in report windows
- creating display trees to build new windows with custom results or custom combinations of standard results

Confusion alert! You can both create and use some display boxes, but you can only use and not create other display boxes.

All display boxes that can be found in a display tree are listed in the DisplayBox Scripting Index, on the Help menu. You can send messages to all of them, but you cannot create all of them.

The JSL Functions Index, on the Help menu, includes all the display boxes that you can create.

If you see a display box listed in the DisplayBox Scripting Index that is not listed in the JSL Functions Index, then you can send messages to it, but you cannot create one. They are typically created as sub-boxes of other display boxes.

For details about scripting 2- and 3-dimensional plots, see the chapters “Scripting Graphs,” p. 237 and “Three-Dimensional Scenes,” p. 273.
Manipulating Displays

Reports in JMP are built in a hierarchical manner by nesting and gluing together different types of rectangular boxes. To manipulate reports with scripts, you first need to learn a bit about how these boxes work. You might want to learn more about how this works if:

- You want to manipulate display boxes from existing reports.
- You want to construct your own reports.

This section introduces JMP reports in general and then discusses how to navigate reports, extract data from them, and change them from JSL.

As you work with displays, a preference might be of use. In the General category of Preferences, select **Report Invalid Display Box Messages** to see possible errors. The default value is off.

When this preference is off and a script contains invalid display box messages, JMP ignores those errors and continues executing the script. When this preference is on, script execution stops when an invalid display box is reached, and an error is sent to the log.

Using this preference while you are developing your script helps you avoid possible problems.

### Introduction to Display Boxes

Here are a few diagrams of the most common types of display boxes in JMP.

**Table 6.1** The most common types of display boxes

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VListBoxes</td>
<td>glue boxes together vertically.</td>
</tr>
<tr>
<td>HListBoxes</td>
<td>glue boxes together horizontally.</td>
</tr>
</tbody>
</table>
Table 6.1 The most common types of display boxes (Continued)

You can nest VListBoxes and HListBoxes together. Here is a VListBox gluing together a VListBox and two HListBoxes.

TableBoxes are special HListBoxes whose elements are string and number columns.

OutlineBoxes create an outline hierarchy.

PictureBoxes glue together axes, frames, and labels to make graphs.

JSL has some other display box types for custom displays: Button Box, Slider Box, Tab Box and Global Box. These are discussed under “Interactive Display Elements,” p. 195. In addition, editable text
boxes (Text Edit Box) and boxes to make trees similar to those from the Diagram platform (Hier Box) are supported.

When a report is displayed, each box asks its children how big they are, so that it can position them and show them in a tidy arrangement.

Here is a report from Bivariate, annotated to show the structure of the display boxes.

Figure 6.1 Display Boxes in a Report

Table 6.5 “Display Boxes and Display JSL Functions,” p. 229, describes the box types and how they arrange their contents. Boxes labeled as “leaf” are those that contain no other boxes inside.

To examine a report’s display box tree

1. Enlarge the report window.
2. In an empty area of the report, hold the CTRL and SHIFT keys and right-click. You can click to the far right of the report, or underneath the report. If your report window is very large, close a few outline nodes until the report is smaller than the window.

3. Select **Show Tree Structure**.

**Figure 6.2** Show Tree Structure

In this example, the red Outline Box represents the closed **Fit Mean** outline node. The green Outline Box represents the open **Bivariate Fit of height By weight** outline node. Other boxes in the display tree are shown in their relationship to other boxes.

You can also obtain the tree structure through a script: send the `<<showtreestructure() message to any report. Or, send the message to a piece of the report (any displaybox object) to see the tree structure for just that part of the report.

**DisplayBox Object References**

A special type of JSL value called a *display box reference* can own or reference a DisplayBox. (Syntax summaries throughout this manual use **db** as a placeholder for any display box reference, **dt** for a data table reference, and **obj** for a scriptable object reference.)
You can create a display box reference with the `Report` message to access the top of the display tree associated with a scriptable platform.

```javascript
variable = platform object<<report;
```

For example, to obtain a reference to a report for a bivariate analysis, you could say:

```javascript
dt= Open("$SAMPLE_DATA/Big Class.jmp");
biv = bivariate(x(height), y(weight), fit mean, fit polynomial(4));
riv = biv<<report;
```

### Confusion alert!
Do not confuse a reference to a `report` with a reference to a `platform`. They are different types of objects and can receive different types of JSL messages. For example, reports can do such things as copy pictures, select display boxes, or close outline nodes. Platforms can do such things as run tests, draw plots, or close entire windows.

### Subscripts

If you want to make a reference to another part of the report, the easiest thing to do is use the subscript operator to find it. You can use any of the following methods, which all work by searching for a complete text string. The "text" argument can also be any expression that evaluates to a text string.

```javascript
db["text"]
```

Finds the outline in `db` that has the title `text`. Note that `text` must be the complete title, not just a substring of the title, but you can use ? as a wildcard character with a substring to match the rest of the title. For example, "? Estimates" to find "Parameter Estimates".

```javascript
db[Outline Box("text")]
```

Finds the outline box containing the `text`.

```javascript
db[Column Box("name")]
```

Finds the column box containing the `text`.

```javascript
db[boxType(n)]
```

Finds the `n`th display box of type `boxType`.

```javascript
db[arg1, arg2, arg3, ...]
```

Matches the last argument to a display box that is contained by the penultimate argument's outline node, which is in turn contained by the antepenultimate argument, and so on. In other words, it is a way of digging down the generations of an outline tree to identify a nested display box.

Typically you would want to assign the part of the report to a variable so that you could send messages to it easily.

```javascript
Variable = r1 ["search string"];  
```

For example, to find the Analysis of Variance report for Bivariate, you could say:

```javascript
r1=riv["Analysis of Variance"];  
```
If you want to identify an item nested several layers down in an outline, use a subscript with more than one argument of any of the above types. After locating the first item, JMP looks inside that item for the next, and so on.

```
db[arg1,arg2,arg3] /* finds the first item, then the next starting after that location, and so on */
db[arg1][arg2][arg3] // same as comma version
```

Note that you can string together subscripts of different types. This would select the 3rd column of the first table of the outline node “Parameter Estimate” under the node “Polynomial Fit Degree=4.”

```
rbiv["Polynomial Fit Degree=4"]["Parameter Estimates"][1][3] << select
<< reshow;
```

**Note:** When numbering Outline Boxes, it does not matter if the outline is closed or not. However, closed If Boxes do cause fragments of the tree to disappear entirely.

### Wildcards

You can use the wildcard character “?” to represent places in the search string where you want to match any sequence of any characters. This is an important technique for writing general scripts that would work with titles that vary according to the columns analyzed.

For example, you could use a script like this to invoke Distribution platform for any column starting with “he”.

```
dist<<Distribution(Y(\Column("he?")));
```

You could also use the wildcard character to find an outline box by a partial title match, and send the first outline box that matches a Close message:

```
Report(dist)[OutlineBox("he?")]<<Close(1);
```

### Sending Messages

The display reference can be used to send scripts to the display elements using the `Send` or `<<` operator. For example, if `r2` is a reference to an outline node, you could ask it to close itself:

```
r2<<close; //closes an outline node
```

**Close** toggles an outline node back and forth between closed and open states, just like clicking the triangle controls in the window. You can include a Boolean argument (1 or 0) for “close or leave closed” or “open or leave open.”

```
rbiv["Fit Mean"]<<close; //toggle
rbiv["Fit Mean"]<<close(1); //close or leave closed
rbiv["Fit Mean"]<<close(0); //open or leave open
```

You can use Select, Reshow, and Deselect to blink the selection highlight on a display box. Notice that you can string together several `<<` clauses to send an object several messages in a row. The results read left to right; for example, here Select is done first, then Reshow, then Deselect, then the other Reshow.

```
for(i=0,i<10,i++,
    db<<select<<reshow<<deselect<<reshow);
```
Learning What You Can do with a Display Box

To determine the messages that a given display box reference can understand, use Show Properties for the object. For example,

```plaintext
show properties(rbiv);
show properties(rbiv[picturebox(1)]);
```

These messages are the same as items in the context-menu for a given object, plus a few more. The next section, “Constructing Display Trees,” p. 190, discusses messages for each display box type in more detail. Show Properties also works with data tables and platforms; see “Learning About Data Tables’ Messages,” p. 94 in the “Data Tables” chapter, and “Learning the Messages an Object Responds to,” p. 162 in the “Scripting Platforms” chapter.

Find a Window Quickly

You can find a window that has gotten buried under others by sending a window message to it:

```plaintext
rbiv<<zoom window;
```

Customizing Reports

The Send To Report and Dispatch commands are used in tandem to customize the appearance of a report. For example, they are used to open and close outline nodes, resize graphics frames, or customize the colors in a graphics frame.

To see examples of Send To Report and Dispatch, run any analysis and change the default appearance of the report. Then, select Script > Save Script to Script Window to see the script.

Send To Report contains a list of commands to be sent to the display tree. In the example below, Send To Report contains two Dispatch commands.

Dispatch is used to send a command to a specific part of a display tree. It has four arguments. The first argument is a list of outline nodes that need to be traversed to find the desired part of the display tree. The second and third arguments work together. The second is the name of a display element, and the third is the display element’s type. These two arguments specify which particular part of the display tree is to be sent a command. The command to send is the fourth argument.

In essence, the Dispatch command specifies an outline node in the report (first argument), further specifies something underneath that outline node (second and third arguments), and specifies a command (fourth argument).

For example, open the Big Class sample data set and run the attached Bivariate script. This generates a report with a fitted line. Then, open the Lack of Fit outline node, and close the Analysis of Variance outline node. Finally, select Script > Save Script to Script Window. The following script appears. (spacing and line breaks are added here for illustration).

```plaintext
Bivariate(Y( :weight), X( :height), Fit Line( {Line Color( {213, 72, 87} )} ),
SendToReport(,
    Dispatch(\"Linear Fit\", \"Lack Of Fit\", OutlineBox, Close(0)),
    Dispatch(\"Linear Fit\", \"Analysis of Variance\", OutlineBox, Close(1))
)
```
The `Send To Report` command contains two `Dispatch` commands. These correspond to your two customizations to the default report. Examine the first `Dispatch` command in detail.

The first argument says to find an outline node named “Linear Fit”. The second and third commands say to further find an Outline Box named “Lack of Fit” underneath the “Linear Fit” outline. The fourth argument is the command to send to this outline box. In this case, the message is `Close(0)`, in other words, open the node.

Note: If there are several outline nodes with identical names, subscripts are assigned to them. For example, if you have a Bivariate analysis with two quadratic fits (resulting in identical titles), when you dispatch a command to the second fit, the subscript [2] is added to the duplicated title.

The best way to deal with `Send to Report` and `Dispatch` commands is to first run a report using the mouse, creating the customizations interactively. Then, examine the script that JMP generates. Remember: the best JSL writer is JMP itself.

**Using the << Operator**

Using the `<<` operator, messages can be sent to displays as they are being constructed, not just by sending messages to already-constructed displays. This lets you disambiguate between evaluating children arguments and option arguments. It also helps make it clearer which argument is an option, and which is a script to run inside the graph.

For example, before version 5, `HListBox` only accepted boxes in the argument list. Now, commands can be inserted in the list of boxes using the `<<` operator. For example, to insert a journal command in an `HListBox`, do the following:

```julia
New Window("Title",
    HListBox(
        ...
        <<journal,
        ...
    )
)
```

As another example, the `GraphBox` constructor accepts the usual named arguments like this:

```julia
NewWindow("Title",
    GraphBox(FrameSize(400,400),XScale(0,25),yScale(0,25),<<BackGroundColor("Red"), script...));
```

However, you can use the send operator to send commands to the `Frame Box` as an alternative to the named arguments.

```julia
NewWindow("Title",
    GraphBox(
        <<FrameSize(400,400),
        <<XAxis(0,25,add ref line(10)),
        <<YAxis(0,25),
```
Nesting and Precedence

When you stack commands, each command is evaluated left to right.

```
box<<command1<<command2<<command3;
```

sends command1 to box, then command2 to box, then command3 to box. Note that any of the commands can change box before the next command is sent.

```
( (box<<command1) <<command2) <<command3
```

sends command1 to box and gets the result. Command2 is sent to that result, and command3 is sent to the result of command2.

```
x = box<<command1<<command2<<command3;
```

The result of command3 is assigned to x. The first two commands are not assigned to x, although they might have changed box.

```
x = Text Box( "nothing" );
Print( x << settext( "the" )
   << settext( "first" )
   << settext( x << gettext() || "thing" )
   << gettext() );
```

"firstthing"

If you stack several commands, you might want to use parentheses to group the commands to be sure your script does what you want it to.

Platform Example

Here is a script to build an analysis report from start to finish using JSL. First, open a data table.

```
dt=open("$SAMPLE_DATA/Big Class.JMP");
```

Now launch a bivariate platform and assign it to the platform reference biv.

```
biv=bivariate(y(weight),x(height)); // a reference to the PLATFORM
```

To find out what you can do with the platform itself, use Show Properties on the platform object:

```
show properties(biv);
```

- **Show Points [Boolean] [Default On]**
- **Fit Mean [Action](Fits a flat line at the mean.)**
- **Fit Line [Action](Fits a regression line to the data.)**
- **Fit Polynomial [ActionChoice] {2,quadratic, 3,cubic, 4,quartic, 5, 6}**
- **Fit Special [Action](Fitting with transformations on X or Y, or constraints on slope or intercept.)**
- **Fit Spline [ActionChoice] {1000000, stiff, 100000, 10000, 1000, 100, 10, 1, 0.1, 0.01, flexible, Other...}{Fitting a flexible curve}**
Fit Each Value [Action](Fits a line that goes through the mean Y value of each set of unique X values.)
Fit Orthogonal [ActionChoice] {Univariate Variances, Prin Comp, Equal Variances, Fit X to Y, Specified Variance Ratio...}(Fitting where both X and Y have error.)
Density Ellipse [ActionChoice] {0.99, 0.95, 0.90, 0.50, Other...}(The bivariate normal contour fitted with the correlation.)

Note that this is a selected portion of the commands available to the Bivariate platform.

The output in the Log window gives a few ideas for messages to send the platform. (For more on this type of scripting, see the chapter “Scripting Platforms,” p. 153.)

biv<<Fit Spline(1000000)<<Fit Mean;
biv<<show points(0); //hide plotting symbols
biv<<show points(1); //reshow them
biv<<fit polynomial(4, 2); // degree 4 color 2
biv<<fit polynomial(2,4); //degree 2 color 4

Next, get the window to a comfortable size and scroll up to the top to see the graph you just tweaked.

biv<<size window(500,700);
biv<<scroll window({0,0});

Now get to work on the report. First, you need to create a reference, and then see what you can do with it.

rbiv=biv<<report; // a reference to the REPORT
show properties(rbiv);

The log lists the messages to use with reports (note that this is a selected portion of the commands available to the report):

Close [Boolean]
Horizontal [Boolean]
Open All Below [Action]
Close All Below [Action]
Open All Like This [Action]
Close All Like This [Action]
Close Where No Outlines [Action]
Show HelpKeys [Boolean]
DisplayBox [Subtable]
Select [Boolean]
Deselect [Boolean]
Reshow [Action] [Scripting Only]
Journal [Action]
Copy Picture [Action]
Page Break [Action]
Class Name [Action] [Scripting Only]
Child [Action] [Scripting Only]
Sib [Action] [Scripting Only]
Parent [Action] [Scripting Only]
Append [Action] [Scripting Only]
Get Text [Action] [Scripting Only]
Get HTML [Action] [Scripting Only]
Get RTF [Action] [Scripting Only]
Get Journal [Action] [Scripting Only]
Open the Fit Mean node of the outline:

```r
rbiv["Fit Mean"]<<close(0);
```

And practice selecting some results (submit each line alone to see its result):

```r
rbiv["Summary of Fit"]<<select;
rbiv["Parameter Estimates"]<<select;
rbiv["Analysis of Variance"]<<select;
```

//and dig way down in the outline tree:

```r
rbiv["Polynomial Fit Degree=2","Parameter ?", columnbox("Estimate")]<<select;
rbiv<<deselect;
```

To get the second Analysis of Variance item, you would do this:

```r
rbiv["Polynomial Fit Degree=2", "Analysis of Variance"]<<select;
```

Now change the format of one of the columns in the 4-degree polynomial Parameter Estimates report.

```r
pe=rbiv["Polynomial Fit Degree=4","Parameter ?"];
est=pe[columnbox("Estimate")];
est<<set format(12,6);
```

The first argument to `Set Format` sets the column width by the number of characters to display. The second sets how many decimal places are shown in the table.
**Figure 6.3 Applying Changes to a Report**

| Parameter Estimates | Term     | Estimate | StdError | t-Ratio | Prob>|T|
|---------------------|----------|----------|----------|---------|-------|
| intercept           | -17.50008| 84.25775 | 0.21     | 0.8368  |
| height              | 1.882428 | 1.348036 | 1.45     | 0.1558  |
| height-62.55^2      | -0.220179| 0.25906  | -0.73    | 0.4876  |
| height-62.55^3      | 0.070948 | 0.008533 | 1.86     | 0.0691  |
| height-62.55^4      | 0.007389 | 0.004231 | 1.75     | 0.0806  |

You can even get a single number out of the table. For example, the estimate for the cubic term:

```plaintext
terms=pe[columnbox("Term")];
for(i=1, i<10 & (terms<<get(i))!="(height-62.55)^3", i++, 0);
estimate = ests<<get(i);
0.0703948227446
```

How does this work? You use a `For`-loop to count down to the row for the term that you want. Recall that the second argument to `For` is a condition; as long as the condition tests true, looping continues. Here the test is "when the string in the Terms column is not ",(height-62.55)^3" and we have not reached the tenth row," so as soon as the string does match, looping stops and i's value is the number for the matching row. You then use i as a subscript to `Get` on the Estimates column.

This method enables you to use a given result as a parameter for further tests. In a process control situation, you might want to keep tabs on a particular result, triggering an e-mail message if it goes outside a certain range.

```plaintext
resume = Expr( /* some script to do next iteration */ );
message="Current estimate is " || char(estimate) || " on " || mdyhms(today());
if(estimate<1, resume, //else:
  mail("john.doe@company.com", "estimate exceeded limit", message));
```

You can also get values from boxes as a matrix, which you can then use for further computations or write to a data table. You can also make data tables directly:

```plaintext
myMatrix=rbiv[tablebox(4)] << get as matrix;
myVector=rbiv[tablebox(4)][columnbox("Sum of Squares")]<get as matrix;
dt<<new column("Sum of Squares",values(myVector));
rbiv[tablebox(4)] < make data table("ANOVA table");
```

Now adjust the scales on the axes.

```plaintext
rbiv[axisbox(1)]<min(70)<<max(170); // adjust Y axis
rbiv[axisbox(2)]<min(50)<<max(70); // adjust X axis
```
Continuing with this example, copy the graph at the top of the report. Note that you need to select the picturebox containing the graph; selecting just the graph would leave its axes behind.

\[ \text{rbiv}[\text{PictureBox(1)}] \ll \text{Copy Picture}; \]

You can also save the picture:

\[ \text{rbiv}[\text{PictureBox(1)}] \ll \text{save picture("myGraph.png",png); } \]
\[ \text{rbiv}[\text{PictureBox(1)}] \ll \text{save picture("myGraph.jpg",jpeg); } \]

Finally, journal the report:

\[ \text{rbiv} \ll \text{journal window}; \]

**Window**

Window finds a window by its title and returns a reference to that window. For example, to close a window titled "Analysis Results" submit the following code.

\[ \text{wdw = Window("Analysis Results");} \]
\[ \text{wdw} \ll \text{closeWindow();} \]

Window() with no arguments returns a list of all windows open in JMP. Window(n) returns the \( n \)th window. Window("title") returns the window with the specified title. If the \( n \)th window or a window with the specified title does not exist, an empty list is returned instead.

**How to Access Built-in Windows**

What if you just want your script to open a built-in window? Perhaps you want the user to locate a data table or launch a platform. To present a JMP window, simply omit required arguments from the JSL command for the task. Some examples:

\[ \text{dt} = \text{Open();} \quad // \text{File > Open window} \]
\[ \text{dt} \ll \text{Summary();} \quad // \text{Tables > Summary window} \]
\[ \text{myFit} = \text{Fit Y by X();} \quad // \text{Analyze > Fit Y by X launch window} \]
\[ \text{myChart} = \text{Chart();} \quad // \text{Graph > Chart launch window} \]

For modal windows such as Open, further script execution waits until the modal window is dismissed. Whenever JMP encounters a script to launch a platform that lacks column assignments and that requires column assignments, the platform's launch window appears, to get column assignments, but continues with the script immediately. That is, JMP does not wait for answers, as in the case of Open, and other modal dialog boxes. See the chapter “Scripting Platforms,” p. 153, for more about launching platforms.

**Using the Pick Windows**

You can prompt the user to pick a file or directory using the Pick File and Pick Directory commands. Both bring up a platform-specific window that has the user navigate to a file or directory and store the path as a string.

\[ \text{path} = \text{Pick Directory (<"prompt message">);} \]
\[ \text{path} = \text{Pick File(<"prompt message">, <"initial directory">, <filter list>);} \]
If no arguments are specified, the standard Open File and Select Folder windows appear. The quoted string Prompt Message defines the title of the window for Pick File. For Pick Directory, the string is used as the title of the directory list in the window.

You can also specify two other arguments for Pick File. The quoted string "Initial Directory" defines which folder the Open File window uses. If a directory is not defined, or if it is defined as an empty string, the default directory is used.

You can also define the filter used for the Open File window, forcing it to show only certain file types. This list must use the following syntax:

```
{"Label1|suffix1;suffix2;suffix3", "Label2|suffix4;suffix5"}
```

Each quoted string adds an entry to the File of Type menu in the File Open window. Label defines the text that is displayed for each menu option. The following list of suffixes defines the file types that are displayed if its corresponding label is selected. Note the use of * to list all files in the window.

```
Pick File(
   "Select JMP File",
   "$SAMPLE_DATA",
   {"JMP Files|jmp;jsl;jrn",
    "All Files|*"})
```

To obtain a list of filenames in a specified directory, use the Files In Directory command.

```
listOfFileNames = Files In Directory(path);
```

To obtain the full pathnames, use the concatenation operator:

```
listOfFileNames = path||Files In Directory(path);
```

The Files in Directory command accepts native and POSIX paths, as well as paths using path variables. It does not recurse by default, but does have an optional recursive argument.

```
main_samples = Files In Directory("$SAMPLE_DATA");
   {"Abrasion.jmp", "AdverseR.jmp", "Alcohol.jmp", ...} 
all_samples = Files In Directory("$SAMPLE_DATA", recursive);
   {..., "Templates/Polar Coordinates.jsl, "Therm.jmp, "Time Series/Air.jmp...}
```

## Constructing Display Trees

You can use constructor functions to construct your own display and install it in a window. This section shows how to put together displays and send messages to them and concludes with some examples.

### Basics

First you need to start a New Window starting with its title, and then you list the items to construct inside the window. All the Display Box constructors end in “Box”. See “Introduction to Display Boxes,” p. 177, to review what each type of display box looks like. You can nest display boxes inside each other as needed.
With an assignment, you can make a reference to a new window, so that you can send messages to it. (Throughout this document, \textit{db} is a placeholder for a display box reference, \textit{dt} for a data table reference, and \textit{obj} for a scriptable object reference.)

When display objects are created or referred to by JSL, they are freely shared references until they are copied into another display box or until you close the window and they disappear. When you plug a display object into another display tree, JMP makes a copy of it that the new box owns.

\textbf{Example}

This example uses \texttt{OutlineBoxes}, \texttt{HListBoxes}, and \texttt{Matrix Boxes} to assemble a partial cheat-sheet for matrix JSL.

\begin{verbatim}
for(i=1,i<12,i++,mcs[matrixbox(i)]<<set format(2,1));
\end{verbatim}
Figure 6.4 A Matrix Example

Observe how placing Matrix Boxes inside Outline Boxes is a convenient way to arrange items neatly. You could add more topics by adding more outline branches following this pattern.

Another example

Or, you could do some programming to avoid iterating Outline Boxes:

```plaintext
a=[0 1 2,3 4 5,6 7 8]; b=[8 7 6,5 4 3,2 1 0]; c=identity(3);
d=j(3,3,0); e=1::4; f=[+ -,- +];
mnames={a,b,c,d};
mnamesq={"a","b","c","d"};
cnames={Identity(3), J(3,3,0), e=1::4, f=[+ -,- +], Diag(a), VecDiag(a), a||b, a/b, a};
cnamesq={"Identity(3)", "J(3,3,0)", "e=1::4", "f=[+ -,- +]", "Diag(a)", "VecDiag(a)", "a||b", "a/b", "a"};
tnames={a', Cholesky(a+b), designf(a), directproduct(a, b), hdirectproduct(a, b), inverse(a-c), sweep(a), solve(a-c, b)};
tnamesq={"a'", "Cholesky(a+b)", "designf(a)", "directproduct(a, b)", "hdirectproduct(a, b)", "inverse(a-c)", "sweep(a)", "solve(a, b)"};

myOBoxM=expr(OutlineBox(mnamesq[i], matrixbox(Eval(mnames[i]))));
myOBoxC=expr(OutlineBox(cnamesq[i], matrixbox(Eval(cnames[i]))));
myOBoxT=expr(OutlineBox(tnamesq[i], matrixbox(Eval(tnames[i]))));

nw=New Window("Matrix Cheat Sheet",
HListBox(
   vb = vListBox(textBox(" "));
   for(i=1,i<=nitems(mnames),i++,vb<<append(myOBoxM));
   OutlineBox("Some matrices", vb),
   cb = vListBox(textBox(" "));
   for(i=1,i<=nitems(cnames),i++,cb<<append(myOBoxC));
   OutlineBox("Some constructors", cb),
   tb = vListBox(textBox(" "));
   nw=New Window("Matrix Cheat Sheet",
   HListBox(
      vb = vListBox(textBox(" "));
      for(i=1,i<=nitems(mnames),i++,vb<<append(myOBoxM));
      OutlineBox("Some matrices", vb),
      cb = vListBox(textBox(" "));
      for(i=1,i<=nitems(cnames),i++,cb<<append(myOBoxC));
      OutlineBox("Some constructors", cb),
      tb = vListBox(textBox(" "));
```
for(i=1,i<=nitems(tnames),i++,tb<<append(myOBoxT));
OutlineBox("Some transformations", tb));
for(i=1,i<=nitems(mnames)+nitems(tnames)+nitems(cnames),i++,nw[matrixbox(i)]<
<set format(5,2));

The result is too large to be shown here, so try this script yourself.

### Updating an Existing Display

Sometimes, you do not know how many display boxes will appear in a future report. For example, you might be writing a generic script that analyzes and reports on one or more variables. You do not know how many display boxes are needed, since the number of variables can change from one run of the script to the next.

#### Append, Prepend, and Delete

Use the `Append` message to add a display box to an existing display. In the script, construct a single, empty box, then `<<Append` boxes to it for each variable in the analysis.

The following code example assumes that there is a list of effect names in the variable `effectsList`, and that each one corresponds to a column in a matrix `varprop`. In other words, `effectsList[1]` is the label for `varprop[0,1]`; `effectsList[2]` is the label for `varprop[0,2]`; and so on.

First, an empty `Outline Box` containing an `HListBox` is made. The interior empty container is given the name `hb`:

```script
New Window( "H List Box Example",
    Outline Box( "Variance Proportions",
        hb = H List Box()
    )
);
```

Then, a `for` loop steps through the `effectsList` and adds a `Number Col Box` for each element of `effectsList`:

```script
for(i=1, i<=NItems(effectsList), i++,
    Eval(substitute(
        expr(hb << append(Number Col Box(effectslist[i], varprop[0,i]))),
        expr(i), i)
    )
);
```

The `Prepend` message works just like `Append`, but adds the item at the beginning of the display box rather than at the end. If the display box is one of several that do not allow appending, then it delegates the command to a child display box that can accept the command. It is fine to apply it to the top of the tree.

For example,

```script
biv = Bivariate(Y(height), X(weight), Fit Line);
(biv<<report)(Outline Box(1))<<Prepend(Button Box("Click Here for curve",
    biv<<Fit Polynomial(2)));
```
creates a Bivariate report with a least squares line. Clicking the button (which appears at the top of the report) adds a quadratic curve.

The same thing can be accomplished appending to the top of the tree:

\[\text{biv = Bivariate(Y(Height), X(Weight), Fit Line); (biv<<report)<<Prepend(Button Box("Click Here for curve", biv<<Fit Polynomial(2)))};\]

The \text{Delete} method for display boxes makes the display box and all its children vanish. This is useful with the \text{Append} and \text{Prepend} methods for building completely dynamic displays. In the example below, a text box is replaced with another text box. In this case, the script could have used \text{Set Text}, but many display boxes cannot change their content.

\[x=\text{New Window ("X", list = vListBox(}
\text{t1 = Text Box("t1"),}
\text{t2 = Text Box("t2")));}
\text{t1 << Delete; list << Append(t1 = Text Box ("t1new");}\]

You can use the \text{Sib Append} message to add a display element to an existing tree. Refer back to the display box tree shown in “To examine a report’s display box tree,” p. 179. Under \text{ListBox(2)}, you see two picture box trees. The one on the left (\text{PictureBox(1)}) holds the bivariate scatterplot. This is easily determined by looking at the bottom of the tree and seeing the two axis boxes holding the height and weight axes. \text{PictureBox (2)} holds the Fit Mean menu, determined by looking at the bottom of this branch and seeing the red line and the “Fit Mean” text box.

Suppose you wanted to insert a text box in between these two boxes. We want to append a sibling to \text{PictureBox (1)}, so we send it the \text{Sib Append} message:

\[\text{biv=Bivariate (Y(Height), X(weight), Fit Line); (biv<<report)[PictureBox(1)]<<SibAppend(TextBox("Hello There"));}\]
Chapter 6
Display Trees

Constructing Display Trees

Updating a numeric column

To update a numeric column in a display box table, use the Set Values command.

numColBox<<Set Values ([matrix])

The matrix argument specifies the new numbers for the table.

Controlling Text Wrap

Generally, JMP automatically wraps text within a Text Box. However, the default wrap point can be overridden with a Set Wrap (n) message, where n is the number of pixels to display before the wrap point.

You can also use bullet points with Bullet Point(1). Sending this message to a text box places a bullet in front of the text and indents subsequent lines within that text box.

Interactive Display Elements

JMP has three special types of display boxes that you do not normally see in a JMP platform: Button Box, Slider Box, and Global Box. These are useful for building custom windows with interactive graphics.

The chapter “Scripting Graphs,” p. 237, shows how Handle and MouseTrap both present controls inside the graph itself. You can also place button, slider, and edit-field controls outside the graph using Button Box, Slider Box, and Global Box. Each create separate display boxes that need to be combined in a new window according to the rules described in “Constructing Display Trees,” p. 190, but for basic effects, you can follow the pattern in the examples shown here and postpone the detailed discussion.

Slider Box draws a slider control for picking any value for the variable that you specify, within the range given by the min and max you specify. Any time the slider is moved, the value given by the current position of the slider is assigned to the global, and the graph updates accordingly. Thus, Slider Box is another way to parametrize a graph.

SliderBox(min, max, global, script, <set width(n)>, <rescale slider(min, max)>);

You can send Set Width and Rescale Slider as commands to a slider object. For example:

ex = .6;
New Window( "Example", mybox = Slider Box( 0, 1, ex, Show( ex ) ) );
mybox << Set Width( 200 ) << Rescale Slider( 0, 5 );

Button Box draws a button with the name that you specify. Any time the button is clicked, the script executes. The button stays alive and remains available for the duration of the window. You might want to use Button Box in combination with sliders, as shown here, or to provide a choice to update a graph to reflect changing data conditions.

ButtonBox("text",script expression);

You can also send a click() command to a button object at any time.

New Window("Hi", hello = Button Box("Hello", Print("hello")));
hello<<click(); //
Global Box shows the name and current value of a JSL global variable. The user can assign a new value to the global by editing the value directly in the window and pressing Enter or Return to commit the change. Graphs using the global automatically update for the new value. If you specify an expression, such as \( \sqrt{4} \), the global box first evaluates it and then stores and displays the result, 2.

```javascript
GlobalBox(global);
```

**Examples**

This example combines a graph with two sliders and a button by gluing the graph box, two horizontal boxes, and a button together in a vertical list box.

```javascript
// Slider LogNormal
lU=1; lS=2;
NewWindow("LogNormal Density",
  VListBox(
    gr=GraphBox(FrameSize(500,300), XScale(0.01,3), yScale(0,4),
      Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
      YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
    text({1,.5},"u ",lU," s ",lS)),
    HListBox(SliderBox(0,4,lU,gr<<reshow),TextBox("Mu")),
    HListBox(SliderBox(0,4,lS,gr<<reshow),TextBox("Sigma")),
    ButtonBox("Reshow",gr<<reshow));
  show(gr);
  gr<<reshow;

Figure 6.6 Example of Sliders and Buttons in a Report Window
```

This script is similar but uses `Global Box` instead of `Slider Box`:
Chapter 6
Display Trees

Constructing Display Trees

// Global LogNormal
lU=1; lS=2;
NewWindow("LogNormal Density",
    VListBox(
        gr=GraphBox(FrameSize(500,300),XScale(0.01,3), yScale(0,4),
            Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
            YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
            text({1,1},"u ",lU," s ",lS," or type new values below"),
            HListBox(GlobalBox(lU)),
            HListBox(GlobalBox(lS)));

Figure 6.7 Example of Using a Global Box Instead of Sliders

The example under “Drag Functions,” p. 270, shows another use of Button Box.

Modal and Non-Modal Windows

Non-modal windows, like the ones used for all JMP launch windows, are identical to reports and other display lists. Each of the following elements can be incorporated into them.

The Dialog() function, which creates modal windows, is deprecated and might not work at all in future versions of JMP. Use New Window() with the Modal argument.

If used in Dialog() instead of New Window(), the following functions cannot have the optional script attached to them. If you want an action associated with any of these controls, place a script as the control’s last argument. For example,

    comboObj = comboBox({"True", "False"}, <<set(1),print(comboObj<<Get))

prints the selected item number (a 1 or a 2 in this case, since there are two items in the combo box) each time the selection is changed.
A second difference between `Dialog()` and `New Window()` controls is in the way that items and strings are specified as arguments. For `Dialog()`, items are generally comma-separated. In `New Window()`, the items must be placed in a list. For example, compare the following two instances of a Combo Box control.

```plaintext
//modal Dialog
dlg = Dialog(
    cb = Combo Box( "True", "False" ), //note that the items are comma separated
    Button( "OK" )
);

//modal New Window
New Window( "Combo Box",
    <<Modal,
    cb = Combo Box( {"True", "False"} ), //note the items are in a bracketed list
    Button Box( "OK" )
);
```

Since both are modal windows, they stop the execution of the script, and the second window is not drawn until the first is dismissed.

The following controls are available for boxes in `New Window()`.

**Check Box**

Check Box ({"item 1", "item 2", ...}, <script>)

Any number of the items in the check box can be selected simultaneously.

**Radio Box**

Radio Box({"item 1", "item 2", ...}, <script>)

Only one item in the radio box can be selected at any time.

**Combo Box**

Combo Box({"item 1", "item 2", ...}, <script>)

Items in a combo box are drawn in a drop-down menu.

**List Box**

List Box({"item 1", "item 2", ...}, <width(n)>, <max selected(n)>, <nlines(n)>, <script>)

Width is measured in pixels. Max selected is the maximum number of items that might be selected in the list box. Nlines is the number of lines to display in the box, with a default value of 3.

You can add items to a list box by either using Append or Insert:

```plaintext
New Window( "test", lb = List Box( {"a", "e"} ) );
lb << append( {"f", "g"} ); // result is a, e, f, g
```
lb << Insert( {"b", "c", "a", "d"}, 1 ); // result is a, b, c, d, e, f, g

Append always adds the list to the end of the list box. Insert adds the list after the position specified.

**Button Box**

```
Button Box("text", <script>)
```

Draws a button containing text.

You can add a tooltip for your button by sending a Set Tip message to it. For example:

```
Button Box( "Hello", <<Set Tip( "World" ) )
```

creates a button named Hello that, when you hover over it with your mouse, displays a tooltip containing the word World.

Other messages for Button Box include <<Set Button Name("string") and <<Get Button Name.

You can also send the message Open Next Outline as a script command, which causes the next outline box to open. If you are sending more than one message to the button box, this must be the first command listed.

You can create a button that contains a menu:

```
New Window( "Test",
    bb = Button Box( "Select a Letter",
        choice = bb << Get Menu Choice;
        Show( choice );
    )
);
bb << Set Menu Items( {"a", "b", "-", "c"} );
```

The "-" in this example creates a menu separator. The separator counts as an item in the list. If you select c from the menu, 4 is returned, not 3.

**Popup Box**

```
Popup Box({"command1", script1, "command2", script2, ...})
```

Creates a red triangle menu. The following example stores a command list in a variable, and then uses Popup Box to display the items.

```
commandList = {
    "command1", print("command1"),
    "command2", print("command2"),
    "command3", print("command3"),
    "command4", print("command4"),
    ",", empty(), //makes a separator line
    "command5", print("command5"),
    "commandThrow", throw("commandThrow1"),
    "commandError", sqrt(1,2,3),
    "commandEnd", print("commandEnd")};
```
New Window("Test Popup",
   Text Box("Popup Test"),
   Popup Box(commandList);
);

Figure 6.8 Sample Red Triangle Menu

Note that you can also disable and re-enable the menu using the message <<enable(Boolean). An argument of 1 turns the menu on, and an argument of 0 turns the menu off. Using the previous example, you would assign the popup box to a variable, and then send messages to it:

New Window("Test Popup",
   Text Box("Popup Test"),
   mymenu = Popup Box(commandList);
);

mymenu << enable(0);

The red triangle is there, but the menu itself is disabled.

**Tab Box**

Tab Box("page title 1", contents of page 1, "page title 2", contents of page 2, ...

Draws a tabbed window pane.

Tab boxes can also be used in display trees, as in the following example.

NewWindow("test tabbed pages",
   TabBox("First Page",
      vlistBox(
         textBox("first line of first page"),
         textBox("Second line of first page")
      ),
   "Second page",
      vlistBox(
         textBox("first line of second page"),
         textBox("Second line of second page")
      ),
   "Third page",

...
vlistBox(
    textBox("first line of Third page"),
    textBox("Second line of Third page")
  );

Figure 6.9 Tab Boxes

You can specify which tab should be selected by sending `<<SetSelected(n)`, where n is the tab number, to the tab box object.

**Text Box**

```
Text Box("text")
```

Draws a non-editable text box. Text Boxes are frequently used as labels of other controls.

**Text Edit Box**

```
Text Edit Box ("text")
```

Draws an editable text box.

You can add a script to a Text Edit Box by sending it a script message. This is usually most convenient at the time the box is created. Simply add the script message as the last argument.

For example, the following script sends a message to the log each time the text edit box is changed.

```
New Window("Text Edit Box",
    TextEditBox("Change Me", <<Script(Print("Changed")))
)
```

By assigning a reference to the Text Edit Box object, its contents can be accessed. The following script echoes the value of the Text Edit box to the log each time it is changed.

```
New Window("Text Edit Box",
    teb=TextEditBox("Change Me", <<Script(Print(teb<<Get Text)))
)
```

**Passwords**

If you need to use a text edit box for your user to enter passwords, you can apply a style to the text edit box to replace all characters enters with an asterisk. For example, the following line creates a text edit box whose value is the string "a", but only an asterisk is shown in the text edit box.

```
q = Text Edit Box( "a", passwordstyle( 1 ), setscript( Print("changed!") ) )
```
When a user types a new string into the text edit box, each character is displayed as an asterisk, and the message "changed!" is printed to the log.

You can also send a text edit box a message to either use password style or to stop using password style.

```plaintext
q << passwordstyle( 1 ) // set the text edit box to password style
q << passwordstyle( 0 ) // set the text edit box to standard style
```

**Col List Box**

Col List Box (<All>, <width(n)>, <max selected(n)>, <nlines(n)>, <script>)

All specifies that all columns in the current data table should be included. Width is measured in pixels. Max selected is the maximum number of items that might be selected in the list box. Nlines is the number of lines to display in the box.

You can send a Get Items message to a col list box to retrieve a list of all columns selected. Here is an example script showing Get Items in use:

```plaintext
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Get Items Demonstration",
   H List Box( 
      chooseme = Col List Box( All, width( 100 ), nlines( 6 ) ),
      Lineup Box( 
         N Col( 1 ),
         Spacing( 3 ),
         Button Box( "Add Column >>",
            listocols << Append( chooseme << GetSelected );
            // Send Get Items to a Col List Box
            Chosen Columns = listocols << GetItems; ),
            Button Box( "<< Remove Column",
            listocols << Remove Selected;
            // Send Get Items to a Col List Box
            Chosen Columns = listocols << GetItems;
            ),
         // listocols is a Col List Box
         listocols = Col List Box( width( 100 ), nlines( 6 ) ),
      ),
      Text Box( " " ),
      // Show what Get Items returns
      stuff = Global Box( Chosen Columns )
   );
```

**Line Up Box**

Line Up Box (NCol(nc), <Spacing(pixels)>, displaybox args)

Display boxes specified in the displaybox arguments are drawn in nc columns. Optional spacing can be specified, in pixels, for the space between columns.
Border Box

Border Box (Left(pix), Right(pix), Top(pix), Bottom(pix), Sides(int), displaybox)

Used to add space around the displaybox argument. Left, Right, Top, and Bottom add space around the displaybox argument. Sides draws border around the box, as described in Table 6.2. Additional effects can also be applied to the borders using Sides, as described in Table 6.3. To add both an effect and a border, add the two numbers.

For example, this code produces a text box with a border at the top and bottom (Draw Border value of 5) and a white background (Effect value of 32):

```new window( "Borders",
    border box( sides( 37 ), text box( "Hello World!" ) )
);
```

Table 6.2 Border Box Values

<table>
<thead>
<tr>
<th>Number</th>
<th>Draw border</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Top</td>
</tr>
<tr>
<td>2</td>
<td>Left</td>
</tr>
<tr>
<td>3</td>
<td>Top and Left</td>
</tr>
<tr>
<td>4</td>
<td>Bottom</td>
</tr>
<tr>
<td>5</td>
<td>Top and Bottom</td>
</tr>
<tr>
<td>6</td>
<td>Left and Bottom</td>
</tr>
<tr>
<td>7</td>
<td>Top, Left, and Bottom</td>
</tr>
<tr>
<td>8</td>
<td>Right</td>
</tr>
<tr>
<td>9</td>
<td>Top and Right</td>
</tr>
<tr>
<td>10</td>
<td>Left and Right</td>
</tr>
<tr>
<td>11</td>
<td>Top, Left, and Right</td>
</tr>
<tr>
<td>12</td>
<td>Bottom and Right</td>
</tr>
<tr>
<td>13</td>
<td>Top, Bottom, and Right</td>
</tr>
<tr>
<td>14</td>
<td>Left, Bottom, and Right</td>
</tr>
<tr>
<td>15</td>
<td>Top, Left, Bottom, and Right</td>
</tr>
</tbody>
</table>
Panel Box

Panel Box ("title", displaybox args)

Encloses the displaybox argument in a labeled border.

Journal Box

The Journal Box function, like other functions ending in Box, constructs a DisplayBox appropriate for gluing together with other Display Boxes to create a display in a window.

The usage is:

```
box = JournalBox("journal text")
```

where "journal text" is text that has been extracted from a journal file.

Since journal text has lots of rules about what boxes can be with other boxes, we recommend that the only way that you obtain journal text is to highlight an area, use the Journal command to make a journal containing only that item, save it. Now open the file in a text editor (you might have to change the file extension to do this). Then paste it into your script as the Journal Box argument. We highly recommend that you use the "\[ ... \]" quoting mechanism so that you do not have to escape double quotes within the journal text.

Another way to get journal text is to send <<GetJournal to displayBoxes.

Below is an example that makes a mosaic plot:

```
NewWindow("MosaicPlot",
TextBox("Here is a mosaic Plot"),
JournalBox("\[ //Note the quoting mechanism here
PictureBox(sub( 
BorderBox(top(12), left(5), bottom(5), right(7), sides(143), options(0), 
xmin(0), ymin(0), sub( 
ScaleBox(ID(2), axis(scaleType(0), scaleOrig(0), scaleWidth(1), 
widthMajor(0.25), 
nbin(4), nminor(0), timeCode(0), ndec(3), ndecSpec(0),
```

<table>
<thead>
<tr>
<th>Add to Number Above</th>
<th>Additional Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Make the border the highlight color as defined in Preferences. The default is blue.</td>
</tr>
<tr>
<td>32</td>
<td>Make the borderbox's background white.</td>
</tr>
<tr>
<td>64</td>
<td>Erase the background of the border box's container.</td>
</tr>
<tr>
<td>128</td>
<td>Draw the border in an embossed style.</td>
</tr>
</tbody>
</table>

Table 6.3 Border Box Additional Effects
Chapter 6
Display Trees

Constructing Display Trees

MinInit(0), MaxInit(1), LinearInit, MajorInit(0.25), MinorInit(0),
NObsInit(0),
options(showMajorTicks, showMinorTicks, showLabels, fixMinimum, fixMaximum),
length(180), sub(  
ScaleBox(ID(1), length(200), sub(  
ListBox(horizontal, near, sub(  
ListBox(horizontal, near, sub(  
CenterBox(vert, sub(  
TextEditBox("age", left))),  
AxisBox(side(R), size(33, 180), locked(false), scales(0, 2, 0, 2), )))},  
ListBox(vertical, near, sub(  
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0),  
ymin(0), sub(  
FrameBox(size(200, 180), border(0), flags(0), markerSize(-1), help  
name("Conting Mosaic"), scales(1, 2, 1, 2), seg(  
MosaicSeg(  
um x(2), num y(6), totals(18, 22),
cross tabs(0.277777, 0.444444, 0.72222, 0.83333, 0.944444, 1, 0.136366,  
0.31818, 0.63636, 0.863636, 0.909090909090909, 1), sum weight(40), ycolors(5, -2142812212, -2138140980,  
-2134077810, -2134096057, 3), vertical ))))),  
NomAxisBox(size(200, 36), sizeID(1, 0), num labels(2), labels(F, M), value(18,  
22), total(40), horizontal, left),  
CenterBox(horiz, sub(  
TextEditBox("sex"))),  
NomAxisBox(size(29, 180), sizeID(0, 2), num labels(6), labels("12", "13",  
"14", "15", "16", "17"), value(8, 7, 12, 7, 3, 3), total(40), vertical,  
left),  
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0),  
ymin(0), sub(  
FrameBox(size(10, 180), border(0), flags(0), markerSize(-1), help  
name("Conting Mosaic Single"), scales(0, 2, 0, 2), seg(  
MosaicSeg(  
um x(1), num y(6), totals(40),
cross tabs(0.2, 0.375, 0.675, 0.85, 0.925, 1),  
sum weight(40), ycolors(5, -2142812212, -2138140980, -2134077810, -2134096057, 3), vertical)  
))))))))))))))))
) //End of quoting mechanism here

Complete Example

The following script generates a sample of many controls illustrated above. The Big Class sample data table is open.

New Window("Window Controls",  
Line Up Box (NCol(2), Spacing(3),  
Panel Box("Panel Box",  
Check Box({"Check Box 1", "Check Box 2"}),  
Radio Box({"Radio Box 1", "Radio Box 2"}),
Display Trees

Chapter 6

Constructing Display Trees

```plaintext
Combo Box(\"Combo Box\")
Button Box(\"Button Box\")
List Box(\"List Box 1\", \"List Box 2\")
Col List Box(all)
```

**Figure 6.10** Example of Many Interactive Display Elements

---

**Getting and Setting values of interactive display elements**

You can use \$\langle\rangle\langle Set Selected (item number, state)\rangle\$ command to pre-select an item. You can use this command separately to a saved display box reference, or you can specify it inline as a list box \$\langle\rangle\$ argument.

To retrieve the selected value, use \$\langle\rangle\langle Get Selected\rangle\$, which returns the value of the selected item. \$\langle\rangle\langle Get Selected Indices\rangle\$ returns the index number of the selected item.

Two examples follow to illustrate all three commands.

```plaintext
antennaList = \{"Dish","Helical","Polarizing","Radiant Array\"\};
//method 1
New Window("Test List",
  listObj = List Box (antennaList, print("iList",
    listObj<<getSelected, listObj<<getSelectedIndices))
); listObj<<Set Selected(2, 1);
//method 2
New Window("Test List",
  listObj = List Box (antennaList, \$\langle\rangle\langle Set Selected(2, 1)\rangle\$, print (\"iList\",
    listObj<<getSelected, listObj<<getSelectedIndices))
)
```

**Advanced Example**

The following example code creates a simplified replica of the Cluster platform launch window, which then launches the platform with the given arguments. Note that some functions (Recall and Help) are not
implemented in the script, so an alert window is shown when they are clicked. In addition, switching from hierarchical to k-means clustering does not change anything like the real Cluster launch window does.

```plaintext
// Launch Window for Cluster Platform

dt = currentDataTable(); nc = ncol(dt);
lbWidth = 130;
methodList ={"Average","Centroid","Ward","Single","Complete"};
notImplemented = expr(New Window("Feature Not Implemented Yet",<<Modal,
       ButtonBox("OK")));

clusterDlg = NewWindow("Clustering",
 BoostBox(left(3),top(2),
   VListBox(
      TextBox("Finding points that are close, have similar values"),
      ListBox(
        PanelBox("Select Columns",
          colListData=ColListBox(All,width(lbWidth),nLines(min(nc,10)))),
        PanelBox("Options",VListBox(
          comboObj=comboBox("Hierarchical","K-Means",<<Set(1)),
          PanelBox("Method",
            methodObj=RadioBox(methodList,<<Set(3))
          ),
          checkObj=checkBox("Standardize Data",<<Set(1,1))
        )
      ),
      PanelBox("Cast Selected Columns into Roles",
        LineupBox(ncol(2),Spacing(3),
          ButtonBox("Y, Columns",
            colListY<<Append(colListData<<GetSelected)),
          colListY = ColListBox(width(lbWidth),nLines(5),numeric),
          ButtonBox("Ordering",
            colListO<<Append(colListData<<GetSelected)),
          colListO = ColListBox(width(lbWidth),nLines(1),numeric),
          ButtonBox("Label",
            colListL<<Append(colListData<<GetSelected)),
          colListL = ColListBox(width(lbWidth),nLines(1)),
          ButtonBox("By",
            colListB<<Append(colListData<<GetSelected)),
          colListB = ColListBox(width(lbWidth),nLines(1))
        ),
      PanelBox("Action",
        LineupBox(ncol(1),
          ButtonBox("OK",
            if ((comboObj<<Get)==1,
              HierarchicalCluster(
                Y(Eval(colListY<<GetItems)),
                Order(Eval(colListO<<GetItems))
              ),
              KMeansCluster(
                Y(Eval(colListY<<GetItems)),
                C(Eval(colListO<<GetItems))
              )
            )))
    )
```
Figure 6.11 The Cluster Launch Window
Send Messages to Constructed Displays

If you assign a construction to a name, that name becomes a reference to the window, which in turn owns the display boxes inside it. Using subscripts, you can then send messages to the display boxes inside the window.

For example, the graphing section shows how to make an interactive sine wave graph. This example automates the interaction, as it were, by sending messages to the frame box inside the window (note the assignment to $tf$).

```julia
amplitude = 1; freq = 1; phase = 0;
t = New Window( "Wiggle Wave",
   Graph Box(FrameSize(500,300),X Scale(-5,5),Y Scale(-5,5),Double Buffer,
      Y Function(amplitude*Sine(x/freq+phase),x);
      Handle(phase,amplitude,phase=x;amplitude = y);
      Handle(freq,.5,freq=x);
      Text({3, 4},"amplitude: ",Round(amplitude,4),
            {3, 3.5},"frequency: ",Round(freq,4),
            {3, 3},"phase: ",Round(phase,4));
   )
   tf = t[framebox( 1 )];
   For(amplitude=-4,amplitude<4,amplitude+=.1,tf << reshow);
```

Use For loops for more complex movement:

```julia
amplitude = 1; freq = 1; phase = 0;
for(i=0,i<1000,i++,
   amplitude+=(Random Uniform()-.5);
   amplitude = if(amplitude>4,4,amplitude<-4,-4,amplitude);
   freq += (random uniform()-.5)/20;
   phase+=(Random Uniform()-.5)/10;tf<<reshow; Wait(0);
)
```

Build Your Own Displays from Scratch

This script uses the Summarize operator to collect summary statistics on the Height column of Big Class and then uses display box constructors to show the results in a nicely formatted window.

```julia
dt=Open("$SAMPLE_DATA/Big Class.jmp");
summarize( a=by(age), c=count,
   sumHt=sum(Height), meanHt=mean(Height),
   minHt=min(Height), maxHt=max(Height));
sr=NewWindow("Summary Results",
   TableBox(
      stringColBox("Age",a),
      NumberColBox("Count",c),
      NumberColBox("Sum",sumHt),
      NumberColBox("Mean",meanHt),
      NumberColBox("Min",minHt),
      NumberColBox("Max",maxHt)));
```

This produces a window called “Summary Results” like this:
Constructing Display Trees

You can use the usual commands for display boxes:

    show properties(sr);
    sr<<journal;

### Construct Display Boxes Containing Platforms

Another type of display that you might want to construct is simply your own combination of results from the analysis platforms in JMP. Simply script the platform inside a display box, assemble the display boxes into a window, and for ease in routing messages to it later, assign the whole thing to a reference.

This example assembles a “six-pack” of capability analysis tests for a convenient QC overview:

```julia
    csp=NewWindow("Capability Sixpack",
    OutlineBox("Capability Sixpack",
        HListBox(
            VListBox(cc=Control Chart(chart Col(Height, Individual Measurement, Moving Range),K Sigma(3))),
            VListBox(dist=Distribution(columns(Height)))));
```
Now you can work with the window by sending messages to the reference \texttt{csp}. This is a display box reference, whose capabilities are similar to those of a \texttt{Report} for a platform. You can use multiple-argument subscripting to locate specific items within the outline tree:

\begin{verbatim}
csp["Control ?", "moving range ?"]<<close;
csp["Dist?","quantiles"]<<close;
\end{verbatim}

Notice that this script not only assigned the whole window to a reference but also assigned the platform-launch scripts to names (\texttt{cc} and \texttt{dist}) within their display boxes. This makes it easy to route messages to the platforms. You could in turn get the reports for these and have yet another way to manipulate display boxes. The following are equivalent messages that reopen the nodes:

\begin{verbatim}
rcc=cc<<report; rdist=dist<<report;
rcc["moving range ?"]<<close;
rdist["quantiles"]<<close;
\end{verbatim}

Naturally, you can send messages directly to the platform references themselves. First find out your options:
show properties(cc);
show properties(dist);

The log shows choices matching the pop-up menus for each platform, as usual. To execute choices from JSL, just direct them as messages to the platform references:

cc<<needle; dist<<normal quantile plot;

---

**Construct a Custom Platform**

An example in “Manipulating expressions,” p. 416 in the “Advanced Concepts” chapter, showed how to the `SubstituteInto` function to plug coefficients for a quadratic polynomial into the quadratic formula and then use the formula to calculate the roots of the polynomial. That example required specifying the coefficients as arguments to `SubstituteInto`.
The section “Modal Windows,” p. 218 shows an example to collect coefficients from the user using a modal dialog box.

This section further develops the example into a complete customized platform that first puts up a dialog box to ask for coefficients, finds the roots, and then displays the results along with a graph in a custom window.

```plaintext
//First, open a window to collect coefficients from the user:
myCoeffs = New Window( "Find the roots for the equation",
  <<Modal,
   H List Box( 
     a = Number Edit Box( 1 ),
     Text Box( "*x^2 + " ),
     b = Number Edit Box( 2 ),
     Text Box( "*x + " ),
     c = Number Edit Box( 1 ),
     Text Box( " = 0" )
   ),
   Button Box( "OK",
     a = a << get;
     b = b << get;
     c = c << get;
     Show( a, b, c );
   ),
   Button Box( "Cancel" )
 );

/* Second, calculate the results: The quadratic formula is x=(-b + - sqrt(b^2 - 4ac))/2a. Plug the coefficients into the quadratic formula: */
x = {Expr( (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a) ), Expr( (-b - Sqrt( b ^ 2 - 4 * a * c )) / (2 * a) )};
//Store the solution list:
xx = Eval Expr( x );

/* Third, test whether real roots were found and make an appropriate display. 
If yes (for example, with the window's defaults), show roots and a graph: */
results = Expr( 
  xmin = xx[1] - 5;
  xmax = xx[2] + 5;
  ymin = -20;
  ymax = 20;

  myResult = New Window( "The roots of a quadratic function",
    V List Box( 
      Text Box( "The real roots for the equation ",
      Text Box(" || Expr( po ) || " = 0" ),
      H List Box( Text Box( "are x=") , Text Box( xxx ) ),
      Text Box( " " ), // to get a blank line
      Graph Box( 
        frameSize( 200, 200 ),
        X Scale( xmin, xmax )
      )
    )
  );
```
Display Trees

Chapter 6

Constructing Display Trees

Y Scale( ymin, ymax ),
Line Style( 2 ),
H Line( 0 ),
Line Style( 0 ),
Y Function( polynomial, x ),
Line Style( 3 ),
Pen Color( 3 ),
V Line( xx[1] ),
V Line( xx[2] ),
Marker Size( 2 ),
Marker( 0, {xx[1], 0}, {xx[2], 0} )
)
)

/* If no (for example, with a=3, b=4, c=5), put up an error window with a
helpful graph: */
error = Expr( New Window( "Error",
    V List Box( Text Box( " " ),
        Text Box( "Polynomial " || po || " has no real roots. " ),
        Text Box( " " ),
        Text Box( "Examine a graph of the function to get an idea why." ),
        Graph Box( framesize( 200, 200 ),
            X Scale( -20, 20 ),
            Y Scale( -20, 20 ),
            Line Style( 2 ),
            H Line( 0 ),
            Line Style( 0 ),
            Y Function( polynomial, x )
        )
    )
)

/* Either way, the script needs to have some strings ready. Rewrite the
polynomial with the coefficients specified: */
polynomial = Expr( a * x ^ 2 + b * x + c );
//Store this instance of the polynomial as a string:
po = Char( Eval Expr( polynomial ) );
//Store the solution list as a string:
xxx = Char( Eval Expr( x ) );
//Now it's ready for the test:
    error,
    results
)

When you run this script, you first see a window like this:
And then you get a results window, with either the roots or an error message.

Figure 6.16  Example: The Custom Platform’s Report

Sheets

Sheet boxes let you create a grid of plots. H Sheet Box and V Sheet Box contain display boxes and arrange them in columns and rows. The general approach is to first consider what display boxes that you want, and in what arrangement. Then, create either an H or V Sheet Box and send it a Hold message for each plot. Finally, create interior H or V Sheet Boxes and tell each one which plot it should hold.

Here is an example of creating a sheet with four plots: a bivariate plot, a distribution, a tree map, and a bubble plot.

First, open the data table and create a new window.

```julia
Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Example",
```

Use a V Sheet Box to organize the window into two columns.

```julia
  V Sheet Box(
```

Send it four Hold messages, one for each plot. The order matters.

```julia
  <<Hold(Bivariate( // Plot 1
```

...
Display Trees

Chapter 6

Constructing Display Trees

Y( :weight ),
X( :height ),
Fit Line(),

<<Hold(Distribution( // Plot 2
  Continuous Distribution(
    Column( :height ),
    Horizontal Layout( 1 ),
    Outlier Box Plot( 0 )
  )
)),

<<Hold(Tree Map(Categories( :age ))), // Plot 3
<<Hold(Bubble Plot( // Plot 4
  X( :height ),
  Y( :weight ),
  Sizes( :age ),
  Coloring( :sex ),
  Circle Size( 6.226 ),
  All Labels( 0 )
)),

Finally, add two H Sheet Boxes to the V Sheet Box and tell each one which plot it should hold. Each H Sheet Box holds two side-by-side plots. They are held by a V Sheet Box, so the H Sheet Boxes are displayed vertically.

H Sheet Box(
  Sheet Part("",
    Excerpt Box( 1, {Picture Box( 1 )} )
  ),
  Sheet Part("Distribution of height",
    Excerpt Box( 2, {Picture Box( 1 )} )
  )
),
H Sheet Box(
  Sheet Part("",
    Excerpt Box( 3, {Picture Box( 1 )} )
  ),
  Sheet Part("My Title Here",
    Excerpt Box( 4, {Picture Box( 1 )} )
  )
),

);
Finally, a note on the titles for Sheet Part. If you include an empty string as the title, the sheet part's title is the default report title. For example, “Bivariate Fit of weight By height”. You can define your own title by enter your title as the string. For example, “My Title Here”.

**Note:** You must include a title, either an empty string or a character string. This argument is mandatory. If you want a blank title, use a string of one or more spaces.

**Journals**

It is relatively easy to put things into a journal window. It it more difficult to see how to manipulate the journal itself. The following examples show some journal manipulations.

Suppose you start with a report, generated in this case from the Big Class.jmp data table.

```julia
biv=bivariate (y(weight), x(height));
riv=biv << report;
```

To journal the report results, use the journal window message.

```julia
riv<<journal window;
```

To save the journal to a file,

```julia
riv<<Save Journal("Macintosh HD:users:username:Documents:test.jrn");
```

To save the journal as HTML,

```julia
riv<<Save HTML("Macintosh HD:users:username:Documents:test.htm");
```

(Note that the above examples use Macintosh standards for filenames. Windows users should adapt the examples to use filename conventions appropriate for your operating system or use the POSIX operating system, which is respected on all platforms.)

When using a By variable in a script, the result is a list of references to the analysis results for each By group. In order to journal all By member parts of the report, you need to use parent messages to get to the top of the report.

For example, the following code creates a Bivariate report with a By group and then journals the entire report:

```julia
biv=bivariate (y(weight), x(height),by(Sex));
((report(biv[1]) << parent) << parent) << save journal("test.jrn");
```

**Picture Display Type**

JSL has a Picture data type, used to store pictures of JMP output or formulas. You can take a picture of anything in a displaybox, or create a picture of a text formula as it would look in the Formula Editor.

To create picture data, send a Get Picture message to a displaybox.

```julia
displaybox<<Get Picture;
```
The function `Expr As Picture` evaluates its argument and creates a picture of the expression, using the same formatting mechanism as the formula editor. If you have a literal expression as the argument, remember to enclose it in `Expr()` so that JMP just takes a picture of the result, rather than evaluate the expression.

For example,

```js
New Window("Formula", Expr As Picture(expr(a+b*c+r/d+exp(x))));
```

Once you have the picture, there are two ways of using it.

1. Incorporate it into a new Display tree, using a `displayBox` constructor.
2. Write it to a file using `Save Picture`.

```js
picture<<Save Picture("path", type)
```

where `type` can be `WMF` (Windows), `EMF` (Windows), `PICT` (Macintosh), `BMP` (Windows), `JPEG` or `JPG`, or `PNG`.

---

**Modal Windows**

JSL supports both modal and non-modal windows.

- Modal windows must be answered immediately. Clicking outside the windows produces an error sound. Script execution stops until the user responds to the window.
- Non-modal windows are similar to JMP reports. They do not have to be answered immediately. Platform Launch windows are non-modal.

**Note:** The `Dialog()` function is deprecated and might not work at all in future versions of JMP. Use either `New Window()` with the `Modal` argument, or `Column Dialog()` for column selection.

Using `New Window()` instead of `Dialog()` means that you can use standard display box constructors, instead of the specialized constructors for `Dialog()`.

**Constructing Modal Windows**

When you submit a script with a modal window, JMP draws the window, waits for the user to make choices and click `OK`, and then stores a list of the variables with their values. `Modal` means that the user has no choice but to respond to the window. Any attempt to click outside the window produces an error sound, and script execution is suspended until the user clicks `OK` or `Cancel`.

The `Column Dialog` function is specifically intended to prompt users to choose columns from the current (topmost) data table. Windows created by `Column Dialog` are also modal windows.

A few tips for using modal windows in scripts:

1. Put all the modal windows near the beginning of the script, if possible. This way, all the user interaction can be accomplished at once, and then users can leave JMP to finish its work unattended.
2. Make sure your modal windows give the user enough information. Do not just present a number field. Tell users how the number is used. If there are limits for valid responses, say so.

**General-Purpose Modal Window**

A very simple modal window might request a value for one variable:

```julia
New Window( "Set a Value",
<<Modal,
  Text Box("Set this value"),
  variablebox = Number Edit Box( 42 ),
  Button Box( "OK"),
  Button Box( "Cancel"))
```

Notice that the argument to `EditNumber` is the default value, 42. Also notice that a dialog box without at least one `Button` makes no sense, so JMP adds a button if you script a modal window without one.

---

**Figure 6.17** Sample Modal Dialog Box

![Sample Modal Dialog Box](image)

If you click **OK**, the dialog box closes, it returns `{Button(1)}`, and the script continues. To reference the number set, use `variablebox<<get`.

If you click **Cancel**, the dialog box closes, it returns `{Button(-1)}`, and the script continues. See “Throwing and Catching Exceptions,” p. 387 in the “Advanced Concepts” chapter, if you need a way to cancel out of a script.

**Comparing Dialog and New Window**

The following two code example produce the same window using the older `Dialog()` method and the recommended `New Window()` using the `Modal` argument. Although `New Window()` takes more work to get the user's input, you can construct a modal window just like any other window. You also have more control over the contents and the function of the window.

**Dialog**

```julia
Dialog(
  HList(
    VList(
      "Radio Frequency Embolism Projection",
      Lineup( 2,
        "Lower Spec Limit", lsl = EditNumber( 230 ),
    )))
)```
"Upper Spec Limit", usl = EditNumber( 340 ),
"Threshold", threshold = EditNumber( 275 )
),
HList(
  VList(
    "Type of Radio",
    type = RadioButtons( "RCA", "Matsushita", "Zenith", "Sony" )
  ),
  VList(
    "Type of Antenna",
    antenna = RadioButtons( "Dish", "Helical", "Polarizing", "Radiant Array" )
  ),
synch = Check Box( "Emission Synchronization", 0 ),
"Title for plot",
title = EditText( "My projection" ),
HList(
  "Quality",
  quality = Combo Box( "Fealty", "Loyalty", "Piety", "Obsequiousness"
)
),
VList( Button( "OK" ), Button( "Cancel" )
);

New Window
New Window( <<Modal,
  H List Box(
    V List Box(
      Text Box( "Radio Frequency Embolism Projection" ),
      Lineup Box(2,
        Text Box( "Lower Spec Limit" ),
lsl_box = Number Edit Box( 230 ),
        Text Box( "Upper Spec Limit" ),
        usl_box = Number Edit Box( 340 ),
        Text Box( "Threshold" ),
        threshold_box = Number Edit Box( 275 )
      ),
    H List Box(
      V List Box(
        Text Box( "Type of Radio" ),
        rb_box1 = Radio Box( {"RCA", "Matsushita", "Zenith", "Sony"} )
      ),
      V List Box(
        Text Box( "Type of Antenna" ),
        rb_box2 = Radio Box( {"Dish", "Helical", "Polarizing", "Radiant Array"} )
      )
    )
  )
);
cb_box1 = Check Box( "Emission Synchronization", 0 ),
Text Box( "Title for plot" ),
title_box = Text Edit Box( "My projection" ),
H List Box(  
Text Box( "Quality" ),
\cb_box2 = Combo Box( "Fealty", "Loyalty", "Piety", "Obsequiousness"
)
V List Box( Button Box( "OK" ), Button Box( "Cancel" ) )
)

lsl = lsl_box << get;
usl = usl_box << get;
threshold = threshold_box << get;
radio_type = rb_box1 << get;
antenna = rb_box2 << get;
synch = cb_box1 << get;
title = title_box << gettext;
quality = cb_box2 << get;

Figure 6.18 Results from Dialog (left) and New Window (right)

The OK button in Dialog() returns all the variables that you set, while the OK button in New Window() returns only the status of the button.
Data Column Dialog Boxes

Column Dialog, a variant of the Dialog command, lets you prompt for column selections from the current data table or contextual data table, which must already be open.

For example:

```jl
dt = Open( "$SAMPLE_DATA/Big Class.JMP" );
r = Column Dialog(
    Col ID = ColList( "X, Treatment", Max Col( 1 ) ),
    Group = ColList( "Group Factors" ),
    Split = ColList( "Y, Response" ),
    w = ColList( "Weight" ),
    HList( "Alpha", alpha = EditNumber( .05 ) )
);
```

Figure 6.19 Column Dialog

This example returns a list similar to this one, depending on the user's choices:

```
{Col ID = {}, Group = {}, Split = {}, w = {}, alpha = 0.05, Button( -1 )}
```

For each destination list, a ColList clause must be a direct argument of ColumnDialog (not nested inside some other argument). An optional MaxCol(n) argument restricts the number of data columns that can be chosen to n. The resulting list contains the “name” list that is enclosed in parentheses. Lists are always returned, although they can sometimes be empty lists. You can include as many as twelve ColList clauses.

Other items permitted in a Dialog command are permitted in Column Dialog also, and have the same functionality. The OK, Cancel, and Remove buttons and the list of columns to choose from are both added automatically.
You can specify the minimum and maximum number of columns that are allowed in a column dialog box with the \texttt{MinCol} and \texttt{MaxCol} parameters. You can also specify the modeling type of the columns that are allowed to be selected. You can set the width of the list using \texttt{Select List Width(pixels)} parameter. To set the width of the column list, use \texttt{Width(pixels)} inside the \texttt{Col List()} function.

For example, the following code generates a column dialog box that only allows the selection of exactly one numeric column.

```
rt_Dlg = Column Dialog(
    cv = ColList( "Response To Test", MaxCol( 1 ), MinCol( 1 ), DataType( Numeric ) )
);
```

\textbf{Figure 6.20} Restricting Selection of Columns

The \texttt{DataType} choices are \texttt{Numeric}, \texttt{Character}, and \texttt{RowState}.

In addition, use the \texttt{Columns} specification to pre-fill some column selections. For example,

```
dlg = Column Dialog(
    xCols = ColList( "X, Factors", Columns( :height ) ),
    yCols = ColList( "Y, Response", Columns( :weight, :age ) )
);
```

assigns \texttt{height} to the \texttt{X} role and \texttt{weight} and \texttt{age} to the \texttt{Y} role.

\section*{Arrangement}

\textbf{Note:} This section pertains only to the deprecated \texttt{Dialog()} function and the \texttt{Column Dialog()} function. \texttt{New Window()} creates modal windows using the standard display box constructors.

JMP automatically creates a dialog box window of the appropriate size and shape and arranges items in the order specified.

Items can be grouped and aligned by using \texttt{HList} and \texttt{VList} containers. An \texttt{HList} top-aligns and spaces its contents in a horizontal row. \texttt{HListing} a pair of \texttt{VLists} produces a top-aligned, spaced pair of columns. A \texttt{VList} left-aligns and spaces its contents in a vertical column. \texttt{VListing} a pair of \texttt{HLists} produces a left-aligned, spaced pair of rows.

\texttt{Line Up} arranges items in the number of columns that you specify. JMP automatically figures out the proper spacing.
Display Trees
Modal Windows

Dialog(
   VList(
      LineUp(2,
         "Set this value", variable=EditNumber(42),
         "Set another value", var2=EditNumber(86)),
      HList(Button("OK"), Button("Cancel")))
);

Figure 6.21 Default Dialog Arrangements for Windows (left) and Macintosh (right)

{variable = 42, var2 = 86, Button(1)}

Note that JMP does exert some control over OK and Cancel button positions to ensure that dialog boxes are consistent with what the operating system (Windows or Macintosh) expects. In certain cases, JMP needs to override your HList, VList, and LineUp settings for Button(OK) and Button(Cancel). Do not be alarmed if the result is slightly different from what you expect.

Assign Dialogs to Variables

Note: This section pertains only to the deprecated Dialog() function and the Column Dialog() function. A variable for a New Window() function contains a reference to the window itself.

If you assign the Dialog() function to a variable, JMP stores the list returned by the Dialog() function in that variable. Calling that variable calls the list, not the Dialog().

   myValues=Dialog(HList(
      VList(
         "Lower Spec Limit", lsl=EditNumber(230)),
      VList(
         Button("OK"),
         Button("Cancel"))));
   myValues; // returns the list of values

If you want to store the Dialog() function itself for later use, quote it with Expr.

   myDialog=expr(Dialog(HList(
      VList(
         "Lower Spec Limit", lsl=EditNumber(230)),
      VList(
         Button("OK"),
         Button("Cancel")))));
   myDialog; // draws the dialog box to gather values
Unload Results from Column Dialogs

Note: This section pertains only to the deprecated Dialog() function and the Column Dialog() function.

Column Dialog() returns a list, and values are not set directly.

```plaintext
result=dialog(
  v list(  
    line up (2,
      "Alpha (0-1)", a=edit number (0.05),
      "Sigma (0-5)", sd=edit number (1),
      "Effect (0-5)", eff=edit number (2),
      "Sample (2-100)", n=edit number (2)),
  h list(  
    button("Cancel"),
    button("OK"))));

{a = 0.03, sd = 2, eff = 2, n = 50, Button(1)}
```

When you are ready to use some of the values, you have to unload them from the list returned:

```plaintext
sd = result["sd"];
eff = result["eff"];
```

Use EvalList to evaluate an entire list of assignments all at once:

```plaintext
RemoveFrom(result,5); // since Button() is undefined
EvalList(result);
```

Constructing Dialogs and Column Dialogs

Table 6.4 shows describes the display box constructors used in Dialogs and Column Dialogs. Note the following:

- A Column Dialog automatically includes the list of columns in the data table.
- Both windows automatically include an OK button if no buttons are defined.
- A Column Dialog also automatically includes Cancel and Remove buttons.

Table 6.4 Dialog and Column Dialog Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog</td>
<td>Dialog( contents of window )</td>
<td>Draws a modal window that includes any of the items listed in this table. The contents must be separated from each other by commas, rather than semicolons.</td>
</tr>
<tr>
<td>Column Dialog</td>
<td>Column Dialog( contents of window )</td>
<td>Draws a modal window for the user to make column role assignments. The contents must be separated from each other by commas, rather than semicolons.</td>
</tr>
</tbody>
</table>
### Table 6.4 Dialog and Column Dialog Constructors (Continued)

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Col List</strong></td>
<td>( \text{var=} \text{Col List}( \text{&quot;role&quot;}, \text{&lt;MaxCol}(n)&gt;, \text{&lt;Datatype}(\text{type})&gt; ) )</td>
<td>Available only for Column Dialog(). Creates a selection destination with a role button; the user's choice are returned in a list item of the form ( \text{var}=[\text{choice 1}, \text{choice 2}, \ldots, \text{choice n}] ). You can specify the minimum or maximum number of columns using MaxCol(( n )) or MinCol(( n )). You can specify the required data type of the column using Datatype(type). The choices for type are Numeric, Character, or Rowstate.</td>
</tr>
<tr>
<td><strong>HList</strong></td>
<td>( \text{HList}( \text{item}, \text{item}, \ldots ) )</td>
<td>Top-aligns and spaces the items in a horizontal row. Placing a pair of VLists within an HList produces a top-aligned, spaced pair of columns.</td>
</tr>
<tr>
<td><strong>VList</strong></td>
<td>( \text{VList}( \text{item}, \text{item}, \ldots ) )</td>
<td>Left-aligns and spaces the items in a vertical column. Placing a pair of HLists within a VList produces a left-aligned, spaced pair of rows.</td>
</tr>
<tr>
<td><strong>Line Up</strong></td>
<td>( \text{Line Up}(n, \text{item}_1, \text{item}_2, \ldots, \text{item}_n) )</td>
<td>Lines up the items listed in ( n ) columns, where ( \text{item}_{ij} ) is the ( j )th item of the ( i )th row.</td>
</tr>
<tr>
<td><strong>Button</strong></td>
<td>( \text{Button( \text{&quot;OK&quot;} )}, \text{Button( \text{&quot;Cancel&quot;} )} )</td>
<td>Draws an OK or a Cancel button. If OK is clicked, Button(1) is returned. If Cancel is clicked, Button(-1) is returned.</td>
</tr>
<tr>
<td><strong>string</strong></td>
<td>&quot;string&quot;</td>
<td>Draws text in the window. For example, can include a labeling string before an Edit Number field. All strings must be quoted.</td>
</tr>
<tr>
<td><strong>Edit Number</strong></td>
<td>( \text{var=} \text{Edit Number}( \text{number} ) ), ( \text{var( Edit Number(\text{number}) )} )</td>
<td>Produces an edit field for a number with number as the default value. When OK is clicked, the number entered in the field is assigned to the variable. Use var( Edit Number( ... ) for Column Dialog()). Either syntax works for Dialog().</td>
</tr>
</tbody>
</table>
Scripting the Script Editor

Even the script editor window is a display tree in JMP, which means you can write a JSL script to write and save another JSL script.

There is no `New Script` command. Instead, to open a new script window, you use the `New Window` command and then send it a message to tell it that it's a script window:

```julia
ww = New Window("Window Title", <<Script, "Initial Contents");
```

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Text</td>
<td><code>var=Edit Text( &quot;string&quot;, &lt;width(x)&gt; )</code></td>
<td>Produces an edit field for a string with <code>string</code> as the default value. You can also specify the minimum width of the box in pixels. The default width is 72 pixels. When <code>OK</code> is clicked, the text entered in the field is assigned to the variable. Use <code>var(Edit Text(...))</code> for <code>Column Dialog()</code>. Either syntax works for <code>Dialog()</code>.</td>
</tr>
<tr>
<td>Radio Buttons</td>
<td><code>var=Radio Buttons( &quot;choice1&quot;, &quot;choice2&quot;, ... )</code></td>
<td>Produces a vertical, left-justified list of radio buttons with the choices specified. The first choice is the default. When <code>OK</code> is clicked, the button that is selected is assigned to the variable. Choices must evaluate to quoted text strings. Use <code>var(RadioButtons(...))</code> for <code>Column Dialog()</code>. Either syntax works for <code>Dialog()</code>.</td>
</tr>
<tr>
<td>Check Box</td>
<td>`var=CheckBox(&quot;Text after box&quot;, &lt;1</td>
<td>0&gt; )`</td>
</tr>
<tr>
<td>Combo Box</td>
<td><code>var=ComboBox( &quot;choice1&quot;, &quot;choice2&quot;, ... )</code></td>
<td>Produces a menu with the choices listed. The first choice is the default. Choices evaluate to quoted text strings. Choices can also be inside a list. Use <code>var(ComboBox(...))</code> for <code>Column Dialog()</code>. Either syntax works for <code>Dialog()</code>.</td>
</tr>
</tbody>
</table>
The last argument is optional. If you include a string, the new script window contains that string.

In the New Window example above, \texttt{ww} is a reference to the DisplayBox that is the entire window. To write to a script window, you first need to get a reference to the part of the display box that you can write to, which is called a ScriptBox:

\begin{verbatim}
ed = ww[scriptbox(1)];
\end{verbatim}

Using the reference \texttt{ed}, you can add text, remove text, get the text that is already there.

\begin{verbatim}
ed << get text();
"Initial Contents"
ed << set text("aaa=3;\!N");
ed << append text("bbb=1/10;"); ed << append text("\!Nccc=4/100;");
ed << get line text(2); ed << set line text(2, "bbb = 0.1;");
ed << get line count(); ed << get lines();
ed << reformat();
ed << run();
ww << close window(nosave);
\end{verbatim}

Table 6.5 contains a summary of display boxes that appear in reports. The JSL function to create each display box is listed for those that can also be constructed. Many display boxes cannot be constructed through JSL, but are still part of the display tree that can be accessed through JSL.
Table 6.5 Display Boxes and Display JSL Functions

<table>
<thead>
<tr>
<th>Box Type</th>
<th>JSL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis Box</td>
<td></td>
<td>Contains the axis: tick marks, tick labels, and axis label.</td>
</tr>
<tr>
<td>Border Box</td>
<td><code>Border Box(&lt;Left(pix)&gt;, &lt;Right(pix)&gt;, &lt;Top(Pix)&gt;, &lt;Bottom(Pix)&gt;, &lt;Sides(0)&gt;, db)</code> on p. 486 in the &quot;JSL Syntax Reference&quot;</td>
<td>A container that can be used to add space on one, two, three, or all four sides.</td>
</tr>
<tr>
<td>Button Box</td>
<td><code>Button Box(title, &lt;&lt;Set Icon(&quot;path&quot;), script)</code> on p. 486 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a button with the <code>title</code> and executes the <code>script</code> when the button is clicked.</td>
</tr>
<tr>
<td>Cat Axis Box</td>
<td></td>
<td>An axis box for a categorical axis.</td>
</tr>
<tr>
<td>Cell Plot Box</td>
<td></td>
<td>Contains a cell plot.</td>
</tr>
<tr>
<td>Center Box</td>
<td></td>
<td>Centers the display box that it contains.</td>
</tr>
<tr>
<td>Check Box</td>
<td><code>Check Box(list, &lt;script&gt;)</code> on p. 487 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box to show one or more check boxes.</td>
</tr>
<tr>
<td>Check Box Box</td>
<td></td>
<td>Contains a single check box within the check box.</td>
</tr>
<tr>
<td>Col List Box</td>
<td>`Col List Box(&lt;all&gt;, &lt;width(pix)&gt;, &lt;maxSelected(n)&gt;, &lt;nlines(n)&gt;, &lt;script&gt;, &lt;MaxItems(n)&gt;, &lt;MinItems(n)&gt;, &lt;character</td>
<td>numeric&gt;, &lt;On Change(expr)&gt;)` on p. 487 in the &quot;JSL Syntax Reference&quot;.</td>
</tr>
<tr>
<td>Combo Box</td>
<td><code>Combo Box(list, &lt;script&gt;)</code> on p. 488 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box to show a combo box with a menu.</td>
</tr>
</tbody>
</table>
Table 6.5 Display Boxes and Display JSL Functions (Continued)

<table>
<thead>
<tr>
<th>Box Type</th>
<th>JSL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context Box</td>
<td>Context Box(displayBox, ...) on p. 488 in the &quot;JSL Syntax Reference&quot;</td>
<td>Defines a scoped evaluation context. Each Context Box is executed independently of each other. Creates an Eval Context Box.</td>
</tr>
<tr>
<td>Crosstab Box</td>
<td></td>
<td>Container for a contingency table.</td>
</tr>
<tr>
<td>Display 3D Box</td>
<td></td>
<td>Container for a three-dimensional Scene Box.</td>
</tr>
<tr>
<td>Excerpt Box</td>
<td>Excerpt Box(report, subscripts) on p. 489 in the &quot;JSL Syntax Reference&quot;</td>
<td>Returns a display box containing the excerpt designated by the report held at number report and the list of display subscripts subscripts. The subscripts reflect the current state of the report, after previous excerpts have been removed.</td>
</tr>
<tr>
<td>Expr As Picture</td>
<td>Expr As Picture(expr(...)) on p. 489 in the &quot;JSL Syntax Reference&quot;</td>
<td>Converts expr() to a picture as it would appear in the Formula Editor. Creates a Pict Box.</td>
</tr>
<tr>
<td>Frame Box</td>
<td></td>
<td>Contains a graphics frame.</td>
</tr>
<tr>
<td>Global Box</td>
<td>Global Box(global) on p. 489 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a box to display the current value of the global variable; the value is directly editable, and editing the value automatically forces graphs to update.</td>
</tr>
<tr>
<td>Graph 3D Box</td>
<td>Graph 3D Box(properties) on p. 490 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box with 3-D content. Creates a Scene Box.</td>
</tr>
<tr>
<td></td>
<td>Graph Box(properties, script) on p. 490 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a graph with axes. Might create display boxes such Axis Box and Frame Box.</td>
</tr>
<tr>
<td></td>
<td>H Center Box(display box) on p. 490 in the &quot;JSL Syntax Reference&quot;</td>
<td>Returns a display box with the display box argument centered horizontally with respect to all other sibling display boxes. Creates a Center Box.</td>
</tr>
<tr>
<td></td>
<td>H List Box(display box, ...) on p. 490 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a display box that contains other display boxes and displays them horizontally. Creates a List Box.</td>
</tr>
<tr>
<td>Box Type</td>
<td>JSL Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>H Sheet</td>
<td>Box]&lt;&lt;Hold(report), display boxes) on p. 490 in the &quot;JSL Syntax Reference&quot;</td>
<td>Returns a display box that arranges the display boxes provided by the arguments in a horizontal layout. The &lt;&lt;Hold() message tells the sheet to own the report(s) that are excerpted.</td>
</tr>
<tr>
<td>HierBox</td>
<td>Hier Box(&quot;text&quot;, Hier Box(...), ...) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a node of a tree (similar to Diagram output) containing text. Hier Box can contain additional Hier Boxes, allowing you to create a tree. The text can be a Text Edit Box.</td>
</tr>
<tr>
<td>Icon Box</td>
<td>Icon Box(name) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box containing an icon, where the quoted string name can be the name of any JMP icon. For example, Icon Box(&quot;Nominal&quot;) constructs a display box that contains the Nominal icon.</td>
</tr>
<tr>
<td>If Box</td>
<td>If Box(Boolean, display boxes) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box whose contents are conditionally displayed.</td>
</tr>
<tr>
<td>Journal Box</td>
<td>Journal Box(&quot;Journal Text&quot;) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box that displays the quoted string journal box. We recommend that you do not generate the journal text by hand.</td>
</tr>
<tr>
<td>Line Up Box</td>
<td>Lineup Box(&lt;NCol(n), &lt;Spacing(pixels)&gt;, display boxes, ...) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box to show an alignment of boxes in n columns.</td>
</tr>
<tr>
<td>List Box</td>
<td>List Box(OperItem(&quot;item&quot;, ...), &lt;width(pixels)&gt;, &lt;maxSelected(n)&gt;, &lt;nLines(n)&gt;, &lt;script&gt;) on p. 491 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box to show a list box of selection items.</td>
</tr>
<tr>
<td>List Box Box</td>
<td>Created by Col List Box().</td>
<td></td>
</tr>
<tr>
<td>Matrix Box</td>
<td>Matrix Box(matrix) on p. 492 in the &quot;JSL Syntax Reference&quot;</td>
<td>Displays the matrix given in the usual array form.</td>
</tr>
</tbody>
</table>
### Table 6.5 Display Boxes and Display JSL Functions (Continued)

<table>
<thead>
<tr>
<th>Box Type</th>
<th>JSL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Col Box</td>
<td>Number Col Box(&quot;title&quot;, numbers) on p. 492 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a column named <em>title</em> with numeric entries given in list or matrix form.</td>
</tr>
<tr>
<td>Number Col Edit Box</td>
<td>Number Col Edit Box(&quot;title&quot;, numbers) on p. 492 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a column named <em>title</em> with numeric entries given in list or matrix form. The numbers can be edited.</td>
</tr>
<tr>
<td>Number Edit Box</td>
<td>Number Edit Box(value) on p. 492 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates an editable number box that initially contains the <em>value</em> argument.</td>
</tr>
<tr>
<td>Outline Box</td>
<td>Outline Box(&quot;title&quot;, display box, ...) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a new outline named <em>title</em> containing the listed display boxes.</td>
</tr>
<tr>
<td>Page Break Box</td>
<td>Page Break Box() on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a display box that forces a page break when the window is printed.</td>
</tr>
<tr>
<td>Panel Box</td>
<td>Panel Box(&quot;title&quot;, display box) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a display box labeled with the quoted string <em>title</em> that contains the listed display boxes.</td>
</tr>
<tr>
<td>Pict Box</td>
<td>Picture Box(picture) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Contains a picture object. Created by any function that places a picture into a report window (for example, Picture Box()).</td>
</tr>
<tr>
<td>Plot Col Box</td>
<td>Plot Col Box(title, numbers) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Returns a display box labeled with the quoted string <em>title</em> to graph the <em>numbers</em>. The numbers can be either a list or a matrix.</td>
</tr>
<tr>
<td>Popup Box</td>
<td>Popup Box(&quot;command1&quot;, script1, &quot;command2&quot;, script2, ...) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a red triangle menu. The single argument is an expression yielding a list of an even number of items alternating between the command string and the expression that you want evaluated when the command is selected. If the command is an empty string, a separator line is inserted.</td>
</tr>
</tbody>
</table>
### Table 6.5 Display Boxes and Display JSL Functions (Continued)

<table>
<thead>
<tr>
<th>Box Type</th>
<th>JSL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio Box</td>
<td>Radio Box({&quot;item&quot;,...}, &lt;script&gt;) on p. 493 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs a display box to show a set of radio buttons. The optional script is run every time a radio button is selected.</td>
</tr>
<tr>
<td>Scene Box</td>
<td>Scene Box(x size, y size) on p. 494 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates an x by y-sized scene box for 3-D graphics.</td>
</tr>
<tr>
<td>Script Box</td>
<td>Script Box(&lt;script&gt;, &lt;width&gt;, &lt;height&gt;) on p. 494 in the &quot;JSL Syntax Reference&quot;</td>
<td>Constructs an editable box that contains the quoted string script. The editable box is a script window and can both be edited and run as JSL.</td>
</tr>
<tr>
<td>Scroll Box</td>
<td>Scroll Box(&lt;size(h,v)&gt;, &lt;flexible(Boolean)&gt;, displayBox, ...) on p. 494 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a display box that positions a larger child box using scroll bars.</td>
</tr>
<tr>
<td>Sheet Box</td>
<td></td>
<td>Container for one or more Sheet Panel Boxes.</td>
</tr>
<tr>
<td>Sheet Panel Box</td>
<td></td>
<td>Container for a single panel of a sheet layout.</td>
</tr>
<tr>
<td></td>
<td>Sheet Part(title, display box) on p. 494 in the &quot;JSL Syntax Reference&quot;</td>
<td>Returns a display box containing the display box argument with the quoted string title as its title.</td>
</tr>
<tr>
<td></td>
<td>Slider Box(min, max, global, script, &lt;set width(n)&gt;, &lt;rescale slider(min, max)&gt;) on p. 495 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates an interactive slider control.</td>
</tr>
<tr>
<td>Spacer Box</td>
<td>Spacer Box(&lt;size(h,v)&gt;, &lt;color(color)&gt;)) on p. 495 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a display box that can be used to maintain space between other display boxes, or to fill a cell in a LineUp Box.</td>
</tr>
<tr>
<td>String Col Box</td>
<td>String Col Box(&quot;title&quot;, {&quot;string&quot;, ...}) on p. 495 in the &quot;JSL Syntax Reference&quot;</td>
<td>Creates a column in the table containing the string items listed.</td>
</tr>
<tr>
<td>Box Type</td>
<td>JSL Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>String Col Edit Box</td>
<td><code>String Col Edit Box(&quot;string&quot;, ...) on p. 495 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Creates a column in the table containing the string items listed. The string boxes are editable.</td>
</tr>
<tr>
<td>Tab Box</td>
<td><code>Tab Box(&quot;page title1&quot;, contents of page 1, &quot;page title 2&quot;, contents of page 2, ...) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Creates a tabbed window pane. The arguments are an even number of items alternating between the name of a tab page and the contents of the tab page. Creates a Tab List Box.</td>
</tr>
<tr>
<td>Table Box</td>
<td><code>Table Box(display box, ...) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Creates a report table with the display boxes listed as columns.</td>
</tr>
<tr>
<td>Tab List Box</td>
<td></td>
<td>Contains Tab Pane boxes. Created by Tab Box.</td>
</tr>
<tr>
<td>Tab Pane Box</td>
<td></td>
<td>Contains the display boxes shown on a single tab pane in a tab box.</td>
</tr>
<tr>
<td>Text Box</td>
<td><code>Text Box(&quot;text&quot;, &lt;arguments&gt;) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Constructs a box that contains the quoted string text.</td>
</tr>
<tr>
<td>Text Edit Box</td>
<td><code>Text Edit Box(&quot;text&quot;, &lt;script&gt;) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Constructs an editable box that contains the quoted string text. The optional script argument attaches a script to the text box, either by adding the script as a second argument, or by sending the Set Script message.</td>
</tr>
<tr>
<td>V Center Box</td>
<td><code>V Center Box(display box) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Returns a display box with the display box argument centered vertically with respect to all other sibling display boxes. Creates a Center Box.</td>
</tr>
<tr>
<td>V Sheet Box</td>
<td><code>V Sheet Box(&lt;Hold(report), display boxes) on p. 496 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Returns a display box that arranges the display boxes provided by the arguments in a vertical layout. The &lt;&lt;Hold() message tells the sheet to own the report(s) that is excerpted.</td>
</tr>
<tr>
<td>Web Browser Box</td>
<td><code>Web Browser Box(url) on p. 497 in the &quot;JSL Syntax Reference&quot;</code></td>
<td>Creates a display box that contains a web page.</td>
</tr>
</tbody>
</table>
Table 6.6 Subscripts for a Display Box

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>$db[&quot;text&quot;]$</td>
<td>Finds the outline in $db$ that has the title $text$. Note that $text$ must be the complete title, not just a substring of the title, but you can use ? as a wildcard character with a substring to match the rest of the title. For example, &quot;$? Estimates&quot; to find &quot;Parameter Estimates&quot;.</td>
</tr>
<tr>
<td></td>
<td>$db[Outline Box(&quot;text&quot;)]$</td>
<td>Finds the outline box containing the $text$.</td>
</tr>
<tr>
<td></td>
<td>$db[Column Box(&quot;name&quot;)$</td>
<td>Finds the column box containing the $text$.</td>
</tr>
<tr>
<td></td>
<td>$db[boxType(n)]$</td>
<td>Finds the $n$th display box of type $boxType$.</td>
</tr>
<tr>
<td></td>
<td>$db[arg1, arg2, arg3, ...]$</td>
<td>Matches the last argument to a display box that is contained by the penultimate argument’s outline node, which is in turn contained by the antepenultimate argument, and so on. In other words, it is a way of digging down the generations of an outline tree to identify a nested display box.</td>
</tr>
</tbody>
</table>
Display Trees
Syntax Reference
You can run a script inside a graph, which draws inside the graph. You can do this with almost any graph from an analysis platform, or you can create your own new graphs. In both cases, you are storing a JSL script inside a graph, and the script runs each time you display the graph.

For scripting 3-dimensional plots, see the chapter “Three-Dimensional Scenes,” p. 273.
Contents

Adding Scripts to Graphs ........................................... 239
Ordering Graphics Elements Using JSL ............................ 240
Adding a Legend to a Graph ........................................ 240
Creating New Graphs From Scratch ............................... 241
Making Changes to Graphs ........................................ 242
Graphing Elements .................................................. 243
  Plotting Functions ................................................ 243
  Getting the Properties of a Graphics Frame .................... 249
  Adding a Legend ............................................... 249
Drawing Lines, Arrows, Points, and Shapes ...................... 249
  Lines ......................................................... 249
  Arrows ..................................................... 251
  Markers .................................................... 252
  Pies and Arcs .............................................. 254
Regular Shapes: Circles, Rectangles, and Ovals ............... 255
Irregular Shapes: Polygons and Contours ....................... 258
Adding text ....................................................... 260
Colors .......................................................... 261
Transparency ...................................................... 263
Fill patterns ...................................................... 264
Line types ....................................................... 264
Drawing With Pixels .............................................. 265
Interactive graphs ................................................. 266
Handle .......................................................... 266
MouseTrap ....................................................... 269
Drag Functions ................................................... 270
Troubleshooting .................................................. 272
Adding Scripts to Graphs

If you context-click (right-mouse-button) on the graphics frame, you can enter or paste JSL commands. The script usually contains drawing commands that run in the context of the graphics frame. For example:

1. Open the Big Class.jmp data table.
2. Select Analyze > Fit Y by X.
3. Choose Height for X and Weight for Y, and click OK.
4. Right-click inside the graph.
5. Select Customize from the context menu.
6. Click the plus button to add a new graphics script.
7. Type this text and click OK.

```javascript
text({55,160},"Hello World");
```

Now the graph has a text element at the graph's x-coordinate 55 and y-coordinate 160.

---

**Figure 7.1** Adding a Script to a Graph Interactively

By default, graphics scripts appear on top of the data.

For example, add this line to the graph:

```javascript
Fill Color("Green"); Rect(57, 175, 65, 110, 1);
```

A solid green rectangle appears. The list of scripts shows the order in which each script is drawn, so the first item on the list is drawn first. If you want the rectangle behind everything, move it to the top. If you want it on top of everything, move it to the bottom. You can arrange all the scripts in a graph into the drawing order that you prefer. Any new script is initially added directly after the item in the list that is selected.

**Tip:** To use a script that references a column name, use `Column(colname)` or a colon (`:colname`) to scope it properly.
Adding a Legend to a Graph

Interactively, you add a legend using the **Row Legend** command. To accomplish this through JSL, send a **Row Legend** message to the display. In this message, specify which column you want to base the legend on, and whether the legend should affect colors and markers.

For example, using **Big Class**, submit the following JSL to turn on a legend based on the age column, setting both colors and markers by values in the age column.

```julia
biv = Bivariate( Y( :height), X( :weight ));
```

---

**Scripting Graphs**

**Chapter 7**

**Adding a Legend to a Graph**

**Hint:** To see the JSL for the above actions, select **Script > Save Script to Script Window** from the red triangle menu.

**Ordering Graphics Elements Using JSL**

You can also add graphics elements using JSL instead of doing so interactively:

```julia
frame box <<Add Graphics Script(order, script)
```

When you add graphical elements using JSL, they are drawn on top of whatever is already in the graph. The optional *order* argument specifies in what order to draw the graphics element. *Order* can be either the keyword *Back* or an integer that specifies drawing order for a number of graphics elements. For example, if you add an oval to a scatterplot, the oval is drawn on top of the markers. The keyword *Back* causes the oval to be drawn in the background.

To specify the drawing order for a number of graphics elements, use an integer for the *order* argument to determine where each is drawn in relation to the others. The following script first adds a blue oval behind the points in the plot, and then adds a red oval in front of the blue oval, but still behind the points.

```julia
dt = Open( "$SAMPLE_DATA/big class.jmp" );

op = dt << Overlay Plot(
    X( :height ),
    Y( :weight ),
    Separate Axes( 1 )
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
    1,
    Fill Color( "Blue" );
    Oval( 60, 140, 65, 90, 1 );
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
    2,
    Fill Color( "Red" );
    Oval( 50, 120, 65, 100, 1 );
);
```

---

**Adding a Legend to a Graph**

Interactively, you add a legend using the **Row Legend** command. To accomplish this through JSL, send a **Row Legend** message to the display. In this message, specify which column you want to base the legend on, and whether the legend should affect colors and markers.

For example, using **Big Class**, submit the following JSL to turn on a legend based on the age column, setting both colors and markers by values in the age column.

```julia
biv = Bivariate( Y( :height), X( :weight ));
```
Creating New Graphs From Scratch

Graphics scripts are set up within the `Graph Box` command within a `New Window` command.

```
New Window("window title", <Editable|Dialog>, Graph Box( named arguments,..., script));
```

There are two optional keywords after the window name. `Editable` treats the window like a Journal window (so report items can be dragged and dropped onto them). `Dialog` treats the window like a modal window, so it shows up with a gray background on the Macintosh.

`Graph Box` takes named arguments for:

- `FrameSize(horizontal, vertical)`, // size in pixels
- `XScale(xmin, xmax), YScale(ymin, ymax)`, // range of x, y axes
- `X Axis(messages), Y Axis (messages)`, // the usual axis controls
- `Double Buffer` // for smooth animation
- `XName, YName` // names for x, y axes

Here is a script that plays a smoothed random walk around the frame. Because this uses random numbers, your display might differ.

```
New Window( "Smoothed Random Walk",
Graph Box(
    FrameSize( 200, 200 ),
    X Scale( -20, 20 ),
    Y Scale( -20, 20 ),
    x = 0;
    y = 0;
    xi = 0;
    yi = 0;
    For( i = 0, i < 2000, i++,
        xi = .9 * xi + Random Normal() / 10;
        yi = .9 * yi + Random Normal() / 10;
        xx = x + xi;
        yy = y + yi;
        xx = If( 
            xx < -20, -20,
            xx > 20, 20,
            xx
        );
        yy = If(
            yy < -20, -20,
            yy > 20, 20,
            yy
        );
    
    
    ));
```

Report( biv )[Frame Box(1)] << Row Legend( “age”, color(1), marker(1));

The `color()` and `marker()` arguments are optional. By default, they mimic the behavior of the window: colors are on by default, and markers are off.

To use a continuous scale if your variable is nominal or ordinal, use the `Continuous Scale(1)` option with the `color(1)` option.
Scripting Graphs

Creating New Graphs From Scratch

```
yy > 20, 20,
    yy
); Line( {x, y}, {xx, yy} );
    x = xx;
    y = yy;
)

Figure 7.2 Creating a Graph

Making Changes to Graphs

You can also make changes to graphs through scripting. For example, create a window with a graph and get a reference to the report:

```julia
open("$SAMPLE_DATA/Big Class.jmp"); biv = Bivariate( Y( :weight ), X( :height ), Fit Line ); rbiv = biv<<report;
```

Using subscripting, you can send messages to any part of that report. Use **Show Tree Structure** to discover how to reference them. See “To examine a report's display box tree,” p. 179 in the “Display Trees” chapter for details about the tree structure. See “Subscripts,” p. 181 in the “Display Trees” chapter on using subscripts.

This line re-sizes the graph to 400 pixels by 400 pixels:

```
rbiv[frame box(1)]<<frame size(400,400);
```

To see a list of possible messages for any given display box object, use **Show Properties**. For example, here is a partial list of messages that you can send to an axis:

```
Show Properties(rbiv[axis box(1)]);
```

**Axis Settings [Action]** *(Bring up the Axis window to change various settings.)*
**Revert Axis [Action]** *(Restore the settings that this axis had originally.)*
**Add Axis Label [Action]**
Remove Axis Label [Action]
Save To Column Property [Action] (Save the Axis settings as an Axis property in the data column associated with this axis.)
Set Width [Action] [Scripting Only]
Axis [Subtable] [Scripting Only]
Min [Numeric]
Max [Numeric]
Inc [Numeric]
Tick Font [Action]
Interval [Enum] [Scripting Only] {Numeric, Year, Month, Week, Day, Hour, Minute, Second}

For example, to change the font of both axis labels (weight and height in the example above, which are both text edit boxes attached to the axis boxes) to 12-point, italic Arial Black:

```
rbin[Text Edit Box( 1 )] << set font( "Arial Black" );
rbin[Text Edit Box( 1 )] << set font style( "Italic" );
rbin[Text Edit Box( 1 )] << set font size( 12 );
rbin[Text Edit Box( 2 )] << set font( "Arial Black" );
rbin[Text Edit Box( 2 )] << set font style( "Italic" );
rbin[Text Edit Box( 2 )] << set font size( 12 );
```

---

### Graphing Elements

You can use the following commands inside `Graph Box` statements. This chapter focuses on the JSL that is specific to graphing, but you can also use general script commands like `For`, `While`, and so on.

#### Plotting Functions

A `YFunction` operator is used to draw smooth functions. The first argument is the expression to be plotted, and the second argument is the name of the X variable in the expression.

```
New Window( "Sine Function",
  Graph Box(  
    FrameSize( 200, 100 ),
    X Scale( -10, 10 ),
    Y Scale( -1, 1 ),
    xName( "x" ),
    yName( "Sine(x)" ),
    Y Function( Sine( x ), x )
  )
);
```
You can use `For` to overlap several sine waves:

```plaintext
New Window( "Overlapping Sine Waves",
    Graph Box(
        FrameSize( 200, 100 ),
        X Scale( -10, 10 ),
        Y Scale( -1, 1 ),
        For( i = 1, i <= 4, i += .1,
            Y Function( Sine( x / i ), x )
        )
    );
```

Similarly, an `XFunction` is for drawing a plot where the symbol is varied on the Y variable.

```plaintext
New Window( "Overlapping Sine Waves",
    Graph Box(
        FrameSize( 100, 200 ),
        X Scale( -1, 1 ),
        Y Scale( -10, 10 ),
        For( i = 1, i <= 4, i += .2,
            X Function( Sine( y / i ), y )
        )
    );
```
**Figure 7.5** Overlapping Sine Waves Along the X-Axis

ContourFunction is an analogous way to represent a three-dimensional function in a two-dimensional space. The final argument specifies the value(s) for the contour line(s), and it can be a value, an indexed range of values using ::, or a matrix of values.

```plaintext
New Window( "Bird's eye view of the egg-carton function",
Graph Box(
    FrameSize( 300, 300 ),
    X Scale( -10, 10 ),
    Y Scale( -10, 10 ),
    Pen Color( "black" );
    Pen Size( 2 );
    Contour Function( Sine( y ) + Cosine( x ), x, y, (0 :: 20) / 5 );
    Pen Color( "red" );
    Pen Size( 1 );
    Contour Function( Sine( y ) + Cosine( x ), x, y, (-20 :: 0) / 5 );
)
);```
Normal Contour draws normal probability contours for $k$ populations and two variables. The first argument is a scalar probability or a matrix of probability values for the contours, and subsequent arguments are matrices to specify means, standard deviations, and correlations. The mean and standard deviation matrices have dimension $k \times 2$. The correlation matrix should be $k \times 1$, where the first row pertains to the first contour, the second row to the second contour, and so on. The first column is for $x$ and the second column for $y$. For example:

```
Normal Contour(
    [ prob1,
      prob2,
      prob3, ...],
    [ xmean1 ymean1,
      xmean2 ymean2,
      xmean3 ymean3, ...],
    [ xsd1 ysd1,
      xsd2 ysd2,
      xsd3 ysd3, ...],
    [ xycorr1,
      xycorr2,
      xycorr3, ...]);
```

The following script draws contours at probabilities 0.1, 0.5, 0.7, and 0.99 for two populations and two variables. The first population has $x$ mean 0 and $y$ mean 1, with standard deviation 0.3 along the $x$ axis and 0.6 along the $y$-axis, and with correlation 0.5. The second has $x$ mean 4 and $y$ mean 6, with standard deviation 0.8 along the $x$ axis and 0.4 along the $y$-axis, and with correlation 0.9.

```
New Window( "Normal contours",
            Graph Box(
                X Scale( -5, 10 ),
                Y Scale( -5, 10 ),
```
Normal Contour is thus a generalized way to accomplish effects like Bivariate’s density ellipses, which are demonstrated to good effect with the Football sample data (just open the data table Football and run its stored Bivariate script).

### Gradient Function

The syntax of the gradient function is

```plaintext
Gradient Function(expression, xname, yname, [zlow, zhigh], ZColor([colorLow, colorHigh]), <XGrid(min, max, incr)>, <YGrid(min, max, incr)>)
```

This function fills a set of rectangles on a grid according to a color determined by the expression value as it crosses a range corresponding to a range of colors. To implement it, use the following syntax.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>the expression to be contoured, which is a function in terms of the two variables that follow</td>
</tr>
<tr>
<td>xname, yname,</td>
<td>the two variable names used in the expression</td>
</tr>
<tr>
<td>[zlow, zhigh],</td>
<td>the low and high expression values the gradient is scaled between</td>
</tr>
<tr>
<td>ZColor([colorLow, colorHigh])</td>
<td>the colors corresponding to the low value and the high value</td>
</tr>
<tr>
<td>&lt;XGrid(min, max, incr)&gt;,</td>
<td>optional specification for the grid of values</td>
</tr>
<tr>
<td>&lt;YGrid(min, max, incr)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The ZColor values must be numeric codes, rather than names. You can use the color menu indices (0=black, 1=grey, 2=white, 3=red, 4=green, 5=blue, and so on) found in “Colors,” p. 261.

The following example script uses the Gradient function, with the picture showing two frames of the animation.
phase = 0.7;
New Window( "Gradient Function",
    a = Graph(
        FrameSize( 400, 400 ),
        X Scale( -5, 5 ),
        Y Scale( -5, 5 ),
        DoubleBuffer,
        Gradient Function(
            phase * Sine( x ) * Sine( y ) + (1 - phase) * Cosine( x ) * Cosine( y ),
            x,
            y,
            [-1 1],
            zcolor( [0, 2] )
        )
    )
);
    b = a[FrameBox( 1 )];
For( i = 1, i <= 5, i++,
    For( phase = 0, phase < 1, phase += 0.05,
        b << reshow;
        Wait( 0.01 );
    );
    For( phase = 1, phase > 0, phase -= 0.05,
        b << reshow;
        Wait( 0.01 );
    );
);

Figure 7.8 Gradient Function
Getting the Properties of a Graphics Frame

There are several functions that are useful for getting properties of an existing graphics frame:

- **H Size** Returns the horizontal size of the graphics frame in pixels.
- **V Size** Returns the vertical size of the graphics frame in pixels.
- **X Origin** Returns the distance from the left to right edge of the display box.
- **X Range** Returns the x-value for the left edge of the graphics frame.
- **Y Origin** Returns the y-value for the bottom edge of the graphics frame.
- **Y Range** Returns the distance from the bottom to top edges of a display box.

Examples:

The first line calculates the right edge, and the second line calculates the top edge.

```plaintext
rightEdge = X Origin() + X Range();
topEdge = Y Origin() + Y Range();
```

Adding a Legend

You can add a legend to a graph, using the `Row Legend` command. The following example uses the `Fitness.jmp` sample data file, and sets colors and markers based on the Age column, and adds a legend to the plot.

```plaintext
biv = Bivariate(Y(:Oxy), X(:Runtime)); //generate a scatterplot
Report(biv)[Frame Box(1)] << Row Legend (:Age, color(1), marker(1));
```

The `marker()` parameter has binary arguments. If you specify 1, the markers are set to each point.

You can specify the following settings for colors:

- **Color(0)** turns the color legend on.
- **Color(1)** turns the color legend off.

Drawing Lines, Arrows, Points, and Shapes

Lines

`Line` draws lines between points.

```plaintext
New Window( "Five-point star"),
Graph Box(  
    framesize( 300, 300 ),
    X Scale( -1.1, 1.1 ),
    Y Scale( -1.1, 1.1 ),
    Line(  
        {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )},
    )
)  
```
You can either specify the points in two-item lists as demonstrated above or as matrices of \( x \) and then \( y \) coordinates. Matrices are flattened by rows, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. The following would be equivalent:

```javascript
Line({1,2}, {3,0}, {2,4}); // several \( \{x,y\} \) lists
Line([1 3 2],[2 0 4]);     // row vectors
Line([1,3,2], [2,0,4]);    // column vectors
Line([1 3 2], [2,0,4]);    // one of each
```

Thus, the star example could also be drawn this way. Note that it must use full `Matrix({...})` notation rather than \([ \]\) shorthand since the entries are expressions.

```javascript
new window("Five-point star",
  graph box(framesize(300, 300), xscale(-1.1, 1.1), yscale(-1.1, 1.1),
    line(          // the x coordinates
      matrix({
        cos(1*pi()/10), cos(9*pi()/10), cos(17*pi()/10),
        cos(5*pi()/10), cos(13*pi()/10), cos(1*pi()/10))},
      // the y coordinates
      matrix({
        sin(1*pi()/10), sin(9*pi()/10), sin(17*pi()/10),
        sin(5*pi()/10), sin(13*pi()/10), sin(1*pi()/10)}))));
```
HLine draws a horizontal line across the graph at the \( y \)-value you specify. Similarly VLine draws a vertical line down the graph at the \( x \)-value you specify. Both commands support drawing multiple lines by using a matrix of values in the \( y \) argument. These are illustrated in the example under “MouseTrap,” p. 269.

**Arrows**

Similarly, Arrow draws an arrow from the first point to the second. The default arrowhead is scaled to 1 plus the square root of the length of the arrow. To set the length of the arrowhead, add an (optional) first argument, specifying the length of the arrowhead, in pixels.

For example, the following script draws arrowheads with the default length.

```plaintext
New Window( "Hurricane",
    Graph Box(
        FrameSize( 100, 100 ),
        X Scale( -100, 100 ),
        Y Scale( -100, 100 ),
        For( r = 35, r < 100, r += 20,
            ainc = 2 * Pi() * 3 / r;
            For( a = 0, a < 2 * Pi(), a += ainc,
                x = r * Cosine( a );
                y = r * Sine( a );
                aa = a + ainc * 45 / r;
                rr = r - r / 6;
                x2 = rr * Cosine( aa );
                y2 = rr * Sine( aa );
                Arrow( {x, y}, {x2, y2} );
            );
        );
    );
);
```

**Figure 7.10 Drawing Arrows**

This script compares drawing with a specified length (19 pixels) and drawing with the default arrow head size.

```plaintext
New Window( "Arrow Heads",
    Graph Box(
        Frame Size( 300, 300 ), X Scale( 0, 100 ), Y Scale( 0, 220 ),
```
x = 10; y1 = 10; y2 = y1 + 10;

For( i = 1, i < 10, i++,
    Pen Color( "Red" );
    Arrow( {x, y1}, {x, y2} );
    y2 += 10; y1 += 100; y2 += 100;
    Pen Color( "Blue" );
    Arrow( 20, {x, y1}, {x, y2} );
    x += 10; y1 -= 100; y2 -= 100;
    Text Color( "Red" );
    Text( {10, 80}, "Without Length Arg" );
    Text Color( "Blue" );
    Text( {10, 200}, "With Length Arg" );
    );
)

Figure 7.11 Arrowhead Sizes

As with Line, you can either specify the points in two-item lists as demonstrated above or as matrices of x and then y coordinates.

Markers

Marker draws a marker of the type that you specify (1–15) in the first argument at the point that you specify in the second argument. Marker Size scales markers from 0–6 (dot–XXXL). To set markers to the preferred size, use a value of −1.
Chapter 7
Scripting Graphs
Drawing Lines, Arrows, Points, and Shapes

```
ymax = 20;
New Window( "The markers",
    Graph Box(
        FrameSize( 300, 400 ),
        X Scale( -2, ymax - 5 ),
        Y Scale( -2, ymax + 3 ),
        For( j = 1, j < 7, j++,
            Marker Size( j );
            For( i = 0, i < (ymax + 1), i++,
                Marker( i, {j * 2, i} );
                Text( {0, i}, i );
                Text( {j * 2, ymax + 2}, j );
            );
        );
    ));
```

Figure 7.12 Drawing Markers

You can also include a row state argument before, after, or instead of the marker ID argument. By using Combine States, you can set multiple row states inside Marker. Try substituting each of these lines in the graph script above:

```
marker(i, color state(i), {j*2, i});
marker(color state(i), i, {j*2, i});
marker(combine states(colorstate(i),markerstate(i),hiddenstate(i)),{j*2, i});
```

Again, points can also be specified as matrices of x and then y coordinates.
Pie and Arc draw wedges and arc segments. The first four arguments are $x_1, y_1, x_2,$ and $y_2,$ the coordinates of the rectangle to inscribe. The last two arguments are the starting and ending angle in degrees, where 0 degrees is 12 o'clock and the arc or slice is drawn clockwise from start to finish.

New Window( "Pies and Arcs",
Graph Box(
  framesize( 400, 400 ),
  X Scale( 0, 9 ),
  Y Scale( 0, 9 ),
  Fill Color( "Black" ), // top left
  Pie( 1.1, 7.9, 3.9, 5.1, 45, 270 ),
  Text( erased, {1.75, 6}, "1,8,4,5,45,270" ),
  Arc( 1, 8, 4, 5, 280, 35 ),
  Fill Color( "Red" ), // top right
  Pie( 7.9, 7.9, 5.1, 5.1, 270, 360 ),
  Text( erased, {5.75, 6}, "8,8,5,5,270,360" ),
  Arc( 8, 8, 5, 5, 370, 260 ),
  Fill Color( "BlueCyan" ), // bottom left
  Pie( 1.1, 1.1, 3.9, 3.9, 50, 360 ),
  Text( erased, {1.75, 2}, "1,1,4,4,50,360" ),
  Arc( 1, 1, 4, 4, 370, 40 ),
  Fill Color( "Purple" ), // bottom right
  Pie( 7.9, 1.1, 5.1, 3.9, 270, 45 ),
  Text( erased, {5.75, 2}, "8,1,5,4,270,45" ),
  Arc( 8, 1, 5, 4, 55, 260 )
) );
Figure 7.13 Drawing Pies and Arcs

Regular Shapes: Circles, Rectangles, and Ovals

Circles

Circle draws a circle with the center point and radius given. Subsequent arguments specify additional radii.

New Window( "Circles",  
Graph Box( framesize( 200, 200 ),  
  Circle( {50, 50}, 10, 12, 25 )  
),
);

Note that a circle is always a circle, even if you re-size the graph to a different aspect ratio. If you want your circle to change aspect ratios (in other words, cease being a circle) when the graph is resized, use an oval instead.

If you do not want your circle to resize if the graph is resized, specify the radius in pixels instead:

```graph
New Window("Circles",
    Graph Box(framesize(200, 200),
        Circle({50, 50}, PixelRadius(10), PixelRadius(12), PixelRadius(25))));
```

**Rectangles**

Rect draws a rectangle from the diagonal coordinates you specify. The coordinates can be specified either as four arguments in order (left, top, right, bottom), or as a pair of lists ([left, top], [right, bottom]).

```graph
New Window("Rectangles",
    Graph Box(framesize(200, 200),
        Pen Color(1); Rect(0, 40, 60, 0);
        Pen Color(3); Rect(10, 60, 70, 10);
        Pen Color(4); Rect(50, 90, 90, 50);
        Pen Color(5); Rect(0, 80, 70, 70);
    )
)
```

Rect has an optional fifth argument, fill. Specify a zero to get an unfilled rectangle, and a one to get a filled rectangle. The rectangle is filled with the current fill color. The default value for fill is 0.

Any negative fill argument produces an unfilled frame inset by one pixel:

```graph
New Window("Framed rectangle",
    Graph Box(framesize(200, 200),
        Rect(0, 40, 60, 0, -1)
    )
)
```
Ovals

Oval draws an oval inside the rectangle given by its x1, y1, x2, and y2 arguments:

```graph
new window("Ovals",
  graph box(framesize(200,200),
    pen color(1); oval(0,40,60,0);
    pen color(3); oval(10,60,70,10);
    pen color(4); oval(50,90,90,50);
    pen color(5); oval(0,80,70,70));
)
```

Oval also uses the optional fifth argument, fill. Specify a zero to get an unfilled rectangle, and a one to get a filled oval. The oval is filled with the current fill color. The default value for fill is 0.

Figure 7.15 shows rectangles and ovals, drawn both filled and unfilled. Notice that filled rectangles do not have outlines, while ovals do. If you want a filled rectangle with an outline, you must draw the filled rectangle, and then draw an unfilled rectangle with the same coordinates.

Figure 7.15 Rectangles and Ovals, Unfilled and Filled
Irregular Shapes: Polygons and Contours

Polygons

Polygon works similarly to Line, connecting points but also returning to the first point to close the polygon and filling the resulting area. You can specify the points as individual points in two-item lists (as shown for Marker, above) or as matrices of x and then y coordinates. Matrices are flattened by rows, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. First set up the matrices of points, then call them inside Polygon().

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5, 6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25, 25.625, 25];
gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40, 39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
New Window( "The JMP man",
  Graph Box( framesize( 300, 300 ),
    X Scale( -10, 80 ),
    Y Scale( -10, 80 ),
    Pen Color( "black" );
    Fill Color( "blue" );
    Polygon( gCoordX, gCoordY );
    Fill Color( "black" );
    Circle( {18, 58}, 9, "FILL" );
  ));
```
A related command, `In Polygon`, tells whether a given point falls inside the polygon specified. This code checks some points from the JMP man pictured in Figure 7.16:

```plaintext
In Polygon(0,60, GcoordX,GCoordY);  //returns 0
In Polygon(30,38, GcoordX,GCoordY); //returns 1
```

Or you can add `In Polygon` to the JMP man script. Run this script, and then click various locations in the picture and watch the Log window.

```plaintext
new window("The JMP man",
    graph box(framesize(300,300), xscale(-10,80),yscale(-10,80),
        pen color("black"); fill color("black");
        polygon(gCoordX, gCoordY);
        mousetrap({},print(if(in polygon(x,y,gCoordX,gCoordY),"in","out"))));
```

### Contours

`Contour` draws contour lines using a grid of coordinates. Its syntax is:

```plaintext
Contour(xVector,yVector,zGridMatrix,zContour,<zColors>);
```

Given an $n$ by $m$ matrix `zGridMatrix` of values on some surface, defined across the $n$ values of `xVector` by the $m$ values of `yVector`, this function draws the contour lines defined by the values in `zContour` in the colors defined by `zColors`.

```plaintext
// testContour
x = (-10 :: 10) / 5;
y = (-12 :: 12) / 5;
grid = J( 21, 25, 0 );
z = [-.75, -.5, -.25, 0, .25, .5, .75];
```
260  

Scripting Graphs

Chapter 7

Drawing Lines, Arrows, Points, and Shapes

\[
zcolor = [3, 4, 5, 6, 7, 8, 9];
For( i = 1, i <= 21, i++,
   For( j = 1, j <= 25, j++,
      grid[i, j] = \sin((x[i])^2 + (y[j])^2)
   )
);
Show( grid );
New Window( "Hat",
   Graph Box( X Scale( -2, 2 ), Y Scale( -2.4, 2.4 ), Contour( x, y, grid, z, zcolor )
);

Figure 7.17  Drawing Contour Lines

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>-2.5</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>-1.5</td>
</tr>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

Adding text

You can use Text to draw text at a given location. The point and text can be in any order and repeated. You can precede the point and text with an optional first argument, Center Justified, Right Justified, Erased, Boxed, Counterclockwise, or Clockwise. Erased is for “erasing” whatever would otherwise obscure the text in a graph. It paints a background-colored rectangle behind the text. In the example below, notice how the erased text appears inside a white box over the green Rect. The other effects are self-explanatory.

\[
mytext = New Window( "Text",
   Graph Box(
      framesize( 200, 200 ),
      Y Scale( 0, 15 ),
      X Scale( 0, 10 ),
      Text Size( 9 );
      Text Color( "blue" );
      Text( {5, 1}, "Left Justified" );
      Text( Center Justified, {5, 2}, "Center Justified" );
      Text( Right Justified, {5, 3}, "Right Justified" );
      Fill Color( 4 );
      Rect( 5, 8, 9, 5, 1 );
      Text( Erased, {6, 6}, "Erased" );
);
Text( Boxed, {6, 10}, "Boxed" );
Text( Clockwise, {4, 10}, "Clockwise" );
Text( Counterclockwise, {3, 5}, "Counterclockwise" );
)
);

Figure 7.18 Drawing Text in a Graph Box

There is a variant of the text function that draws a string inside the rectangle specified by four coordinates specified as arguments, wrapping as needed. The syntax is

\[
\text{text( \{left, top, right, bottom\}, string)}
\]

Colors

Five commands control colors. Fill **Color** sets the color for solid areas, Pen **Color** for lines and points, Back **Color** for the background of text (similar to the box around the erased text above), Background **Color** for the graph's background color, and Font **Color** for added text.

Fill colors preempt pen colors for drawn shapes. You do not get both, as in some drawing packages. To get both a fill and a penline, draw two shapes, one with fill and one without. A color can be chosen with a single numeric argument, or a color name in quotation marks, or an RGB value. The standard colors can be chosen with numbers 0–15 (both 0 and 15 are black) or by their names.
Larger numbers cycle through shading variations of the same color sequence. A script demonstrating this appears under “Colors and Markers,” p. 143 in the “Data Tables” chapter. Values outside the range 0–84 are not accepted.

Table 7.2 How Numbers Map to Colors

<table>
<thead>
<tr>
<th>Number</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16–31</td>
<td>dark shades</td>
</tr>
<tr>
<td>32–47</td>
<td>light shades</td>
</tr>
<tr>
<td>48–63</td>
<td>very dark shades</td>
</tr>
<tr>
<td>64–79</td>
<td>very light shades</td>
</tr>
<tr>
<td>80–84</td>
<td>shades of gray, light to dark</td>
</tr>
</tbody>
</table>

If you prefer to use RGB values, type a list with the percentages for each color in red, green, blue order.

```pen color({.38,.84,.67}); // a lovely teal```
RGB Color and Color to RGB convert color values between JMP color numbers and the Red-Green-Blue system. For example, to find the RGB values for JMP color 3 (red):

```
Color to RGB(3);
{0.941176470588235, 0.196078431372549, 0.274509803921569}
```

Likewise, HLS Color and Color to HLS convert color values between JMP color numbers and the Hue-Lightness-Saturation system.

Finally, Heat Color returns the JMP color that corresponds to a value in any color theme that is supported by Cell Plot, Tree Map, and so on. The syntax is:

```
Heat Color(n,<<"theme")
```

The theme message is optional, and the default value is "Blue to Gray to Red". You can specify any color theme, including custom color themes. You can also create and use an anonymous color theme. For example,

```
Heat Color( z, <<"", {{1, 1, 0}, {0, 0, 1}} })
Heat Color( z, <<"", {blue, green, yellow} })
```

**Transparency**

In a graphics environment (like a Frame Box), use the Transparency function to set the level of transparency. The argument, alpha, can be any number between zero and one. The value 0 means clear and drawing has no effect, while the value 1 means completely opaque and is the usual drawing mode. Intermediate values build semi-transparent color layers on top of what has already been drawn below it. The following example script illustrates transparency with rectangles.

```
New Window( "Transparency",
Graph Box(framesize( 200, 200 ),
Pen Color( "gray" ); Fill Color( "gray" );
Transparency( 0.25 );
Rect( 0, 40, 60, 0, 1 );

Pen Color( "red" ); Fill Color( "red" );
Transparency( 0.5 );
Rect( 10, 60, 70, 10, 1 );

Pen Color( "green" ); Fill Color( "green" );
Transparency( 0.75 );
Rect( 50, 90, 90, 50, 1 );

Pen Color( "blue" ); Fill Color( "blue" );
Transparency( 1 );
Rect( 0, 80, 70, 70, 1 );
);
```
Fill patterns

The Fill Pattern function has been deprecated and is now obsolete. It can be present in a script without causing errors, but has no effect.

Line types

You can also control Line Style by number (0–4) or name (Solid, Dotted, Dashed, DashDot, DashDotDot).

```
linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
New Window( "The line styles",
    Graph Box(
        FrameSize( 200, 200 ),
        X Scale( -1, 5 ),
        Y Scale( -1, 5 ),
        For( i = 0, i < 5, i++,
            Line Style( i );
            H Line( i );
            Text( {0, i + .1}, i );
            Text( {1, i + .1}, linestyles[i + 1] );
        )
    )
);
```
To control the thickness of lines, set a Pen Size and specify the line width in pixels. The default is 1 for single-pixel lines. For printing, think of Pen Size as a multiplier for the default line width, which varies according to your printing device.

```plaintext
pen size(2); // double-width lines
```

**Drawing With Pixels**

You can also draw using pixel coordinates. First you set the Pixel Origin in terms of graph coordinates, and then use Pixel Move To or Pixel Line To commands in pixel coordinates relative to that origin. The main use for Pixel commands is for drawing custom markers that do not vary with the size or scale of the graph. You can store a marker in a script and then call it within any graph. This example uses Function to store pixel commands in a script with its own parameters, x and y.

```plaintext
ballotBox = Function( {x, y},
    Pixel Origin( x, y );
    Pixel Move To( -5, -5 );
    Pixel Line To( -5, 5 );
    Pixel Line To( 5, -5 );
    Pixel Line To( -5, -5 );
    Pixel Line To( 5, 5 );
    Pixel Line To( -5, 5 );
    Pixel Move To( 5, 5 );
    Pixel Line To( 5, -5 );
);
New Window( "Custom markers",
    Graph Box( framesize( 200, 200 ),
        ballotBox( 10, 10 );
        ballotBox( 15, 90 );
        ballotBox( 20, 50 );
        ballotBox( 80, 50 );
        ballotBox( 60, 70 );
    );
```

---

**Figure 7.20** Line Styles
Interactive graphs

Handle and MouseTrap are functions for making interactive graphs that respond to clicking and dragging. Handle lets you parametrize a graph by adding a handle-marker that can be dragged around with the mouse, executing the graph’s script at each new location. MouseTrap is similar, but it takes its parameters from the coordinates of a click, without dragging a handle. The main difference is that Handle only catches mousedowns at the handle-marker’s location, but MouseTrap catches mousedowns at any location.

Another approach is to place buttons or slider controls outside the graph with Button Box, Slider Box, or Global Box.

Handle

Handle places a marker at the coordinates given by the initial values of the first two arguments and draws the graph using the initial values of the parameters. You can then click and drag the marker to move the handle to a new location. The first script is executed at each mousedown to update the graph dynamically, according to the new coordinates of the handle. The second script (optional, and not used here) is executed at each mouseup, similarly; see the example for “MouseTrap,” p. 269.

```vbnet
// Normal Density
mu = 0;
sigma = 1;
rsqrt2pi = 1 / Sqrt( 2 * Pi() );
New Window( "Normal Density",
    Graph Box(      
        FrameSize( 500, 300 ),      
        X Scale( -3, 3 ),      
        Y Scale( 0, 1 ),      
        Double Buffer,
    )
) ;
```

Figure 7.21 Drawing Custom Markers
Y Function( Normal Density( (x - mu) / sigma ) / sigma, x );
Handle(
    mu,
    rsqrt2pi / sigma,
    mu = x;
    sigma = rsqrt2pi / y;
);
Text( {1, .7}, "mu ", mu, {1, .65}, "sigma ", sigma );
)

In the sample Scripts folder, you can find scripts for showing the Beta Density, Gamma Density, Weibull Density, and LogNormal Density. The output for the normal is show below. Since we cannot show you the picture in motion, be sure to try this yourself.

---

**Figure 7.22** Normal Density Example for Handle

![Normal Distribution Example](image)

To avoid errors, be sure to set the initial values of the handle’s coordinates, as in the first line of this example. If you want to use some function of a handle’s coordinates, such as in the normal density example, you should adjust the arguments for Handle. Otherwise, the handle marker would run away from the mouse. For example:

YFunction(a*x^b);
handle(a,b,a=2*x;b=y)

Suppose you drag the marker from its initial location to (3,4). The parameter \(a\) is set to 6 and \(b\) to 4, the graph is redrawn as \(Y = 6x^4\), and the handle is now drawn at (6,4), several units away from the mouse. To compensate, you would adjust the first argument to handle, for example.

handle(a/2,b,a=2*x;b=y)
To generalize, suppose you define the `Handle` parameters as functions of the handle's \((x, y)\) coordinates. For example, \(a = f(x); b = g(y)\). If \(f(x) = x\) and \(g(y) = y\), then you would specify simply \(a, b\) as the first two arguments. If not, you would solve \(a = f(x)\) for \(x\) and solve \(b = g(y)\) for \(y\) to get the appropriate arguments.

You can use other functions to constrain `Handle`. For example, here is an interactive graph to demonstrate power functions that uses `Round()` to prevent bad exponents and to keep the intercepts simple.

```plaintext
a = 3; b = 2;
New Window( "Intercepts and powers",
    Graph Box( FrameSize( 200, 200 ), X Scale( -10, 10 ), Y Scale( -10, 10 ),
              Y Function( Round( b ) + x ^ (Round( a )), x );
              Handle( a, b, a = x; b = y; );
              Text( {a, b}, " Move me" );
              Text( {-9, 9}, "y=", Round( b ), " + x^", Round( a ) );
    )
);
```

Figure 7.23 Intercepts and Powers Example for `Handle`

Handle and `For` can be nested for complex graphs.

```plaintext
a=5; b=5;
NewWindow("powers",
    Graph Box("FrameSize(200,200),XScale(-10,10),yScale(-10,10),Double Buffer,
        for(i=0,i<1.5,i+=.2,
            pen color(1+10*i);
            text color(1+10*i);
            YFunction(i*x^round(a),x);
            Handle(a,b,a=x;b=y);
            h=9-10*i;
            text({-9,h},"*i*x^",round(a),", i=",i));
    )
);
```

You can use more than one handle in a graph:

```plaintext
amplitude = 1; freq = 1; phase = 0;
NewWindow("Sine Wave",
```
Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),Double Buffer, YFunction(amplitude*sine(x/freq+phase),x); Handle(freq,amplitude,freq=x;amplitude=y); Handle(phase,.5,phase=x); Text({3, 4}, "amplitude: ", Round( amplitude, 4 ), {3, 3.5}, "frequency: ", Round( freq, 4 ), {3, 3}, "phase: ", Round( phase, 4 )));

MouseTrap

MouseTrap takes parameters for a graph from the coordinates of a mouse click. The first script is executed after eachmousedown and the second script after each mouseup to update the graph dynamically, according to the new coordinates of the handle. As with Handle, it is important to set the initial values for the MouseTrap's coordinates. If you include both MouseTraps and Handles in a graph, put the Handles before the MouseTraps so they have a chance to catch clicks before a MouseTrap does.

This example uses both MouseTrap and Handle to draw a three-dimensional function centered on the MouseTrap coordinates, where the single contour line takes its value from a Handle.

```
x0=0;y0=0;z0=0;
New Window("Viewing a 3-D function in Flatland",
    Graph Box(FrameSize(300,300),XScale(-5,5),yScale(-5,5),DoubleBuffer,
    ContourFunction(exp(-(x-x0)^2)*exp(-(y-y0)^2)*(x-x0),x,y,z0/10);
    handle(-4.5,z0, z0=round(y*10)/10); // get the z-cut values from a handle
    vline(-4.5);text size(9);text(Counterclockwise,{-4.6,-4},
    "Drag to set the z-value for contour cut: z = " || char(z0/10));
    markersize(2);marker(2,{x0,y0});
    mousetrap(x0=x;y0=y); //set the origin to the click-point
    text({-4.25,-4.9},"Click any location to set the function's centerpoint.");
```

You might use MouseTrap to collect points in a data table, such as for visually interpolating points in a graph. Here is an example illustrating a script that could be adapted and added to a data plot (such as a scatterplot from Fit Y by X) for that purpose:

```
dt = new Table("dat1");
Current Data Table(dt);
NewColumn("XX",Numeric);
NewColumn("YY",Numeric);
x=0; y=0;
add point = expr(
    dt<<addRows(1);
    row()=nrow();
    :xx = x;
    :yy = y);
NewWindow("Add Points",
    Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),
    for each row(marker({xx,yy}));
    MouseTrap({},add point)));
Notice that the first script argument is empty. At mousedown, nothing happens. The second script, add point, is executed at mouseup to add a data point. This means that if you click, drag, and release, the point that is added to your data set is the point where you let go of the mouse button, not the point where you pressed it down.

Drag Functions

There are five Drag functions to perform similar functions to Handle and MouseTrap but with more than one point at a time. For \( n \) coordinates in matrices listed as the first two arguments:

- **Drag Marker** draws \( n \) markers.
- **Drag Line** draws a connected line with \( n \) vertices and \( n - 1 \) segments.
- **Drag Rect** draws a filled rectangle using the first two coordinates, ignoring any further coordinates.
- **Drag Polygon** draws a filled polygon with \( n \) vertices.
- **Drag Text** draws a text item at the coordinates, or if there is a list of text items, draws the \( i \)th list item at the \( i \)th \((x,y)\) coordinate. If there are fewer list items than coordinate pairs, the last item is repeated for remaining points.

The syntax for these commands:

```plaintext
dragMarker (xMatrix, yMatrix, dragScript, mouseupScript)
dragLine   (xMatrix, yMatrix, dragScript, mouseupScript)
dragRect   (xMatrix, yMatrix, dragScript, mouseupScript)
dragPolygon(xMatrix, yMatrix, dragScript, mouseupScript)
dragText   (xMatrix, yMatrix, "text", dragScript, mouseupScript)
```

They all must have L-value arguments for the coordinates, in other words, literal matrices or names of matrix values that are modified if you click a vertex and drag it to a new position. The script arguments are optional, and behave the same as with Handle. However, there is no \( x \) nor \( y \) that is modified as in Handle.

The Drag operators are ways to display data that the user can adjust and then capture the adjusted values. Consider the earlier script to draw the JMP man. Drag Polygon makes it possible to draw an editable JMP man; using a matching Drag Marker statement makes the vertices more visible. And, similar to the Mouse Trap example, you can save the new coordinates to a data table. Notice how : and :: operators avoid ambiguity among matrices and data table columns with the same names.

You could just as easily put storepoints in the fourth argument of Drag Polygon or Drag Marker, but that would create a data table after each drag, and you probably just want a single data table when you are finished. Regardless, the values in \( gCoordX \) and \( gCoordY \) update with each drag.

```plaintext
::i = 1;
storepoints = Expr(
  mydt = New Table( "My coordinates_" || Char( i ) );
i++;
  New Column( "GCoordX", Numeric );
  New Column( "GCoordY", Numeric );
  mydt << add rows( N Row( GcoordX ) );
  :GCoordX << values( ::GcoordX );
  :GCoordY << values( ::GcoordY );
```

Perhaps you think the JMP Man needs to lose some weight. Here is how he looks before and after some judicious vertex-dragging. Clicking the button after re-shaping the JMP Man executes the `storepoints` script to save his new, slender figure in a data table of coordinates.
This example uses two operators discussed under “Constructing Display Trees,” p. 190 in the “Display Trees” chapter:

- **Button Box**, which creates controls outside the graph,
- **V List Box**, which glued the graph box and the button box together in the same graph window.

**Troubleshooting**

If your interactive graphs do not work as expected, make sure that you supply initial values for the Handle or MouseTrap coordinates (and other globals as needed), and that the values make sense for the graph.
JSL includes commands for scripting three-dimensional scenes derived from OpenGL. Although not a complete OpenGL implementation, JSL’s 3-D scene commands enable complex, interactive plots to be constructed and viewed. The Surface Plot platform in JMP is built using JSL scene commands.
Contents

About JSL 3-D Scenes ........................................... 275
JSL 3-D Scene Boxes ........................................... 275
Setting the Viewing Space .................................... 278
  Setting Up a Perspective Scene ............................. 279
  Setting up an Orthographic Scene ....................... 280
Changing the View ............................................... 281
  The Translate Command .................................... 281
  The Rotate Command ....................................... 281
  The Look At Command .................................... 283
  The ArcBall ................................................. 284
Graphics Primitives ........................................... 285
  Primitives Example ....................................... 287
  Controlling the Appearance of Primitives ............... 288
  Other uses of Begin and End ............................. 294
Drawing Spheres, Cylinders, and Disks ..................... 294
Drawing Text .................................................. 295
Using the Matrix Stack ...................................... 297
Lighting and Normals ......................................... 300
  Creating Light Sources .................................. 300
  Lighting Models .......................................... 302
  Normal Vectors .......................................... 302
  Shading Model ............................................ 303
  Material Properties ..................................... 304
  Alpha Blending ........................................... 304
  Fog ...................................................... 305
  Example ................................................. 305
Bézier Curves ................................................ 306
Using the mouse .............................................. 309
Parameters ..................................................... 311
About JSL 3-D Scenes

JMP’s 3-D scene language is built on top of the OpenGL® API, extending, replacing, and leaving out various parts of the OpenGL API, and as such is not an implementation that is certified or licensed by Silicon Graphics, Inc. under the OpenGL API.

This chapter documents JMP’s JSL commands for creating 3-D scenes but is not a tutorial on OpenGL programming. If you are not familiar with OpenGL programming, you might want supplemental material. If you are familiar with OpenGL programming, you still need to read this chapter because some items are nonstandard.

JMP ships with sample files in the Scene3D subfolder of Sample Scripts to get you started and give you some ideas. Some of the example scripts are similar to some of the examples in this chapter; some are almost complete applications.

The Web site opengl.org is a good jumping off point for information, as is your favorite search engine.

JMP’s Scene 3-D language does some work for you that the OpenGL API requires you to do for yourself. JMP makes text easy, gives you a built-in arcball controller, and makes sure the matrix operations that belong on the model view stack and projection stack go on their respective stacks. JMP uses its own display list manager so your scenes can be journaled and played back later, and provides a pick mechanism that calls back to your JSL code to tell you what object in your scene is under the mouse, with almost no extra effort on your part. At this time, JMP does not provide access to some features, like texturing.

OpenGL is a trademark of Silicon Graphics, Inc.

JSL 3-D Scene Boxes

These commands are necessary to set up and configure a 3-D scene.

Like all displays in JMP (detailed in the “Display Trees” chapter), 3-D scenes must be placed in a display box (in this case, a Scene Box). This box is then placed in a window. Therefore, a simple 3-D scene script has the following form.

```julia
myScene=Scene Box(300, 300); //create a 300 by 300 pixel scene box
...(commands to set up the scene)...
New Window (“3-D Scene”, myScene); //draw the scene in a window
...(commands that manipulate the scene)
```

The scene can be sent messages that construct elements in the scene. Typical messages alter the viewer’s vantage point, construct physical elements in the scene itself, or manipulate lights and textures. These messages are maintained in a display list and are manipulated in one of two ways:

- They are sent as messages to the scene, which immediately adds them to the scene’s internal display list.
- They are sent as messages to a display list stored in a global variable, which is called by the scene’s display list later.
For example of commands sent to the scene's display list directly, consider the following small script. Each of the commands is explained in detail later in the chapter.

```plaintext
scene = SceneBox(400, 400); // make a scene box.
NewWindow( "Example 1", scene ); // put the scene in a window.
scene << Perspective( 45, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -4.5 ); // move to (0,0,-4.5) to draw
scene << color(1,0,0); // set the RGB color of the text
scene << Text( center, baseline, 0.2, "Hello, World." ); // add text
scene << Update; // update the scene
```

The first two lines create a scene and place it in a window. The `Perspective` command defines the viewing angle and field depth. By sending it as a message to the scene, it is immediately added to the scene's display list. Since the "Hello World" text is to be drawn at the origin (0, 0, 0), the `Translate` command is added to the display list to move the camera back a bit so that the origin is in the field of vision. The color is set to red with the `Color` command, the text is drawn, and the `Update` command causes the scene to be rendered (in other words, causes the display list that contains the commands to be drawn.)
Equivalently, the commands to construct the display can be accumulated in a display list stored in a global variable, which is then sent to the scene all at once. To define a global variable as a display list, assign it using the \texttt{Scene Display List} function. For example, to use the global \texttt{greeting} as a display list, issue the command

\begin{verbatim}
greeting=Scene Display List();
\end{verbatim}

Display commands can then be sent as messages to \texttt{greeting}. An equivalent “Hello World” example using a display list follows.

\begin{verbatim}
// create a display list and send it commands
greeting = Scene Display List();
greeting << color(1,0,0); //set the RGB color of the text
greeting << Text( center, baseline, 0.2, "Hello, World." ); //add text

//draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box.
New Window( "Example 1", scene ); // put the scene in a window.
scene << Perspective( 45, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -4.5 ); // move to (0,0,-4.5) to draw
scene << Call List(greeting); //send the display list to the scene
scene << Update; // update the scene
\end{verbatim}

Note which commands were separated into the display list, and which were applied to the scene directly. Those that manipulate the camera (\texttt{Translate} and \texttt{Rotate}) are applied to the scene. Those that define the object (\texttt{Color} and \texttt{Text}) were relegated to the display list. This is done so that the display list can be called many times to replicate the object at different positions.
Setting the Viewing Space

3-D scenes can be rendered in two ways. Orthographic projections place the elements in a box, where coordinates are not changed to accommodate the perspective of the viewer. Perspective projections modify the display to simulate the position of the elements in relation to the position of the viewer. For example, two parallel lines (like railroad tracks) stay parallel in orthographic projections, but seem to connect at a distance in perspective projections.

As another example, imagine looking at a tube edge-on (like a telescope). In an orthographic projection, the tube would appear as a thin circle. In a perspective projection, the circle would have a thickness; the hole at the far end of the tube would appear smaller than the close hole, and the interior of the tube is visible.

Therefore, the viewable space of an orthographic projection is a rectangular shape, while that of a perspective projection is the frustum of a pyramid (that is, a pyramid whose top has been sliced off).
In general, perspective projections give a more realistic view of the world, since it mimics the way an eye or a camera sees. Orthographic projections are important when it is essential to preserve dimensions, such as an architectural CAD program.

**Setting Up a Perspective Scene**

To set up a perspective scene in JSL, send the `Perspective` command to a display list.

```
Perspective (angle, near, far)
```

where `angle` is the viewing angle, `near` is the distance to the near plane, and `far` is the distance to the far plane, as illustrated in the drawing above. A couple of things need to be remembered when defining the viewing space.

- Items outside the viewing space (for example, closer than the `near` plane or farther than the `far` plane) are not drawn. They are clipped off.
- The ratio of `far` to `near` needs to be small so that the rendering engine can effectively determine which items should be drawn “on top of” other items, simulating closeness of items. The `near` parameter must be greater than zero.

The “Hello World” example contains the line
Three-Dimensional Scenes
Setting the Viewing Space

scene << Perspective( 45, 3, 7 ); // define the camera

This defines a 45 degree viewing angle, with a near plane 3 units from the viewer and a far plane 7 units
from the viewer.

The viewing angle functions in the same way as a wide angle or telephoto lens on a camera. Small viewing
angles zoom into a drawing, while wide angles zoom out. In other words, a small viewing angle maps the
screen space onto a small portion of the scene, resulting in apparently larger scene elements. A large viewing
angle maps the screen space onto a large portion of the scene, resulting in apparently small screen elements.
The size of scene elements can therefore be manipulated using the \textit{angle} parameter of the \texttt{Perspective}
function. The picture here shows the hello world script with perspective angles of 45 and 90 degrees.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{hello_world.png}
\caption{Changing the Perspective}
\end{figure}

\begin{center}
\textbf{Hello, World.}
\end{center}

\begin{verbatim}
scene << Perspective( 45, 3, 7 );
\end{verbatim}

As an alternative to the \texttt{Perspective} command, you can define the actual viewing frustum with the
\texttt{Frustum} command.

\begin{verbatim}
Frustum( left, right, bottom, top, near, far);
\end{verbatim}

The frustum's viewing volume is defined by \texttt{(left, bottom, near)} and \texttt{(right, top, near)} which specify the \texttt{(x, y, z)}
coordinates of the lower left and upper right corners of the near clipping plane; \texttt{near} and \texttt{far} give the
distances from the viewpoint to the near and far clipping planes.

\section*{Setting up an Orthographic Scene}

Orthographic scenes are specified in ways similar to perspective scenes. Issue the command

\begin{verbatim}
Ortho(left, right, bottom, top, near, far)
\end{verbatim}

which specifies the four corners of the near plane, the distance to the near plane, and the distance to the far
plane.

If you are dealing with a simple 2-D environment, you can set up a two-dimensional orthographic scene
with the command
Changing the View

One of the advantages of creating a 3-D scene is the ease that they can be viewed from different angles and positions. The Translate and Rotate commands let you set the position from which you view the scene. In addition, you can use the ArcBall command to enable the user to change the viewing angle interactively.

The Translate Command

You have actually seen the Translate command in earlier sample scripts. It sets the position from which the scene is viewed. The arguments give the amount to move from the current position in the x, y, and z direction.

Translate (x, y, z)

For example,

Translate( 0.0, 0.0, -2 );

moves the origin two units in the negative z direction.

Initially, the origin and camera were at the same place. Now, the camera can see the origin because the camera faces down the negative z-axis.

The Rotate Command

The Rotate command is used to modify the viewing angle of a scene. It has the following format.

Rotate (degrees, xAxis, yAxis, zAxis)

This rotates by degrees around the axis described by the vector (xAxis, yAxis, zAxis). For example, to rotate a model 90 degrees about the x-axis, use Rotate( 90, 1, 0, 0 ).

You can also specify the three axis values in a matrix. For example, Rotate( 90, [1, 0, 0] ).

Note: The Rotate command uses degrees, in contrast to JMP’s trigonometric functions, which use radians.

Translate and Rotate are also used to position objects with respect to each other. The first Translate or Rotate can be thought of as positioning everything that follows with respect to the camera. Subsequent Translate and Rotate commands are used to position objects, such as spheres, cylinders, disks, and display lists in Call List and ArcBall commands. For example, suppose you have a display list named table and another named chair. Your scene might look like this:
The following example uses the Rotate command inside a For loop to continuously change the viewing angle of a scene. It draws a cylinder that swings around a central point. This central point is shown by a small sphere.

```cpp
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 600, 600 );

// put the scene in a window.
NewWindow( "Example 1", scene );

for (i=1, i<360, i++,
    scene << clear;
    // the lens is 45 degrees, near is 1 units from the camera, far is 10.
    scene << Perspective( 45, 1, 10 );
    scene << Translate( 0.0, 0.0, -2 );
    scene << Rotate(i,1,0,0);
    scene << Rotate(i*3, 0, 1, 0);
    scene << Rotate (i*3/2, 0, 0, 1);
    scene << Color(0, 1, 0); //green for cylinder
    scene << Cylinder(0.5, 0.5, 0.5, 40,10);
    scene << Color(0, 0, 0); //black for sphere
    scene << Sphere(0.01, 10, 5);
    scene << Update;
)
```
Note the use of the Update command at the end of the scene messages. This command tells JMP to make the displayed screen agree with the current state of the display list. It is important to clear the list at the beginning (so the list does not contain the old angles as well as the current) and update the scene after each change.

**The Look At Command**

The Look At command is an alternative way to set the camera view.

```plaintext
Look At( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

The Look At command puts the camera at the eye coordinates and points it toward the center coordinates. The up vector describes how the camera is rotated on its line of sight. Because the model is typically constructed at the origin, a JMP scene should have either a Look At or a Translate command near its beginning to move the camera away from the origin.

First, clear the scene box of any commands from the previous frame.

```plaintext
scene << clear;
```

Then use one of these projections:

```plaintext
scene << perspective(45,2,10);
```
Three-Dimensional Scenes

Chapter 8

Changing the View

```plaintext
scene <<frustum(-.5,.5,-.5,.5,1,10);
scene <<ortho(-2,2,-2,2,1,10);
scene <<ortho2d(-2,2,-2,2);
```

**Note:** If you use the `ortho2d` projection, you should not also set the camera position using either `Translate` or `Look At`.

Finally, use either `Translate` or `Look At` to set the camera position:

```plaintext
scene <<Translate(0.0, 0.0, -4.5);
/* the camera faces down the negative Z axis. move it back so 0,0,0 is in view. */
scene <<Look At( /*eye*/ 3,3,3, /*center*/ 0,0,0, /*up*/ 1,0,0 );
/*this is much easier. */
```

Once the scene and camera position are set, add your model.

The ArcBall

Sometimes you want a scene to rotate based on the movements of the mouse. The Surface Plot platform in JMP is an example of a 3-D scene that rotates based on mouse movements.

An ArcBall creates a sphere around the 3-D scene and enables the user to click on the sphere's surface and drag it around, thus causing the scene to rotate.

Use an ArcBall instead of a `CallList` command to place the scene in an ArcBall. Scenes that are attached to an ArcBall automatically respond to clicks and drags of the mouse. Custom programming is not needed. However, rotations made in the arcball are not saved. (Technically, the ArcBall is surrounded by an implicit `Push Matrix` and `Pop Matrix` block, so the movements are gone after it returns. See “Using the Matrix Stack,” p. 297 for details of pushing and popping.)

For example, examine the script from “Primitives Example,” p. 287. Change the single line

```plaintext
scene << CallList(shape); //send the display list to the scene
```

so that it reads

```plaintext
scene << ArcBall(shape,2); //send the display list to an arcball
```

This displays the script with an associated arcball with diameter 2. When you run the script and the window appears, Right-click and select **Show ArcBall > Always** from the menu that appears.

**Note:** ArcBall comes from an article by Shoemake (1994) found in *Graphics Gems IV*, published by Academic Press.

This sets the display so that the ArcBall is always showing. Click and drag on the ArcBall to rotate the scene. The popup menu with **Background Color**, **Use Hardware Acceleration**, and **Show ArcBall** is always available, whether the scene is displayed through a platform, in a journal, or through JSL.

**Note:** The ArcBall does not have to be showing to react to mouse commands. It is shown here for display purposes only.
You can also set the display state of the ArcBall in JSL using the `Show ArcBall` command.

```
scene << Show Arcball (state)
```

where `state` is `During Drag`, `Always`, or `Never`.

**Figure 8.9** Showing the Arc Ball

---

### Graphics Primitives

All scenes in JSL are built with a small number of graphics primitives. These fundamental elements function as the building blocks for complicated scenes.

Every graphics primitive involves specifying vertices. In some cases, the vertices are simply drawn as points. In others, the vertices are connected to form polygons. To draw a primitive, you must specify the type of primitive and the coordinates and properties of the vertices involved. In JSL, this specification is accomplished through the `Begin` and `End` statements.

```
scene<<Begin(primitive type);
...(commands specifying vertices and their properties)...
scene<<End();
```

To specify the coordinates of the vertices, use the `vertex` command.

```
scene<<Begin(primitive type);
scene<<Vertex(x, y, z);
...
scene<<End();
```

The options for `primitive type` are the following. In these examples, assume that v0, v1, and so on, have been specified between a `Begin` and `End` pair, similar to the following.

```
scene<<Begin(primitive type);
scene<<Vertex(x0,y0,z0)//specify vertex v0
scene<<Vertex(x1,y1,z1)//specify vertex v1
...
Three-Dimensional Scenes
Graphics Primitives

```cpp
scene<<Vertex(xn,yn,zn)//specify vertex vn
scene<<End();
```

### Table 8.1: Primitive Types

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTS</td>
<td>Draws a point at each of the vertices.</td>
<td>![Points Diagram]</td>
</tr>
<tr>
<td>LINES</td>
<td>Draws a series of (unconnected) line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If n is odd, the last vertex is ignored.</td>
<td>![Lines Diagram]</td>
</tr>
<tr>
<td>POLYGON</td>
<td>Draws a polygon using the points v0,..., vn as vertices. Three vertices must exist, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices do not satisfy these conditions, the results are unpredictable.</td>
<td>![Polygon Diagram]</td>
</tr>
<tr>
<td>TRIANGLES</td>
<td>Draws a series of (disconnected) triangles using vertices v0, v1, v2, then v3, v4, v5, and so on. If the number of vertices is not an exact multiple of 3, the final one or two vertices are ignored.</td>
<td>![Triangles Diagram]</td>
</tr>
<tr>
<td>LINE_STRIP</td>
<td>Draws a line segment from v0 to v1, then from v1 to v2, and so on. Therefore, n vertices specify n–1 line segments. Nothing is drawn unless there is more than one vertex. There are no restrictions on the vertices describing a line strip; the lines can intersect arbitrarily.</td>
<td>![Line Strip Diagram]</td>
</tr>
<tr>
<td>LINE_LOOP</td>
<td>Same as LINE_STRIP, except that a final line segment is drawn from the last vertex to the first, completing a loop.</td>
<td>![Line Loop Diagram]</td>
</tr>
</tbody>
</table>
Table 8.1 Primitive Types (Continued)

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>QUADS</code></td>
<td>Draws a series of quadrilaterals (four-sided polygons) using vertices v0, v1, v2, v3, then v4, v5, v6, v7, and so on. If the number of vertices is not a multiple of 4, the final one, two, or three vertices are ignored.</td>
</tr>
<tr>
<td><code>QUAD_STRIP</code></td>
<td>Draws a series of quadrilaterals (four-sided polygons) beginning with v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6, and so on. The number of vertices must be at least 4 before anything is drawn, and if odd, the final vertex is ignored.</td>
</tr>
<tr>
<td><code>TRIANGLE_STRIP</code></td>
<td>Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. There must be at least three vertices for anything to be drawn.</td>
</tr>
<tr>
<td><code>TRIANGLE_FAN</code></td>
<td>Same as <code>TRIANGLE_STRIP</code>, except that the vertices are v0, v1, v2, then v0, v2, v3, then v0, v3, v4, and so on.</td>
</tr>
</tbody>
</table>

Primitives Example

The following short example illustrates the use of a graphics primitive.

```cpp
// create a display list and send it commands
shape = Scene Display List();
shape << Color(1, 0, 0); // set the RGB color of the text
shape << Begin(POLYGON);
shape << Vertex(0, 0, 0);
shape << Vertex(0, 3, 0);
shape << Vertex(3, 3, 0);
```
Three-Dimensional Scenes
Graphics Primitives

shape << Vertex(5, 2, 0);
shape << Vertex(4, 0, 0);
shape << Vertex(2, -1, 0);
shape << End();

//draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box.
New Window( "Primitive", scene ); // put the scene in a window.
scene << Perspective( 90, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -5 ); // move to (0,0,-5) to draw
scene << Call List(shape); //send the display list to the scene
scene << Update; // update the scene

The first section of the script creates a display list named shape. Inside this display list, a polygon is defined using six vertices. The second section of the script creates a scene box and a new window. It then uses the Call List function to put the list in the display. Note that all the z-coordinates are zero, which makes sure the polygon lies in a plane. Polygons that do not lie in a plane can cause unpredictable results.

Experiment with the line

    shape <<Begin(POLYGON);

by changing it to some of the other primitive types. For example, changing it to

    shape <<Begin(TRIANGLES);

results in a different picture.

Figure 8.10 Polygon (left) and Triangles (right)

Controlling the Appearance of Primitives

JSL has several commands that let you tailor-make the appearance of primitive drawing objects. You can also specify the widths of lines and their stippling pattern (that is, whether they are dashed, dotted, and so on.)
Size and Width

To set the point size of rendered objects, use the Point Size command.

Point Size \( (n) \)

where \( n \) is the number of pixels. Note that this might not be the actual number of pixels rendered, depending on other settings such as anti-aliasing and your hardware configuration.

Set the line width using the Line Width command

Line Width\( (n) \)

where \( n \) is the number of pixels. The argument \( n \) must be larger than zero and is, by default, one.

Stippling Pattern

To make stippled lines, use the Line Stipple command.

Line Stipple\( (\text{factor, pattern}) \)

Factor is a stretching factor. Pattern is a 16-bit integer that turns pixels on or off. Use Enable(LINE_STIPPLE) to turn the effect on.

To construct a line stippling pattern, write a 16-bit binary number that represents the stippling pattern that you desire. Note that the pattern should read from right to left, so your representation might seem backward to the way it is rendered. Convert the binary number to an integer and use this as the pattern parameter.

For example, imagine you want the dotted line pattern \( 0000000011111111 \). This is equal to 255 in decimal notation, so use the command

\[
\text{Line Stipple(1, 255)}
\]

The factor parameter expands each binary digit to two digits. In the example above, \( \text{Line Stipple(2, 255)} \) would result in \( 00000000000000001111111111111111 \).

For example, the following script draws three lines, each of different widths (the Line Width commands) and stippling patterns.

```cpp
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 200, 200 );

// put the scene in a window.
New Window( "Stipples", scene );
scene << Ortho(-2,2,-2,2,-1,1);
scene << color(0,0,0);
//set the RGB color of the text
scene << Enable(LINE_STIPPLE);
scene << Line Width(2);
scene << Line Stipple(1, 255);
scene << Begin(LINES);
```
scene << Vertex(-2, -1, 0);
scene << Vertex(2, -1, 0);
scene << End();

scene << Line Width(4);
scene << Line Stipple(1, 32767);

scene << Begin(LINES);
scene << Vertex(-2, 0, 0);
scene << Vertex(2, 0, 0);
scene << End();
scene << Line Width(6);
scene << Line Stipple(3, 51);

scene << Begin(LINES);
scene << Vertex(-2, 1, 0);
scene << Vertex(2, 1, 0);
scene << End();
scene << Update;

---

**Figure 8.11** Stipples

```
.. .. .. .. ..
```

---

**Note:** Stipple patterns “crawl” on rotating models because they are in screen pixels, not model units, and lines in the model change length on the screen even though nothing changes in model units.

**Fill Pattern**

Polygons are rendered with both a front and a back, and the drawing mode of each side is customizable. This enables the user to see the difference between the back and front of the polygon.

To set the drawing mode of a polygon, use the *Polygon Mode* command.

```
Polygon Mode (face, mode)
```

where *face* can be FRONT, BACK, or FRONT_AND_BACK, and *mode* can be POINT, LINE, or FILL.
Figure 8.12 Points, Line, and Fill Modes

For example, the following script creates a display list that defines a triangle. This display list is used three times in conjunction with Translate, Rotate, and Color commands to draw triangles in three positions. In addition, the Polygon Mode command changes the drawing mode of each triangle. Note there is no explicit call to the FILL mode, since it is the default.

The following table dissects the script, showing how the Translate and Rotate commands accumulate to manipulate a single display list.

### Table 8.2 Translate and Rotate Commands

<table>
<thead>
<tr>
<th>Code from above script</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape = Scene Display List();</td>
<td>Creates a display list</td>
</tr>
<tr>
<td>shape &lt;&lt; Begin(TRIANGLES);</td>
<td>Creates a display list named shape that holds vertices for the triangles. All the z vertices are zero since this is a two dimensional scene</td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(0, 0, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(-1, 2, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; Vertex(1, 2, 0);</td>
<td></td>
</tr>
<tr>
<td>shape &lt;&lt; End();</td>
<td></td>
</tr>
<tr>
<td>scene = Scene Box( 200, 200 );</td>
<td>Put the scene in a display box, and create a new window.</td>
</tr>
<tr>
<td>New Window( &quot;Fill Modes&quot;, scene );</td>
<td></td>
</tr>
<tr>
<td>scene &lt;&lt; Ortho2d(-2,2,-2,2);</td>
<td></td>
</tr>
<tr>
<td>scene &lt;&lt; Color(1,0,0);</td>
<td>Draw the first triangle in red.</td>
</tr>
<tr>
<td>scene &lt;&lt; Call List(shape);</td>
<td></td>
</tr>
<tr>
<td>scene &lt;&lt; Update;</td>
<td></td>
</tr>
<tr>
<td>// update the scene to see the triangle</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8.2 Translate and Rotate Commands

<table>
<thead>
<tr>
<th>Code from above script</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>scene &lt;&lt; Rotate (90, 0, 0, 1); scene &lt;&lt; Translate (-0.5, 0, 0); scene &lt;&lt; Color(0, 0.5, 0.5); scene &lt;&lt; Polygon Mode(FRONT_AND_BACK, LINE); scene &lt;&lt; Call List(shape); scene &lt;&lt; Update;</td>
<td>Draw the second triangle in teal. Note that we first rotate the triangle.</td>
</tr>
<tr>
<td>// update the scene to see the triangle</td>
<td>And then translate it.</td>
</tr>
</tbody>
</table>
Some developers use the fill mode in concert with the line mode to draw a filled polygon with a differently colored border. However, due to the way the figures are rendered, they sometimes do not line up correctly. The **Polygon Offset** command is used to correct for this so-called “stitching” problem.

**Polygon Offset** *(factor, units)*

To enable offsetting, use `Enable(POLYGON_OFFSET_FILL)`, `Enable(POLYGON_OFFSET_LINE)`, or `Enable(POLYGON_OFFSET_POINT)`, depending on the desired mode. The actual offset values are calculated as \( m \times \text{factor} + r \times \text{units} \), where \( m \) is the maximum depth slope of the polygon and \( r \) is the smallest value guaranteed to produce a resolvable difference in window coordinate depth values. Start with `Polygon Offset(1,1)` if you need this.

An example of **Polygon Offset** is in the Surface Plot platform, when a surface and a mesh are displayed on top of each other, or a surface and contours displayed on top of each other. In either case, the surface would interfere with the lines if the lines were not moved closer or the surface moved farther from the viewer.

### Table 8.2 Translate and Rotate Commands

<table>
<thead>
<tr>
<th>Code from above script</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>scene &lt;&lt; Rotate (90, 0, 0, 1); scene &lt;&lt; Translate (-0.5, -1, 0); scene &lt;&lt; Color(0, 0, 0); scene &lt;&lt; Point Size(5); //large points so they are visible scene &lt;&lt; Polygon Mode(FRONT_AND_BACK, POINT); scene &lt;&lt; Call List(shape); scene &lt;&lt; Update; // update the scene to see the triangle</td>
<td>Draw the third triangle as black points. First rotate. And then translate to get the final picture.</td>
</tr>
</tbody>
</table>
Other uses of Begin and End

Although vertices are typically specified between begin and end statements, there are other commands that are valid. These commands are discussed in other sections of this chapter.

- **Vertex** adds a vertex to the list
- **Color** changes the current color
- **Normal** sets the normal vector coordinates
- **Edge Flag** controls drawing of edges
- **Material** sets material properties
- **Eval Coord** and **Eval Point** generate coordinates
- **Call List** executes a display list.

Drawing Spheres, Cylinders, and Disks

There are several pre-defined commands that allow for quick rendering of spheres, cylinders, and disks. The advantage of these commands is not only their ease-of-use, but that they have special lighting properties (their "normals") built in.

Construction

The following commands are used to construct cylinders, disks, partial disks, and spheres.

**Cylinders**

\[ \text{Cylinder( } \text{baseRadius, topRadius, height, slices, stacks } \) \]

- **baseRadius** is the radius of the cylinder's base. Similarly, **topRadius** is the radius of the top. **height** is the height of the cylinder.
- **slices** can be 10 for a reasonably accurate cylindrical shape. Using QuadricNormals(Smooth) helps the appearance.
- **stacks** sets the number of vertices available for lighting reflections. Use a larger value for **stacks** for accurate "hot-spots".

**Disks**

The following command draws a paper-thin disk with an **innerRadius** hole in the middle.

\[ \text{Disk( } \text{innerRadius, outerRadius, slices, loops } \) \]

Like Cylinder, **slices** controls the accuracy of the curve and **loops** makes more vertices (for lighting accuracy).

- **Partial Disk(** **innerRadius, outerRadius, slices, loops, startAngle, sweepAngle** **)**
The Partial Disk command works like Disk, but with a slice of the disk removed. Specify the part of the disk that is showing using startAngle and sweepAngle.

**Spheres**

The following command draws a sphere with the specified radius.

\[
\text{Sphere}( \text{radius, slices, stacks} )
\]

The slices can be thought of as longitudes and stacks as latitudes. About 10 of each make a nicely drawn sphere.

**Lighting**

It is not necessary to make specific calculations of normal vectors (as is the case for customized surfaces) for spheres, disks, and cylinders. However, you can use the following commands to tailor the automatic lighting.

Quadric Normals(mode) tells what type of normal should be automatically generated. The parameter mode can be None, Flat, or Smooth. Flat makes faceted surfaces. Smooth makes the normals at each vertex be the average of the adjacent polygons.

Quadric Orientation(mode) determines which way the normals point. The parameter mode can be Inside or Outside.

Quadric Draw Style(mode) specifies the drawing mode. The parameter mode can be Fill, Line, Silhouette, or Point.

JMP uses the values that you set for Quadric Normals, Quadric Orientation, and Quadric Draw Style for subsequently generated cylinders, disks, and spheres.

---

**Figure 8.13 Draw Styles**

- Fill
- Line
- Silhouette
- Point

---

**Note:** Other OpenGL documentation refers to quadric objects. JMP has only one, and always uses it.

**Drawing Text**

As shown in the “Hello World” example above, text is added to a scene using the Text command.
Text( horz, vert, size, string, <billboard>)

- horz can be Left, Center, or Right justification.
- vert can be Top, Middle, Baseline, or Bottom justification.
- size represents the height of a capital letter M in model coordinates.
- string is the text to draw.
- billboard is an optional argument that causes the text to rotate with the model. Text with this option always faces the viewer.

The font is always the JMP Text font. You can change the text font from the preferences menu, but because of the way JMP caches fonts for scenes, changes might not take effect until JMP is restarted.

**Note:** Text is not part of the standard OpenGL definition.

### Using Text with Rotate and Translate

The following example uses the text command in conjunction with the Translate and Rotate commands.

```c
/* make a scene box...holds an OpenGL scene * /
scene = SceneBox( 600, 600 );

/* put the scene in a window */
NewWindow( "Example 2", scene );

scene << Perspective( 45, 3, 7 );
   /* the "lens" is 45 degrees, near is 3 units from the camera, far is 7 */
scene << Translate( 0.0, 0.0, -4.5 );
   /* move the world so 0,0,0 is visible in the camera */
scene << Rotate( 30, 0, 1, 0 );
   /* rotate the first text about the Y (vertical on screen) axis */
scene << Color( 1, 0, 0 );
   /* pure red */
scene << Text( center, baseline, .2, "First Red String" );
scene << Translate( 0.0, 0.0, -2.0 );
   /* the next string is even farther away from the camera */
scene << Rotate( 30, 0, 1, 0 );
   /* rotate the second text about the Y (vertical on screen) axis */
scene << Color( 0, 1, 0 );
   /* pure green */
scene << Text( center, baseline, .2, "Second Green very long string" );
scene << Update;
   /* update the displaybox in the window using the current display list */```

```
Figure 8.14 Rotating and Translating Text Strings

Note the green string is extending backwards beyond the far clipping plane. Change the 7 to 10 in the Perspective command to see the complete string.

Using the Matrix Stack

JMP 3-D scenes use a matrix stack to keep track of the current transform. The stack is initialized to the identity matrix, and each time a translate, rotate, or scale command is given, the top matrix on the stack is changed.

Confusion alert! Unlike many OpenGL implementations, JMP does not use a transposed matrix.

The JSL example below uses Push Matrix and Pop Matrix to position pieces of the toy top and then return to the origin. This is faster than using the Translate command a second time in reverse.

Figure 8.15 Drawing With a Matrix Stack

toyTop = SceneDisplayList();
toyTop<<PushMatrix;
    toyTop<<Translate(0,0,.1);
There are some cases where you want to replace the current matrix on the stack. For these cases, use the Load Matrix command.

Load Matrix(m)
where $m$ is a 4x4 JMP matrix that is loaded onto the current matrix stack.

Similar is the Mult Matrix command

```
Mult Matrix(m)
```

When the Mult Matrix command is issued, the matrix on the top of the current matrix stack is multiplied by $m$.

The following matrices perform some simple commands.

Translation:

```
[1 0 0 x
0 1 0 y
0 0 1 z
0 0 0 1]
```

In the following rotation matrices, $c = \cos(\text{angle})$ and $s = \sin(\text{angle})$.

Rotation about $x$-axis:

```
[1 0 0 0
0 c -s 0
0 -s c 0
0 0 0 1]
```

Rotation about $y$-axis:

```
[c 0 s 0
0 1 0 0
-s 0 c 0
0 0 0 1]
```

Rotation about $z$-axis:

```
[c -s 0 0
s c 0 0
0 0 1 0
0 0 0 1]
```

For example, here are two equivalent (except for the translation being opposite) ways to translate and rotate a display list.

```
// first way uses matrix
gl<<Push Matrix;
xt = identity(4); // translate this one left by .75
xt[1,4]=-0.75;
xr = Identity(4); // rotate this one, cos needs radians, not degrees
xr[2,2]=cos(3.14159*a/180);
xr[2,3]=-sin(3.14159*a/180);
xr[3,2]=sin(3.14159*a/180);
xr[3,3]=cos(3.14159*a/180);
yr = Identity(4);
yr[1,1]=cos(3.14159*a/180);
yr[1,3]=sin(3.14159*a/180);
```
Three-Dimensional Scenes

Lighting and Normals

The following methods enable you to add lighting, materials, and normal vectors to your shapes. Using these methods, models can appear shiny or light-absorbing.

Creating Light Sources

Light sources are specifications of a color, position, and direction. JSL allows for up to eight lights (numbered 0 to 7) defined by the Light command, where \( n \) is the number of the light.

\[
\text{Light( } n, \text{ parameter, value, ... value )}
\]

Note: To turn each light on, issue an Enable (Lighting) and an Enable (light\( n \)) command, where \( n \) is the light number. Then, move the light to a position in the scene with a Light(\( n \), POSITION, \( x \), \( y \), \( z \)) command.

The value of parameter can be any one of those shown in Table 8.3. The table shows default values for each parameter.
Table 8.3 Light Parameters and Default Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBIENT</td>
<td>(0, 0, 0, 1)</td>
<td>Ambient RGBA intensity</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>(1, 1, 1, 1)</td>
<td>diffuse RGBA intensity</td>
</tr>
<tr>
<td>SPECULAR</td>
<td>(1, 1, 1, 1)</td>
<td>specular RGBA intensity</td>
</tr>
<tr>
<td>POSITION</td>
<td>(0, 0, 1, 0)</td>
<td>((x, y, z, w)) position</td>
</tr>
<tr>
<td>SPOT_DIRECTION</td>
<td>(0, 0, -1)</td>
<td>((x, y, z)) direction of spotlight</td>
</tr>
<tr>
<td>SPOT_EXPONENT</td>
<td>0</td>
<td>spotlight exponent</td>
</tr>
<tr>
<td>SPOT_CUTOFF</td>
<td>180</td>
<td>spotlight cutoff angle</td>
</tr>
<tr>
<td>CONSTANT_ATTENUATION</td>
<td>1</td>
<td>constant attenuation factor</td>
</tr>
<tr>
<td>LINEAR_ATTENUATION</td>
<td>0</td>
<td>linear attenuation factor</td>
</tr>
<tr>
<td>QUADRATIC_ATTENUATION</td>
<td>0</td>
<td>quadratic attenuation factor</td>
</tr>
</tbody>
</table>

**Note:** The default values for DIFFUSE and SPECULAR in this table only apply to Light 0. For other lights, the default value is (0, 0, 0, 1) for both parameters.

The first three parameters (AMBIENT, DIFFUSE, and SPECULAR) are used to color the light. DIFFUSE is the parameter that is most closely associated with the physical color of the light. AMBIENT refers to the property of the light when it functions as a background light. SPECULAR alters the way a light is reflected off a surface.

Specify the position of the light using the POSITION parameter. Nonzero values of the fourth \((w)\) coordinate position the light in homogenous object coordinates.

Light in the real-world decreases in intensity as distance from the light increases. Since a directional light is infinitely far away, it does not make sense to attenuate its intensity as a function of distance. However, JSL attenuates a light source by multiplying the contribution of the source by an attenuation factor

\[
\text{attenuation factor} = \frac{1}{c + ld + qd^2}
\]

where \(c = \text{CONSTANT\_ATTENUATION}\), \(l = \text{LINEAR\_ATTENUATION}\), and \(q = \text{QUADRATIC\_ATTENUATION}\).

To create a spotlight, limit the shape of the light to a cone. Use the SPOT_CUTOFF parameter to define the side of the cone, as shown in the following illustration.
In addition to the cutoff angle, you can control the intensity and direction of the light distribution in the cone. SPOT_DIRECTION specifies the direction for the spotlight to point; SPOT_EXPONENT influences how concentrated the light is.

**Lighting Models**

Lighting models are specified with the `Light Model` command.

```
Light Model( parameter, value,...,value )
```

Light models specify three attributes of lights.
- The global ambient light intensity
- Whether the viewpoint is local or is an infinite distance away
- Whether lighting calculations should be performed differently for the front and back faces of objects.

Table 8.3 “Light Parameters and Default Values,” p. 301 shows the three valid parameters for the `Light Model` command.

**Table 8.4 Light Model Parameters and Default Values**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIGHT_MODEL_AMBIENT</td>
<td>(0.2, 0.2, 0.2, 1)</td>
<td>Ambient RGBA intensity of the entire scene</td>
</tr>
<tr>
<td>LIGHT_MODEL_LOCAL_VIEWER</td>
<td>0 (false)</td>
<td>How specular reflection angles are computed</td>
</tr>
<tr>
<td>LIGHT_MODEL_TWO_SIDE</td>
<td>0 (false)</td>
<td>Nonzero values imply two-sided lighting</td>
</tr>
</tbody>
</table>

**Normal Vectors**

Normal vectors point in a direction perpendicular to a surface. For a plane, all normals are the same. For a more complicated surface, normals are more complicated. JSL enables you to specify the normal vector for each vertex. These normals specify the orientation of the surface in space, necessary for lighting calculations. Accurate normals assure accurate lighting.
The normal vector is of length 1 and is perpendicular to the vertex. Typically, a vertex is shared between several polygons and a smooth shaded effect is desired, so the perpendicular at the vertex is calculated as a (possibly weighted) average of the polygon’s normals. It is important to calculate the “outward” normal for polygons unless two-sided shading is enabled because only the outer face of the polygon is illuminated. With a scaled polygon, the normal’s length is not 1 after scaling, and the lighting is wrong.

Normal vectors are set at the same time the surface is constructed, and are specified with the Normal command. Use the Enable(NORMALIZE) command to have the normals re-normalized to 1 each time the scene is drawn.

**Shading Model**

The shading model of a polygon is set using the Shade Model command.

\[ \text{Shade Model (mode)} \]

where mode can be SMOOTH (the default) or FLAT. SMOOTH shading interpolates the colors of the primitive from one vertex to the next. FLAT mode duplicates the color of one vertex across the entire primitive.

The following script changes the color at each of a triangle’s vertices. The FILL shade model interpolates the color of the interior automatically.

```cpp
// make a scene box...holds an OpenGL scene.
scene = SceneBox(200, 200);

// put the scene in a window.
NewWindow("Shade Model", scene);
scene << clear;
scene << Ortho2D (-1,1,-1,1);

scene << Shade Model(SMOOTH);
scene << Polygon Mode (FRONT_AND_BACK, FILL);
scene << Begin(TRIANGLES);
scene << color(0, 0, 0); // black
scene << Vertex(-1, -1, 0);
scene << Color(0, 1, 1); // cyan
scene << Vertex(0, 1, 0);
scene << Color (1, 1, 1); // white
scene << Vertex(1, -1, 0);
scene << End();

scene << Update;
```
Material Properties

To set the material properties of a surface, use the Material command.

Material( face, parameter, value,...value )

face can be Front, Back, or Front_and_back. (Note that the material properties can be set separately for the front and back faces of a polygon.)

Table 8.5 shows the parameters and default values for Material parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBIENT</td>
<td>(0.2, 0.2, 0.2, 1.0)</td>
<td>Ambient color of material</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>(0.8, 0.8, 0.8, 1.0)</td>
<td>Diffuse color of material</td>
</tr>
<tr>
<td>AMBIENT_AND_DIFFUSE</td>
<td></td>
<td>Both AMBIENT and DIFFUSE</td>
</tr>
<tr>
<td>SPECULAR</td>
<td>(0.0, 0.0, 0.0, 1.0)</td>
<td>Specular color of material</td>
</tr>
<tr>
<td>SHININESS</td>
<td>0</td>
<td>Specular exponent that can range from 0 to 128.</td>
</tr>
<tr>
<td>EMISSION</td>
<td>(0, 0, 0, 1)</td>
<td>Emissive color of material</td>
</tr>
</tbody>
</table>

Alpha Blending

The BlendFunc command allows for alpha blending. To use it, send a BlendFunc message to a scene, for example:

    scene << BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)

SRC_ALPHA and ONE_MINUS_SRC_ALPHA are OpenGL constants that tell BlendFunc to use alpha to blend against the existing display buffer. Disabling z-buffer testing or rendering primitives from back to front
might be needed for some applications. By default, the z-buffer tests prevent anything from drawing behind a transparent polygon after it is drawn.

Complete details of all the constants available to `BlendFunc` (many of which are not useful to the JSL programmer) are available in the OpenGL documentation at opengl.org.

**Fog**

Fog enables figures to fade into the distance, making for more realistic models. All types of geometric figures can be fogged. To turn fog on, enable the `FOG` parameter.

```
Enable (FOG)
```

**Example**

The following example uses several of the concepts presented in this section, including lighting, fog, and normalization. It draws a spinning cylinder that is affected by two lights.

```
scene = SceneBox( 300, 300 ); // make a scene box
New Window( "Cylinder", scene ); // put the scene in a window.

for (i=1, i<360, i++,

    scene << Clear;
    // the lens is 45 degrees, near is 3 units from the camera, far is 7.
    scene << Perspective( 50, 1, 10 );
    // move the world so 0,0,0 is visible in the camera
    scene << Translate( 0.0, 0.0, -2 );

    scene<<Enable(Lighting);
    scene<<Enable(Light0);
    scene<<Enable(Light1);
    scene<<Light(Light0,POSITION,1,1,1,1); //near viewer
    scene<<Light(Light0,DIFFUSE,1,0,0,1); //red light
    scene<<Light(Light1,POSITION,-1,-1,-1,1);//behind object
    scene<<Light(Light1,DIFFUSE,.5,.5,1,1); //blue-gray light

    scene<<Enable(Fog);
    scene<<Enable(NORMALIZE);

    scene << Rotate(i,1,0,0);
    scene << Rotate(i*3, 0, 1, 0);
    scene << Rotate ((i*3/2, 0, 0, 1);
    scene << Cylinder(0.5, 0.5, 0.5, 40,10);
    scene << Update;
)
```
A complete discussion of Bézier curves is beyond the scope of this book. JSL has several commands for defining and drawing curves and their associated meshes.

To define a one-dimensional map, use the Map1 command.

\[
\text{Map1}(\text{target, } u_1, u_2, \text{ stride, } \text{ order, } \text{ matrix})
\]

The target parameter defines what the control points represent. Values of the target parameter are shown in Table 8.6. Note that you must use the Enable command to enable the parameter.
Bézier Curves

The second two parameters \((u_1, u_2)\) define the range for the map. The \textit{stride} value is the number of values in each block of storage (in other words, the offset between the beginning of one control point and the beginning of the next control point). The \textit{order} should equal the degree of the curve plus one. The \textit{matrix} holds the control points.

For example, \texttt{Map1(MAP1_VERTEX_3, 0, 1, 3, 4, \textless 4x3 matrix\gg)} is typical for setting the two end points and two control points to define a Bézier line.

You use the \texttt{MapGrid1} and \texttt{EvalMesh1} commands to define and apply an evenly spaced mesh.

\texttt{MapGrid1(\textit{un}, \textit{u1}, \textit{u2})}

sets up the mesh with \textit{un} divisions spanning the range \textit{u1} to \textit{u2}. Code is simplified by using the range 0 to 1.

\texttt{EvalMesh1(\textit{mode}, \textit{i1}, \textit{i2})}

actually generates the mesh from \textit{i1} to \textit{i2}. The \textit{mode} can be either \texttt{POINT} or \texttt{LINE}. The \texttt{EvalMesh1} command makes its own \texttt{Begin} and \texttt{End} clause.

The following example script demonstrates a one-dimensional outlier. A random set of control points draws a smooth curve. Only the first and last points are on the curve. Using \texttt{NPOINTS=4} results in a cubic Bézier spline.

```plaintext
boxwide=500;
boxhigh=400;
gridsize=100; // bigger for finer divisions
NPOINTS = 4;
```

<table>
<thead>
<tr>
<th>\textit{target Parameter}</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP1_VERTEX_3</td>
<td>((x, y, z)) vertex coordinates</td>
</tr>
<tr>
<td>MAP1_VERTEX_4</td>
<td>((x, y, z, w)) vertex coordinates</td>
</tr>
<tr>
<td>MAP1_INDEX</td>
<td>color index</td>
</tr>
<tr>
<td>MAP1_COLOR_4</td>
<td>R, G, B, A</td>
</tr>
<tr>
<td>MAP1_NORMAL</td>
<td>normal coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_1</td>
<td>(s) texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_2</td>
<td>(s, t) texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_3</td>
<td>(s, t, r) texture coordinates</td>
</tr>
<tr>
<td>MAP1_TEXTURE_COORD_4</td>
<td>(s, t, r, q) texture coordinates</td>
</tr>
</tbody>
</table>

Table 8.6 Map1 Target Parameters and Default Values
/* We suggest you use only values between 2 and 8 (inclusively). Numbers
beyond these might be interpreted differently, depending on implementation.
This value is the degree+1 of the fitted curve */

points = J(NPOINTS, 3, 0);
    // create an array of x,y,z triples
for( x = 1, x <= NPOINTS, x++,
    points[x, 1] = (x-1)/(NPOINTS-1) - .5;
    // x from -.5 to +.5
    points[x, 2] = randomuniform() - .5;
    // y is random in same range
    points[x, 3] = 0;
    /* z is always zero, which causes the curve to stay in a plane */
);

spline = SceneBox(boxwide,boxhigh);

spline << ortho( -.6, .6, -.6, .6, -2, 2 );
    /* data from -.5 to .5 in x and y; this is a little larger */

spline<<Enable(MAP1_VERTEX_3);
spline<<MapGrid1(gridsize, 0, 1);
spline<<Color(.2,.2,1); // blue curve

spline<<Map1( MAP1_VERTEX_3, 0, 1, 3, NPOINTS, points );
spline<<Line Width(2); // not-so-skinny curve
spline<<EvalMesh1(LINE, 0, gridsize ); // also try LINE, POINT

spline<<Color(.2, 1, .2);
spline<<Point Size(4); // big fat green points

// show the points and label them
for( i=1, i <= NPOINTS, i++,
spline<<Begin(POINTS);
spline<<Vertex(points[i,1], points[i,2], points[i,3]);
spline<<End;
spline<<Push Matrix;
spline<<Translate(points[i,1], points[i,2], points[i,3]);
spline<<Text(center, bottom, .05,char(i));
spline<<Pop Matrix;
);

New Window("Spline", spline);

http://www.tinaja.com/glib/bezconn.pdf offers an explanation of connecting cubic segments so that both
the slope and the rate of change match at the connection point. This example does not illustrate doing so;
there is only one segment here.
Two-Dimensional Evaluators

Two-dimensional evaluators follow their one-dimensional counterparts, and are used in a similar way.

\[ \text{Map2(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, matrix)} \]
\[ \text{Eval Coord2(u, v)} \]

Values for the \text{target} parameter are the same as those shown in Table 8.6 "Map1 Target Parameters and Default Values," p. 307 with \text{Map1} replaced with \text{Map2} appropriately. The \text{u1, u2, v1, and v2} values specify the range of the two-dimensional mesh.

For example, \text{Map2(MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, <16x3 matrix>)} is typical for setting the 16 points that define a Bézier surface.

Use the \text{MapGrid2} and \text{EvalMesh2} commands to define and apply an evenly spaced mesh.

\[ \text{MapGrid2(un, u1, u2, vn, v1, v2)} \]
\[ \text{EvalMesh2(mode, i1, i2, j1, j2)} \]

sets up the mesh with \text{un} and \text{vn} divisions spanning the range \text{u1} to \text{u2} and \text{v1} to \text{v2}. Code is simplified by using ranges that span 0 to 1.

\[ \text{EvalMesh2(mode, i1, i2, j1, j2)} \]

actually generates the mesh from \text{i1} to \text{i2} and \text{j1} to \text{j2}. The \text{mode} can be \text{POINT}, \text{LINE}, or \text{FILL}. The \text{EvalMesh2} command makes its own \text{Begin} and \text{End} clause.

Using the mouse

Mouse activity is supported through two feedback functions. The \text{Patch Editor.jsl} sample script uses these functions to support the dragging and dropping of points. Part of that script, the call-back function for mouse activity, is explained below. To run the script, open \text{PatchEditor.jsl} in the \text{Scene3D} folder inside the \text{Sample Scripts} folder.

\[ \text{topClick2d = Function( } \{ x, y, m, k \}, \]
\[ \text{dragfunc( x, boxhigh - y, m, 1, 2 ); 1; \)}
\[ \}) ; \]
\[ \text{frontClick2d = Function( } \{ x, y, m, k \}, \]
\[ \text{dragfunc( x, boxhigh - y, m, 1, 3 ); 1; \)}
\[ \}) ; \]
\[ \text{rightClick2d = Function( } \{ x, y, m, k \}, \]
\[ \text{dragfunc( x, boxhigh - y, m, 2, 3 ); 1; \)}
\[ \}) ; \]
\[ \text{Click3d = Function( } \{ x, y, m, k, hitlist \}, \]
\[ \text{If( m == 1,} \]
\[ \text{If( N Items( hitlist ) > 0,} \]
\[ \text{CurrentPoint = hitlist[1][3], /* first matrix in the list is the closest; 3rd element of matrix is ID*/} \]
\[ \text{call function for mouse activity} \]
\[ \text{)} ; \]
\[ \text{)} ; \]
\[ \text{)} ; \]
Three-Dimensional Scenes

Using the mouse

```plaintext
CurrentPoint = 0
); makePatch();
); 0; /* only cares about initial mouse down. return 1 if drag, release is
needed, but then arcball does not happen. */
);

/* after one of the 3 Click2d functions figures out which axis of the model is
represented by the screen X, Y, pass in to this common code */
dragfunc = Function( {x, y, m, ix, iy}, /* ix, iy are the index of the X, Y, or
Z part of the coord in the points matrix */
If( CurrentPoint > 0,
    points[CurrentPoint, ix] = (x / boxwide) * (orthoright - ortholeft) +
    ortholeft;
    points[CurrentPoint, iy] = (y / boxhigh) * (orthotop - orthobottom) +
    orthobottom;
    makepatch();
)
);

When a 2-D function is called, the arguments are X, Y, M, K.
• X and Y are the coordinates of the mouse.
• M shows the state of the mouse and button. M=0 says that the mouse button is up. M=1 says that the
  button was just pressed. M=2 says that the button is down and the mouse is moving. M=3 says that the
  button was just released.
• K is related to the keys Shift, Alt, and control. K=1 for the Shift key, K=2 for the Control (command) key,
  and K=3 for the Alt (Option) key.

The 3-D function is called similarly. The arguments are X, Y, M, K, hitlist, where hitlist is a list of
matrices
[znear, zfar, id1, id2, id3, ...]
znear, zfar is the Z distance from the camera of the near and far edge of the object. The matrices are
sorted from near to far by the midpoint of znear, zfar. The ids in the list are the pushname, loadname,
and popname values you just put in the display list.

The drag functions use a return value to tell if mouse processing should continue. That is the trailing “1”
you see in the functions. Anything else stops the mouse tracking. This is needed because the 2-D and 3-D
functions do not run in parallel. You might want the 2-D to return 0 and the 3-D to return 1 so the tracking
would happen in 3-D rather than 2-D.

Pick Commands

This SceneBox callback gets the 2-D mouse coordinates, and then uses pick to determine the “named”
object under the mouse. For example, hitlist is a 5x5 pixel pick box around x, y; up to 1000 items
returned, but just the leaf names. The format of the return is determined by the last parameter (1 returns a
simple array, 0 returns a sorted (by depth) list of arrays).
Track2d=function({x, y, m, k},
    hitlist = theSceneBox<<pick( x, y, 5, 5, 1000, 1 );
    if ( nrow(hitlist) > 0, // something IS in the pick box
        ... hitlist[1..n] // are names that you put in the display list
    ) );

Contrast this with a call back Track3d function, where the pick rectangle is always 1x1 and picking only happens when the mouse moves. This is almost always what you want, but points are difficult to pick because the 1x1 pick area is the same small size as the point. This function lets you pick without a mouse move.

The Track3d function always provides a depth-sorted list of arrays; each array can describe multiple names in a hierarchy (pushname and popname construct a hierarchy of objects). The sorting can be very slow when thousands of objects are selected. The final argument (1, above) controls whether the pick function replaces the sorted list of arrays with a simple array. The simple array contains only the “leaf” names, not higher level names.

Parameters

Parameters enable you to specify special modes and settings. To enable a parameter, use the Enable(parameter) command. To disable a parameter, use the Disable(parameter) command. Available parameters are shown in Table 8.7.

**Table 8.7 Parameters**

<table>
<thead>
<tr>
<th>ALPHA_TEST</th>
<th>LIGHT5</th>
<th>MAP2_TEXTURE_COORD_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO_NORMAL</td>
<td>LIGHT6</td>
<td>MAP2_TEXTURE_COORD_4</td>
</tr>
<tr>
<td>BLEND</td>
<td>LIGHT7</td>
<td>MAP2_VERTEX_3</td>
</tr>
<tr>
<td>CLIP_PLANE0</td>
<td>LIGHTING</td>
<td>MAP2_VERTEX_4</td>
</tr>
<tr>
<td>CLIP_PLANE1</td>
<td>LINE_SMOOTH</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>CLIP_PLANE2</td>
<td>LINE_STIPPLE</td>
<td>POINT_SMOOTH</td>
</tr>
<tr>
<td>CLIP_PLANE3</td>
<td>MAP1_COLOR_4</td>
<td>POLYGON_OFFSET_FILL</td>
</tr>
<tr>
<td>CLIP_PLANE4</td>
<td>MAP1_INDEX</td>
<td>POLYGON_OFFSET_LINE</td>
</tr>
<tr>
<td>CLIP_PLANE5</td>
<td>MAP1_NORMAL</td>
<td>POLYGON_OFFSET_POINT</td>
</tr>
<tr>
<td>COLOR_LOGIC_OP</td>
<td>MAP1_TEXTURE_COORD_1</td>
<td>POLYGON_SMOOTH</td>
</tr>
<tr>
<td>COLOR_MATERIAL</td>
<td>MAP1_TEXTURE_COORD_2</td>
<td>POLYGON_STIPPLE</td>
</tr>
<tr>
<td>CULL_FACE</td>
<td>MAP1_TEXTURE_COORD_3</td>
<td>SCISSOR_TEST</td>
</tr>
<tr>
<td>DEPTH_TEST</td>
<td>MAP1_TEXTURE_COORD_4</td>
<td>STENCIL_TEST</td>
</tr>
<tr>
<td>DITHER</td>
<td>MAP1_VERTEX_3</td>
<td></td>
</tr>
<tr>
<td>FOG</td>
<td>MAP1_VERTEX_4</td>
<td></td>
</tr>
<tr>
<td>LIGHT0</td>
<td>MAP2_COLOR_4</td>
<td></td>
</tr>
<tr>
<td>LIGHT1</td>
<td>MAP2_INDEX</td>
<td></td>
</tr>
<tr>
<td>LIGHT2</td>
<td>MAP2_NORMAL</td>
<td></td>
</tr>
<tr>
<td>LIGHT3</td>
<td>MAP2_TEXTURE_COORD_1</td>
<td></td>
</tr>
<tr>
<td>LIGHT4</td>
<td>MAP2_TEXTURE_COORD_2</td>
<td></td>
</tr>
</tbody>
</table>
Matrix algebra is a compact notation to represent operations on matrices, which are two-dimensional arrays of numbers. JSL supports matrices as a data type, and implements many matrix operations. Most statistical methods are expressed in compact matrix notation, and using matrices in JSL is a powerful and compact way to program statistical expressions.

For example, least squares regression can be expressed compactly as an expression involving matrix multiplication and inversion:

$$b = (X'X)^{-1}X'y$$

This could be implemented by a JSL expression that looks just like it:

```julia
b = Inv(X'*X)*X'*y;
```

**Notation**

This chapter follows the standard notational practice of representing a matrix with an uppercase bold variable (for example, $A$). Often lowercase bold letters represent matrices that are vectors (for example, $x$).
## Contents

Basics .......................................................... 315
Definitions .................................................. 315
Constructing Matrices ...................................... 315
Numeric Operators .......................................... 317
Manipulating Values ....................................... 319
Subscripts ................................................... 321
Matrix Operators ............................................ 324
Solving Linear Systems .................................... 325
Additional Construction Functions ...................... 327
Decompositions and Normalizations ...................... 329
Build Your Own Matrix Operators ....................... 333
Matrices With the Rest of JMP ............................ 334
Matrices and Data Tables .................................. 334
Matrices and Reports ....................................... 335
Statistical Examples ......................................... 335
Regression Example ......................................... 335
ANOVA Example ............................................ 336
Basics

Definitions

A matrix is a rectangular array of rows and columns of numbers. JSL supports matrix values and a number of matrix operators. Statistical methods are often implemented in matrix expressions.

Scalar is used in this chapter to refer to a simple non-matrix numeric value. If a matrix has only one row or one column, then it is commonly called a row vector or column vector, respectively, or sometimes just vector.

Some of this chapter reintroduces operators and functions that you already know from their use in scalar arithmetic. Pure numeric functions work elementwise on matrices, though there are exceptions, notably multiply and divide, which are not elementwise. The square root of a matrix is the matrix of square roots of the elements, but there are also several types of matrix roots available from other functions, Cholesky and Eigen and SVD.

Constructing Matrices

Use brackets to specify a matrix literal, with the values separated by blanks (or other “white” characters) and the rows separated by either commas or semicolons. For example, a 4-row by 3-column matrix would be specified like this:

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}
\]

The numbers inside might have decimal parts, might be signed, and might be in scientific notation. They are internally stored in double precision floating point. Since white spaces are ignored in JSL, you can use as many spaces, tabs, and returns as you want to make matrices easier for you to read.

\[
r = \begin{bmatrix}
10 & 12 & 14
\end{bmatrix}; \quad \text{row vector}
\]
\[
c = \begin{bmatrix}
11, 13, 15
\end{bmatrix}; \quad \text{column vector}
\]
\[
b = \begin{bmatrix}
20
\end{bmatrix}; \quad \text{1-by-1 matrix}
\]
\[
e = []; \quad \text{empty matrix}
\]

You can specify just signs to get matrices with +1 or –1 elements:

\[
B = \begin{bmatrix}
+ & + & + \\
+ & - & - \\
+ & - & +
\end{bmatrix};
\]
\[
\begin{bmatrix}
1 & 1 & 1 \\
1 & -1 & -1 \\
1 & -1 & 1
\end{bmatrix}
\]

Matrices can contain only numbers, unlike lists, which can contain expressions and other lists.
Constructing Matrices From Expressions

If you have numbers of numeric expressions stored in lists, you can convert it to a matrix by using the Matrix function. Just specify it as a list of expressions, or if you have several rows, a list of lists. For example:

\[
A = \text{matrix}([[1,2,3], [4,5,6], [7,8,9], [10,11,12]]);
\]

When specifying matrices in this way, elements can be expressions that resolve to numbers:

```plaintext
first={1,2,3}; second={4,5,6}; third={7,2*4,3^2};
A = \text{matrix}([first, second, third,}
\{2*5, \text{floor(exp(0)+exp(1)+exp(2))}, \text{floor(pi()*4)}]);
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}
\]
```

Special Matrix Constructors

The Identity function constructs an identity matrix of the dimension that you specify. An identity matrix is a square matrix of zeros except for a diagonal of ones. The only argument specifies the dimension.

```plaintext
\text{Identity}(3);
\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
```

The J function constructs a matrix with the number of rows and columns that you specify as the first two arguments, whose elements are all the third argument, for example:

```plaintext
J(3,4,5);
\[
\begin{bmatrix}
5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5
\end{bmatrix}
\]
```

The Index function generates a row vector of integers from the first argument to the last argument. Double colon :: is an infix operator to do the same thing:

```plaintext
6::10;
\{6,7,8,9,10\}
\text{Index}(1,5);
\{1,2,3,4,5\}
```

The optional increment argument changes the default increment of +1.

```plaintext
\text{index}(0.1, 0.4, 0.1)
```

produces

\{0.1, 0.2, 0.3, 0.4\}

The increment can also be negative, so `\text{index}(6, 0, -2)` produces

\{6, 4, 2, 0\}

The default value of the increment is 1, or -1 if the first argument is higher than the second.
Note that :: preceding an argument (used as a prefix operator rather than an infix operator) is a scoping operator to force its argument to be interpreted as a JSL global variable, not a data table column, as discussed under “Global Scoping Operator,” p. 40 in the “JSL Building Blocks” chapter. To use :: for Index, you must place it between two arguments.

The Shape function reshapes an existing matrix across rows to be the specified dimensions. For example the following changes the 3x4 matrix a into a 12x1 matrix.

```
a=\[1 \ 1 \ 1, \ 2 \ 2 \ 2, \ 3 \ 3 \ 3, \ 4 \ 4 \ 4\];
shape(a, 12, 1)
\[1,1,1,2,2,2,3,3,3,4,4,4\]
```

**Deleting Rows and Columns**

Deleting rows and columns is accomplished by these assignments:

```
A[k, 0] = [];  // to delete the kth row
A[0, k] = [];  //to delete the kth column
```

**Inquiry Functions**

NCol and NRow return the number of columns and rows in a matrix (or data table), respectively:

```
NCol([1 2 3, 4 5 6]); // returns 3, for 3 columns
NRow([1 2 3, 4 5 6]); // returns 2, for 2 rows
```

To determine whether a value is a matrix, use the Is Matrix function, which returns a 1 if the argument evaluates to a matrix:

```
if(IsMatrix(A), clauseForMatrix, elseClause);
```

**Numeric Operators**

**Basic Arithmetic**

You can add, subtract, multiply, and divide matrices, but be aware that the standard multiply operator becomes a matrix multiplier, rather than an elementwise multiplier.

```
A=[1 2 3, 4 5 6, 7 8 9, 10 11 12];
B=[0 1 2, 2 1 0, 0 1 1, 2 0 0];
C=[1 2 3 4, 4 3 2 1, 0 1 0 1];
D=[0 1 2, 2 1 0, 1 2 0];
R = A+B; // matrix addition
\[
\begin{bmatrix}
1 & 3 & 5 \\
6 & 6 & 6 \\
7 & 9 & 10 \\
12 & 11 & 12
\end{bmatrix}
\]
R = A-B; // matrix subtraction
\[
\begin{bmatrix}
1 & 1 & 1 \\
2 & 4 & 6
\end{bmatrix}
\]
```
The usual * and / infix operators, and the equivalent Multiply and Divide functions, do matrix multiplication and matrix division when given two matrix arguments. Given a matrix and a scalar, they perform elementwise multiplication and division. You can also use the Matrix Mult function to indicate matrix multiplication.

To force elementwise multiplication, use :*, or the equivalent EMult function. Recall that while multiplication of scalars is commutative (\(ab = ba\)), multiplication of matrices is not.

To force elementwise division, use :/, or the equivalent EDiv function.

\[R = A \ast C; // \text{matrix multiplication,}\]
\[\begin{bmatrix} 9 & 11 & 7 & 9, \\ 24 & 29 & 22 & 27, \\ 39 & 47 & 37 & 45, \\ 54 & 65 & 52 & 63 \end{bmatrix}\]
\[R = A / D; // \text{matrix division,}\]
\[\begin{bmatrix} 1.5 & 0.5 & 0, \\ 3 & 2 & 0, \\ 4.5 & 3.5 & 0, \\ 6 & 5 & 0 \end{bmatrix}\]
\[R = A :* B; // \text{elementwise multiplication}\]
\[\begin{bmatrix} 0 & 2 & 6, \\ 8 & 5 & 0, \\ 0 & 8 & 9, \\ 20 & 0 & 0 \end{bmatrix}\]
\[R = C * 2; // \text{scalar multiplication}\]
\[\begin{bmatrix} 2 & 4 & 6 & 8, \\ 8 & 6 & 4 & 2, \\ 0 & 2 & 0 & 2 \end{bmatrix}\]
\[R = C / 2; // \text{scalar division}\]
\[\begin{bmatrix} 0.5 & 1.5 & 2, \\ 2 & 1.5 & 1.5, \\ 0 & 0.5 & 0.5 \end{bmatrix}\]
\[R = A :/ B; // \text{elementwise division}\]
\[\begin{bmatrix} . & 2 & 1.5, \\ 2 & . & ., \\ . & 8 & 9, \\ 5 & . & . \end{bmatrix}\]

**Scalar Numeric Library Functions**

The other arithmetic operators and functions work elementwise on matrices:

\[B = \text{Sqrt}(A); // \text{elementwise square root}\]
\[\begin{bmatrix} 1 & 1.414213562373095 & 1.732050807568877, \\ 2 & 2.23606797749979 & 2.449489742783178, \\ 2.645751311064591 & 2.828427124746193, \\ 3.162277660168379 & 3.3166247903554 & 3.464101615137754 \end{bmatrix}\]
Most of the pure numeric functions can be applied to matrices, resulting in a matrix of results. You can mix conformable matrix arguments with scalar arguments. Among the functions that work this way are:

- `Sqrt`, `Root`, `Log`, `Exp`, `^ Power`, `Log10`
- `Abs`, `Mod`, `Floor`, `Ceiling`, `Round`, `Modulo`
- `Sine`, `Cosine`, `Tangent`, `ArcSine`, other trig functions...
- `Normal Distribution`, other probability functions...

**Manipulating Values**

**Concatenation**

`Concat` combines two matrices side by side to form a larger matrix. The number of rows must agree. Double vertical bar ( || ) is an infix operator equivalent for horizontal concatenation.

\[
\text{Identity}(2) || J(2,3,4);
\begin{bmatrix}
1 & 0 & 4 & 4 & 4 \\
0 & 1 & 4 & 4 & 4
\end{bmatrix}
\]

\[B=[1,1]; B || \text{Concat}(\text{Identity}(2), j(2,3,4));\]
\[
\begin{bmatrix}
1 & 1 & 0 & 4 & 4 & 4 \\
1 & 0 & 1 & 4 & 4 & 4
\end{bmatrix}
\]

`VConcat` stacks two matrices on top of each other to form a larger matrix. The number of columns must agree. Vertical-bar-slash ( |/ ) is an infix operator equivalent for vertical concatenation.

\[
\text{Identity}(2) |\!\!/ J(3,2,1); \quad \text{// or} \quad \text{VConcat}(\text{Identity}(2), J(3,2,1));
\begin{bmatrix}
1, & 0, & 1, & 1, & 1, & 1 \\
0, & 1, & 0, & 1, & 1, & 1
\end{bmatrix}
\]

Both `Concat` and `VConcat` support concatenating to empty matrices, scalars, and lists, for example:

\[a=[]; a || [1]; \quad \text{// yields [1]}\]
\[a || \{2\}; \quad \text{// yields [2]}\]
\[a || [3 4 5]; \quad \text{// yields [3 4 5]}\]

There are two in place concatenation operators ||= and |/= which a||=b is equivalent to a=a||b and a|/=b is equivalent to a=a|/b.

**Diagonals**

`Diag` creates a diagonal matrix from a square matrix or a vector. A diagonal matrix is a square matrix whose nondiagonal elements are zero.

\[D=[ 1 -1 1 ];\]
\[\text{Diag}(D);
\begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix}\]
Matrices
Basics

\[
\text{Diag([1,2,3,4]);}
\]
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 4
\end{bmatrix}
\]

\[
\text{Diag([1,2,3,4],[5]);}
\]
\[
\begin{bmatrix}
1 & 2 & 0 \\
3 & 4 & 0 \\
0 & 0 & 5
\end{bmatrix}
\]

\text{VecDiag} \text{ creates a vector from the diagonal elements of a matrix.}

\[
v=\text{vecdiag(}
\begin{bmatrix}
1 & 0 & 0 & 1 \\
5 & 3 & 7 & -1 \\
9 & -9 & 8 & 8 \\
14 & 5 & -4 & 3
\end{bmatrix};
\]
\[
\begin{bmatrix}
1, & 3, & 8, & 3
\end{bmatrix}
\]

\text{VecQuadratic} \text{ is used to calculate the hats in regression that go into the standard errors of prediction or the Mahalanobis or } T^2 \text{ statistics for outlier distances. The first argument is a symmetric matrix, usually an inverse covariance matrix; the second argument is a rectangular matrix with the same number of columns as the symmetric matrix argument.}

\[
\text{VecQuadratic(Sym, X)} \text{ is equivalent to calculating } \text{VecDiag(X*Sym*X')}.
\]

\text{Trace} \text{ returns the sum of the diagonal elements for a square matrix:}

\[
\text{D}=[0,1,2,1,0,1,2,0];
\]
\[
\text{trace(D); // returns 1}
\]

\text{Transpose}

\text{Transpose} \text{ transposes the rows and columns of a matrix. Back-quote (') is a postfix operator equivalent for Transpose, like the common prime or superscript-T notation (} A' \text{ or } A^T).}

\[
\text{A}=[1,2,3,4,5,6,7,8,9,10,11,12];
\]
\[
\text{A'};
\]
\[
\begin{bmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12
\end{bmatrix}
\]
\[
\text{Transpose([1,2,3,4]);}
\]
\[
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
\]

Matrices and Data Tables

Messages to data tables or data table columns can either get data table values as matrices or write values from matrices into data tables. See the section “Converting Between Matrices and Data Tables,” p. 110 in the “Data Tables” chapter.
Subscripts

Use the subscript operator to pick out elements or submatrices from matrices. Subscript is usually written as a bracket notation after the matrix to be subscripted, with arguments for row and column.

Single Element

The expression $A[i,j]$ extracts the element in row $i$, column $j$, returning a scalar number. The equivalent functional form is Subscript$(A,i,j)$.

```plaintext
P = [1 2 3, 4 5 6, 7 8 9];
P[2,3];     // returns 6
Subscript(P,2,3);  // equivalent to P[2,3]
```

Matrix or List Subscripts

You can also give lists or matrices of subscript values, and the result is a matrix of the selected rows and columns. The following expressions select the 2nd and 3rd rows with the 2nd and 1st columns.

```plaintext
P[[2 3],[2 1]]; // matrix subscripts
P[{2,3},{2,1}]; // list subscripts
```

both resulting in:

```
[5  4,
 8 7]
```

Single Subscripts

A single subscript addresses matrices as if all the rows were turned into columns connected end-to-end in a single column. This makes the double subscript $A[i,j]$ the same as the single subscript $A[(i-1) \times \text{ncol}(A)+j]$.

Here is an assortment of ways to get the matrix $[10,14,18]$ using single subscripts:

```plaintext
Q = [2 4 6, 8 10 12, 14 16 18];
Q[{5,7,9}];
Q[[5 7 9]];:
ii = {5,7,9}; Q[ii];
Subscript(Q,ii);
```

This script returns the values 1–9 from the matrix $P$ in order:

```plaintext
for(i=1,i<=3,i++,
    for(j=1,j<=3,j++,
        show(P[i,j])));
```

Selecting Whole Rows or Columns

A subscript argument of zero selects all rows or columns. For example:

```plaintext
P[0,2];     // results in column 2: [2,5,8]
```
Assignment Through Subscripts

You can insert new values into matrices using subscripts. The subscripts can be single indices, matrices or lists of indices, or the zero index representing all rows or columns. The number of selected rows and columns for the insertion must either match the dimension of the inserted argument, or the argument can be a scalar inserted repeatedly into the indexed positions.

\[ P[2,3]=99; \] // insert a 99 into row 2, column 3
\[ p[[1\ 2],[2\ 3]]=\begin{bmatrix} 66 & 77 & 88 & 99 \end{bmatrix}; \] // result P is \[ \begin{bmatrix} 1 & 66 & 77 & 88 & 99, 7 & 8 & 9 \end{bmatrix} \]
\[ P[0,2] = \begin{bmatrix} 11 & 22 & 33 \end{bmatrix}; \] // inserts the three values into column 2
\[ p[3,0] = \begin{bmatrix} 100 & 102 & 104 \end{bmatrix}; \] // inserts the three values into row 3
\[ p[2,0]=99; \] // replaces the second row with all 99s

Operator Assignment

You can use operator-assignment (such as \( += \)) on matrices or subscripts of matrices. For example, the following statement adds 1 to the \( i-j \)th element of the matrix:
\[ P[i,j] += 1; \]

Operator assignments to subscripted matrices can map very generally. For example to multiply each negative element of a matrix by \(-1\):
\[ A[Loc(A<0)] *= -1; \] // another way of doing \( A=Abs(A) \)

Ranges of Rows or Columns

It is common to need to work with a range of subscripts. The \texttt{Index} operator, mentioned previously and written with a double-colon, is used to create matrices of ranges.

\[ 4::7; \] // creates the matrix \[ \begin{bmatrix} 4 & 5 & 6 & 7 \end{bmatrix} \]
\[ Index(4,7); \] // the equivalent function notation for 4::7
\[ T[1::3,10::14]; \] // refers to rows 1 to 3, columns 10 to 14

\texttt{Loc}

\texttt{Loc} creates a matrix of positions that locate where \( A \) is true, that is, nonzero and nonmissing.

\[ A = \begin{bmatrix} 0 & 1 & 0 & 3 & 4 & 0 \end{bmatrix}; \]
\[ I = Loc(A); \] // returns I as \[ \begin{bmatrix} 2 & 4 \end{bmatrix} \], the nonzero positions
\[ B = \begin{bmatrix} 2 & 0 & 0 & 0 & 5 & 6 \end{bmatrix}; \]
\[ I = Loc(A<8); \] // returns the indices where true, \[ \begin{bmatrix} 1 & 5 \end{bmatrix} \]

The following script replaces negative values in \( A \) with zero:
\[ A[Loc(A<0)]=0; \]
**Loc Min and Loc Max**

Loc Min and Loc Max return the position of the minimum and maximum elements of a matrix. Elements of a matrix are numbered consecutively, starting in the first row, first column, and proceeding left to right. For example,

\[
A = \begin{bmatrix} 1 & 2 & 2 & 2 & 4 & 4 & 1 & 1 & 1 \end{bmatrix}; \\
B = [6, 12, 9];
\]

Show( Loc Max( A ) );
Show( Loc Min( B ) );

results in the output

\[
\text{Loc Max}(::A) = 5 \\
\text{Loc Min}(::B) = 1
\]

**Loc Sorted**

Loc Sorted creates a matrix of positions in A that have values less than or equal to given values in B. For example,

\[
A = \begin{bmatrix} 2 & 4 & 6 & 8 \end{bmatrix}; \\
B = [2, 5, 8, 9]; \\
I = \text{Loc Sorted}(A, B);
\]

results in an I of

\[
[1, 2, 4, 4]
\]

indexing the values 2, 4, 8, and 8 of A. The returned values are always 1 or greater. Loc Sorted is mainly used in conjunction with interpolation routines that need to do fast lookups for boundary values. Note that A must be sorted in ascending order.

**Comparisons, Range Checks, and Logical Operators**

JMP's comparison, range check, and logical operators also work with matrices and produce matrices of elementwise Boolean results. You can compare a matrix with a conformable matrix, or a matrix with a scalar.

\[
A<B; \quad // \text{less than} \\
A<=B; \quad // \text{less or equal} \\
A>B; \quad // \text{greater than} \\
A>=B; \quad // \text{greater or equal} \\
A==B; \quad // \text{equal to} \\
A!=B; \quad // \text{not equal to} \\
A<B<C; \quad // \text{continued comparison (range check)} \\
A|B; \quad // \text{logical OR} \\
A&B; \quad // \text{logical AND}
\]

You can use the Any or All operators to summarize matrix comparison results. Any returns a 1 if any element is nonzero. All returns a 1 if all elements are nonzero.

\[
[2, 2] == [1, 2] \quad // \text{returns } [0, 1], \text{therefore:} \\
\text{All}([2, 2] == [1, 2]) \quad // \text{returns } 0 \\
\text{Any}([2, 2] == [1, 2]) \quad // \text{returns } 1
\]
Min or Max return the minimum or maximum element from the matrix or matrices given as arguments. Using the matrices A and B defined earlier:

\[
\begin{align*}
A &= \begin{bmatrix} 1 & 2 & 3, & 4 & 5 & 6, & 7 & 8 & 9, & 10 & 11 & 12 \end{bmatrix}; \\
B &= \begin{bmatrix} 0 & 1 & 2, & 2 & 1 & 0, & 0 & 1 & 1, & 2 & 0 & 0 \end{bmatrix}; \\
\text{min}(A); & // \text{ returns } 1 \\
\text{max}(A); & // \text{ returns } 12 \\
\text{min}(A,B); & // \text{ returns } 0 \\
\end{align*}
\]

**Ranking and Sorting**

Rank returns a column vector of ranks for the elements of a row or column vector. These ranks can be used to produce a sorted column vector. For example:

\[
\begin{align*}
E &= \begin{bmatrix} 1 & -2 & 3 & -4 & 0 & 5 & 1 & 8 & -7 \end{bmatrix}; \\
R &= \text{Rank}(E); // \text{ returns } [9,4,2,5,7,1,3,6,8] \\
\text{sortedE} &= E[R]; // \text{ returns } [-7,-4,-2,0,1,1,3,5,8] \\
\end{align*}
\]

Since single subscripts result in column vectors, if you want a sorted row vector, you need to transpose the result. For example, \text{sortedE} = E[R]`

Ranking Tie(vector) returns a vector of ranks of the values of vector, with ranks for ties averaged. Similarly, Ranking(vector) returns a vector of ranks of the values of vector, low to high as 1 to \( n \), but the ties are ranked arbitrarily.

You can also use Sort Ascending and Sort Descending to obtain a copy of the matrix supplied sorted as specified, as described for lists.

**Summarizing Columns**

There are several functions that return a row vector based on summary values for each column. For example, given this matrix:

\[
mymatrix = \begin{bmatrix} 11 & 22, & 33 & 44, & 55 & 66 \end{bmatrix};
\]

the following row vectors are returned:

\[
\begin{align*}
\text{V Max}(\text{mymatrix}) & \begin{bmatrix} 55 & 66 \end{bmatrix} \\
\text{V Min}(\text{mymatrix}) & \begin{bmatrix} 11 & 22 \end{bmatrix} \\
\text{V Mean}(\text{mymatrix}) & \begin{bmatrix} 33 & 44 \end{bmatrix} \\
\text{V Sum}(\text{mymatrix}) & \begin{bmatrix} 99 & 132 \end{bmatrix} \\
\text{V Std}(\text{mymatrix}) & \begin{bmatrix} 22 & 22 \end{bmatrix} \\
\end{align*}
\]

**Matrix Operators**

This section discusses JMP's capabilities for solving various matrix equations.
Solving Linear Systems

JMP has four related operators for solving linear systems: Inverse, GInverse, Solve, and Sweep.

**Inverse or Inv**

Inverse returns the matrix inverse for the square, nonsingular matrix argument. Inverse can be abbreviated Inv also. For a matrix A, the matrix product of A and Inverse(A) (often denoted A(A⁻¹)) returns the identity matrix, though as you can see in the example, there can be small discrepancies due to floating point limitations.

\[
\begin{align*}
A &= \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 6 & 7 \end{bmatrix}; \\
A_{\text{Inv}} &= \text{Inverse}(A); \\
A \times A_{\text{Inv}}; & \quad \text{returns} \quad \begin{bmatrix} 1.110223025 \times 10^{-16} & 0.110223025 \\ 0.110223025 & 1 \end{bmatrix}
\end{align*}
\]

**GInverse**

A (Moore-Penrose) generalized inverse of a matrix A is any matrix G such that:

\[
\begin{align*}
AGA &= A \\
GAG &= G \\
(GA)\top &= AG \\
(GA)\top &= GA
\end{align*}
\]

The GInverse function accepts any matrix, including non-square ones, and uses singular-value decomposition to calculate the Moore-Penrose generalized inverse. This is useful in inverting a matrix that is not full rank. For example, a solution of the system

\[
\begin{align*}
x + 2y + 2z &= 6 \\
2x + 4y + 4z &= 12 \\
x + y + z &= 1
\end{align*}
\]

is found using the script

\[
\begin{align*}
A &= \begin{bmatrix} 1 & 2 & 2 \\ 2 & 4 & 4 \\ 3 & 6 & 7 \end{bmatrix}; \\
B &= \begin{bmatrix} 6 \\ 12 \\ 1 \end{bmatrix}; \\
\text{Show}(\text{GInverse}(A) \times B);
\end{align*}
\]

**Solve**

Solve solves a linear system, that is it finds the vector x for the square, nonsingular matrix A and vector b you specify such that \( x = A^{-1}b \). The matrix A and vector b must have the same number of rows.

\[
\begin{align*}
A &= \begin{bmatrix} -4 & 2 & 3 & 2 & 0 & 4 & -1 \end{bmatrix}; \\
b &= \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}; \\
x &= \text{solve}(A,b); \\
\text{Show}(x);
\end{align*}
\]

\[
\begin{align*}
A^x &= \begin{bmatrix} -16.999999999999998 & 4.999999999999998 \\ 18.999999999999999 \\ 0.999999999999999 \\
\end{bmatrix}
\end{align*}
\]
Solve(A, b) is the same as Inverse(A) * b, but Solve is more efficient and is preferred. Note that results can be rounded differently on different machines.

**Sweep**

The Sweep function is a way to invert parts of a square matrix, identified by pivot indices, in such a way that if you sequence through all the pivot indices, you end up with the matrix inverse. Normally the matrix must be positive definite (or negative definite) so that the diagonal pivots never go to zero.

Suppose matrix E consists of smaller matrix partitions, A, B, C, and D, to look like this:

\[
E = \begin{bmatrix} A & B \\ C & D \end{bmatrix}
\]

\[
E = (A|B) / (C|D);
\]

Then the command:

\[
\text{Sweep}(E, \{\ldots\}); // where \{\ldots\} indicates partition A
\]

produces the matrix result equivalent to:

\[
\begin{bmatrix}
A^{-1} & A^{-1}B \\
-CA^{-1} & D - CA^{-1}B
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{inv}(A) \mid \text{inv}(A) * B \\
-C*\text{inv}(A) \mid D-C*\text{inv}(A) * B
\end{bmatrix}
\]

The submatrix in the A position becomes the inverse. The submatrix in the B position becomes the solution to \(Ax = B\). The submatrix in the C position becomes the solution to \(xA = -C\).

One reason why Sweep is particularly useful is that it is sequential and reversible. That is, \(A = \text{Sweep}(A, \{i, j\})\) is the same as \(A = \text{Sweep(\text{Sweep}(A, i), j)}\): it is sequential. And \(A = \text{Sweep(\text{Sweep}(A, i), i)}\) restores A to its original values: it is reversible.

If you have a cross-product matrix partitioned as follows:

\[
C = \begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}
\]

Then after sweeping through the indices of \(X'X\), the result is:

\[
\begin{bmatrix}
(X'X)^{-1} & (X'X)^{-1}X'y \\
yX(X'X)^{-1} & yy - y'X(X'X)^{-1}X'y
\end{bmatrix}
\]

The partitions are recognizable to statisticians as the least squares estimates for the model \(Y = Xb + e\) in the upper right, the sum of squared errors in the lower right, and a matrix proportional to the covariance of the estimates in the upper left. This property, when combined with the properties of being sequential and
reversible, make the `sweep` function very useful in doing the partial solutions needed for stepwise regression.

The `index` argument is a vector that lists the rows (or equivalently the columns) on which you want to sweep the matrix. For example, if `E` is a $4 \times 4$ matrix, to sweep on all three rows to get $E^{-1}$ requires these commands:

```plaintext
E=[ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
sweep(E,[1,2,3,4]);
[ 0.56 -0.44 -0.02 -0.02, -0.44 0.56 -0.02 -0.02, -0.02 -0.02 0.34 -0.16, -0.02 -0.02 -0.16 0.34]
inverse(E); // notice that these are the same
[ 0.56 -0.44 -0.02 -0.02, -0.44 0.56 -0.02 -0.02, -0.02 -0.02 0.34 -0.16, -0.02 -0.02 -0.16 0.34]
```


The `sweep` operator does not check whether the matrix given is positive definite. If the matrix is not positive definite, then it works as long as no zero pivot diagonals are encountered. If zero (or near-zero) pivot diagonals are encountered on a full sweep, then the result is a $g2$ generalized inverse if the zero pivot row and column are zeroed.

`sweep` is further demonstrated in the “ANOVA Example,” p. 336.

**Determinant**

`det` returns the determinant of a square matrix. The determinant of a $2 \times 2$ matrix is the difference of the diagonal products, as demonstrated below. Determinants for $n \times n$ matrices are defined recursively as a weighted sum of determinants for $(n-1) \times (n-1)$ matrices. The determinant must be nonzero for the matrix to have an inverse.

```plaintext
F=[1 2,3 5];
Det(F); // returns -1
```

**Additional Construction Functions**

**Design Matrices**

`design` creates a matrix of design columns for a vector, one column for each unique value in the vector, with indicator values for that value. The design columns have elements 0 and 1. For example, `x` below has values 1, 2, and 3, then the design matrix has a column for 1s, a column for 2s, and a column for 3s. Each
row of the matrix has a 1 in the column representing that row's value. So, the first row (1) has a 1 in the 1s column (the first column) and 0s elsewhere; the second row (2) has a 1 in the 2's column and 0s elsewhere; and so on. A design matrix has as many columns as the vector has unique values.

```plaintext
x=[1, 2, 3, 2, 1];
Design(x);
[1 0 0,
0 1 0,
0 0 1,
0 1 0,
1 0 0]
```

A variation is `DesignNom`, which removes the last column and subtracts it from the others. Thus, the elements of `DesignNom` matrices are 0, 1, and –1, and the `DesignNom` matrix has one less column than the vector has unique values. This operator is used to make full-rank versions of design matrices for effects.

```plaintext
DesignNom(x);
[1 0,
0 1,
-1 -1,
0 1,
1 0]
```

`DesignNom` is further demonstrated in the “ANOVA Example,” p. 336.

In order to facilitate ordinal factor coding, a third Design function, `DesignOrd()`, which functions like `Design()` but produces a full-rank coding such that the row for the low level is all zero, and succeeding levels pack one more 1 across the row of the design matrix.

`Design`, `DesignNom`, and `DesignOrd` support a second argument that specifies the levels to be looked up and their order. This feature allows design matrices to be created one row at a time. The result, as with the one-argument version, is a design matrix of 0s and 1s, each column indicating the level. In the case of `DesignNom`, -1's appear in all columns indicating the last level.

```plaintext
r = Design(values, levels); //create a design matrix of indicator columns
r = DesignNom(values, levels); //create a full-rank design matrix of indicator columns
```

The levels argument is a list or matrix of levels to be looked up. The values argument is a numeric or character value, matrix of numeric values, or list of numeric or character values.

The result has the same number of rows as there are elements in the values argument. If there is only one value, the result has one row. Also, the result always has the same number of columns as there are items in the levels argument, or in the case of `DesignNom` and `DesignOrd`, one less.

If a value is not found, the whole row is zero.

```plaintext
Design(20, [10 20 30]);
[0 1 0]
Design(30, [10 20 30]);
[0 0 1]
DesignNom(20, [10 20 30]);
[0 1]
DesignNom(30, [10 20 30]);
[-1 -1]
```
Chapter 9

Matrices

Matrix Operators

DesignOrd(20, [10 20 30]);

[2 0]

Design([20, 10, 30, 20], [10 20 30])

[ 0 1 0,
  1 0 0,
  0 0 1,
  0 1 0]

Design({"b", "a", "c", "b"}, {"a", "b", "c"});

[ 0 1 0,
  1 0 0,
  0 0 1,
  0 1 0]

Direct Products

Direct Product finds the direct product (or Kronecker product) of two square matrices or scalars. The direct product of an \( m \times m \) matrix and an \( n \times n \) matrix is the \( mn \times mn \) matrix whose elements are the products of terms, one from \( A \) and one from \( B \). The direct product of two incidence matrices can be useful in determining the existence of certain classes of design. For example:

\[
G = \begin{bmatrix}
1 & 2 \\
3 & 5
\end{bmatrix};\]

\[
H = \begin{bmatrix}
2 & 3 \\
5 & 7
\end{bmatrix};\]

Direct product(G,H);

\[
\begin{bmatrix}
2 & 3 & 4 & 6 \\
5 & 7 & 10 & 14 \\
6 & 9 & 10 & 15 \\
15 & 21 & 25 & 35
\end{bmatrix}
\]

H Direct Product finds the horizontal direct product, which is the row-by-row direct product of two matrices with the same number of rows or scalars. For example:

HDirectProduct(G,H);

\[
\begin{bmatrix}
2 & 3 & 4 & 6 \\
15 & 21 & 25 & 35
\end{bmatrix}
\]

HDirect Product is useful in constructing the design matrix columns for interactions.

\[
XA = \text{Design Nom}(A);\]

\[
XB = \text{Design Nom}(B);\]

\[
XAB = \text{HDirect Product}(XA,XB);\]

\[
X = J(NRow(A),1)||XA||XB||XAB;
\]

Decompositions and Normalizations

Eigenvalues

Eigen does eigenvalue decomposition for the symmetric matrix that you specify, returning a list of matrices. The first matrix in the returned list is a vector of eigenvalues. The second matrix is a matrix of column eigenvectors.
Matrices

Chapter 9

Matrix Operators

\[ A = \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}; \]

\[ \text{Eigen}(A); \]

\[ \{(2, 2, 1.999999999999999, -2), \]
\[ \{-0.70710678118655 -0.28867513459481 -0.40824829046386 0.5, \]
\[ 0.707106781186547 -0.28867513459481 -0.40824829046386 0.5, \]
\[ -1.281975124e-16 -0.28867513459481 0.816496580927726 0.5, \]
\[ 0 0.866025403784439 0 0.5\} \]

Since the function returns a list of matrices, you might want to assign it to a list of two global variables, so that the vector of eigenvalues is assigned to one global and the matrix of eigenvectors to another:

\[ \{\text{evals}, \text{evecs}\} = \text{Eigen}(A); \]

Eigenvalue decomposition finds, for some \( n \times n \) matrix \( A \), all scalars \( \lambda \) (lambda) and vectors \( x \) such that the equation \( Ax = \lambda x \) has a nonzero solution \( x \). The \( \lambda \)'s are called eigenvalues, and the corresponding \( x \)'s are called eigenvectors. This is equivalent to solving \( (A - \lambda I)x = 0 \). You can reconstruct \( A \) from eigenvalues and eigenvectors by a statement such as:

\[ \text{newA} = \text{evecs} * \text{diag}(\text{evals}) * \text{evecs}'; \]

The eigenvector matrices are orthonormal, such that the inverse is the transpose: \( E'E = EE' = I \).

Eigenvectors are uniquely determined only if the eigenvalues are unique. Zero eigenvalues correspond to singular matrices. Inverses can be obtained by inverting the eigenvalues and reconstituting with the eigenvectors. Moore-Penrose (see “\( \text{GInverse} \)” p. 325) generalized inverses can be formed by inverting the nonzero eigenvalues and reconstituting. However, in practice, you must use judgment as to whether a very small eigenvalue is effectively zero.

The eigenvalue decomposition allows one to view any square-matrix multiplication as a rotation (multiplication by an orthonormal matrix), a scaling (multiplication by a diagonal matrix), and a reverse rotation (multiplication by the orthonormal inverse, which is the transpose), or in notation:

\[ A*x = E'*\text{diag}(M)*E*x; \quad \text{// E rotates, diag(M) scales, E' reverse-rotates} \]

Eigenvalue decompositions are used in many statistical techniques, notably in principal components and canonical correlation, where the transformation associated with the largest eigenvalues are transformations that maximize variances.

**Cholesky Decomposition**

\( \text{Cholesky} \) does Cholesky decomposition, where a positive definite matrix \( A \) is reexpressed as the product of a real nonsingular lower triangular matrix \( L \) and its transpose: \( A = LL' \).

\[ E = \begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix}; \]

\[ \text{L} = \text{Cholesky}(E); \]

\[ \begin{bmatrix} 2.23606797749979 & 0 & 0 & 0 \\ 1.78854438199832 & 1.341640786499874 & 0 & 0 \\ 0.447213595499958 & 0.149071198499986 & 1.9436506316151 & 0 \\ 0.447213595499958 & 0.149071198499986 & 0.914659120760047 & 1.714985851425088 \end{bmatrix} \]

To verify the results:
Cholesky is often useful for reconstituting expressions into a more manageable form. For example, since eigenvalues are available only for symmetric matrices in JMP, then the eigenvalues of the product $AB$ could not be obtained directly, because though $A$ and $B$ can be symmetric, the product is not. However, you can usually rephrase the problem in terms of eigenvalues of $L'BL$ where $L$ is the Cholesky root of $A$, which has the same eigenvalues. Another use is in reordering matrices within Trace expressions. Expressions like $\text{Trace}(A^*B^*A^*)$ might involve huge operations counts if $A$ has lots of rows, but if $B$ is small and can be factored into $L'L'$ by Cholesky, then it can be reformulated to $\text{Trace}(A^*L^*L^*A^*)$, which is equal to $\text{Trace}(L^*A^*AL)$, which is a much smaller number of operations, being just the sum of squares of the elements of $AL$.

You can use the function `Chol Update()` to efficiently update a Cholesky decomposition. If $L$ is the Cholesky root of an $n \times n$ matrix $A$, then after calling `cholUpdate(L, C, V)`, $L$ will be replaced with the Cholesky root of $A+V^*C^*V'$ where $C$ is an $m \times m$ symmetric matrix and $V$ is an $n \times m$ matrix.

Example:

```plaintext
// The inner product of a design matrix
exS = [16 1 0 11 -1 12, 1 11 -1 -1 1 0, 0 -1 12 -1 1 9, -1 -1 1 -1 9 -1, 12 1 0 9 -1 12];
// Conduct the Cholesky decomposition
exAchol = Cholesky( exS );

// Two column vectors to be applied to change the design matrix
exV = [1 1, 0 0, 0 1, 0 0, 0 0, 0 1];

/* The first column vector is added to one of the rows in the design matrix
The second column vector is subtracted from one of the rows in the design matrix */
exC = [1 0, 0 -1];

// Update the Cholesky decomposition manually
exAnew = exS + exV * exC * exV';
exAcholnew = Cholesky( exAnew );

// Update the Cholesky decomposition more efficiently
exAcholnew_test = Chol Update( exAchol, exV, exC );

// Results are the same
Show( exAcholnew_test );
Show( exAcholnew );
```
Matrices
Chapter 9
Matrix Operators

Singular Value Decomposition

SVD finds the singular value decomposition of a matrix. That is, for a matrix \( A \), SVD returns a list of three matrices \( U \), \( M \), and \( V \) such that \( U^* \text{diag}(M)^*V = A \). When \( A \) is taller than it is wide, \( M \) is more compact, without extra zero diagonals. Singular value decomposition reexpresses \( A \) in the form \( U^*V^\dagger \), where \( U \) and \( V \) are matrices that contain orthogonal column vectors (perpendicular, independent column vectors) and \( S \) is an \( n \times n \) diagonal matrix containing the nonnegative square roots of the eigenvalues of \( A^*A \), the singular values of \( A \). Singular value decomposition is the basis of correspondence analysis.

\[
\begin{bmatrix}
1 & 2 & 1 & 0 \\
2 & 3 & 0 & 1 \\
1 & 0 & 1 & 5 \\
0 & 1 & 5 & 1
\end{bmatrix};
\]

\{U,M,V\}=svd(A);
newA=U*diag(M)*V\';

\[
\begin{bmatrix}
1 & 2 & 1.6895265e-15 \\
2 & 3 & 1.2632499e-15 \\
1 & 4.5864769e-16 & 1 \\
5.955833e-16 & 1 & 5 & 1
\end{bmatrix}
\]

Orthonormalization

Ortho does orthonormalization of the columns of a matrix using the Gram-Schmidt method. That is, it orthogonalizes the columns and then divides the vectors by their magnitudes to normalize them. The column vectors of orthogonal matrices are unit-length and are mutually perpendicular (their dot products are zero). For example:

\[
\begin{bmatrix}
1 & -1,1 & 0 & 0 & 1 \\
0.408248290463863 & 0.70710678118655 \\
0.408248290463863 & 0.707106781186547 \\
-0.81649658092773 & 3.145090223e-16
\end{bmatrix}
\]

\[
B= \text{ortho}([1 -1,1 0,0 1]);
\]

\[
\begin{bmatrix}
0.408248290463863 & -0.35355339059327 \\
0.408248290463863 & 0.353553390593274 \\
-0.81649658092773 & 1.572545112e-16
\end{bmatrix}
\]

You can verify that these vectors are orthogonal by multiplying \( B \) by its transpose, which should yield the identity matrix.

\[
C=B^\dagger*B;
\]

\[
\begin{bmatrix}
1 & -3.205621207e-16 \\
-3.205621207e-16 & 1
\end{bmatrix}
\]

By default, vectors are normalized. That is, they are divided by their magnitudes, which scales them to have length 1. Include the option \text{Scaled(0)} to turn scaling off:

\[
\text{ortho}([1 -1,1 0,0 1], \text{scaled(0))};
\]

\[
\begin{bmatrix}
0.408248290463863 & -0.35355339059327 \\
0.408248290463863 & 0.353553390593274 \\
-0.81649658092773 & 1.572545112e-16
\end{bmatrix}
\]

Include the option \text{Centered(1)} to create vectors whose elements sum to zero. This is useful when constructing a matrix of contrasts.

\[
\text{result=ortho}([1 -1,1 0,0 1], \text{centered(1))};
\]

\[
\begin{bmatrix}
0.408248290463863 & -0.70710678118655 \\
0.408248290463863 & 0.707106781186547 \\
-0.81649658092773 & 3.145090223e-16
\end{bmatrix}
\]
You can verify that the elements of each column sum to zero by premultiplying by a vector of ones to sum the columns:

\[ J(1,3) \cdot \text{result}; \]
\[ \begin{bmatrix} 1.110223025e-16 & 2.034867198e-16 \end{bmatrix} \]

**Orthogonal Polynomials**

OrthoPoly returns orthogonal polynomials for a vector of indices representing spacings up to the order given. Orthogonal polynomials can be useful for fitting polynomial models where otherwise some regression coefficients might be highly correlated.

\[ \text{OrthoPoly([1 2 3],2);} \]
\[ \begin{bmatrix} -0.70710678118655 & 0.408248290463862, \\ 0 & -0.81649658092773, \\ 0.707106781186547 & 0.408248290463864 \end{bmatrix} \]

The order must be less than the dimension of the vector. A Scale option produces vectors of unit length, as described above for Ortho.

**QR Decomposition**

QR factorization is useful for numerically stable matrix work. QR returns a list of two matrices. The typical usage is

\[ \{Q, R\} = \text{QR}(X); \]

where Q and R hold the results. For an \( m \times n \) matrix, QR creates an orthogonal \( m \times m \) matrix \( Q \) and an upper triangular \( m \times n \) matrix \( R \) such that \( X = Q \cdot R \).

**Updating Inverse Matrices**

If you have an inverse of an \( M^T \cdot M \) matrix and want to recalculate it to add or drop one or more rows, then the operator Inv Update(S, X, w) uses an updating formula to recalculate it efficiently. This is used in drop-1 influence diagnostics, and also in candidate design evaluation.

The first argument, \( S \), is the matrix to be updated. The second argument, \( X \), is the matrix of rows to be added or dropped, The third argument, \( w \), is either \( 1 \) to add or \( -1 \) to delete the row or rows.

It is equivalent to calculating \( S - w \cdot S \cdot X^T \cdot \text{Inv}(I + w \cdot X \cdot S \cdot X^T) \cdot X \cdot S \), where \( I \) is an identity matrix and Inv(A) means an inverse matrix of A.

Note that multiple rows can be added or deleted.

**Build Your Own Matrix Operators**

Recall from “Macros,” p. 414 in the “Advanced Concepts” chapter, that you can store your own operations in macros. For example, you can make your own matrix operation Norm to find the magnitude of a vector:

\[ \text{norm}=\text{function}([x], \sqrt{x^T \cdot x}); \]

You could similarly make Normalize to divide a vector by its magnitude:
Matrices With the Rest of JMP

Matrices and Data Tables

You can use several methods to move information between a matrix structure and a JMP data table. This way, you can use JMP's capabilities for matrix algebra to perform your own calculations on numbers already stored in JMP data tables, and you can in turn save results back to data tables.

Moving Data from a Data Table to a Matrix

If \( dt \) is a variable containing a reference to a data table, then \( \text{Get As Matrix} \) creates a matrix \( A \) with the values from all numeric columns in the data table:

```julia
dt=Open("$SAMPLE_DATA/Big Class.jmp");
A = dt<<GetAsMatrix;
```

If \( \text{col} \) is a variable containing a reference to a data column in a data table, then \( \text{Get As Matrix} \) creates a matrix \( A \) as a column vector of values from the column.

```julia
col=Column("Height");
A = col<<GetAsMatrix;
```

Of course there are many other ways to do this, such as using \( \text{For} \) loops:

```julia
n=nrow(dt);
a=j(n,1,0);
for(i=1,i<=n,i++,a[i]=col[i]);
```

There are also commands to get certain rows of a data table. \( \text{GetAsMatrix} \) supports column list arguments, either names or character strings.

```julia
dt=current data table();
x=dt<<Get As Matrix("height", "weight");
```

or

```julia
x=dt<<Get As Matrix({height, weight});
```

\( \text{dt<<Get Selected Rows} \) returns a matrix of the currently selected rows in the data table. Note that this can be an empty matrix.

\( \text{dt<<Get Rows Where (expression)} \) returns a matrix of row numbers where the expression is true.

Moving Data from a Matrix to a Data Table

If \( \text{col} \) is a variable containing a reference to a data column in a data table, and \( x \) is a column vector of values, then the following expression copies the values of \( x \) into the column, for as many values are in \( x \).

```julia
\text{col}<<\text{SetValues}(x);
```
In addition, the `As Table (matrix)` command creates a new data table from a matrix argument. The columns are named Col1, Col2, and so forth.

**Matrices and Reports**

You can extract matrices of values from reports in windows. See “Manipulating Displays,” p. 177 in the “Display Trees” chapter to learn how to refer to elements of reports, such as TableBoxes and ColumnBoxes.

If `tableBox` is a variable containing a reference to a table box, then `Get As Matrix` creates a matrix `A` with the values from all numeric columns in the table:

```
A = tableBox<<GetAsMatrix;
```

If `colBox` is a variable containing a reference to a numeric column box in a report table, then `Get As Matrix` creates a matrix `A` as a column vector of values from the column.

```
A = colBox<<GetAsMatrix;
```

For example, to obtain the table of parameter estimates in a bivariate report, you could submit this:

```
biv = Bivariate(X(height), Y(weight), Fit Line);
col = (biv<<report)["Linear Fit", "Parameter Estimates", ColumnBox("Estimate")];
beta = col<<GetAsMatrix;
[-127.145248610915, 3.711354893859555]
```

**Statistical Examples**

These examples demonstrate how to use JSL's matrix algebra capabilities to compute statistics.

**Regression Example**

Suppose that you want to implement your own regression calculation, rather than use the facilities built into JMP. Because of the compact matrix notation, it might require only a few lines of code:

```
Y = [ 98, 112.5, 84, 102.5, 102.5, 50.5, 90, 77, 112, 150, 128, 133, 85, 112];
X = [65.3, 69, 56.5, 62.8, 63.5, 51.3, 64.3, 56.3, 66.5, 72, 64.8, 67, 57.5, 66.5];
X = J(nrow(X),1) || X; // put in an intercept column of 1s
beta = Inv(X'*X)*X'*Y; // the least square estimates
resid = Y-X*beta; // the residuals, Y - predicted
sse = resid'*resid; // sum of squared errors
show(beta, sse);
```

This could easily be expanded into a script that gets its data from a data table, calculates additional results, and shows the results in a report window:

```
// open the data table
bigClass = open("$SAMPLE_DATA/Big Class.jmp");

// get data into matrices
```
Statistical Examples

```plaintext
x = (Column("Age")<getValues) || (Column("Height")<getValues);
x = j(nrow(x),1,1)||x;
y = Column("Weight")<getValues;

// regression calculations
xpx = Inv(x'x);
bet = xpxx'y; // parameter estimates
resid = y-x bet; // residuals
sse = resid'x resid; // sum of squared errors
dfe = nrow(x)-ncol(x); // degrees of freedom
mse = sse/dfe; // mean square error, error variance estimate

// additional calculations on estimates
stdb = sqrt(vecDiag(xpxix) mse); // standard errors of estimates
alpha = .05;
qt = Students t Quantile(1-alpha/2,dfe); // upper 95% confidence limits
betau95 = betat + qt*stdb;
betal95 = betat - qt*stdb; // lower 95% confidence limits
tratio = betat/stdb; // Student's T ratios
probt = (1-TDistribution(abs(tratio),dfe))*2; // p-values

// present results
newWindow("Big Class Regression",
    tableBox(
        StringColBox("Term",{"Intercept","Age","Height"}),
        NumberColBox("Estimate", bet),
        NumberColBox("Std Error", stdb),
        NumberColBox("TRatio", tratio),
        NumberColBox("Prob>|t|", probt),
        NumberColBox("Lower95%", betal95),
        NumberColBox("Upper95%", betau95)));
```

ANOVA Example

You can also implement your own one-way ANOVA. This example steps through this in some detail with a simple problem involving a three-level factor, perhaps indicating Low, Medium, and High doses, and a response measurement. Thus it solves the general linear model:

\[ Y = a + bX + \epsilon \]

where \( Y \) is a vector of responses, \( a \) is the intercept term, \( b \) is a vector of coefficients, \( X \) is a design matrix for the factor, and \( \epsilon \) is an error term.

```plaintext
factor = [1,2,3,1,2,3,1,2,3];
y = [1,2,3,4,3,2,5,4,3];

First you need to build a design matrix for the factor.

designNom(factor);
[ 1 0,
  0 1,
]```
You also need to add a column of 1s to the design matrix, for the intercept term. You can do this by concatenating \( J \) and \( \text{Design Nom} \):

\[
x = J(9,1,1) || \text{designNom(factor)};
\]

Now, to solve the normal equation, you need to construct a matrix \( M \) with partitions like this:

\[
\begin{bmatrix}
X'X & X'y \\
y'X & y'y
\end{bmatrix}
\]

You can do this in one step by concatenating the pieces:

\[
M=(x'x || x'y) \\
\quad / \\
(y'x || y'y);
\]

Now you sweep \( M \) over all the columns in \( X'X \) for the full fit model, and over the first column only for the intercept-only model:

\[
\text{FullFit}=\text{sweep}(M,[1,2,3]); \quad \text{// full fit model}
\]
\[
\text{InterceptOnly}=\text{sweep}(M,[1]); \quad \text{// model with intercept only}
\]

Recall that some of the standard ANOVA results are calculated by comparing the results of these two models. This example focuses on the full fit model, which produces this swept matrix:

\[
\begin{bmatrix}
0.111111111111111 & 0 & 0 & 3, \\
0 & 0.222222222222222 & -0.111111111111111 & 0.333333333333333, \\
0 & -0.111111111111111 & 0.222222222222222 & 0, \\
-3 & -0.33333333333333 & 0.333333333333333 & 0
\end{bmatrix}
\]

Read the model coefficients from the upper right partition of the matrix (notice that the lower left partition is the same except that the signs are reversed): 3, 0.333, 0. Thus the coefficient for the intercept term is 3, the coefficient for the first level of the factor is 0.333, the coefficient for the second level is 0, and by virtue
of the use of Design Nom the coefficient for the third level is the difference, –0.333. The lower right partition of the matrix holds the sum of squares, 11.333.

You could modify this into a generalized ANOVA script by replacing some of the explicit values in the script with parameters.

These results match those from the Fit Model platform:

---

**Figure 9.1 Create an ANOVA Report**

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Sum of Squares</th>
<th>Mean Square</th>
<th>F Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>2</td>
<td>0.846887</td>
<td>0.423448</td>
<td>0.1785</td>
</tr>
<tr>
<td>Error</td>
<td>6</td>
<td>11.33333</td>
<td>1.88889</td>
<td>0.8424</td>
</tr>
<tr>
<td>C. Total</td>
<td>8</td>
<td>11.33333</td>
<td>1.88889</td>
<td></td>
</tr>
</tbody>
</table>

**Parameter Estimates**

<table>
<thead>
<tr>
<th>Term</th>
<th>Estimates</th>
<th>Std Error</th>
<th>t Ratio</th>
<th>Prob &gt;</th>
<th>P</th>
<th>Wald</th>
<th>Prob &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>0.459123</td>
<td>0.06</td>
<td>0.056</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor[1]</td>
<td>0.3323332</td>
<td>0.847984</td>
<td>0.31</td>
<td>0.51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor[2]</td>
<td>0.947984</td>
<td>0.6253</td>
<td>0.00</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is how to construct this display.

1. Build a data table, according to the methods described in the “Data Tables” chapter:

   ```
   factor = [1, 2, 3, 1, 2, 3, 1, 2, 3];
   Y = [1, 2, 3, 4, 3, 2, 5, 4, 3];
   dt = New Table( "foo" )
   dt << New Column( "y", Set Values( ::y ) );
   dt << New Column("factor", Nominal, Values( ::factor ) );
   ```

2. Run a model, as described in “Launching Platforms,” p. 164 in the “Scripting Platforms” chapter:

   ```
   Fit Model( 
   Y( :y ),
   Effects( :factor ),
   Personality( Standard Least Squares ),
   Run Model( 
   y << {Plot Actual by Predicted( 0 ), Plot Residual by Predicted( 0 ),
   Plot Effect Leverage( 0 )}
   )
   );
   ```
3. Use JSL techniques for navigating displays, as discussed in “DisplayBox Object References,” p. 180 in the “Display Trees” chapter.

```js
ranova = Report( Fit Least Squares[1] );
ranova[OutlineBox(5)] << Close(0);
ranova[OutlineBox(6)] << Close(1);
ranova[OutlineBox(8)] << Close(1);
ranova[OutlineBox(4), NumberColBox( 2 )] << select;
ranova[OutlineBox(5), NumberColBox( 1 )] << select;
```
This chapter discusses scripting features that are particularly useful for production settings, such as the following:

- a Datafeed for capturing real-time data from a laboratory instrument
- using SAS or R through JSL
- using JMP with Excel
- connecting to databases
- sharing scripts and adding custom functionality using add-ins
- controlling JMP externally by OLE automation

Some general JSL commands that might be of particular interest for use in a production setting include Caption, Speak, Print, Write, and Mail. These commands are described in “Functions that communicate with users,” p. 55 in the “JSL Building Blocks” chapter.
## Contents

Real-Time Data Capture .................................................. 343
- Create a Datafeed Object ........................................... 343
- Manage a Datafeed with Messages ................................. 344
- Examples of Datafeed ................................................ 346

Dynamic Link Libraries (DLLs) ........................................ 349

Using Sockets in JSL ..................................................... 351
- Socket-Related Commands ........................................... 352
- Messages for Sockets ................................................ 352

Database Access (Windows Only) ..................................... 354

Working with SAS ......................................................... 355
- Make a SAS DATA Step .............................................. 355
- SAS Variable Names ................................................ 355
- Connect to a SAS Metadata Server ............................... 356
- Preferences ........................................................... 358
- Sample Scripts ....................................................... 358

Working with R ........................................................... 359
- Installing R ............................................................ 359
- JMP to R Interfaces .................................................. 360
- R JSL Scriptable Object Interfaces ............................... 360
- Conversion Between JMP Data Types and R Data Types ...... 361
- Troubleshooting ..................................................... 363
- Examples ............................................................. 364

Working with Excel ....................................................... 366

JMP Add-Ins ............................................................. 366
- Workflow for Creating an Add-In ................................. 367
- More About Installation and Registration ....................... 369
- Addin.def File Specification ...................................... 371
- Example JMP Add-In ............................................... 372

OLE Automation ........................................................ 373
- Automating JMP through Visual Basic .......................... 373
- Automating JMP through Visual C++ ......................... 381
Real-Time Data Capture

A Datafeed is a real-time method to read data continuously, such as from a laboratory measurement device connected to a serial port. A Datafeed object sets up a concurrent thread with a queue for input lines that arrive in real time and are processed with background events. You set up scripts to interpret the lines and push the data to data tables, or do whatever else your process requires.

For example, submit this to get records from com1: and list them in the log.

```julia
feed = Open Datafeed(
    Connect( Port( "com1:" ), Baud( 9600 ), DataBits( 7 ) ),
    Set Script( Print( feed << getLine ) )
);
```

Create a Datafeed Object

To create a DataFeed object, use the `Open DataFeed` command with arguments specifying details about the connection:

```julia
feed = Open DataFeed( options );
```

No arguments are required. You can simply create the Datafeed object and then send messages to it. Messages might include connecting to a port or setting up a script to process the data coming in. However, you would typically set up the basic operation of the data feed in the `Open DataFeed` command and subsequently send messages as needed to manage the data feed. Any of the options below work both as options inside `Open DataFeed` or as messages sent to a Datafeed object.

It's a good idea to store a reference to the object in a global variable, such as `feed` above, so that you can easily send messages to the object. You can store a reference to an existing object by using a subscript; for example, to store a reference to the second data feed created:

```julia
feed2 = Data Feed[2];
```

Options

(Windows only) To connect to a live data source, use `Connect( )` and specify details for the port. Each setting takes only one argument; in this syntax summary the symbol `|` between argument choices means “or.” The `Port` specification is needed if you want to connect; otherwise the object still works, but is not connected to a data feed. The last three items, `DTR_DSR`, `RTS_CTS`, and `XON_XOFF`, are Boolean to specify...
which control characters are sent back and forth to indicate when the Datafeed is ready to get data. Typically you would turn on at most one of them.

```javascript
feed = Open Datafeed(
    Connect(
        Port( "com1:" | "com2:" | "lpt1:" | ... ),
        Baud( 9600 | 4800 | ... ),
        Data Bits( 8 | 7 ),
        Parity( None | Odd | Even ),
        Stop Bits( 1 | 0 | 2 ),
        DTR_DSR( 0 | 1 ), // Data Terminal Ready
        RTS_CTS( 0 | 1 ), // Request To Send | Clear To Send
        XON_XOFF( 1 | 0 ) // Transmitter On | Transmitter Off
    )
);
```

This command creates a scriptable data feed object and stores a reference to it in the global variable `feed`. The `Connect` argument starts up a thread to watch a communications port and collect lines. The thread collects characters until it has a line, and then appends it to the line queue and schedules an event to call the script.

### Confusion alert!

For Datafeed purposes, a line of data is a single value (a datum). A line is not to be confused with a data table row, which can contain values for many variables on a subject.

Set Script attaches a script to the Datafeed object. This script is triggered by the On DataFeed handler whenever a line of data arrives. The argument for Set Script is simply the script to run, or a global containing a script.

```javascript
feed = Open Datafeed( set script( myScript ) );
feed = Open Datafeed( set script( Print( feed << getLine ) ) );
```

A Datafeed script typically uses Get Line to get a copy of one line and then does something with that line. Often it parses the line for data and adds it to some data table.

### Manage a Datafeed with Messages

A Datafeed object responds to several messages, including Connect and Set Script. These are detailed above as arguments for Open Datafeed. They can also be sent as messages to a Datafeed object that already exists:

```javascript
feed << connect( port( "com1:" ), baud( 4800 ), databits( 7 ), parity( odd ),
    stopbits( 2 ) );
feed << set script( myScript );
```

The following messages could also be used as arguments to On Data Feed. However, it would be more common to send them as messages to a Datafeed object that is already present.

You can send lines to a Datafeed from a script. This is a quick way to test a Datafeed. Include a text argument or a global that stores text:
Here is a test script to queue five lines of data:

```javascript
feed << Queue Line( "11" );
feed << Queue Line( "22" );
feed << Queue Line( "33" );
feed << Queue Line( "44" );
feed << Queue Line( "55" );
```

To get the first line currently waiting in the queue, use a `Get Line` (singular) message. When you get a line, it is removed from the queue. Five lines were queued with the test script above, and `Get Line` returns the first line and removes it from the queue:

```javascript
feed << Get Line
"11"
```

To empty all lines from the queue into a list, use `Get Lines` (plural). This returns the next four lines from the test script in list `{ }` format.

```javascript
myList = feed << GetLines;
{ "22", "33", "44", "55" }
```
To stop and later restart the processing of queued lines, either click the Stop and Restart buttons in the Datafeed window, or send the equivalent messages:

```
feed << Stop;
feed << Restart;
```

To close the Datafeed and its window:

```
feed << Close;
```

To disconnect from the live data source:

```
feed << Disconnect
```

### Examples of Datafeed

#### Reading Data

Here is a typical Datafeed script. It expects to find a string 3 characters long starting in column 11. If it does, it uses it as a number and then adds a row to the data table in the column “thickness.”

```
feed = Open Datafeed( );
myScript = Expr(
  line = feed << Get Line;
  If( Length( line ) >= 14,
    x = Num( SubString( line, 11, 3 ) );
    If( !Is Missing( x ),
      Current Data Table( ) << Add Row( {thickness = x} )
    );
  );
);
```

Assign the script to the data feed object by using Set Script:

```
feed << Set Script( myScript );
```

#### Set up a Live Control Chart

Here is a sample script that sets up a new data table and starts a control chart based on the data feed.

```
// make a data table with one column
dt = New Table( "Gap Width" );
 dc = dt << New Column( "gap", Numeric, Best );

// set up control chart properties
dt << Set Property( "Control Limits", {XBar( Avg( 20 ), LCL( 19.8 ),
   UCL( 20.2 ) )} );
dt << Set Property( "Sigma", 0.1 );

// make the data feed
feed = Open Datafeed( );
feedScript = Expr(
  line = feed << get line;
```
Real-Time Data Capture

```julia
z = Num( line );
Show( line, z ); // if logging or debugging
If( !Is Missing( z ),
  dt << AddRow( {:gap = z} )
);
);
feed << SetScript( feedScript );

// start the control chart
Control Chart(
  Sample Size( 5 ),
  K Sigma( 3 ),
  Chart Col( gap,
    XBar( Connect Points( 1 ),
      Show Points( 1 ),
      Show Center Line( 1 ),
      Show Control Limits( 1 )
    ),
    R( Connect Points( 1 ),
      Show Points( 1 ),
      Show Center Line( 1 ),
      Show Control Limits( 1 )
    )
  )
);

// Either start the feed from the device or test-feed some data
// to see it work (comment out one of the lines):
// feed<<connect(Port("com1:"), Baud(9600));
For( i = 1, i < 20, i++,
  feed << Queue Line( Char( 20 + Random Uniform( ) * .1 ) )
);
```

Store the Script in a Data Table

You can further automate the production setting by placing a Datafeed script such as the one above in an On Open data table property. A property with this name is run automatically each time the table is opened (unless you set a preference to suppress execution). If you save such a data table as a Template, opening the template runs the Datafeed script and saves data to a new data table file.
### Table 10.1 DataFeed messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Data Feed</td>
<td><code>feed = Open Datafeed( commands )</code></td>
<td>Creates a data feed object. Any of the following can be used as commands inside <code>Open DataFeed</code> or sent as messages to an existing data feed object. Data Feed is a synonym.</td>
</tr>
<tr>
<td>Data Feed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Script</td>
<td><code>feed &lt;&lt; Set Script( script )</code></td>
<td>Assigns the script that is run each time a line of data is received.</td>
</tr>
<tr>
<td>Show Window</td>
<td><code>feed &lt;&lt; Show Window( Boolean )</code></td>
<td>Specifies whether to show a status window for the data feed.</td>
</tr>
<tr>
<td>Get Line</td>
<td><code>feed &lt;&lt; Get Line</code></td>
<td>Returns and removes one line from the Datafeed queue.</td>
</tr>
<tr>
<td>Get Lines</td>
<td><code>feed &lt;&lt; Get Lines</code></td>
<td>Returns as a list and removes all lines from the Datafeed queue.</td>
</tr>
<tr>
<td>Queue Line</td>
<td><code>feed &lt;&lt; Queue Line( string )</code></td>
<td>Sends one line to the end of the Datafeed queue.</td>
</tr>
<tr>
<td>Stop</td>
<td><code>feed &lt;&lt; Stop</code></td>
<td>Stops processing queued lines.</td>
</tr>
<tr>
<td>Restart</td>
<td><code>feed &lt;&lt; Restart</code></td>
<td>Restarts processing queued lines.</td>
</tr>
<tr>
<td>Close</td>
<td><code>feed &lt;&lt; Close</code></td>
<td>Closes the Datafeed object and its window.</td>
</tr>
<tr>
<td>Connect</td>
<td>`feed &lt;&lt; Connect(</td>
<td>(Windows only) Sets up port settings for the connection to the device. The symbol</td>
</tr>
<tr>
<td></td>
<td>Port(&quot;com1:&quot;</td>
<td>&quot;lpt1:&quot;</td>
</tr>
<tr>
<td>Disconnect</td>
<td><code>feed &lt;&lt; Disconnect</code></td>
<td>(Windows only) Disconnects the device from the Datafeed queue but leaves the Datafeed object active.</td>
</tr>
</tbody>
</table>
Dynamic Link Libraries (DLLs)

You can extend JMP functionality by using JSL to load a DLL and call functions exported by that DLL. There is one JSL command and six messages that implement this functionality.

\[
\text{dll_obj} = \text{Load DLL("path"}
\]

Load DLL() loads the DLL in the specified path.

Use the Declare Function message to define it and then call it.

\[
\text{dll_obj} \ll \text{Declare Function("name", Convention(named_argument), Alias("string"), Arg(type, "string"), Returns(type), other_named_arguments)}
\]

The Alias defines an alternate name that you can use in JSL. For example, if you declared Alias("MsgBox") for a function that is named "Message Box" in the DLL, then you would call it as follows:

\[
\text{result} = \text{dll_obj} \ll \text{MsgBox(...}
\]

The named arguments for Convention are as follows:

- STDCALL or PASCAL
- CDECL

The type argument for both Arg and Returns can be one of the following:

**Table 10.2** Types for Arg and Returns

<table>
<thead>
<tr>
<th>Int8</th>
<th>UInt8</th>
<th>Int16</th>
<th>UInt16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>UInt32</td>
<td>Int64</td>
<td>UInt64</td>
</tr>
<tr>
<td>Float</td>
<td>Double</td>
<td>AnsiString</td>
<td>UnicodeString</td>
</tr>
<tr>
<td>Struct</td>
<td>IntPtr</td>
<td>UIntPtr</td>
<td>ObjPtr</td>
</tr>
</tbody>
</table>

See “Dynamic Link Libraries (DLLs),” p. 675 in the “Messages” appendix for the specifications for all the arguments for the Declare Function message.

Finally, use the UnLoadDLL message to unload the DLL:

\[
\text{dll_obj} \ll \text{UnLoadDLL}
\]

Example

This example uses If() to ensure that it fails silently on Macintosh.

\[
\text{If( Host is( Windows ),}
    \text{dill_obj} = \text{Load DLL(}
        \text{If(}
            \text{File Exists(}
                \text{"C:/Winnt/System32/User32.DLL"
            )}
        )
    )
\]


Dynamic Link Libraries (DLLs)

```
), "C:/Winnt/System32/User32.DLL",
File Exists(
    "C:/Windows/System32/User32.DLL"
), "C:/Windows/System32/User32.DLL",
throw
);

dll_obj << DeclareFunction(
    "MessageBoxW",
    Convention( STDCALL ),
    Alias( "MsgBox" ),
    Arg( IntPtr, "hWnd" ),
    Arg( UnicodeString, "message" ),
    Arg( UnicodeString, "caption" ),
    Arg( UInt32, "uType" ),
    Returns( Int32 )
);
result = dll_obj << MsgBox(
    0,
    "Here is a message from JMP.",
    "Call DLL",
    321
);
Show( result );

dll_obj << UnLoadDLL
```

Other DLL Messages

The Show Functions message sends any functions that have been declared using Declare Function to the log:

```
dll_obj << Show Functions;
```

If you are writing your own DLL, you can create functions in it using JSL. The Get Declaration JSL message sends any JSL functions in your DLL to the log:

```
dll_obj << Get Declaration JSL;
```

For Simple DLL Functions

If your DLL contains functions that are very simple, the Call DLL message declares the specified function with the specified signature, as shown in Table 10.3.

```
dll_obj << Call DLL(function_name, signature, arguments)
```

**Table 10.3** Signature Values

<table>
<thead>
<tr>
<th>Signature</th>
<th>Parameter is of Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;v&quot;</td>
<td>void</td>
</tr>
<tr>
<td>&quot;c&quot;</td>
<td>string</td>
</tr>
</tbody>
</table>
Then, the function is called, passing in the argument.

Example

```javascript
// Load the DLL, given the specified path
dll_obj = Load DLL( "D:\Release\MyDLL.DLL" );
// Call the function inside the DLL named MyExportedFcn.
// This function takes one numeric as input and in this case,
// the value of the input parameter is to be 654.
dll_obj << CallDLL( "MyExportedFcn", "n", 654 );
dll_obj << UnLoadDLL();
```

Effectively, what these two lines accomplish is to call `MyExportedFcn`, passing 654 as input to the function. Conceptually, it is as if the JSL script executed `MyExportedFcn(654);`

### Using Sockets in JSL

Another tool that can be useful in establishing a live Datafeed is JSL Sockets. You can create two types of sockets using JSL:

- **Stream**  
  Stream sockets create a reliable connection between JMP and another computer. The other computer might be running JMP, or it might be a vending machine, data collector, printer, or other device that is capable of socket communication. Some devices implement their interface as an HTTP web server.

- **Datagram**  
  Datagram sockets create a less reliable connection between JMP and another computer. Datagrams are connectionless and information might arrive multiple times, not at all, and out of order. A datagram connection does not include all the overhead of a stream connection that does guarantee reliability. Since datagrams are connectionless, the destination address must be supplied each time (and for the same socket that can be different each time).

Once a socket is created, it can do two things: wait for a connection from another socket, or make a connection to another socket. Here is a simple example program that makes a connection to another computer's web server to get some data:

```javascript
tCall = Socket();
tCall << connect( "www.jmp.com", "80" );
tCall << Send( Char To Blob( "GET / HTTP/1.0~0d~0a~0d~0a", "ASCII~HEX" ) );
tMessage = tCall << Recv( 1000 );
text = Blob To Char( tMessage[3] );
Show( text );
tCall << Close();
```

### Table 10.3 Signature Values (Continued)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Parameter is of Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>“n”</td>
<td>int</td>
</tr>
</tbody>
</table>
The first line creates a socket and gives it a reference name (tCall). By default, a stream socket is created. You can designate the type of socket to create with an optional argument: socket(STREAM) or socket(DGRAM).

The second line connects the tCall socket to port 80 (which is generally the HTTP port) of the JMP Web site.

The third line sends a GET request to the JMP web server; this message tells the JMP web server to send the JMP home page back. The / that follows the word GET should be the path to the page to be opened. A / opens the root page.

The fourth line receives up to 1000 bytes from the JMP web server and stores a list of information in tMessage. Each socket call returns a list. The first element of the list is the name of the call, and the second is a text message, which might be ok or a longer diagnostic message. Additional elements, if present, are specific to the call. In this case, the third element in the list is the data received.

The fifth line converts the binary information received into a character string. tMessage[3] is the third item in the list returned by Recv; it is the data from the JMP web server.

The sixth line displays the data in the log.

The last line closes the socket. The web server has already closed the far end, so this socket either needs reconnecting or proper disposal (close).

The Sample Scripts folder contains several examples of scripts using sockets.

**Socket-Related Commands**

Before creating and using a socket, you might need to retrieve information about the end that you want to connect to. GetAddrInfo( ) and GetNameInfo( ) each takes an address argument and an optional port argument and returns a list of information. For example:

```javascript
Print( Get Addr Info( "www.sas.com" ) );
Print( Get Addr Info( "www.sas.com", "80" ) );
Print( Get Name Info( "149.173.5.120" ) );
{"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120", "0"}}
{"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120", "80"}}
{"Get Name Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "www.sas.com", "0"}}
```

Sometimes there can be more than one answer. In that case, the sublist might be repeated one or more times. These functions can be quite slow; you probably should not try to build a data table of every Web site name with it. For IPV6 compatibility, you should generally use names like “www.sas.com” rather than the numerical form of an address.

**Messages for Sockets**

Once you have created a socket with socket( ), there are many messages that you can send to it.
connect  Connects to a listening socket. Returns a list: {"connect", "ok"} if the connection was successful; or an error if not (for example, {"connect", "CONNREFUSED: The attempt to connect was forcefully rejected."}).

close  Closes the connection when you are finished with it. Returns a list (for example, {"Close", "ok"}).

send  Sends a STREAM message to the other end of the socket.

sendto  Sends a DGRAM message to the other end of the socket.

recv  Receives a STREAM message. The data comes back in a list, along with some other information.Recv takes a required numeric argument that specifies the number of bytes to accept.

recvfrom  Receives a DGRAM message.

ioctl  Controls the socket's blocking behavior. By default, JMP sockets block if no data is available; the socket does not return control to the JSL program until data is available. This makes scripts easy to write, but not particularly robust if the remote end of the connection fails to supply the data. A socket that is set for non-blocking behavior always returns immediately, either with an "ok" return code and some data, or with a "WOULDBLOCK: ..." return code, which means if it were a blocking socket, it would have to wait (block progress of the next JSL statement) until data became available.

Important: Background operations that use a JSL callback avoid this issue; a socket used in a background recv, recvfrom, or accept is set to non-blocking and is polled during wait statements and when JMP is otherwise idle.

Ioctl returns a list. For example, {"ioctl", "ok"}, or {"ioctl", "NOTCONN: The socket is not connected."} if the socket has not been bound (see bind, below) or connected already.

bind  Tells the server socket what address the client socket listens on. Bind associates a port on the local machine with the socket. This is required before a socket can Listen. (See below). Bind is not usually used on sockets that connect; the operating system picks an unused port for you. Bind is needed for a server because anyone that wants to connect to the server needs to know what port is being used. A common port is 80, the HTTP port. Bind returns a list. For example, {"bind", "ok"}, or {"bind", "ADDRNOTAVAIL: The specified address is not available from the local machine."} if you try binding to a name that is not on your machine. Another socket can connect to this socket if it knows your machine name and the number.

listen  Tells the server socket to listen for connections. A listening socket is listening for connections from other sockets. You only need to put the socket into listen mode once. Accept (see below) is used to accept a connection from another socket. Listen returns a list. For example, {"listen", "ok"}, or {"listen", "INVAL: The socket is (or is not, depending on context) already bound to an address. or, Listen was not invoked prior to accept. or, Invalid host address. or, The socket has not been bound with Bind."} if your bind call did not succeed.

accept  Tells the server socket to accept a connection and return a new connected socket. Accept returns a list stating what happened, and if successful, a new socket that is connected to the socket at the other end. For example, the returned list might be {"Accept", "ok", "localhost", socket( )}. In this case localhost is the name of the machine that just connected, and the fourth parameter is a socket that you can use to send or recv a message.

getpeername  Asks about the other end of a connection. GetPeerName returns a list with information about the other end's socket: {"getpeername", "ok", "127.0.0.1", "4087"}.
If this is a server socket, you can discover the address and port of the client that connected. If this is a client socket, you re-discover the name and port of the server that you used in the connect request.

### Database Access (Windows Only)

JMP supports ODBC access to SQL databases through JSL with the `Open Database` command.

```jsl
dt = Open Database(
    "Connect Dialog" | "DSN=...", // data source
    "sqlStatement" | "dataTableName" | "SQLFILE=...", // SQL statement
    "outputTableName" // new table name
);
```

The first argument is a quoted connection string to specify the data source. It should be either of the following:

- "Connect Dialog" to display the ODBC connection dialog box
- "DSN=" and then the data source name and any other information needed to connect to the data source.

For example:

"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/9/Support Files English/Sample Import Data;"

The second argument is a double-quoted string that can be one of the following:

1. An SQL statement to execute. For example, the second argument might be a `SELECT` statement in a quoted string like the following:
   
   "SELECT AGE, SEX, WEIGHT FROM BIGCLASS"

2. The name of a data table. In this case, the effect is an SQL "SELECT * FROM" statement for the data table indicated. For example, `Open Database` would in effect execute the statement "SELECT * FROM BIGCLASS" if you specify this for the second argument:
   
   "BIGCLASS"

3. "SQLFILE=..." and a path to a text file containing an SQL statement to be executed. For example, with the following argument, JMP would attempt to open the file mySQLFile.txt from the C:\ directory and then execute the SQL statement in the file.
   
   "SQLFILE=C:\mySQLFile.txt"

The third argument, `outputTableName`, is optional and specifies the name of the output table to be created, if any. Note that `Open Database` does not always return a data table. The return value might be null. Whether it returns a data table depends on the type of SQL statement executed. For example, a `SELECT` statement would return a data table, but a `DROP TABLE` statement would not.
To save a table back to a database through JSL, send the data table reference a `Save Database( )` message:

```julia
dt << Save Database("connectInfo", "TableName");
```

The first argument works the same way as it does in `Open Database`. Note that some databases do not allow you to save a table over one that already exists. If you want to replace a table in a database, use a DROP TABLE SQL statement in an Open Database command:

```julia
Open Database("connectinfo", "DROP TABLE TableName");
```

The following script opens a database with an SQL query, saves it back to the database under a new name, and then deletes the new table.

```julia
dt = Open Database("Connect Dialog", "SELECT age, sex, weight FROM !"Bigclass!"", "My Big Class");
dt << Save Database("Connect Dialog", "MY_BIG_CLASS");
Open Database("Connect Dialog", "DROP TABLE BIGCLASS.MY_BIG_CLASS");
```

**Note:** When you import data from an ODBC database, a table variable is added that can contain user ID and password information. To prevent this from happening, set the following JSL-only preference:

```julia
pref(ODBC Hide Connection String(1));
```

---

**Working with SAS**

JMP has several ways of interacting with the SAS System.

**Make a SAS DATA Step**

Sending `<<Make SAS Data Step` to a data table returns the text for a SAS DATA Step that can recreate the data table in SAS. For example,

```julia
Current Data Table( ) << Make SAS Data Step
```

prints a DATA Step to the log that can be used in the SAS Program Editor.

Sending `<<Make SAS Data Step Window` produces this code in a window with a .SAS suffix, so that it can be easily sent to SAS.

**SAS Variable Names**

`SAS Open For Var Names( )` opens a SAS data set only to obtain the names of its variables, returning those names as a list of strings.

The rules for SAS variable names are more strict than those of JMP. The `SAS Name` function converts JMP variable names to SAS variable names, changing special characters and blanks to underscores, and various other transformations to produce a valid SAS name.

```julia
result = SAS Name(name);
result = SAS Name({list of names});
```
If the argument is a list of names, the result is a blank-separated character string of names. For example,

```javascript
SAS Name({"x 1", "x 2"})
```

produces

```
  " x_1 x_2"
```

### Connect to a SAS Metadata Server

You can connect to a SAS server and work directly with SAS data sets. Making connections and interacting with SAS data sets is scriptable through JSL. For more information, see the *Using JMP*.

#### Make the Connection

First, use the `Metaconnect` command to connect to a SAS metadata server:

```javascript
connected = Meta Connect("MyMetadataServer", port)
```

If you supply only the machine name (for example, `myserver.mycompany.com`) and the port, you are prompted to provide the authentication domain, your user name, and your password. You can also specify all that in JSL:

```javascript
connected = Meta Connect("MyMetadataServer", port, "authdomain", "user name", "password")
```

When you are finished using the SAS metadata server, use `Meta Disconnect()` to disconnect the connection. No arguments are necessary; the command closes the current metadata server connection.

You can see the repositories that are available on a metadata server and set the one that you want to use:

```javascript
Meta Get Repositories();
{“Foundation”}
Meta Set Repository("Foundation”);
```

Note that if there is only one repository available, it is selected automatically and you do not need to explicitly set it.

Once your repository is set, you can view the servers that are available:

```javascript
mylist = Meta Get Servers();
{“SASMain”, “Schroedl”, “SASMain Ja”, “SASMain zh”, “SASMain ko”, “SASMain_fr”, “SASMain de”, “SASMain Unicode”}
```

Next, set your SAS connection. You can also use this command to connect directly to a local or remote server instead of using a metadata server:

```javascript
conn = SAS Connect("SASMain”);
```

To see the libraries on the server, use the `SAS Get Lib Ref` command:

```javascript
librefs = SAS Get Lib Refs();
```
If the library containing the data that you want is not assigned, assign it:

librefs = SAS Assign Lib Refs("My Library", "c:\public\data");

**Open SAS Data Sets**

First, assign a SAS library reference:

SAS Assign Lib Refs("MyLib", "c:\public");

The first argument is any name that you want to use to refer to the library reference. The second is the path on the server where the data sets are located.

Next, get the list of data sets in the selected library:

datasets=SAS Get Data Sets("MyLib");

{"ANDORRA", "ANDORRA2", "ANYVARNAME", "BOOKS", "BOOKSCOPYNOT", "BOOKS_VIEW",
"CATEGORIES", "DATETIMETESTS", "MOREUGLY", "NOTTOOUGLY", "PAYPERVIEW",
"PUBLISHERS", "PURCHASES", "PURCHASES_FULL"},

Now you can open a data set:

dt=SAS Import Data("BOOKS", "PURCHASES");

or

dt=SAS Import Data(librefs[1], datasets[12]);

or

dt=SAS Import Data("BOOKS.PURCHASES");

Once you have a reference to a library, you can get information about any SAS data sets in that library. For example, you can get a list of variables:

bookvars=SAS Get Var Names("BOOKS.PURCHASES");

{"purchaseyear", "purchasemonth", "purchaseday", "bookid", "catid",
"pubid", "price", "cost"}

With that information, you can choose to import only part of the data set by specifying the variables to import.

dt=SAS Import Data(librefs[1], datasets[12], columns(bookvars[1], bookvars[2],
bookvars[4]));

**Save SAS Data Sets**

To save a JMP data table or an imported SAS Data Set, use the SAS Export Data( ) command:

SAS Export Data(dt, librefs[1], datasets[4], ReplaceExisting);

**Run a Stored Process**

To get a reference to a stored process:

stp=Meta Get Stored Process("Samples/Stored Processes/Sample: Hello World");
There is no way to acquire a list of stored processes through JSL; you must know the path to the stored process that you want to run.

To run it, send the stored process a message:

\[ \text{stp}<<\text{run}(\text{);} \]

**Submit SAS Code from JMP**

You can also directly submit SAS code and get back SAS results. For example:

\[ \text{SAS Submit("proc print data=sashelp.class; run;");} \]

Two optional arguments control whether you see the output and the SAS log in JMP:

\[ \text{SAS Submit("SAS Code" } <,\text{No Output Window(True|False)} <,\text{Get SAS Log(True|False)>);} \]

You can also see the SAS Log at any time using the command

\[ \text{SAS Get Log( );} \]

\[ \text{SAS Get Log( )} \]

returns the contents, which can be placed in a JSL variable and used like any JSL string.

**Preferences**

SAS Integration preferences are not scriptable.

**Sample Scripts**

In your Sample Scripts folder is a folder named SAS Integration that contains examples. To run the stored process scripts successfully, the stored processes need to be placed on your SAS Metadata Server. The stored processes can be found in the `sampleStoredProcesses.spk` file, also in this folder.

To import `sampleStoredProcesses.spk` into your SAS Metadata Server:

**Caution:** We recommend that you import these stored processes into a SAS Metadata Server that is used for testing rather than into a production system.

1. Run SAS Management Console.
2. Connect to your SAS Metadata Server using an account with administrative privileges.
3. Expand the **BI Manager** node in the left pane of the SAS Management Console.
4. Navigate to the folder in the tree under which you would like to create the imported sample stored processes.
5. Right-click on that folder in either the left pane or the right pane of the SAS Management Console and select **Import**.

   The Import Wizard appears.
6. Enter the full path to `sampleStoredProcesses.spk` or use the **Browse** button to navigate to it.
7. Select **All Objects** in the Import Options section of the wizard.
8. Click **Next**.

   The next panel reports that during the import process, you must specify values for **Application servers** and **Source code repositories**.
9. Click **Next**.

   Select which of the application servers defined in your SAS Metadata Server that you would like to use to execute the imported stored processes.
10. Select an application server from the drop-down list under **Target**.
11. Click **Next**.

   Select the source code repository (directory) defined on your SAS Metadata Server where you would like the SAS code for the imported stored processes to be placed.
12. Select a source code repository from the drop-down list under **Target Path**.
13. Click **Next**.

   The next panel gives a summary of what occurs if you click **Import**.
14. Review the information about the panel, and if it looks correct, click **Import**.
15. During the Import process, you might be asked to provide login credentials for connecting to the metadata server to perform the import. Provide credentials with administrative privileges and click **OK**.

After the import completes, you will find a folder named **BIP Tree** under the folder that you imported the stored processes into. Under **BIP Tree** is a folder named **JMP Samples**, and in the **JMP Samples** folder are two sample stored processes: **Shoe Chart** and **Diameter**.

Please note that the paths to the sample stored processes needs to be adjusted in the sample scripts **storedProcessHTML.jsl** and **storedProcessJSL.jsl** to match the folder into which you imported the sample stored processes. Otherwise, these scripts will not work correctly.

---

**Working with R**

You can interact with R using JSL:

- Submit statements to R from within a JSL script.
- Exchange data between JMP and R.
- Display graphics produced by R.

Textual output and error messages from R appear in the log window.

**Installing R**

R must be installed on the same computer as JMP. JMP is not distributed with a copy of R. You can download R from the Comprehensive R Archive Network Web site:

http://cran.r-project.org
Because JMP is supported as both a 32-bit and a 64-bit Windows application, you must install the corresponding 32-bit or 64-bit version of R. For the supported version of R, see the system requirements on the JMP Web site: http://www.jmp.com/support/system_requirements_jmp.shtml

How JMP Finds R

JMP delays loading R until a JSL-based script requires access to it. When JMP needs to load R, it follows the standard steps for finding R on a Windows computer:

1. Look up the environment variable R_HOME.
   
   If the variable exists, load R from the specified directory.

2. If the environment variable R_HOME does not exist, look up the InstallPath value in the Windows registry under the following key:

   HKEY_LOCAL_MACHINE\SOFTWARE\R-core\R

   For 32-bit JMP running on a 64-bit machine, the InstallPath value is under the following key:

   HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\R-core\R

   If the InstallPath value exists, load R from the specified directory.

3. If the InstallPath value does not exist, an error message states that R could not be found.

Testing Your Setup

To test that your computer is able to run JSL-based scripts that use R, run the following JSL script:

```julia
R Init();
R Submit("x <- 1:5
 x
 ");
R Term();
```

You should see the following output in the log:

```
[1] 1 2 3 4 5
```

JMP to R Interfaces

The following JMP interfaces are provided to access R. The basic execution model is to first initialize the R connection, perform the required R operations, and then terminate the R connection. In most cases, these functions return 0 if the R operation was successful, or an error code if it was not. If the R operation is not successful, a message is written to the log. The single exception to this is R Get(), which returns a value.

R JSL Scriptable Object Interfaces

The R interfaces are also scriptable using an R connection object. A scriptable R connection object can be obtained using the R Connect() JSL function.
Conversion Between JMP Data Types and R Data Types

Table 10.4 shows what JMP data types can be exchanged with R using the R Send( ) function. Sending lists to R recursively examines each element of the list and sends each base JMP data type. Nested lists are supported.

Table 10.4 Equivalencies Between JMP and R Data Types for R Send()

<table>
<thead>
<tr>
<th>JMP Data Type</th>
<th>R Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Double</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Matrix</td>
<td>Double Matrix</td>
</tr>
<tr>
<td>List</td>
<td>List</td>
</tr>
<tr>
<td>Data Table</td>
<td>Data Frame</td>
</tr>
<tr>
<td>Row State</td>
<td>Integer</td>
</tr>
<tr>
<td>Datetime</td>
<td>Date and Time</td>
</tr>
<tr>
<td>Duration</td>
<td>Time/Duration</td>
</tr>
</tbody>
</table>

Example

```r
R Init( );
X = 1;
R Send( X );
S = "Report Title";
R Send( S );
M = [1 2 3, 4 5 6, 7 8 9];
R Send( M );
R Submit( "
X
S
M"
);
R Term( );
```

Table 10.5 shows what JMP data types can be exchanged with R using the R Get( ) function. Getting lists from R recursively examines each element of the list and sends each base R data type. Nested lists are supported.

Table 10.5 Equivalencies Between JMP and R Data Types for R Get()

<table>
<thead>
<tr>
<th>R Data Type</th>
<th>JMP Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>Numeric</td>
</tr>
</tbody>
</table>
Table 10.5 Equivalencies Between JMP and R Data Types for R Get()

<table>
<thead>
<tr>
<th>R Data Type</th>
<th>JMP Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical (Boolean)</td>
<td>Numeric (0</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Integer</td>
<td>Numeric</td>
</tr>
<tr>
<td>Date and Time</td>
<td>Datetime</td>
</tr>
<tr>
<td>Time/Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>Factor</td>
<td>Expanded to a list of Strings or a Numeric matrix</td>
</tr>
<tr>
<td>Data Frame</td>
<td>Data Table</td>
</tr>
<tr>
<td>List</td>
<td>List of converted R data types</td>
</tr>
<tr>
<td>Matrix</td>
<td>Numeric Matrix</td>
</tr>
<tr>
<td>Numeric Vector</td>
<td>Numeric Matrix</td>
</tr>
<tr>
<td>String Vector</td>
<td>List of Strings</td>
</tr>
<tr>
<td>Graph</td>
<td>Picture</td>
</tr>
<tr>
<td>Time Series</td>
<td>Matrix</td>
</tr>
</tbody>
</table>

**JMP Scoping Operators and R**

A JMP object sent to R using `R Send()` uses the same JMP reference as the name of the R object that gets created. For example, sending the JMP variable `dt` to R creates an R object named `dt`. The colon and double colon scoping operators (:` and `::`) are not valid in R object names, so these are converted as follows:

- A single colon scoping operator is replaced with a period (`.`).
  
  For example, sending `nsref:dt` to R creates a corresponding R object named `nsref.dt`.
- A double colon scoping operator (designating a global variable) is ignored.
  
  For example, sending `::dt` to R creates a corresponding R object named `dt`.

**Using R Name() with R Send()**

The R `Name()` option to `R Send()` has an argument that is a quoted string that contains a valid R object name. The JMP object sent to R becomes an R object with the name specified. For example:

```r
R Send( jmp_var_name, R Name( "r_var_name") );
R Submit( "print(r_var_name") )
```
Example

This example creates a variable x in the Here namespace, a variable y in the global namespace, and a variable z that is not explicitly referenced to any namespace. The variable z defaults to Global unless Names Default To Here(1) is on. These variables are then passed to R.

```r
Here:x = 1;
::y = 2;
z = 3;

//Initiate the R connection
R Init();

//Send the Here variable to R
//Here:x creates the R object Here.x
R Send( Here:x );
R Submit( "print(Here.x)" );
/* Note that the JMP log labels the output with the original JMP variable reference Here:x. */

//::y creates the R object y
R Send( ::y );
R Submit( "print(y)" );

//To use a different name for the R object, use the R Name() option
R Send( Here:x, R Name( "localx" ) );
R Submit( "print(localx)" );
/*The R Name option to the R Send() command creates the R object named "localx" corresponding to the JMP variable "Here:x". Again the log shows the original corresponding JMP variable name.*/

//z creates the R object z
R Send( z );
R Submit( "print(z)" );
```

Troubleshooting

Recording Output

On Windows, if you want to record output to the graphics window, send the following R code using R Submit():

```r
windows.options( record = TRUE);
```

Character Vectors

A JMP list of strings is not the same as an R character vector. If you send a list of strings to R, it becomes a list of strings in R, not a character vector. You can use the R function Unlist to convert it:

```r
R Init();
```
X = {"Character", "JMP", "List"};
R Send( X );
R Submit( "class(X)" );
/* R output is:
[1] "list"
*/
R Submit( "Y<-unlist(X)
    class(Y)"");
/* Object Y is now a character vector. The R output is:
[1] "character"
*/
R Term();

**Element Names**

A feature of an R list (called attributes) lets you associate a name with each element of the list. You can use the name to access that element instead of having to know the position of it in the list.

In the following example, the list that is created in R has two elements named x and y that are created using the `list()` function of R. When you bring the R list into JMP and then send it back to R, the names are lost. Therefore in R, you cannot access the first matrix using `pts$x`. Instead, you must use the index using `pts[[1]]`.

R Init();
R Submit("pts <- list(x=cars[,1], y=cars[,2])
    summary(pts)");
JMP_pts = R Get(pts);
R Send(JMP_pts);
R Submit("Summary(JMP_pts)");
R Term();

**Examples**

**Sending a Data Table to R**

This example initiates an R connection, sends a data table to R, prints it to the log, and closes the R connection.

R Init();
dt = Open("$SAMPLE_DATA/Big Class.jmp",invisible);
R Send(dt); // Sends the opened data table represented by dt to R;
R Submit("print(dt)");
Creating Objects in R

This example initiates an R connection, creates an R object, retrieves the object into JMP, and closes the R connection.

```r
R Init();
R Submit("L3 <- LETTERS[1:3]
d <- data.frame(cbind(x=1, y=1:15), Group=sample(L3, 15, repl=TRUE))
");
R Get( d ) << NewDataView;
R Term();
```

Using R Functions and Graphics

This example initiates an R connection and plots the normal density function in R using the R graphics device. Then the graph is retrieved from R and displayed in JMP. Finally, the R connection is closed.

```r
R Init();
R Submit("\[plot(function(x) dnorm(x), -5, 5, main = "Normal(0,1) Density")\]");
picture = R Get Graphics( "png" );
New Window( "Picture", picture );
Wait( 10 );
R Term();
```

Simple Matrix Addition in R

This example initiates an R connection, sends a matrix to R, creates a matrix in R, adds them together, returns the new matrix to JMP, and closes the R connection.

```r
R Init();
X = J( 2, 2, 1 );
R Send( X );
R Submit("X                                     #Prints X to the log
Y <- matrix(1:4, nrow=2, byrow=TRUE)  #Makes a 2x2 matrix object Y
Y                                     #Prints Y to the log
Z <- X + Y                            #Matrix object Z is addition of X and Y
");
Z = R Get( Z );
R Term();
Show(Z);
```
A Bootstrap Sample

See the file JMPtoR_bootstrap.jsl in the sample scripts folder for an example script. This script performs a bootstrap simulation by using the JMP to R Project integration.

The script produces a JMP window that asked the user to specify the variable to perform bootstrapping over. Then the user selects a statistic to compute for each bootstrap sample. Finally, the data is sent to R using the R interface in JSL.

The boot package in R is used to call the `boot()` function and the `boot.ci()` function to calculate the sample statistic for each bootstrap sample and the bootstrap confidence interval.

The results are brought back to JMP and displayed using the JMP Distribution platform.

Working with Excel

You can script the Profiler interface to Excel, although not the Transfer to JMP interface. The basic syntax is as follows:

```javascript
excel_obj = Excel Profiler(
    Workbook( "excel_workbook_path" ),
    Model( "name_of_model" )
);
```

The `Model` argument is optional. If you supply only the workbook, you are prompted to select the model. You do not need to specify the worksheet, because the model is found no matter where it is located.

Once you have an Excel Profiler object, you can send the following messages to run the JMP prediction profiler using the Excel model. For example:

```javascript
excel_obj <<Prediction Profiler( 1 );
```

JMP Add-Ins

You can write complex scripts that create new platforms and interact with users and JMP. By using the JMP Add-In architecture, you can also easily distribute your complex scripts so that any JMP user can access them without having to open and run scripts.

A JMP add-in is a collection of files in a folder that has been registered with JMP. Registering an add-in saves the information about your add-in so that JMP knows where it is, what it is, where to place it on the menu, and so on.

JMP add-ins have two essential requirements:

- A unique identifier. The unique ID of a JMP Add-In is a string up to 64 characters in length and must meet the following requirements:
  - begin with a letter
  - consist only of letters, numbers, periods, and underscores
JMP Add-In unique IDs are case-insensitive. The convention of using reverse-DNS names (for example, `com.mycompany.myaddin`) as the ID is strongly recommended to ensure uniqueness, but not required.

- A filepath to an existing folder. This folder is referred to as the add-in's home folder.

In addition to these required features, a JMP Add-In can include the following:

- A user-friendly name that can be displayed in the JMP user interface wherever add-in names are displayed, instead of the more restrictive unique ID.
- A text file named `addin.def` that contains name-value pairs specifying the add-in's unique ID, friendly name, and other information. This file is used for automatic registration of a JMP add-in.
- A specifically formatted XML file named `addin.jmpcust` at the root level of its home folder that contains menu and toolbar customizations for the add-in.
- A JSL script named `addinLoad.jsl` at the root level of the home folder. This script is run when JMP loads the add-in (normally whenever JMP is started).
- A JSL script named `addinUnload.jsl` at the root level of the home folder. This script is run when JMP unloads the add-in (normally whenever JMP is shut down).

**Note:** When you write an add-in script, the `Names Default To Here()` option is enabled for you. Doing so ensures that all unqualified (not specifically referenced to a namespace) JSL variables are in the `Here` namespace, and local only to the script. See “Advanced Scoping and Namespaces,” p. 390 in the “Advanced Concepts” chapter for more information about `Names Default To Here()`.

### Workflow for Creating an Add-In

The basic steps of creating an add-in are as follows:

1. “Set Up Your Add-In,” p. 367
2. “Create a Menu Entry and a Toolbar Button for the Add-In,” p. 368
3. “Create the Add-In File,” p. 369
4. “Test Your Add-In,” p. 369
5. “Deploy Your Add-In,” p. 369

### Set Up Your Add-In

1. Create a folder and put your JSL file(s) for your add-in into it, along with any support files that are required (for example, data tables or icons).
2. In your add-in's home folder, create a file named `addin.def` that contains these three lines:
   ```
   id=com.yourcompany.youraddin
   name=friendly name
   autoload=1
   ```
The \textit{id} is the unique ID. We recommend using reverse DNS (for example, \texttt{com.yourcompany.youraddin}).

The \textit{name} is a display name. It does not need to be unique, and it can provide a friendly description of what your add-in does.

Setting \textit{autoload} to 1 (true) ensures that your add-in is loaded automatically during JMP start-up when it is deployed.

Only the \textit{id} is required.

For more information about the \texttt{addin.def} file, see “Addin.def File Specification,” p. 371.

3. Register your add-in with JMP by opening your \texttt{addin.def} file in JMP.

   Or, you can register it using JSL:

   \begin{verbatim}
   registerAddin(
      // the unique ID for your add-in:
      "com.yourcompany.youraddin",
      // the folder that contains the add-in files:
      "full-path-to-folder-with-JSL-files"
   );
   \end{verbatim}

   You want to register your add-in immediately so that you can test it easily during development. Once the add-in is registered with JMP, you can reference its folder using the \texttt{$ADDIN\_HOME(uniqueID)$} path variable. For example, you could use the \texttt{Include} function like this:

   \begin{verbatim}
   include("$ADDIN\_HOME(com.yourcompany.youraddin)\addin\_script.jsl");
   \end{verbatim}

4. Write any scripts that you need for your add-in and save them in this folder. You can include images, DLLs, icons, data tables, or anything else your add-in requires.

\section*{Create a Menu Entry and a Toolbar Button for the Add-In}

1. In JMP (on Windows only), select \textbf{View > Customize > Menus and Toolbars}.
2. Click \textbf{Change}.
3. Select the \textbf{JMP Add-In} option, and then select your add-in (for example, \texttt{com.yourcompany.youraddin}) from the list.
4. Click \textbf{OK}.
5. Create a menu item in the \textbf{Add-Ins} menu. See the “Personalizing JMP” chapter in the \textit{Using JMP} book for details about creating menu items.
6. For the action, select \textbf{Run JSL in this file}.
7. Select the \textbf{Use add-in Home Folder} check box, and then select your add-in from that drop-down.
8. Click \textbf{Browse}.
9. Select the JSL file that drives your add-in.
10. Click \textbf{Open}.

   The path to your add-in script appears in the field.

11. (Optional) Select an icon for your menu or toolbar item. The icon must be in the folder that contains everything for your add-in.
12. Click OK to close the Menu Editor.

Your add-in's menu and toolbar items now appear in the menu, and the addin.jmpcust file is now in your add-in's home folder. You might see backup menus as well, that you can delete. The only menu file that you need is addin.jmpcust.

See the Using JMP book for details about using the Menu Editor.

Create the Add-In File

1. In your add-in's home folder, select all of the files in that folder (including the addin.def and addin.jmpcust files), and create a ZIP archive of all of the files.

**Note:** Do not select the folder to create your ZIP archive. Doing so creates an add-in that cannot be recognized by JMP. Instead, be sure to select the files within the folder to create the ZIP archive.

2. Change the extension of the archive from .zip to .jmpaddin so that JMP can recognize it. The name of the archive can be anything.

Test Your Add-In

Test your add-in thoroughly. Unregister your add-in, so that you can test installing and using it.

1. Select View > Add-Ins.
2. Select your add-in and click Unregister.
3. Install your add-in by either selecting File > Open in JMP, or double-clicking your .jmpaddin file.
4. Ensure that the menu and toolbar button run your script correctly and that the script itself runs correctly.

Deploy Your Add-In

When your add-in is finished, distribute the .jmpaddin file and tell people how to find and install it.

More About Installation and Registration

There are several ways to install and register an add-in.

**JSL Registration**

When you are developing an add-in, use the Register Addin() JSL function. This approach is especially useful for quickly getting started in developing an add-in.

See Register Addin(unique_id, home_folder, <named_parameters>) on p. 614 in the "JSL Syntax Reference" and Unregister Addin(unique_id) on p. 621 in the "JSL Syntax Reference".

If a file named addin.def is found in the specified home folder, values from that file are used for any optional parameters that are not included in the Register Addin() function. The addin.def file is used
only for values that are not provided in the `Register Addin()` function. This function is useful while developing, but not necessary, since the `addin.def` file is enough to register an add-in.


**Combined Installation and Registration**

Once an add-in file is ready, you can install it directly. This method is typically how users install your add-ins.

Package all of the files for your add-in into a ZIP archive and give it the file extension `.jmpaddin`. The ZIP archive must include an `addin.def` (or `addin.jmpaddindef`) file at the root level. Distribute this ZIP file to JMP users. When JMP users open this file in JMP, the files are extracted into the appropriate JMP add-ins install folder, and the add-in is registered.

1. Select **File > Open**.
2. In the Open window, navigate to the folder that contains a JMP add-in.
3. Select the `.jmpaddin` file.
4. Click **Open**.

You can also double-click the file to install it.

**Network Registration**

If your add-in is located on a network folder that is accessible to other JMP users in your organization, you can distribute a `.def` file (or `.jmpaddindef`) that includes the path to the home folder on the network. When JMP users open that file in JMP, the add-in is registered using the network location. This approach is helpful for the centralized administration of add-ins.

**Automatic Discovery**

When JMP starts, JMP reads in the `addinRegistry.xml` file, which contains information about previously registered JMP add-ins. Then, JMP looks for any other add-ins that can be installed automatically in the folders shown in Table 10.6.

**Table 10.6 Add-In Folder Locations**

<table>
<thead>
<tr>
<th>On Windows</th>
<th>On Macintosh</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%ALLUSERSPROFILE%\SAS\JMP\AddIns</code> (add-ins for all users)</td>
<td><code>/Library/Application Support/JMP/AddIns</code> (add-ins for all users)</td>
</tr>
<tr>
<td><code>%LOCALAPPDATA%\SAS\JMP\AddIns</code> (add-ins for the current user)</td>
<td><code>~/Library/Application Support/JMP/AddIns</code> (add-ins for the current user)</td>
</tr>
</tbody>
</table>

JMP looks for any first-level subfolders of those `AddIns` folders that contain a file named `addin.def`. For each `addin.def` that JMP finds, JMP performs the following steps:
1. Reads in the name-value pairs in the file.
2. Checks to see whether an add-in with the unique ID specified in the file is already registered. If not, the add-in is added to the list of known add-ins. The modification date of addin.def is saved, along with the other information about the add-in.
3. If JMP already knows about an add-in with the specified unique ID, JMP compares the modification date on addin.def with the date that was previously saved for this add-in.
   If JMP has no saved date (the add-in was explicitly registered last time) or the saved date is earlier than the current modification date, JMP updates its stored information for the Home folder, friendly name, and JMP Version for the add-in, as well as the current modification date of addin.def.

To avoid folder name conflicts, the name of the AddIns subfolder for an add-in should match the unique ID of the add-in.

Home Folder for Discovered Add-Ins

The Home Folder for discovered add-ins does not have to be the AddIns subfolder in which the addin.def file was found. The addin.def file can be the only file in that subfolder and have a Home setting that points to some other location where the add-in files actually reside.

Troubleshooting

If an automatically discovered add-in has the same unique ID as an add-in that was explicitly registered, the automatically discovered add-in is used.

Provide Your Own Installation Program

You can provide your own installer that installs your JMP Add-In (possibly in combination with other files) on the user's computer. For your add-in to be discovered by JMP, you need to create a folder in one of the areas that JMP looks for add-ins, and place an appropriate addin.def file there. See “Automatic Discovery,” p. 370.

Addin.def File Specification

The addin.def file is a simple text file that contains name-value pairs that provide registration information about a JMP Add-In.

The addin.def file can alternatively be named addin.jmpaddindef. The longer extension allows opening the file through the operating system (for example, by double-clicking the file) to open JMP and automatically register the add-in. This ability is useful where add-in files are accessed from a network shared folder within a company, because the add-in can be deployed by distributing only the addin.jmpaddindef file.

Descriptions of Recognized Name-Value Pairs

id Required. The unique ID for your add-in. The string can contain up to 64 characters. The string must begin with a letter and contain only letters, numbers, periods, and underscores. Reverse-DNS names are recommended to increase the likelihood of uniqueness.
name Optional. The name that can be displayed in the JMP user interface wherever add-in names are displayed, instead of the unique ID. This name is displayed if no localized names are provided or when JMP is run under a language for which you did not provide a localized name.

name_xx Optional. Allows the user-friendly name to be localized for different languages, where xx is the two-letter ISO 639-1 code for the language. If you include localized names, you should still include a language-neutral name in case JMP is running under regional settings for which you do not have a localized name.

home Optional. The path to the add-in files. The Home Folder for the add-in is assumed to be the folder where addin.def is located. You need to include a setting for home only if the Home Folder is somewhere else (for example, a network shared folder).

home_win Optional. The path to the add-in files to be used when JMP is running on Windows. Overrides the value specified for home on Windows, if any.

home_mac Optional. The path to the add-in files to be used when JMP is running on the Macintosh. Overrides the value specified for home on Macintosh, if any.

autoLoad Optional, Boolean. The default value is True (1). Determines whether this add-in is initially configured to load automatically during JMP start-up.

JMPVersion Optional. Valid values are integers corresponding to the JMP major version that this add-in is restricted to, or the keyword All. All is the default value.

minJMPVersion Optional. Valid values are integers corresponding to the JMP major version that is the minimum version that the add-in supports.

maxJMPVersion Optional. Valid values are integers corresponding to the JMP major version that is the maximum version that the add-in supports. Use this setting only if there is a known incompatibility between your add-in and a specific version of JMP. You should provide a new version of the add-in for later versions of JMP.

Example addin.def File

id="com.mycompany.myaddin"
name="My Add-In’s Friendly Name"
name_fr="My Add-In's French Name"
name_de="My Add-In's German Name"
home="\server\share\myjmpaddin"
loadsAtStartup=1
MinJMPVersion=9

Example JMP Add-In

In the Sample Scripts folder is a sample add-in named Simple Calculator.jmpaddin. The Sample Scripts folder is typically in these locations:

- On Windows: C:\Program Files\SAS\JMP\9\Support Files <language>\Sample Scripts
- On Macintosh: /Library/Application Support\JMP\9<language>\Sample Scripts

To see what the add-in contains, change the extension to .zip and unzip it into a new folder. To see how it works, change the extension back to .jmpaddin and install.
The add-in contains the following files:

adinn.def

Provides the specification for JMP to register the add-in. It contains only these two lines:

```plaintext
id="com.jmp.sample.calculator"
name="Simple Calculator"
```

adinn.jmpcust

Provides the menu customization file that is created when you interactively create a custom menu. This example places the add-in menu item into the default Add-Ins menu.

calculator.jsl

A JSL script that creates a basic calculator.

calc_icon.gif

The image used as the calculator’s icon.

## OLE Automation

Most of JMP can be driven through OLE automation. Please see the Automation Reference.pdf in JMP/9/Support Files English/Documentation for details about automating JMP. This document introduces how to automate JMP through Visual Basic and using Visual C++ with MFC. It also contains details for the methods and properties that JMP exposes to automation clients like Visual Basic and Visual C++.

The JMP\9\Support Files English\Automation Samples folder contains several example Visual Basic.Net, Visual C#.Net, and Visual C++.Net programs that automate features in JMP.

## Automating JMP through Visual Basic

### Starting a JMP Application

The first step in automating JMP is to start it up. However, it’s important to look at the resources available to help you with the JMP methods and properties. JMP provides a type library that allows automation controllers like Visual Basic (VB) to display a list of the methods and properties that JMP exposes, along with parameters that the methods require. This library is called JMP.TLB.

There are two steps to make the JMP type library available to VB.

1. Select Project > References in VB. A list of applications that are known to VB appears. If JMP is not in that list, select Browse. A file window asks you to locate a .tlb (Type library) file. Find the icon for the JMP type library in the JMP directory. Select this library and click OK.

2. Open the object browser by selecting View > Object Browser in VB. Select JMP from the drop down list box.
Now you can see the JMP automation classes and constants. You can now select a class, and the methods available to that class appear in the right list box for the object browser. If you select a method, a short helper string appears at the bottom of the window. This string lists the parameters for the method. Constants are used when methods require a restricted set of parameters, typically denoting a specific action.

Now that you have access to the type library information, write the necessary code to instantiate JMP. This is done with `CreateObject`. In global declarations for the VB project, create a variable of type `JMP.Application`. This is done as:

```vbnet
Dim MyJMP As JMP.Application
```

Now dimension some other variables. Good examples are `DataTable`, `Distrib`, `Oneway`, and `JMPDoc`. These are specified with `JMP.DataTable`, `JMP.Distribution`, `JMP.Oneway`, and `JMP.Document` respectively.

To create a JMP session, make it visible, and load a data table, add the following code to your VB script.

```vbnet
Dim JMPDoc As JMP.Document
Set MyJMP = CreateObject("JMP.Application")
MyJMP.Visible = True
Set JMPDoc = MyJMP.OpenDocument("C:\Program Files\SAS\JMP\9\Support Files\English\Sample Data\Big Class.jmp")
```

The `Dim` statement indicates the type of variable. This declaration should go in the general declarations section of your VB project, though. If you do not do this, the JMP objects are destroyed when the variable goes out of scope at the end of the procedure.

JMP comes up invisible by default, as required by automation guidelines. Therefore, one of your first moves should be to make it visible, as shown in the above code.

### Launching an Analysis

Now that you have a data table open, you can launch an analysis and manipulate it. Each analysis must first be created. Then, the required parameters for the analysis must be specified. Optional settings can also be specified. Then the analysis is launched. Additional option processing can then be done on the analysis object after the launch.

```vbnet
Dim Oneway As JMP.Oneway
Set Oneway = JMPDoc.CreateOneway
Oneway.LaunchAddY ("Height")
Oneway.LaunchAddX ("Age")
' Set an option before the launch
Oneway.Quantiles (True)
' Create the initial analysis output
Oneway.Launch
Oneway.MeansAnovaT (True)
Oneway.MeansStdDev (True)
Oneway.UnequalVariances (True)
Oneway.NormalQuantilePlot (True)
Oneway.SetAlpha (0.05)
Oneway.Save(JMP.OnewaySaveConstants.oscCentered)
Oneway.Save(JMP.OnewaySaveConstants.oscStandardized)
```
OLE Automation

Oneway.CompareMeans(JMP.OnewayCompareConstants.occAllPairs, True)
Oneway.CompareMeans(JMP.OnewayCompareConstants.occEachPair, True)

The first step is to create the analysis object, which is done by calling the CreateOneway method of the document class. Next, X and Y columns are selected, and then Launch is called to create the actual One-way analysis. Each analysis platform has a distinct creation method, which you can view under the Document object in the object browser. In many cases, it is possible to specify options before the Launch of the object, so the analysis output uses the options that are already set. In this example, most option processing is done after the launch of the analysis, which shows the options popup in the display. As you can see, most methods are a simple setting of options, like you might do from a menu. SetAlpha takes a parameter, since you do not want to open a window for interaction during automation. CompareMeans takes two parameters, one for the type of comparison and one for the toggle to indicate on or off. The Save method takes a predefined constant (viewable in the object browser) that tells the Oneway analysis what to save.

Most analysis methods work this way, although some like Bivariate produce additional objects when methods are called. An example is:

Set Fit = Bivar.FitLine
Fit.ConfidenceFit (True)
Fit.ConfidenceIndividual (True)

Here, the FitLine method produces an object of type Fit. This object has methods and properties of its own, which can be manipulated. Remember, the new object created by FitLine can be manipulated only while its variable is in scope.

If a method produces an object that can also be automated, the object browser indicates this. For FitLine, the object browser specifies that the return type is As Fit.

Since this is not a predefined type like short or BSTR, you can probably guess that this is an object. If you look farther down the object browser, you see Fit as an object type. This confirms that an object is produced, and also gives you the methods that Fit supports.

Creating and Populating a Data Table

New data tables can be created with the (appropriately named) NewDataTable method of the Application object. A filename is assigned at creation time. This method returns a column object, which must be retained as long as you want to add rows. By default, 20 rows are created. The SetCellVal method can be used to populate individual cells, and AddRows can be used to add rows as needed. Here is an example:

Dim Col As Object
Set DT = JMP.NewDataTable("C:\test.jmp")
Set Col = DT.NewColumn("Col1", dtTypeNumeric, 0, 8)
DT.Visible = True

'You must add rows before populating the table with data
DT.AddRows 20,0

'Set Cell values to increments of 1.5
For i = 1 To 10
Extending JMP
Chapter 10
OLE Automation

Col.SetCellVal i, i * 1.5
Next i
DT.Visible = False
For i = 11 To 20
  Col.SetCellVal i, i * 1.5
Next i
DT.Visible = True

' This adds 5 rows to the end of the table
DT.AddRows 5, 0
' This adds 5 rows after row 2
DT.AddRows 5, 2

' Now save the data table using the previously specified filename
DT.Document.Save

' If you wanted to create a subset of the table, with only rows 1-3
' you could do the following
' Note: you could also create subsets using specific columns by adding the
' columns to a list using the AddToList member function of Datatable
Dim NewDT As JMP.Datatable
Dim DTDoc As JMP.Document
DT.SelectRows 1,3
Set NewDT = DT.Subset

' Now save the new table
Set DTDoc = NewDT.Document
DTDoc.SaveAs("C:\MySubset.jmp")

Example Programs


The ANALYSIS example program shows simple automation cases for almost all of the JMP platforms. The example code tests the features of a platform, but it does not pretend to do meaningful statistical analyses. Its purpose is for teaching automation coding. It is recommended that you make the JMP type library visible to the VB project. The first section of this document describes this process, which lets you see the methods and properties exposed by the automation platforms within JMP.

Likewise, the DATATAB example shows how to exercise the methods available for data table automation. No attempt is made to produce meaningful output.

The TIMPORT program shows the steps necessary to get a text file imported into JMP as a data table. Once this has been done, the data table can be manipulated just like the example in DATATAB, and analyses can be performed on the data just like in the ANALYSIS program.

The ODBC Demo program shows a simple example of importing a dBase file into JMP using ODBC access.
The WordDemo program shows the commands necessary to take a graphic section from a JMP report, copy it to the clipboard, and then insert it into a Microsoft Word document.

The FitModel and DOE examples show operators that are specific to those areas of JMP, and whose platform operator differs slightly from other platforms.

The sample code for all five example programs assumes the data files reside in the default SAMPLE DATA directory. If you move your sample data files, you need to change the path information in the VB samples.

If there are differences between this document’s examples of Visual Basic code and that in the sample programs, preference should be given to the sample program code.

An Example: Automating JMP From Excel 2007

This example automates JMP using a macro within an Excel 2007 worksheet. The macro code is written in Visual Basic. It starts JMP in a visible state when the Excel worksheet is initially opened. The Excel worksheet is then imported into JMP using the ODBC automation interface. Once the worksheet data is in JMP, changes to individual worksheet cells are sent to JMP and changed in the JMP data table.

The first time a row value in Excel changes, JMP generates a Control Chart. Subsequent changes to the excel worksheet result in changes to the Control Charts. This is because Control Chart output is dynamically linked to the JMP data table, which in this example is dynamically updated by Excel. Every fifth time the Excel worksheet changes, a method is called in JMP to generate a .PNG file for the Control Chart. This allows users without JMP to view the output through a web browser. Finally, when the Excel worksheet closes, JMP shuts down through automation.

Begin by opening Microsoft Excel. To create a Visual Basic script for an Excel workbook, select Visual Basic from the Developer ribbon. The Visual Basic editor opens in a separate window. On the left side of the Visual Basic editor, there is a pane entitled VBA Project. This pane shows the sheets that might have Visual Basic code associated with them, as well as the workbook itself.

![Figure 10.5 VBA Project for Excel](image_url)

Code written for the workbook usually works for any of the sheets within the workbook.

There are three sections involved in the coding for this example. First, there are some variables that are global in scope that are declared in the module1.bas file. This allows these variables to be referenced in other code modules. A module can be inserted into the Visual Basic project by context-clicking on the VBA project icon and selecting Insert > Module. Type the following code into the module. The code declares...
instances of a JMP application, a JMP data table, and a flag to keep track of whether a document is open or not.

    Public MyJMP as JMP.Application 'The JMP Application Object
    Public DT As JMP.DataTable 'The JMP Data Table object
    Public DocOpen as Boolean 'A flag indicating “JMP Table Open”

The next segment updates JMP when cells in the Excel worksheet change. It is called automatically because Excel generates the **Worksheet_change** event whenever a cell is changed, deleted, or added.

The Excel VBA Project Browser shows the sheets that are currently part of the workbook. The code below should be placed in the sheet that sends data to JMP. Double-click on the sheet icon in the VBA Project Window to bring up the code for that particular sheet.

    Private Sub Worksheet_change(ByVal Target as Range)
        Dim Col as JMP.Column
        If(DocOpen) Then
            If(Target.Row = 1) Then
                Return
            End If
        End If
        If(DT.NumberRows < Target.Row - 1) Then
            DT.AddRows Target.Row - DT.NumberRows - 1, Target.Row
        End If
        If(Not IsArray(Target.Value) And Not IsEmpty(Target.Value)) Then
            Set Col = DT.GetColumnByIndex(Target.Column)
            Col.SetCellVal Target.Row - 1, Target.Value
        End If
    End Sub

This code first checks to make sure JMP has a data table open. If the change is happening to the first row, then it is ignored because this is the column name in JMP. So, if a column name is changed in Excel, the corresponding change is *not* reflected in JMP. Code that would deal with heading changes could be inserted here, but is omitted in this example.

Next, if the row that has changed is beyond the number of rows that JMP is currently tracking in the data table, then the **AddRows** method is called to create more rows.

Finally, if the operation is on a single value and does not seem to signal a deletion, the JMP data table cell value is changed to the value that is passed into **Worksheet_Change**.

The main module is associated with the workbook. In the VBA Project Browser, the workbook code area is typically assigned the name **ThisWorkbook**, but this name can be easily changed. The following code goes into this area.

    'Public(Global Variables) that all Workbook subroutines can access
    Public Counter As Integer 'counter to update Control Chart every 5 changes
    Public JMPDoc As JMP.Document 'instance of JMP Document
    Public CChart As JMP.ControlChart 'instance of Control Chart
    Public ChartOpen as Boolean 'Flag to set if chart is open
    Public DB As AUTODB
'Shut Down JMP before closing the workbook
Private Sub Workbook_BeforeClose(Cancel as Boolean)
    DocOpen = False
    MyJMP.Quit
End Sub

'As soon as the workbook is opened via File Open, load JMP for Automation
Private Sub Workbook_Open()
    Set MyJMP = CreateObject("JMP.Application") 'Create an instance of JMP
    MyJMP.Visible=True 'Make this instance of JMP visible
    Counter = 0 'initialize counter that counts changes
    DocOpen = False 'no document open yet
    ChartOpen = False 'no charts open yet, either

    'CHANGE THIS PATH TO POINT TO THE EXCEL WORKSHEET
    Set DB = MyJMP.NewDatabaseObject
    DB.Connect ("DSN=Excel Files;DBQ=C:\Book2.xls;")
    Set DT = DB.ExecuteSQLSelect("SELECT * FROM ""Sheet1$""")
    DB.Disconnect
    Set JMPDoc = DT.Document
    DocOpen = True 'Set flag to say that the document is open
End Sub

'This is the most important part.
'After the first piece of data has been changed, generate a control chart.
'After every 5 changes to Excel worksheet cells, generate a new PNG of the
Control Chart.

Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Source As Range)
    Counter = Counter + 1
    'Save the control chart to a PNG every time 5 elements get updated
    If (Counter Mod 5 = 0 Or Counter = 1) Then

        'If the Control Chart has not been created yet, do so
        If Not (ChartOpen) Then
            Set CChart = JMPDoc.CreateControlChart 'create chart
            CChart.LaunchAddProcess "Column 1" 'Add column
            CChart.LaunchAddSampleUnitSize 5
            CChart.LaunchSetChartType jmpControlChartVar
            CChart.Launch 'launch the chart
            ChartOpen = True 'set flag to remember that a chart is open
        End If
        CChart.SaveGraphicOutputAs "C:\ControlChart.png", jmpPNG
    End If
End Sub

The Workbook_Open subroutine is called when the Excel table is initially loaded. It initializes some
variables, starts JMP, and tells JMP to open (through ODBC) the same Excel file that is currently loaded
into Excel 2007. Note that JMP opens the Excel file as a database object rather than opening it as a file. This
is necessary because JMP does not open a file that is already open in another application.
The **Workbook_Change** event is generated every time a user changes the data in any cell in any worksheet in the workbook. This sample assumes that there is only one active worksheet in the workbook. The first time the user changes a cell value in the worksheet, the **Workbook_Change** subroutine creates a Control Chart in JMP using the current data table.

In this sample, the **Workbook_change** subroutine also creates a PNG graphic file of the Control Chart output and updates it on the disk every fifth time a change is made to the workbook. This just gives some ideas on how Excel events and JMP automation can be used together to create output.

Finally, the **Workbook_BeforeClose** subroutine is invoked when the Excel workbook is closed, but before the window goes away. The code within this subroutine instructs JMP to close down as well.

Note that there are some limitations in this method. This example is good if the only activities that occur with the data are additions or changes. The Excel **Worksheet_Change** event is very limited in the reporting that it provides. In particular, cell-by-cell updating of a JMP data table can be difficult in instances where deletion, drag and drop, or block replication needs support.

If these are problem cases, it is probably better to rely on a brute-force approach. One way is to reload the data into JMP every time a certain number of changes occur. An example is shown here.

```vba
Private Sub Workbook_SheetChange(ByVal Sh as Object, ByVal Source as Range)
    Counter = Counter + 1
    If (Counter Mod 10 = 0) Then
        'If there is a previous chart of Table opened, close it first
        If(DocOpen) Then
            JMPDoc.Close False, ""
            CChart.CloseWindow
        End If
        Set JMPDoc = MyJMP.OpenDocument(InstallDir + "C:\BOOK1.XLS")
        Set DT = JMPDoc.GetDataTable
        DocOpen = True
        'Now, create the control chart.
        'This one is keyed to the data in "Column 1".
        'If 5 or more values are changed,
        'JMP should generate a new Chart and save it as a
        'PNG file to disk.
        'The PNG file can be viewed with Internet Explorer.
        Set CChart = JMPDoc.CreateControlChart
        CChart.LaunchAddProcess "Column 1"
        CChart.LaunchAddSampleUnitSize 5
        CChart.LaunchSetChartType jmpControlChartVar
        CChart.Launch
        CChart.SaveGraphicOutputAs "C:\ControlChart.png", jmpPNG
        End If
    End If
End Sub
```

This sample reloads the data every time there are 10 changes to the Excel Workbook. First, it removes JMP Control Charts and data tables that were previously created. Next, it loads the new data and creates a Control Chart.
This sample works best for small amounts of data. If very large Excel files are involved, this approach is not efficient because of the reloading of the table into JMP.

**Automating JMP through Visual C++**

Using C or C++ to create an automation client can be a long, tedious task. However, if you use the support provided by MFC in Microsoft Visual C++, the task is considerably easier. There are several steps that must be performed in order to get to a state where you can launch the automation server application (JMP in this case). The AutoClient application that is included in the Visual C++ Sample directory contains some code that provides ideas on how to get started. The Microsoft sample application CALCDRIV also shows a MFC-based automation client. CALCDRIV is typically included with Visual C++, and on MSDN CDs.

AutoClient shows how to start up JMP and drive a Bivariate analysis and the data table. The sample is much smaller than any of the Visual Basic samples. However, the mechanics behind all the automation calls that you might want to use are the same as the examples with Bivariate and the data table. The following steps are based on the Visual C++ Version 5.0 UI.

**Steps for Automating JMP**

1. Create your application, either manually or through App Wizard. Specify support for OLE automation. Even if you are not automating your own application, you need to include the OLE headers and initialization code. If you are retrofitting an existing application, you need to make sure that you include OLE support. This usually means including afxole.h in your application, and calling AfxOleInit() in your application InitInstance routine. Consult the MFC OLE documentation for details about this.

2. Bring up the Class Wizard and select the Automation tab. Select the Add Class drop down list and then the From a Type Library option. Navigate to the JMP install directory until you find JMP.TLB. Select this type library.

3. You are prompted to confirm the classes that you want to use in your project. If you are unsure what objects (and interfaces) that you want, select them all by Shift-clicking. Select the names for the files where the class wizard generates interface stubs and header information. Class Wizard is generating wrapper classes based on the MFC ColeDispatchDriver class. This gives you easy access to the OLE Invoke automation function without having to know a lot of the technical details. Select OK. Class Wizard generates the two files (.h and .cpp). You should include the .h file in whatever .cpp files use the JMP automation objects. For example, your View class implementation file.

4. The Class View of your Workspace now shows the Interface classes that you have imported. You can examine the methods and properties for each class through this class view.

5. To start JMP, define a variable of type IJMPAutoApp that persist for the length of the automation session. Call CreateDispatch on this variable, passing in the JMP ProgID ("JMP.Application") as the lone parameter. At this point, when the code executes JMP starts.

6. Call SetVisible(TRUE) on the JMP object created in step 5. If you do not want to see JMP execute, do not do this step. However, for debugging it is necessary.

7. Now you can use the JMP application object to spawn further objects, which themselves can spawn more objects. The first thing you probably want to do is load a Data table. To load an existing JMP data table, call the OpenDocument method on the JMP object created in step 5. If successful, this method
returns a dispatch pointer that can be attached to an object of type IJMPDoc using the 
AttachDispatch method.

8. The IJMPDoc object provides the methods to launch the analysis and graphing platforms. Once you 
create an analysis and attach the dispatch pointer, you can specify the data table columns to use in the 
analysis and then you can launch it. Once the analysis is launched, you can manipulate it using the 
properties and methods specific to that particular type of analysis. Code that is taken from the sample 
application that describes steps 5–8 is shown below:

**Example Program**

```cpp
//Note, no error handling is done in this example
IJMPAutoApp m_DispDriver;
IJMPDoc m_Doc;
IAutoBivar m_Bivar;
IAutoFit m_FitLine;

//Create the initial dispatch driver that uses the IJMPAutoApp 
//interface specification (taken from jmpauto.h)
m_DispDriver.CreateDispatch("JMP.Application");

if (m_DispDriver)
{
  //If JMP successfully started, make it visible
  m_DispDriver.SetVisible(TRUE);

  //Now open a data table as a document. The document interface 
  //pointer that is returned is then attached to our Doc dispatch  
  //driver class that uses the IJMPDoc interface specification.
  m_Doc.AttachDispatch(m_DispDriver.OpenDocument("C:\\JMPDATA\\BIGCLASS.JMP");
}

//First, call CreateBivariate on the Doc interface to create 
//a dispatch object to a Bivariate analysis. If there is already 
//a previous dispatch interface in m_Bivar, MFC releases it 
//in AttachDispatch.
m_Bivar.AttachDispatch(m_Doc.CreateBivariate( ));

//Now add Height and Weight as the columns to analyze
m_Bivar.LaunchAddX("Height");
m_Bivar.LaunchAddY("Weight");

//Launch the analysis
m_Bivar.Launch( );

//Create a FitLine. Since the Fit can be automated, attach the dispatch 
//pointer that is returned from FitLine( ) to a DispatchDriver object
m_FitLine.AttachDispatch(m_Bivar.FitLine( ));

//Now do a few more fits. This example does not automate these fit
/objects, although they do support automation.
m_Bivar.FitPolynomial(3.0);
m_Bivar.FitSpline(1000.0);

// Now manipulate the first FitLine object
m_FitLine.ConfidenceFit(TRUE);
m_FitLine.ConfidenceIndividual(TRUE);
This chapter includes advanced techniques, such as throwing and catching exceptions, defining your own functions, and using complex expressions.
Contents

Advanced Programming Concepts .................................................. 387
    Throwing and Catching Exceptions ........................................... 387
    Functions ............................................................................. 388
    Recursion ............................................................................. 389
    Includes ................................................................................. 390
    Loading and Saving Text Files .................................................. 390
Advanced Scoping and Namespaces .................................................... 390
    Names Default To Here ............................................................ 390
    Scoped Names ......................................................................... 393
    Namespaces ............................................................................ 396
    Referencing Namespaces and Scopes ......................................... 401
    Resolving Named Variable References ....................................... 403
    Best Practices for Advanced Scripting ....................................... 405
Scripting BY Groups ................................................................... 405
Lists and Expressions .................................................................... 406
    Stored expressions .................................................................... 406
    Macros .................................................................................. 414
    Manipulating lists .................................................................... 414
    Manipulating expressions ........................................................ 416
Encryption ................................................................................... 420
XML Parsing Operations .................................................................. 422
Pattern Matching and Regular Expressions ...................................... 424
Projects ...................................................................................... 427
Hexadecimal and BLOB Functions ................................................. 430
Web ............................................................................................ 432
Scheduling Actions ...................................................................... 433
Additional Numeric Operators ....................................................... 434
    Derivatives ............................................................................. 434
    Algebraic Manipulations ........................................................ 436
    Maximize and Minimize .......................................................... 437
Advanced Programming Concepts

This section covers some more advanced programming techniques that can be useful for developing complex scripts.

- “Throwing and Catching Exceptions,” p. 387
- “Functions,” p. 388
- “Recursion,” p. 389
- “Includes,” p. 390
- “Loading and Saving Text Files,” p. 390

Throwing and Catching Exceptions

A script can stop itself by executing the `Throw()` function. If you want to escape from part of a script when it is in an error condition, you can enclose it in a `Try()` expression.

Try takes two expression arguments. It starts by evaluating the first expression, and if or when the first expression throws an expression by evaluating `Throw`, it does the following:

1. Immediately stops evaluating that first expression.
2. Returns nothing
3. Evaluates the second expression.

`Throw` does not require an argument but has two types of optional arguments. If you include a character-valued expression as an argument, throwing stores that string in a global named `exception_msg`; this is illustrated in the first example below.

Examples

For example, you can use Try and Throw to escape from deep inside For loops.

```plaintext
a = [1 2 3, 4 5 6, 7 8 9];
b = a;
nr = nrow(a);
nc = ncol(a);
//a[2,3]=2; //uncomment this line to see the "Missing b" outcome

try(
    sum = 0;
    for(i=1,i<=nr,i++,
        for(j=1,j<=nc,j++,
            za = a[i,j]; if(isMissing(za),throw("Missing a");)
            zb = b[j,i]; if(isMissing(zb),throw("Missing b");)
            sum += za*zb;
        )
    ),
    show(i,j,exception_msg); throw();
);```

Advanced Concepts

Chapter 11

Advanced Programming Concepts

```plaintext
i = 2;
j = 3;
exception_msg = "Missing a";
```

You can also use `Try` and `Throw` to catch an exception that JMP itself throws:

```plaintext
try(
    dt=open("My dataset.jmp"); // a file that cannot be opened
    summarize( a =by(age), c=count, meanHt=mean(Height));
    show(a, c, meanHt),
    print("This script does not work without the data set"); throw();
);
```

You do not have to use `Try` to make use of `Throw`. In this example, `Throw` is not caught by `Try` but still stops a script that cannot proceed:

```plaintext
dt=new table(); // to get an empty data table
if (nrow(CurrentDataTable())==0, throw("!Empty Data Table");)
```

### Functions

JSL also has a function called `Function` to extend the macro concept with a local context arguments. Suppose that you want to create a function that takes the square root but tolerates negative arguments, returning zero rather than errors. You first specify the local arguments in a list with braces `{}` and then state the expression directly. You do not need to enclose the expression in `Expr` because `Function` stores it as an expression implicitly.

```plaintext
myRoot = function({x}, if(x>0, sqrt(x), 0));
a = myRoot(4); // result in a is 2
b = myRoot(-1); // result in b is 0
```

Functions are stored in globals, the same as values. This means that you cannot have both a root function and a root value. It also means that you can redefine a function anytime except when you are inside the function itself.

When a function is called, its arguments are evaluated and given to the local variables specified in the list forming the first argument. Then the body of the function, the second argument, is evaluated.

The values of the arguments are for the temporary use of the function. When the function is exited, the values are discarded. The only value returned is the return value. If you want to return several values, then return a list instead of a single value.

In defined functions, the stored function is not accessible directly, even by the `Name Expr` command. If you need to access the function expression in your script, you have to create the function within an `expr()` clause. For example,

```plaintext
makeFunction = expr(myRoot=function({x}, if (x>0, sqrt(x), 0)));
d=substitute(
    NameExpr(MakeFunction),
    expr(x), expr(y)
);
show(d);
makeFunction;
```
Local Symbols

You can declare variables as local to a function so that they do not affect the global symbol space. This is particularly useful for recursive functions, which need to keep separate the values of the local variables at each level of function call evaluation.

As shown above, a function definition looks as follows.

```
functionName=Function({arg1, ...}, body);
```

You can also have the function definition default all the unscoped names to be local.

```
functionName=Function({arg1, ...}, {Default Local}, body);
```

The use of `Default Local` localizes all the names that:

- Are not scoped as globals (for example, ::name)
- Are not scoped as data table column names (for example, :name)
- Occur without parentheses after them (for example, are not of the form name(...))

For example, the following function sums three numbers.

```
add3 = Function({a, b, c}, {temp}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

The following function does the same thing, automatically finding locals.

```
add3 = Function({a, b, c}, {Default Local}, temp=a+b; temp+c);
X=add3(1, 5, 9);
```

In both cases, the variable `temp` is not a global, or, if it is already a global, remains untouched by evaluating the functions.

**Note:** If you use an expression initially as local, then use it as a global, JSL changes the context. However, an expression used globally stays resolved globally regardless of its future use.

Recursion

The `Recurse` function makes a recursive call of the defining function. For example, you can make a function to calculate factorials. A factorial is the product of a number, the number minus 1, the number minus 2, and so on, down to 1.

```
myfactorial=function({a}, if (a==1, 1, a*recurse(a-1)));
myfactorial(5);
```

You can define recursive calculations without using `Recurse`. For example, you could replace `recurse` by `myfactorial`, and the script would still work. However, `Recurse` offers these advantages:

- It avoids name conflicts when a local variable has the same name as the function.
- You can recurse even if the function itself has not been named (for example, assigned to a global variable, such as `myfactorial` above).
Includes

The `include` function opens a script file, parses the script in it, and executes it.

```javascript
include("pathname");
```

For example,

```javascript
include("$SAMPLE_SCRIPTS/myStartupScript.jsl");
```

There is an option to obtain the parsed expression from the file, rather than evaluating the file.

```javascript
include("pathname", <<Parse Only);
```

Another named option creates a namespace that the included script runs in. This namespace is an anonymous namespace and it is independent from the parent script's namespace.

```javascript
Include("file.js1", <<New Context);
```

See “Advanced Scoping and Namespaces,” p. 390 for information about using namespaces with your scripts.

Loading and Saving Text Files

The `Load Text File` and `Save Text File` commands allow manipulation of text files from JSL. Note that the paths in the following code are strings.

```javascript
text = Load Text File( "path" );
Save Text File( "path", text );
```

You can load a text file from a Web site:

```javascript
Load Text File( "URL", <blob> );
```

The URL is a quoted string that contains the URL for the text file. The text file is returned as a string. If you add the optional named argument `blob`, a blob is returned instead.

Advanced Scoping and Namespaces

Scripts that are used in production environments need to use more advanced scoping techniques to avoid collisions between scripts. JMP provides three, progressively more advanced techniques:

- The `Names Default To Here()` function. If you have simple scripting needs, this single command might be sufficient. See “Names Default To Here,” p. 390.
- Scopes that are pre-defined by JMP. See “Scoped Names,” p. 393.
- Namespaces that you can create for your scripts. See “Namespaces,” p. 396.

Names Default To Here

If you write production scripts, you need to insulate the script from the current user environment. Otherwise, the variables that you use might interact with variables used by the user and by other scripts.
The way to do this is to keep your names in a local environment, which you can do by setting an execution mode with the statement:

```
Names Default To Here(1);
```

Unqualified names in a script with the `Names Default To Here` mode turned on are private to that script. However, the names persist as long as the script persists, or as long as objects created by or holding the script are still active. We recommend that all production scripts start with `Names Default To Here(1)` unless there is a specific reason not to do so. When the script uses an unqualified name in this mode, that name is resolved in the local namespace.

To refer to global variables, scope the name specifically as a global variable (for example, `::global_name`). To refer to columns in a data table, scope with name specifically as a data table column (for example, `:column_name`).

In JMP 8 and earlier, the only method to insulate scripts was to use lengthy names that were less likely to collide with names in other scripts. Using `Names Default To Here(1)` makes this technique unnecessary.

`Local()` creates local scopes only in specific contexts within a script and cannot enclose a longer script with interacting functions, while `Names Default To Here(1)` creates a local scope for an entire script.

If you have simple scripting needs, this single command might be sufficient.

### Handling Unqualified Named Variable References

The `Names Default To Here()` function determines how unqualified named variable references are resolved. Explicitly scoping a variable using `here:var_name` always works, whether `Names Default To Here()` is on or off. See “Scoped Names,” p. 393 for details about `here` and other scopes.

Enabling the `Names Default To Here` mode associates a scope called `Here` with an executing script. The `Here` scope contains all of the unqualified named variables that are created when they are the target of an assignment (as an L-value). In JMP 8 and earlier, these variables normally would have been placed in the `Global` scope. Using a `Here` scope keeps variables in multiple executing scripts separate from each other, avoiding name collisions and simplifying the scripting and management of variable name collisions. You can still share information using the `Global` scope.

### Names Default To Here and Global Variables

Run this example script one line at a time to see how the `Names Default To Here()` function changes the resolution of variable names.

**Example Script**

```
a=1;
Names Default To Here(1);
```
a=5;
show(global:a, a, here:a);
  global:a = 1;
a = 5;
  here:a = 5;

1. Run the first line to create a global variable named a that holds the value 1.
2. Run the second line to turn on the Names Default To Here mode.
3. Run the third line to create a new variable named a in the local space that holds the value 5. This line does not change the value assigned to the global variable a.
4. Run the fourth line to see how scoped and unscoped variables are resolved.

The unqualified a is resolved to here:a. If Names Default To Here() were not on, a would be resolved to the global variable named a.

Note that if you use ::a in the Show() function, your output is a little different:

a = 1;
a = 5;
here:a = 5;

Example of Using the Names to Default To Here() Function

You have two scripts with the following definitions, and Names Default To Here() is turned off (the default condition) in both scripts.

Note: Both scripts must be in separate script windows for this example.

// Script 1
a = 1;
show(a);

// Script 2
a = 3;
show(a);

1. Run Script 1. The result is as follows:
   a = 1
2. Run Script 2. The result is as follows:
   a = 3
3. Run only the show(a); line in Script 1. The result is as follows:
   a = 3

The log shows a = 3 because variable a is global, and was last modified by Script 2. This is the default behavior in JMP 9, and it is the only possible behavior in JMP 8 and earlier.

4. Now turn on Names Default To Here() in both scripts.
   Names Default To Here(1);
Note: Names Default To Here is local to a particular script. It is not a global setting.

5. Run Script 1. The result is as follows:
   \[ a = 1 \]
6. Run Script 2. The result is as follows:
   \[ a = 3 \]
7. Run only the \texttt{show(a);} line in Script 1. The result is as follows:
   \[ a = 1 \]

The log shows \( a = 1 \), because a copy of variable \( a \) is maintained for each script.

Note: Problems using this function are generally due to the mixing of unqualified and qualified references to global variables. Always explicitly scoping a name prevents accessing an unintended variable.

**Scoped Names**

Specify where a name is to be resolved by using a scope in the form \texttt{scope:name} where \texttt{scope} indicates how to resolve the name. For example, \texttt{here:name} indicates that the name should be resolved locally. Using the Names Default To Here mode, \texttt{here:name} is equivalent to \texttt{name}. The scope instructs how to look up the name.

The syntax is to use the colon scope operator:

\[ \texttt{scope:name} \]

There are several types of scopes:

- Scope can be a resolution rule. For example, \texttt{here:x} means that \( x \) should be resolved to a name that is local to the script. \texttt{Global:x} means that \( x \) should be resolved to a global name.
- Scope can be a namespace reference variable. For example, \texttt{ref:a} means that \( a \) should be resolved within the namespace that \( \texttt{ref} \) refers to.
- Scope can be a data table reference to look up names as column names. For example, \texttt{dt:height} means that \( \texttt{height} \) should be resolved as a column in the data table that \( \texttt{dt} \) refers to.
- Scope can be the name of a namespace that you created. For example, \texttt{myNamespace:b} where \texttt{myNamespace} is a namespace that you created. "myNamespace":b is equivalent. See “Namespaces,” p. 396.

**Predefined Scopes**

JMP 9 provides support for specialized scopes provided by JMP. Scopes are predefined and cannot be removed or replaced. Each of these scopes have specific roles, depending on its associated object.
Advanced Concepts
Advanced Scoping and Namespaces

Chapter 11
Advanced Scoping and Namespaces

Example of Using the Window Scope

This example uses the Window scope to pass information during execution. Explicitly scoping the variables x and y to this window ensures that JMP does not try to scope x and y in other contexts, such as a data table. The variables x and y are created and used solely inside the Window environment. The Window scope is similar to using Local(), but more useful because Local() is limited in the places that it can be used.

```
New Window( "Example",
    window:gx = 20;
    window:gy = 50;
    Graph Box(;
        Frame Size( 200, 200 ),
        Handle(;
            window:gx,
            window:gy,
            Function( {x, y},
                window:gx = x;
                window:gy = y;
            )
        );
        Circle( {0, 0}, Sqrt( window:gx * window:gx + window:gy * window:gy ) );
    );
);
```
Example of Using the Here Scope

This example uses the Here scope to pass information between windows that are created by the same script. Scoping a variable using Here: is not dependent on turning Names Default To Here() on. The Here: scope is always available.

This script produces two windows and uses two different scopes.

The Launcher window asks the user for two values. Those two values are passed to the Output window, which uses them to graph a function. The Launcher window scopes aBox and bBox to the window: essentially, those two variables (pointers to Number Edit Boxes) exist only in the Launcher window and are not available to the Output window. The values from those two boxes are then copied into variables that are scoped to Here, and so are available to both windows that are produced by this script.

```plaintext
launchWin = New Window( "Launcher", <<Modal,
V List Box(
Lineup Box(
    N Col( 2 ),
    Spacing( 10 ),
    Text Box( "a" ),
    window:aBox = Number Edit Box( 50 ),
    Text Box( "b" ),
    window:bBox = Number Edit Box( 20 ),
),
Lineup Box(
    N Col( 2 ),
    Spacing( 20 ),
    Button Box( "OK",
        // copy values before window goes away
        here:a = window:aBox << Get;
        here:b = window:bBox << Get;
        launchWin << CloseWindow;
    ),
),
```

Figure 11.1 Example of Current Window Namespace
Advanced Scoping and Namespaces

Namespaces

A namespace is a collection of unique names and corresponding values. You can store references to namespaces in variables. Namespace names are global, because JMP has only one namespace map. Namespace references are variables like any other variable that references an object, so they must be unique within their scope or namespace. The members of a namespace are referenced with the : scoping operator, such as my_namespace:x to refer to the object that is named x within the namespace called my_namespace. See “User-Defined Namespace Functions,” p. 396 for details about creating and managing your own namespaces. Namespaces are especially useful for avoiding name collisions between different scripts.

User-Defined Namespace Functions

In JMP 9, you can create your own namespaces to hold related sets of variables and function definitions. There are several functions that you can use to manage namespaces.

New Namespace

nsref = New Namespace( <"nsname">, <{ name = expr, ... }> );
Advanced Concepts

Chapter 11

Advanced Scoping and Namespaces

397

Creates a new namespace called `nsname` (a string expression) and returns a reference to the namespace. All arguments are optional.

`Nsname` is the name of the namespace in the internal global list of namespace names. `Nsname` can be used as the prefix in a scoped variable. The function returns a reference to the namespace, and can also be used as the prefix in a scoped variable reference. If `nsname` is absent, the namespace is anonymous and is given a unique name created by JMP. `Show Namespace()` shows all namespaces and their names, whether assigned or anonymous.

**Important:** If you already have a namespace named `nsname`, it is replaced. This behavior means that while you are developing your script, you can make your changes and re-run the script without having to clear or delete your namespace. To avoid unintentional replacement, you can either use anonymous namespaces, or test to see whether a particular namespace already exists:

```julia
If( !Namespace Exists( "nsname", New Namespace( "nsname" ) ) );
```

A list of named expressions is optional. The names are JMP variables that exist only within the namespace.

**Note:** The named expressions must be a comma-separated list. Separating the expressions with semi-colons causes the list to be ignored.

These namespaces must be uniquely named to prevent collisions in situations where multiple user-defined namespaces are being used. Using anonymous namespace names prevents collisions.

```julia
Namespace
  nsref = Namespace( "nsname" | nsref);
```

Returns a namespace reference. The argument might be any of the following:

- a quoted string that contains a namespace
- a reference to a namespace
- a reference to an object that owns a namespace (such as a data table, a window, or a platform)

If `nsname` is a data table, then the namespace for the data table columns is returned.

**Note:** `Namespace()` returns a reference to a namespace that already exists. It does not create a new namespace.

```julia
Is Namespace
  b = Is Namespace( nsref );
```

Returns 1 (true) if `nsref` is a namespace or 0 (false) otherwise.

```julia
As Scoped
  b = As Scoped( "nsname", var_name);
  nsname:var_name;
```
Advanced Concepts
Advanced Scoping and Namespaces

As `Scoped()` is the function form of a scoped reference. The function returns a reference to the specified variable in the specified scope.

Namespace Exists

\[ b = \text{Namespace Exists}( \text{"nsname"}); \]

Returns 1 (true) if `nsname` exists in the list of global namespaces, or 0 (false) otherwise.

Show Namespaces

\[ \text{Show Namespaces();} \]

Shows the contents of all namespaces contained in the list of global namespaces. Namespaces are not visible unless a reference is made to one, using either the **New Namespace** or **Namespace** functions.

**Namespace Messages**

In addition to the namespace management functions, a namespace also supports a set of messages to access and manipulate its contents.

Note that these messages, as with all message, must be sent to a scriptable object. A namespace name is not a defined scriptable object and cannot be used in a `Send` operation. However, you can use the name of a namespace in variable references. For example, `nsname::var` is equivalent to `nsref::var`.

Table 11.2 defines the messages that are supported by user-defined namespace references.

**Table 11.2 Namespace Messages**

<table>
<thead>
<tr>
<th>Namespace Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ns &lt;&lt; Contains( &quot;var_name&quot; );</code></td>
<td>Returns 1 or 0, depending on whether <code>var_name</code> exists within the namespace.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; Delete;</code></td>
<td>Removes this namespace from the internal global list.</td>
</tr>
<tr>
<td></td>
<td>To delete variables in the namespace, use <code>&lt;&lt;Remove</code>. See the entry for <code>&lt;&lt;Remove</code> in this table.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; First;</code></td>
<td>Returns a quoted string that contains the first variable name used within the namespace.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; Get Contents;</code></td>
<td>Returns a list of key-value pairs, which are each enclosed in a list. Each key is a quoted string that contains a variable name, and each value is the unevaluated expression that the variable contains.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; Get Keys;</code></td>
<td>Returns a list of variable names.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; Get Name;</code></td>
<td>Returns the name of this namespace.</td>
</tr>
<tr>
<td><code>ns &lt;&lt; Get Value( &quot;var_name&quot; );</code></td>
<td>Returns the unevaluated expression that <code>var_name</code> contains in this namespace.</td>
</tr>
</tbody>
</table>
### Table 11.2 Namespace Messages (Continued)

<table>
<thead>
<tr>
<th>Namespace Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns &lt;&lt; Get Values;</td>
<td>Returns a list of unevaluated expressions that each variable in this namespace contains.</td>
</tr>
<tr>
<td>ns &lt;&lt; Get Values( { &quot;var_name1&quot;, &quot;var_name2&quot;, ... } );</td>
<td>Returns a list of unevaluated expressions that each variable in this namespace specified in the list argument contains. If a requested variable name is not found, an error is returned.</td>
</tr>
<tr>
<td>ns &lt;&lt; Insert( &quot;var_name&quot;, expr );</td>
<td>Inserts into this namespace a variable named var_name that holds the expression expr.</td>
</tr>
<tr>
<td>ns &lt;&lt; Lock; ns &lt;&lt; Unlock;</td>
<td>Locks all variables in the namespace and prevents variables from being added or removed. &lt;&lt;Unlock unlocks all of the namespace’s variables.</td>
</tr>
<tr>
<td>ns &lt;&lt; Lock( &quot;var_name&quot;, ... ); ns &lt;&lt; Unlock( &quot;var_name&quot;, ... );</td>
<td>Locks the specified variables in this namespace. If no variables are specified, all variables are either locked or unlocked.</td>
</tr>
<tr>
<td>n = ns &lt;&lt; N Items;</td>
<td>Returns the number of variables contained in this namespace.</td>
</tr>
<tr>
<td>next k = ns &lt;&lt; Next( &quot;var_name&quot; );</td>
<td>Returns the name of the variable that follows the specified variable.</td>
</tr>
<tr>
<td>ns &lt;&lt; Remove( &quot;var_name&quot;, ... );</td>
<td>Removes the specified variable or list of variables.</td>
</tr>
</tbody>
</table>

### Using Namespace References

The following are all equivalent references to a variable that is named b in the namespace that is named nsname that has a reference nsref:

```plaintext
nsref:b
nsname:b
"nsname":b
nsref["b"]
nsref<<Get Value("b") // used as an r-value
```

### Namespaces and Included Scripts

An included script runs in the namespace of the parent script. If the included script has its own namespace definitions, you need to do one of the following:

- manage the namespace names to avoid name collisions
Advanced Concepts

Advanced Scoping and Namespaces

- use anonymous names created by the **New Namespace** function

In either case, you still need to manage variable references to namespaces.

There is also an option for the **Include** function (**New Context**) that creates a namespace that the included script runs in. This namespace is an anonymous namespace and it is independent from the parent script's namespace. For example,

```javascript
Include("file.jsl", <<New Context);
```

This anonymous namespace can be referenced using **Here**.

See “Includes,” p. 390 for more information about the **Include** function.

**Examples of User-Defined Namespaces**

Creating and Using a Basic Namespace with Expressions

This example shows creating an anonymous namespace and using functions and variables within it.

```javascript
new_emp = New Namespace(
    {name_string = "Hello, *NAME*!",
     print_greeting = Function( {a},
       Print( Substitute( new_emp:name_string, "*NAME*", Char( a ) ) )
     )
    );

Note that you must use the fully qualified name for variables defined within the namespace.

ew_emp:print_greeting( 6 );
"Hello, 6!"
```

Complex Number Operations

This example creates a namespace that contains functions to support using 2-element lists to represent complex numbers, and then locks the namespace.

```javascript
If( !Namespace Exists( "complex" ),
    New Namespace( "complex" );

complex:make = Function( {a, b}, Eval List( {a, b} ) );
complex:add = Function( {a, b}, a + b );
complex:subtract = Function( {a, b}, a - b );
complex:divide = Function( {a, b},
    d = b[1] ^ 2 + b[2] ^ 2;
complex:char = Function( {a}, Char( a[1] ) || "+" || Char( a[2] ) || "i" );
);
Namespace( "complex" ) << Lock;
```
Here are examples using functions that are within the above user-defined namespace.

```erlang
cl = complex:make( 3, 4 );
{3, 4}

c2 = complex:make( 5, 6 );
{5, 6}

cml = complex:make( [1, 2, 3], [4, 5, 6] );
{[1, 2, 3], [4, 5, 6]}

cadd = complex:Add( cl, c2 );
{8, 10}

csum = complex:Subtract( cl, c2 );
{-2, -2}

cmul = complex:Multiply( cl, c2 );
{-9, 38}

cdiv = complex:Divide( cl, c2 );
{14.6065573770492, 19.7049180327869}

show( complex:char( cl ) );
complex:char(c1) = "3+4i";
```

**Referencing Namespaces and Scopes**

With the addition of namespaces and scopes to extend the scripting capabilities managing variables, there are a number of factors in how a named variable reference is resolved. Table 11.3 provides additional information about named variable references that are resolved for specific situations.
### Table 11.3 Namespace References\(^a\)

<table>
<thead>
<tr>
<th>Form</th>
<th>Reference Type</th>
<th>Reference Rule</th>
<th>Creation Rule</th>
</tr>
</thead>
</table>
| a    | Unqualified    | If the *Names Default To Here* mode is on, JMP looks for the variable in these locations:  

- Local namespace\(^b\)
- Here namespace
- current data table  

If the *Names Default To Here* mode is off, JMP looks for the variable in these locations:  

- Local namespace\(^b\)
- Here namespace
- Global namespace
- current data table  

| :a   | Current data table | JMP looks for the variable in the current data table. | (Not applicable) |
| ::a  | Global            | JMP looks for the variable in the Global namespace. | JMP creates the variable in the Global namespace. |
| Global:a |                |                |               |
| Shared:a | Qualified        | JMP looks for the variable in the current data table and in the Global namespace. | JMP creates the variable in the Global namespace. |
| ns:a | Qualified         | JMP looks for the variable in the specified namespace. If the variable is not found, an error results. | JMP creates the variable in the specified namespace. Any previous values are replaced. |
| dt:a | Qualified         |                |               |
| Here:a |                |                |               |
| "name":a |                |                |               |
| expr:a |                |                |               |
| ns["a"] | Subscript      | JMP looks for the variable in the specified namespace. If the variable is not found, an error results. | JMP creates the variable in the specified namespace. Any previous values are replaced. |
| ns[expr] |                |                |               |
| Platform:a | Qualified    | JMP looks for the variable in the encapsulating platform. | JMP creates the variable in the encapsulating platform. |
| a    | Qualified         |                |               |
Chapter 11

Advanced Scoping and Namespaces

Resolving Named Variable References

When variables are referenced within a JMP script, JMP resolves the storage location of the variable using a specific set of rules. If the variable is referenced by a qualified name, then the resolution is based on the specific qualification specification. If the variable is referenced by an unqualified name, the situation is a bit more complex. JMP looks through a hierarchy of scopes representing the point of execution with the executing script. This section describes the rules that are used to resolve named variable references.

By default, variable name resolution in JMP 9 works the same way as in JMP 8 and earlier, allowing your current JSL scripts to be executed unchanged. For JMP 9, the difference between qualified and unqualified variable named references is important to understand. A qualified named reference uses the : and :: operators to provide specific information about where a referenced variable resides, or where it is created.

Examples of qualified named references include the following:

:var
::globalvar
datatable:var
nsref:var
"nsname":var

An unqualified named reference provides no explicit information to completely identify where a variable resides or where it is created. No scoping operator (:) or ::) is specified in the reference. To change the behavior of JMP when resolving unqualified named variable references, use the Names Default To Here function. For more details about JMP 8 variable name resolution, see the “Name Resolution,” p. 38 in the “JSL Building Blocks” chapter.
### Table 11.4 Resolving Access to a Variable in JMP 8 versus JMP 9

<table>
<thead>
<tr>
<th></th>
<th>JMP 8</th>
<th>JMP 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If it is followed by a pair of parentheses ( ), look it up as a function.</td>
<td>See “Function Resolution Rules,” p. 42 in the “JSL Building Blocks” chapter.</td>
</tr>
<tr>
<td>2</td>
<td>If it is prefixed by : or an explicit data table reference, look it up as a data table column or table variable.</td>
<td>If it is prefixed by ::, look it up as a global variable.</td>
</tr>
<tr>
<td>3</td>
<td>If it is prefixed by ::, look it up as a global variable.</td>
<td>If it is an explicit scope reference (such as <code>group:vowel</code>), look it up in the user-defined <code>group</code> namespace, which could be represented as a data table or namespace.</td>
</tr>
<tr>
<td>4</td>
<td>Not applicable.</td>
<td>If it is in a Local or Parameter function, look it up as a local variable. If it is in a Local or Parameter function, look it up as a local. If it is nested, repeat until a function call boundary is found.</td>
</tr>
<tr>
<td>5</td>
<td>If it is in a Local or Parameter function, look it up as a local variable.</td>
<td>If it is in a Local or Parameter function, look it up as a local. If it is nested, repeat until a function call boundary is found.</td>
</tr>
<tr>
<td>6</td>
<td>If it is in a user-defined function, look it up as a function argument or local variable.</td>
<td>Look it up in the current scope and its parent scope. Repeat until the Here scope is encountered.</td>
</tr>
<tr>
<td>7</td>
<td>Not applicable.</td>
<td>Look it up as a variable in the Here scope.</td>
</tr>
<tr>
<td>8</td>
<td>Not applicable.</td>
<td>Look it up as a variable in the Here scope.</td>
</tr>
<tr>
<td>9</td>
<td>Look it up as a global variable.</td>
<td>Look it up as a data table column or table variable.</td>
</tr>
<tr>
<td>10</td>
<td>Look it up as a data table column or table variable.</td>
<td>Look it up as a data table column or table variable.</td>
</tr>
<tr>
<td>11</td>
<td>Look it up as a platform launch name (for example, Distribution, Bivariate, Chart, and so on).</td>
<td>Look it up as a platform launch name (for example, Distribution, Bivariate, Chart, and so on).</td>
</tr>
</tbody>
</table>
| 12 | If it is used as the target of an assignment (as an L-value), create and use a global variable. | If it is used as the target of an assignment (as an L-value), test the following:  
  - If it is prefixed by ::, create and use a global variable.  
  - If it is an explicit scope reference, create and use the variable in the specified namespace or scope.  
  - If it is neither of the above, create and use a variable in the Here scope. |
Best Practices for Advanced Scripting

Minimize Polluting the Global Namespace and Prevent Scripts from Interacting

Always start your script with this line:

```
Names To Default To Here(1);
```

Share Variables Across Scripts

Use named namespaces. Namespace names are placed in the global scope.

Use Anonymous Namespaces

Using namespace references to anonymous namespaces avoids possible conflicts with other namespaces.

Scripting BY Groups

By group arguments are supported for these functions: `ColMean()`, `ColStdDev()`, `ColNumber()`, `ColNMissing()`, `ColMinimum()`, `ColMaximum()`.

Any number of BY arguments can be specified, and you can use expressions for the BY arguments. BY arguments must be used in a column formula, or in the context of `ForEachRow()`. The first argument can also be a general numeric expression.

Here are some examples:

```
New Column( "Mean of height by sex", Numeric, Formula( Col Mean( :height, :sex )));

New Column( "Minimum of height by sex and age", Numeric, Formula( Col Minimum( :height, :sex, :age )));

Distribution( Continuous Distribution( Column( :height ) ), By( :sex ));

Tabulate(
    Show Control Panel( 0 ),
    Add Table(
        Column Table(
            Analysis Columns( :height ),
            Statistics( Mean, N, Std Dev, Min, Max, N Missing )
        ),
        Row Table( Grouping Columns( :age, :sex ) )
    )
);
```
Lists and Expressions

Stored expressions

An expression is something that can be evaluated. The first section of the chapter, “Context: Meaning is Local,” p. 36 in the “JSL Building Blocks” chapter, discussed how JMP evaluates expressions. Now you must consider when JMP evaluates expressions.

JMP tends to evaluate things as soon as it possibly can, and it returns a result. If an expression is on the right side of an assignment, the result is what is assigned. Usually, that is what you want and expect, but sometimes you need to be able to delay evaluation.

Quoting and unquoting expressions

The operators to control when expressions are evaluated are Expr and Eval, which you should think of as the procrastination and eager operators. Expr just copies its argument as an expression, rather than evaluating it. Eval does the opposite: it evaluates its argument, and then takes that result and evaluates it again.

Expr and Eval can be thought of as quoting and unquoting operators, telling JMP when you mean the expression itself, and when you mean the result of evaluating the expression.

The following examples all assume these two assignments:

```
x=1; y=20;
```

If you assign the expression `x+y` to `a`, quoting it as an expression with Expr, then whenever `a` is evaluated, it evaluates the expression using the current values of `x` and `y` and returns the result. (Exceptions are the utilities Show, Write, and Print, which do not evaluate expressions for pure name arguments.)

```
a = expr(x+y);
a;
21
```

If you want the expression that is stored in the name, rather than the result of evaluating the expression, use the NameExpr function. See “Retrieve a stored expression, not its result,” p. 407.

```
show(nameExpr(a));
NameExpr(a) = x + y
```

If you assign an extra level of expression-quoting, then when `a` is evaluated, it unpacks one layer and the result is the expression `x+y`.

```
a = expr(expr(x+y));
show(a);
a = Expr(x + y)
```

If you want the value of the expression, then use Eval to unpack all layers:

```
show(eval(a));
Eval(a) = 21
```

You can do this to any level, for example:
Chapter 11
Advanced Concepts
Lists and Expressions

$$a = \text{expr} (\text{expr} (\text{expr} (\text{expr} (x + y))))$$
$$b = a;$$
$$\text{Expr} (\text{Expr} (x + y))$$
$$c = \text{eval} (a);$$
$$\text{expr} (x + y)$$
$$d = \text{eval} (\text{eval} (a));$$
$$x + y$$
$$e = \text{eval} (\text{eval} (\text{eval} (a)));$$
$$21$$

**Store scripts in global variables**

The main use of `Expr` is to be able to store scripts (such as macros) in global variables.

$$\text{dist} = \text{expr} (\text{Distribution} (\text{Column} (\text{height})))$$

Now when you want to do the script, just mention the symbol:

$$\text{dist}$$

You could even put it in a loop to do it many times:

$$\text{for} (i = 0, \text{i < 10}, \text{i = i + 1}, \text{dist});$$

You can use `Eval` to evaluate an expression explicitly:

$$\text{eval} (\text{dist});$$

Note, however, that in column formulas, `eval` only works if it is outermost in the formula. So, for example,

$$\text{Formula} (\text{log} (\text{eval} (\text{columnname} (i))))$$

would generate an error. Instead, use

$$\text{Formula} (\text{Eval} (\text{Substitute} (\text{expr} (\text{log} (\text{xxx})), \text{expr} (\text{xxx}), \text{columnname} (i)))))$$

As another example,

$$\text{Formula} (\text{eval} (\text{columnname} (i)) + 10)$$

generates an error, since `eval` is actually under the `Add` function. Instead, use

$$\text{Formula} (\text{Eval} (\text{Substitute} (\text{expr} (\text{xxx} + 10), \text{expr} (\text{xxx}), \text{columnname} (i)))))$$

**Retrieve a stored expression, not its result**

What if you wanted the symbolic value of a global (such as the expression `Distribution(\text{Column}(\text{height}))` stored in `\text{dist}` above), rather than the evaluation of it (the actual launched platform)? The `Name Expr` function does this. `Name Expr` retrieves its argument as an expression without evaluating it, but if the argument is a name, it looks up the name's expression and uses that, unevaluated.

`Expr` returns its argument exactly, whereas `Name Expr` looks up the expression stored in its argument. `Name Expr` “unpacks” just one layer to get the expression, but does not keep unpacking to get the result.

For example, you would need to use this if you had an expression stored in a name and you wanted to edit the expression:
Advanced Concepts
Lists and Expressions

popVar = Expr( Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() ) );

unbiasedPopVar = substitute( name expr( popVar ), expr( wild()/nrow() ), expr( Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / ( N Row() - 1 ) ) ) );

Compare x, Expr(x), NameExpr(x), and Eval(x) after submitting this script:

```julia
a=1; b=2; c=3;
Expr(x) = a+b+c;
```

<table>
<thead>
<tr>
<th>Command and result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x; 6</td>
<td>Evaluates x to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks all layers).</td>
</tr>
<tr>
<td>Eval(x); 6</td>
<td>Equivalent to simply calling x. Evaluates x to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks all layers).</td>
</tr>
<tr>
<td>NameExpr(x); a+b+c</td>
<td>Returns the expression that was stored in x, which is a+b+c (unpacks the outside layer).</td>
</tr>
<tr>
<td>Expr(x); x</td>
<td>Returns the expression x (packs one layer).</td>
</tr>
</tbody>
</table>

JSL also supports functions to access and traverse expressions, all of them either a name or a literal expression as an argument. In the following, `expressionArg` is either a single name or a compound expression to be taken literally.

`NArg(expressionArg)` finds the number of arguments in expressionArg.

The `expressionArg` can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by `Expr()`.

`NArg (name)` obtains the expression held in name (it is not evaluated) and returns the number of arguments

`NArg (expression)` evaluates expression and returns the number of arguments

`NArg (Expr(expression))` returns the number of arguments to literal expression.

For example, if aExpr = {a+b,c,d,e+f+g};
- `NArg(aExpr)` results in 4.
- `NArg(Expr(aExpr,4))` results in 3.
- `NArg(Expr([{1,2,3,4}]))` results in 4.
Head(expressionArg) returns the head of the expression without any arguments. If the expression is an infix, prefix, or postfix special character operator, then it is returned as the functional equivalent.

The expressionArg can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

For example, if aExpr = expr(a+b);

- r = Head(aExpr) results in Add().
- r = Head (Expr(sqrt(r))) results in Sqrt().
- r = Head({1,2,3}) results in {}.

Arg(expressionArg, indexArg) extracts the specified argument of the symbolic expression, resulting in an expression.

For example,

- Arg(name, i) obtains the expression held in name (it is not evaluated) and finds the ith argument
- Arg(expression, i) evaluates expression and finds the ith argument
- Arg(Expr(expression), i) finds the ith argument of expression

As another example, if aExpr = Expr(12+13*sqrt(14-15));

- Arg(aExpr, 1) yields 12
- Arg(aExpr, 2) yields 13*sqrt(14-15)
- Arg(Expr(12+13*sqrt(14-15)), 2) yields 13*sqrt(14-15)

To extract an argument of an argument inside an expression, you can nest Arg commands:

- Arg(Arg(aExpr, 2), 1) yields the first argument within the second argument of aExpr, or 13.
- Arg(Arg(aExpr, 2), 2) yields Sqrt( 14 - 15 )
- Arg(Arg(Arg(aExpr, 2), 2), 1) yields 14 - 15
- Arg(Arg(Arg(aExpr, 2), 2), 3) yields Empty()

Here is a description of how the last example line unwraps itself:

1. The inner Arg statement is evaluated.
   Arg(aExpr, 2)
   13 * Sqrt( 14 - 15 )

2. Then the next one is evaluated.
   Arg(Arg(aExpr, 2), 2)
   // this is equivalent to Arg( (13 * Sqrt( 14 - 15 ) ), 2)
   Sqrt( 14 - 15 )

3. Finally, the outer Arg is evaluated.
Arg(Arg(Arg(aExpr,2),2),3)
   // this is equivalent to Arg ( (Sqrt( 14 - 15 ) ), 3)
   Empty()

There is only one element to the Sqrt expression, so a request for the third argument yields Empty(). To access the two arguments inside the Sqrt expression, try this:

Arg(Arg(Arg(aExpr,2),2),1),2);

HeadName(expressionArg) returns the name of the head of the expression as a string. If the expression is an infix, prefix, postfix, or other special character operator, then it is returned as the functional equivalent.

The expressionArg can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

For example, if aExpr = expr(a+b);

• r = HeadName (aExpr) results in "Add".
• r = HeadName (Expr(sqrt(r))) results in "Sqrt".
• r = HeadName ({1,2,3}) results in "List".

In previous versions of JMP, other versions of Arg, Narg, Head, and HeadName were implemented, called ArgExpr, NArgExpr, HeadExpr, and HeadNameExpr, respectively. These did the same thing, but did not evaluate their argument. These forms are now deprecated and will not be documented in future versions.

Making lots of substitutions

Eval Insert is for the situation where you want to make a lot of substitutions, by evaluating expressions inside a character string. [In Perl, this is called interpolation.]

With Eval Insert, you specify characters that delimit the start and end of an expression, and everything in between is evaluated and expanded.

There are two functions, one to return the result, the other to do it in-place.

resultString = EvalInsert(string with embedded expressions,startDelimiter,endDelimiter)
EvalInsertInto(string l-value with embedded expressions,startDelimiter,endDelimiter)

The delimiter is optional. The default start delimiter is "^". The default end delimiter is the start delimiter.

xstring = "def";
r = EvalInsert("abc^xstring^ghi"); // results in "abcdefghi";

// in-place evaluation
r = "abc^xstring^ghi";
EvalInsertInto(r); // r now has "abcdefghi";

// with specified delimiter
r = EvalInsert("abc%xstring%ghi","%"); // results in "abcdefghi";
// with different start and end delimiters
r = EvalInsert("abc\[xstring\]ghi","[","]");  // results in "abcdefghi"

**Evaluate expressions inside lists**

Eval List evaluates expressions inside a list and returns a list with the results:

```julia
x = { 1+2, 3+4 };  
y = evallist(x);  // result in y is {3,7}
```

Eval List is useful for loading the list of user choices returned by Column Dialog or New Window with the Modal argument.

**Evaluate expressions inside expressions**

Eval Expr evaluates expressions inside other expressions but returns an expression that includes the results. By comparison, Eval evaluates any expressions inside the expression, and then evaluates the expression. Here is an example where you would need to evaluate the inner expression first:

```julia
// assumes a data table with column named X3
x = expr( distribution(column( expr("X"||char(i)) ))) ;
i = 3;  
y=Eval Expr(x);  // returns Distribution(Column("X3"))
```

However, Eval Expr only unpacks the inside layer then stops, returning the result as an expression. To evaluate further, you need to either call the result in a subsequent step, or else put Eval() around the Eval Expr().

```julia
// two-step method
y=Eval Expr(x);  
y;

// one-step method
eval(eval expr(x));
```

See Table 11.7 “Compare all the operators for controlling evaluation,” p. 413, to learn what would happen if you tried to use Eval directly on x without first doing Eval Expr.

**Parsing strings into expressions, and vice versa**

Parsing is the syntactic scanning of character strings into language expressions. Suppose that you have read in a valid JSL expression into a character string, and now want to evaluate it. The Parse function returns the expression. To evaluate it, use the Eval function.

```julia
x = parse("a=1") ;  // x now has the expression a=1  
eval(parse("a=1"));  // a now has the value 1
```

To go in the reverse, use the Char function, which converts an expression into a character string. Usually the argument to a Char function is an Expr function (or a NameExpr of a global variable), since Char evaluates its argument before deparsing it.

```julia
y = char(expr(a=1));  // results in y having the character value "a=1"
```
z = char(42);

The Char function allows arguments for field width and decimal parameter if the argument is a number. The default is 18 for width and 99 for decimal (Best format).

fortytwo=char(42,5,2); // results in the character value "42.00"

The reverse of Char is not quite as simple. To convert a character string into an expression, you use Parse, but to convert a character string into a number value, you use Num.

parse(y);
num(z);

Table 11.6 Functions to store or evaluate expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>Char(Expr(expression))</td>
<td>Converts an expression into a character string. The expression must be quoted with Expr; otherwise its evaluation is converted to a string.</td>
</tr>
<tr>
<td></td>
<td>Char(name)</td>
<td>Converts a number into its character representation. Width and decimal are optional arguments to specify formatting; the default is 18 for width and 99 for decimal.</td>
</tr>
<tr>
<td></td>
<td>string = char(number, width, decimal)</td>
<td></td>
</tr>
<tr>
<td>Eval</td>
<td>Eval(x)</td>
<td>Evaluates x, and then evaluates the result of x (unquoting).</td>
</tr>
<tr>
<td>Eval Expr</td>
<td>Eval Expr(x)</td>
<td>Returns an expression with all the expressions inside x evaluated.</td>
</tr>
<tr>
<td>Eval List</td>
<td>Eval List(lst)</td>
<td>Returns a list of the evaluated expressions inside list.</td>
</tr>
<tr>
<td>Expr</td>
<td>Expr(x)</td>
<td>Returns the argument unevaluated (expression-quoting).</td>
</tr>
<tr>
<td>NameExpr</td>
<td>NameExpr(x)</td>
<td>Returns the unevaluated expression of x rather than the evaluation of x. NameExpr is like Expr except that if x is a name, NameExpr returns the unevaluated expression stored in the name rather than the unevaluated name x.</td>
</tr>
<tr>
<td>Num</td>
<td>Num(&quot;string&quot;)</td>
<td>Converts a character string into a number.</td>
</tr>
<tr>
<td>Parse</td>
<td>Parse(&quot;string&quot;)</td>
<td>Converts a character string into a JSL expression.</td>
</tr>
</tbody>
</table>

Summary

Table 11.7 compares various ways that you can use the evaluation-control operators with x. Assume that x and i have been assigned as before:

```
x = expr( distribution(column( expr("X"||char(i)) )) );
```
Table 11.7 Compare all the operators for controlling evaluation

<table>
<thead>
<tr>
<th>Commands and results</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x; // or Eval(x);</td>
<td>Eval(x) and simply calling x are equivalent.</td>
</tr>
<tr>
<td>Not Found in access or evaluation of 'distribution', Bad Argument( {&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>Expr(x); x</td>
<td>Returns the expression x (packs an additional layer).</td>
</tr>
<tr>
<td>Name Expr(x); Distribution(Column(Expr(&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>y=Eval Expr(x); Distribution(Column(&quot;X3&quot;))</td>
<td>Evaluates the inner expression but leaves the outer expression unevaluated, so that y is Distribution(Column(&quot;X3&quot;)).</td>
</tr>
<tr>
<td>y; //or Eval(Eval Expr(x)); Distribution[]</td>
<td>Eval(eval expr(x)) and simply calling y are equivalent.</td>
</tr>
<tr>
<td>z = Char(nameexpr(x)); &quot;Distribution(Column(Expr(!&quot;X!&quot;</td>
<td></td>
</tr>
<tr>
<td>Parse(z); Distribution(Column(Expr(&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unquotes the text string and returns an expression.</td>
</tr>
</tbody>
</table>
Macros

Stored expressions can serve as a macro feature. You can store a generalized action as an expression in a global, and then call that global wherever you need that action to be performed. This example has four macros as the arguments to If:

```r
lastStdzdThickness=expr(
    (thickness[nrow()] - col mean(thickness)) / col std dev(thickness));
continue=expr(...<script to read in more data>...);
log=expr(print("In control at "||char(long date(today()))));
break=expr(...<script to shut down process>...);
limitvalue=1;
if(lastStdzdThickness<limitvalue,log;continue,break);
```

Storing the expression (the script itself, not its evaluation at the moment) with `Expr` delays its evaluation until the global is actually called. Any variables, data points, or expressions included in that expression are evaluated on the fly when the expression is evaluated. See “Stored expressions,” p. 406, for detailed rules for storing expressions and later evaluating them.

Manipulating lists

The following operators manipulate lists. They can also be used to manipulate expressions, as shown in the next section, “Manipulating expressions,” p. 416. A summary of commands with explanations is in Table 11.8 “Functions for manipulating lists or expressions,” p. 418.

Most of the function have two variants, one that produces a new value, and one that works in-place directly on its arguments. Here are some example pairs:

```r
A = Remove(A,3); // delete the third item in the list A, storing result in A
Remove From(A,3); // delete the third item in the list A, in place
```
onetwo=Insert({1},2); // onetwo is {1,2}
InsertInto(A,{1,2},4); // puts 1,2 before the current 4th item

Note: If position is omitted in the Insert Into command, items are placed at the end of the list.

a=Shift({1,2,3,4},1); // stores the list {2,3,4,1} in a
Shift Into(a,-1); // a is now {1,2,3,4}

b=Reverse(a); // b is now {4,3,2,1}
Reverse Into(a); // a is now {4,3,2,1}, also

s=Sort List({1,4,2,5,-7.2,pi(),-11,cat, apple, cake});
// s is now sorted list
c={5,pie,2,pi(),-2};
Sort List Into(c); // c is now {-2,2,5,Pi(),pie}

In-place operators

In-place operators are those that operate on lists or expressions directly. They have From or Into in their names (for example, Remove From and Insert Into). They do not return a result; you have to show the list to see the result. The first argument for an in-place operator must be an L-value. An L-value is an entity such as a global variable whose value can be set.

myList={a, b, c, d};
Insert Into(myList,2,3);
show(myList);
myList = {a, b, 2, c, d}

These examples show how to use Insert Into and Remove From with nested lists:

a = {{1, 2, 3}, {"A", "B", "C"}};
Show( a );
   a = {{1, 2, 3}, {"A", "B", "C"}}

Insert Into( a[1], 99, 1 );
Show( a );
   a = {{99, 1, 2, 3}, {"A", "B", "C"}}

Remove From( a[1], 1 );
Show( a );
   a = {{1, 2, 3}, {"A", "B", "C"}}

Not in-place operators

For the not-in-place operators, you must either state the list directly or else quote a name that evaluates to a list. Such operators do not have From or Into in their names. They return manipulated lists or expressions without changing the original list or expression given in the first argument.

myNewList=Insert({a, b, c, d}, 2, 3);
   {a, b, 2, c, d}
oldList={a,b,c,d};
newList=Insert(oldList, 2, 3);
        {a, b, 2, c, d}

Substituting

Substitute and Substitute Into merit further discussion. Both find all matches to a pattern in a list (or expression) and replace them with another expression. Each pattern must be a name. The arguments are evaluated before they are applied, so most of the time you must quote them with an Expr function.

Substitute({a,b,c}, expr(a), 23); // produces {23,b,c}
Substitute(expr(sine(x)),expr(x),expr(y)); // produces sine(y)

To delay evaluating an argument, use NameExpr instead of Expr:

a={quick,brown,fox,jumped,over,lazy,dogs};
b=Substitute(a,Expr(dogs),Expr(cat));
canine=Expr(dogs);equine=Expr(horse);
c=Substitute(a,NameExpr(canine),NameExpr(equine)); show(a,b,c);
    a = {quick,brown,fox,jumped,over,lazy,dogs}
    b = {quick,brown,fox,jumped,over,lazy,cat}
    c = {quick,brown,fox,jumped,over,lazy,horse}

Substitute Into does the same work, in place:

Substitute Into(a,Expr(dogs),Expr(horse));

You can list multiple pattern and replacement arguments to do more than one replacement in a single step:

d=Substitute(a,
    NameExpr(quick),NameExpr(fast),
    NameExpr(brown),NameExpr(black),
    NameExpr(fox),NameExpr(wolf)
);
    {fast,black,wolf,jumped,over,lazy,dogs}

Note that substitutions are done repeatedly over multiple instances of the expression pattern. For example:

Substitute(Expr(a+a), Expr(a), Expr(aaa));

results in:

    aaa + aaa

Manipulating expressions

The operators for manipulating lists can also operate on most expressions. Be sure to quote the expression with Expr. For example:

Remove(Expr(A+B+C+D),2); // results in the expression A+C+D
b=Substitute(Expr(log(2)^2/2), 2, 3); // results in the expression Log(3)^3/3

As with lists, remember that the first argument for in-place operators must be an L-value. An L-value is an entity such as a global variable whose value can be set. In-place operators are those that operate on lists or
expressions directly. They have From or Into in their names (for example, Remove From and Insert Into). They do not return a result; you have to show the expression to see the result.

\[
polynomial = \text{expr}(a \cdot x^2 + b \cdot x + c);
\]
\[
\text{insertinto}(\text{polynomial}, \text{expr}(d \cdot x^3), 1);
\]
\[
\text{show}(\text{polynomial});
\]
\[
\text{polynomial} = d \cdot x^3 + a \cdot x^2 + b \cdot x + c
\]

For the not-in-place operators, you must either state the expression directly or else quote a name that evaluates to an expression using \text{NameExpr}. Such operators do not have From or Into in their names. They return manipulated lists or expressions without changing the original list or expression given in the first argument.

\[
cubic = \text{insert}(\text{expr}(a \cdot x^2 + b \cdot x + c), \text{expr}(d \cdot x^3), 1);
\]
\[
d \cdot x^3 + a \cdot x^2 + b \cdot x + c
\]
\[
quadratic = \text{expr}(a \cdot x^2 + b \cdot x + c);
\]
\[
cubic = \text{insert}(\text{NameExpr}(\text{quadratic}), \text{expr}(d \cdot x^3), 1);
\]
\[
d \cdot x^3 + a \cdot x^2 + b \cdot x + c
\]

Substituting

Substituting is extremely powerful; please review the earlier discussion “Substituting,” p. 416. Here are a few notes regarding substituting for expressions.

\text{Substitute(pattern, name, replacement)} substitutes for names in expressions

\text{NameExpr} looks through the name but copies instead of evaluates:

a = \text{expr}(\text{distribution}(\text{column}(x), \text{normal quantile plot})); \text{show}(\text{NameExpr}(a));
\text{NameExpr}(a) = \text{Distribution}((\text{Column}(x), \text{normal quantile plot})

\text{Substitute} evaluates all its arguments, so they must be quoted correctly:

b = \text{substitute}(\text{NameExpr}(a), \text{expr}(x), \text{expr}(weight)); \text{show}(\text{NameExpr}(b));
\text{NameExpr}(b) = \text{Distribution}((\text{Column}(weight), \text{normal quantile plot})

\text{SubstituteInto} needs an L-value, so the first argument is not quoted:

\text{SubstituteInto}(a, \text{expr}(x), \text{expr}(weight)); \text{show}(\text{NameExpr}(a));
\text{NameExpr}(a) = \text{Distribution}((\text{Column}(weight), \text{normal quantile plot})

\text{Substitute} is useful for changing parts of expressions, such as in the following example that tests the Is functions:

data = \{1, \{1,2,3\}, \text{[1 2 3]}, "abc", x, x(y)};
ops = \{\text{is number}, \text{is list}, \text{is matrix}, \text{is string}, \text{is name}, \text{is expr}\};
m = (\text{n items(data)}, \text{n items(ops)}, 0);
test = \text{expr}(m[r,c] = _\text{op}(\text{data[r]}));
for (r=1, r<=\text{n items(data)}, r++,
    for (c=1, c<=\text{n items(ops)}, c++,
        \text{eval}(\text{substitute}(\text{NameExpr}(\text{test}), \text{expr(_\text{op}, \text{ops}[c])))});
\text{show}(m);
\]
\[
m = [1 0 0 0 0 0,
\]
You can use SubstituteInto to have JMP solve quadratic equations. The following example solves $4x^2 - 9 = 0$:

```plaintext
/* FIND THE ROOTS FOR THE EQUATION: */
/* a*x^2 + b*x + c = 0 */
// The quadratic formula is x=(b +- sqrt(b^2 - 4ac))/2a.
// Use a list to store both the + and - results of the +- operation
x={expr((-b + sqrt(b^2 - 4*a*c))/(2*a)),
   expr((-b - sqrt(b^2 - 4*a*c))/(2*a))};
// Next, plug in the coefficients:
substitute into(x,expr(a),4, expr(b),0, expr(c),-9);
show(x);           //see the result of substitution
show(eval(expr(x))); //see the solution
x = {Expr((-0+Sqrt(0^2-4*4*-9))/(2*4)),Expr((-0-Sqrt(0^2-4*4*-9))/(2*4))}
EvalExpr(x) = {1.5,-1.5}
```

The operators for manipulating lists and expressions are discussed in the previous section, “Manipulating lists,” p. 414, and summarized in Table 11.8.

**Table 11.8** Functions for manipulating lists or expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td>$x = \text{Remove(list</td>
<td>expr)}$</td>
</tr>
<tr>
<td>Remove From</td>
<td>Remove From(list</td>
<td>expr), position) Remove From(list</td>
</tr>
<tr>
<td>Insert</td>
<td>$x = \text{Insert(list</td>
<td>expr, item, position)}$ $x = \text{Insert(list</td>
</tr>
<tr>
<td>Insert Into</td>
<td>Insert Into(list</td>
<td>expr, item, position) Insert Into(list</td>
</tr>
</tbody>
</table>
Table 11.8 Functions for manipulating lists or expressions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>( x = \text{Shift}(\text{list}\mid\text{expr}) ) [= \text{Shift}(\text{list}\mid\text{expr}, n) ]</td>
<td>Shift an item or ( n ) items from the front to the back of the list or expression. Shift items from back to front if ( n ) is negative.</td>
</tr>
<tr>
<td>Shift Into</td>
<td>( \text{Shift Into}(\text{list}\mid\text{expr}) ) [= \text{Shift Into}(\text{list}\mid\text{expr}, n) ]</td>
<td>Shift items in place.</td>
</tr>
<tr>
<td>Reverse</td>
<td>( x = \text{Reverse}(\text{list}\mid\text{expr}) )</td>
<td>Reverse the order of elements of a list or terms of an expression.</td>
</tr>
<tr>
<td>Reverse Into</td>
<td>( \text{Reverse Into}(\text{list}\mid\text{expr}) )</td>
<td>Reverse the order of elements of a list or terms of an expression in place.</td>
</tr>
<tr>
<td>Sort List</td>
<td>( x = \text{Sort List}(\text{list}\mid\text{expr}) )</td>
<td>Sort the elements of a list or the terms of an expression. Numbers sort low, followed by the name value of names, strings, or operators. For example ( 1+2 ) is lower than ( 1-2 ) because the name value Add sorts lower than the name value Subtract. ( [1,2] ) sorts lower than ( [1,3] ), which sorts lower than ( [1,3,0] ). ( [1000] ) sorts lower than ( [&quot;a&quot;] ), but ( [a] ) and ( [&quot;a&quot;] ) sort as equal.</td>
</tr>
<tr>
<td>Sort List Into</td>
<td>( \text{Sort List Into}(\text{list}\mid\text{expr}) )</td>
<td>Sort the elements of a list or terms of an expression in place.</td>
</tr>
<tr>
<td>Sort Ascending</td>
<td>( \text{Sort Ascending}(\text{list}\mid\text{matrix}) )</td>
<td>Returns a copy of a list or matrix with the items in ascending order.</td>
</tr>
<tr>
<td>Sort Descending</td>
<td>( \text{Sort Descending}(\text{list}\mid\text{matrix}) )</td>
<td>Returns a copy of a list or matrix with the items in descending order.</td>
</tr>
<tr>
<td>Loc Sorted</td>
<td>( \text{Loc Sorted}(A, B) )</td>
<td>Creates a matrix of subscript positions where the values in matrix ( A ) match the values in matrix ( B ). ( A ) must be a matrix sorted in ascending order.</td>
</tr>
<tr>
<td>Substitute</td>
<td>( R = \text{Substitute}(\text{list}\mid\text{expr}, \text{Expr}(\text{pattern}), \text{Expr}(\text{replacement}), ...) )</td>
<td>Finds all matches to the pattern in the list or expression, and replaces them with the replacement expression. Each pattern must be a name. The second and third arguments are evaluated before they are applied, so most of the time you must quote them with an \text{Expr} function. To delay evaluating an argument, use \text{Name Expr} instead of \text{Expr}. You can list multiple pattern-replacement pairs for multiple substitutions in one statement.</td>
</tr>
<tr>
<td>Substitute Into</td>
<td>( \text{Substitute Into}(\text{list}\mid\text{expr}, \text{Expr}(\text{pattern}), \text{Expr}(\text{replacement}), ...) )</td>
<td>Substitute in place.</td>
</tr>
</tbody>
</table>
Encryption

If you want to protect a JMP Scripting Language (JSL) file, you can encrypt it so only someone who knows the password can view it; you can also require a password to run it. This is useful in situations when you want to implement controlled sharing of a script.

To encrypt a script:
1. Open the script that you want to encrypt.
2. Select Edit > Encrypt Script.
3. Assign password(s) to encrypt the files:
   – To encrypt a script so that a user can run it without a password, but needs a password to view it, enter only a decrypt password.
   – To encrypt a script so that a user must enter one password to run it and another password to view it, enter both a run and a decrypt password.

Note: The password must consist of single-byte characters; using a text Input Method Editor (IME) does not work.

4. Click OK.
5. If you entered only a decrypt password, click Yes to confirm that you do not want to assign a run password.

The encrypted script opens in a new window. For example:

```plaintext
//e6.0.2
S0FTQ;VGMUTF?J<;LS;B<IRLXC=BV;@NS<TW;LR<ZFOP=JJS>NND@T<V><DZA>SU@MG;LR<ZFOP=JJS>NND@T<V><DZA>
```

6. Save the encrypted script.

To decrypt a JSL script:
1. Open the encrypted script in JMP.
2. Select Edit > Decrypt Script.
3. Enter the decrypt password and click OK.

The decrypted script opens in a new window.

To run an encrypted JSL script:

Note: You must know which data table the script runs on before running an encrypted script. If you do not know the name of the data table, you must decrypt the script before running it.
1. Open the encrypted script in JMP.
2. Select Edit > Run Script.
3. Enter the run password and click OK.

The script runs:
- If the script references a data table, you are prompted to open the data table, and then the script runs.
- If the script requires an empty data table, you must create the table and then run the encrypted script.

Note that entering the run password runs the script, but does not show the script: you must supply the decrypt password to actually view the script.

**Encryption and Global Variables**

Encryption alone does not hide global variables and their values. A Show Globals() command displays them normally. If you want to hide global variables in an encrypted script, you can give them special names.

Any global variable whose name begins with two underscore characters (_ _) is hidden, and Show Globals() displays neither its name nor its value. For example:

```
myvar = 2;
__myvar = 5;
Show Symbols();
//Globals
myvar = 2;
// 2 Globals (1 Hidden)
```

This strategy works whether your script is encrypted or not.

**Encrypting Scripts in Data Tables**

You can also encrypt a script that is saved to a data table.

1. Place the script to be encrypted in a script editor window.
   
   You cannot directly encrypt a script that is already saved to a data table.

2. In the script window, select Edit > Encrypt.
3. Select the entire encrypted script.
4. Select Edit > Copy.
5. Create a new data table script.
6. In the script portion of the window, enter this line:
   
   Include( Char to Blob( "" ) );
7. Place your cursor inside the quotation marks in the Char to Blob function and select Edit > Paste.
8. Click OK.
XML Parsing Operations

JSL has several commands available to parse XML.

Parse XML(string, On Element("tagname", Start Tag(expr), End Tag(expr)))
  parses an XML expression using the On Element expressions for specified XML tags.
value = xmlAttr("attr name")
  extracts the string value of an xml argument in the context of evaluating a Parse XML command.
value = xmlText()
  extracts the string text of the body of an XML tag in the context of evaluating a Parse XML command.

Example

The first part is an XML piece of an Excel spreadsheet that has been saved as XML. This is one row of the Big Class data. The sample XML can be saved as a valid XML document, which the JSL code correctly parses into a one-row data table.

```xml
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
  xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
  xmlns:html="http://www.w3.org/TR/REC-html40">
  <Worksheet ss:Name="Bigclass">
    <Table ss:ExpandedColumnCount="5" ss:ExpandedRowCount="41" x:FullColumns="1"
      x:FullRows="1">
    <Row>
```
The second part is the JSL code that reads the XML file and creates a JMP data table with the information in it.

```javascript
file path = "bigclasssexcel.xml";
file contents = load text file(file path);
Parse XML(file contents,
    OnElement("urn:schemas-microsoft-com:office:spreadsheet^Worksheet",
        StartTag(
            sheetname= xmlAttr("urn:schemas-microsoft-com:office:spreadsheet^Name", "Untitled");
            dt = new table(sheetname);
            row = 1;
            col = 1;
        ),
    ),
    OnElement("urn:schemas-microsoft-com:office:spreadsheet^Row",
        StartTag(
            if (row > 1,// assume that the first row contains column names
                dt << Add Rows(1));
        ),
        EndTag(
            row++;
            col = 1;
        ),
    ),
    OnElement("urn:schemas-microsoft-com:office:spreadsheet^Cell",
        EndTag(
            col++;
        ),
    ),
)
```

```xml
<Cell><Data ss:Type="String">name</Data></Cell>
<Cell><Data ss:Type="String">age</Data></Cell>
<Cell><Data ss:Type="String">sex</Data></Cell>
<Cell><Data ss:Type="String">height</Data></Cell>
<Cell><Data ss:Type="String">weight</Data></Cell>
</Row>
<Cell><Data ss:Type="String">KATIE</Data></Cell>
<Cell><Data ss:Type="Number">12</Data></Cell>
<Cell><Data ss:Type="String">F</Data></Cell>
<Cell><Data ss:Type="Number">59</Data></Cell>
<Cell><Data ss:Type="Number">95</Data></Cell>
</Row>
```
Pattern Matching and Regular Expressions

Pattern matching in JSL is a flexible method for searching and manipulating strings.

You define and use pattern variables just like any JMP variable:

```javascript
i = 3; // a numeric variable
a = "Ralph"; // a character variable
t = textbox("Madge"); // a display box variable
p = ( "this" | "that" ) + patSpan( " " ) + ( "car" | "bus" ); // a pattern variable
```

When the above statement executes, `p` is assigned a pattern value. The pattern value can be used either to construct another pattern or to perform a pattern match. The `patSpan` function returns a pattern that matches a span of characters specified in the argument; `patSpan("0123456789")` matches runs of digits.

```javascript
p2 = "Take " + p + "."; // using p to build another pattern
if( patMatch( "Take this bus.", p2 ), print("matches"), print("no match") ); // performing a match
```

Sometimes all you need to know is that the pattern matched the source text, as above. Other times, you might want to know what matched; for example, was it a bus or a car?

```javascript
p = ( "this" | "that" ) + patSpan( " " ) + ( "car" | "bus" ) >? vehicleType; // conditional assignment ONLY if pattern matches
if( patMatch( "Take this bus.", p ), show( vehicleType ), print( "no match" ) ); // do not use vehicleType in the ELSE because it is not set
```

You could pre-load `vehicleType` with a default value if you do not want to check the outcome of the match with an `if`. The `?>` conditional assignment operator has two arguments, the first being a pattern and the second a JSL variable. `?>` constructs a pattern that matches the pattern (first argument) and stores the result of the match in the JSL variable (second argument) after the pattern succeeds. Similarly, `>>` does not wait for the pattern to succeed. As soon (and as often) as the `>>` pattern matches, the assignment is performed.

```javascript
findDelimString = patLen(3)>>beginDelim + patArb()>?middlePart + expr(beginDelim);
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = PatMatch( testString, findDelimString, "<<<" || middlePart || ">>>" );
```
show(rc, beginDelim, middlePart, testString);

The above example shows a third argument in the `patMatch` function: the replacement string. In this case, the replacement is formed from a concatenation (|| operator) of three strings. One of the three strings, `middlePart`, was extracted from the `testString` by `?>` because the replacement cannot occur unless the pattern match succeeds (rc == 1).

Look at the pattern assigned to `findDelimString`. It is a concatenation of 3 patterns. The first is a `>>` operator that matches 3 characters and assigns them to `beginDelim`. The second is a `?>` operator that matches an arbitrary number of characters and, when the entire match succeeds, assigns them to `middlePart`. The last is an unevaluated expression, consisting of whatever string is in `beginDelim` at the time the pattern is executing, not at the time the pattern is built. Just like `expr()`, the evaluation of its argument is postponed. That makes the pattern hunt for two identical three letter delimiters of the middle part.

Other pattern functions might be faster and represent the problem that you are trying to solve better than writing a lot of alternatives; for example, "a"|"b"|"c" is the same as `patAny("abc")`. The equivalent example for `patNotAny("abc")` is much harder. Similar to `patSpan` (above), `patBreak("0123456789")` matches up to, but not including, the first number.

Here is a pattern that matches numbers with decimals and exponents and signs. It also matches some degenerate cases with no digits; look at the pattern assigned to `digits`.

```
digits = patSpan("0123456789") | "";

number = ( patAny("+-") | "" ) >? signPart +
   ( digits ) >? wholePart +
   ( "." + digits | "" ) >? fractionPart +
   ( patAny("eEdD") + ( patAny("+-") | "" ) + digits | "" ) >?
   exponentPart;
```

```
if( patMatch("-123.456e-78", number), show(signPart, wholePart, fractionPart, exponentPart) );
```

Sometimes data is in fixed fields. The `patTab`, `patRTab`, `patLen`, `patPos`, and `patRPos` functions make it easy to split out the fields in a fixed field string. `PatTab` and `patRTab` work from the left and right end of the string and take a number as their argument. They succeed by matching forward to the specified tab position. For example:

```
p = patPos(10) + patTab(15);
```

`PatPos(10)` matches the null string if it is in position 10. So at match time, the matcher works its way forward to position 10, then `patTab(15)` matches text from the current position (10) forward to position 15. This pattern is equivalent to `patPos(10)+patLen(5)`. Another example:

```
p = patPos(0) + patRTab(0);
```

This example matches the entire string, from 0 characters from the start to 0 characters from the end. the `patRem()` function takes no argument and is shorthand for `patRTab(0)`: it means the remainder of the string. Pattern matching can also be anchored to the beginning of the string like this:

```
patMatch("now is the time", patLen(15) + patRPos(0), NULL, ANCHOR );
```
The above pattern uses NULL rather than a replacement value, and ANCHOR as an option. Both are uppercase, as shown. NULL means that no replacement is done. ANCHOR means that the match is anchored to the beginning of the string. The default value is UNANCHORED.

Patterns can be built up like this, but this is not recursive:

```
   p = "a" | "b"; // matches one character
   p = p + p; // two characters
   p = p + p; // four characters
   patMatch("babb", patPos(0) + p + patRPos(0));
```

A recursive pattern refers to its current definition using expr():

```
   p = "<" + expr(p) + "*" + expr(p) + ">" | "x";
   patMatch("<x*x*x>x*>", patPos(0) + p + patRPos(0));
```

Remember, expr() is the procrastination operator; when the pattern is assigned to the variable p, expr() delays evaluating its argument (p) until later. In the next statement, patMatch performs the pattern match operation, and each time it encounters expr(), it looks for the current value of the argument. In this example, the value does not change during the match). So, if p is defined in terms of itself, how can this possibly work?

p consists of two alternatives. The right hand choice is easy: a single letter x. The left side is harder: <p*>. Each p could be a single letter x, since that is one of the choices p could match, or it could be <p*>. The last few example have used patPos(0) + ... + patRPos(0) to make sure the pattern matches the entire source text. Sometimes this is what you want, and sometimes you would rather the pattern match a subtext. If you are experimenting with these examples by changing the source text, you probably want to match the entire string to easily tell what was matched. The result from patMatch is 0 or 1.

This example uses “Left” recursion:

```
   x = expr(x) + "a" | "b"; // + binds tighter than |
```

If the pattern is used in FULLSCAN mode, it eventually uses up all memory as it expands. By default, the patMatch function does not use FULLSCAN, and makes some assumptions that allow the recursion to stop and the match to succeed. The pattern matches either a “b”, or anything the pattern matches followed by an “a”.

```
   rc = patMatch("baaaaa", x);
```

**Patterns and Case**

Unlike regular expressions, pattern matching is case insensitive. To force case sensitivity, you can add the named argument MATCHCASE to either Pat Match or Regex Match. For example:

```
   string = "abcABC";
   result = Regex Match( string, Pat Regex( "[aBc]+" ) );
   Show( string, result );
   string = "abcABC"
   result = {"abcABC"}
```

result = Regex Match( string, Pat Regex( "[cba]+" ), NULL, MATCHCASE );
Regular Expressions

You can also use standard regular expressions in JMP using the `Regex()` function:

```
Regex(source, regular expression, <format>, <IGNORECASE>);
```

Source is the string that you want to apply the regular expression to. Both arguments must be quoted strings or references to quoted strings.

An example:

```
Regex(
    "Are you there, Alice?, asked Jerry.", // source
    " (here|there) (\w+).+(said|asked) (\w+).\" ");
```

"there, Alice?, asked Jerry."

Since no format was specified, all the text that matched the regular expression is returned. Use the optional format to create a new string using your source and the regular expression:

```
Regex(
    "Are you there, Alice?, asked Jerry.",
    " (here|there) (\w+).+(said|asked) (\w+).\" ,
    " I am \1, \4, replied \2." );
```

"I am there, Jerry, replied Alice."

Format defaults to \0, which is the entire match. \n specifies the nth match made by the regular expression. If the regular expression does not match anything in the source, `Regex` returns numeric missing.

Regular expressions are case sensitive. To perform a case-insensitive match, use the optional switch `IGNORECASE`.

**Tip:** For details about forming regular expressions, use a regular expressions reference.

Projects

JMP projects are fully scriptable. The following script creates a new project, adds groups and files to it, and retrieves the project's name:

```
expj = New Project( "My Project" );
exg1 = expj << Add Group( "Data" );
exg2 = expj << Add Group( "Reports" );
exdt = Open( "$SAMPLE_DATA/Big Class.jmp" );
exg1 << Add Window( exdt );
exrp = Bivariate( X( height ), Y( weight ) );
exg2 << Add Window( exrp );
```
Close( exdt, NoSave );
expj << getname();

To open a project that already has been saved, use the Open Project function. For example,

```julia
prj = Open Project("filepath");
```

Scripting for JMP Projects has been expanded in JMP 9. The changes from JMP 8 to JMP 9 are summarized in Table 11.9.

**Table 11.9** Changes in Project Scripting

<table>
<thead>
<tr>
<th>Item</th>
<th>JMP 8 Behavior</th>
<th>JMP 9 Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Messages</td>
<td>Any project item is returned.</td>
<td>A more appropriate name is shown.</td>
</tr>
<tr>
<td></td>
<td>Example</td>
<td>Example</td>
</tr>
<tr>
<td></td>
<td><code>g1 = p &lt;&lt; Find(&quot;Group1&quot;);</code></td>
<td><code>g1 = p &lt;&lt; Find(&quot;Group1&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>Project:Group1</code></td>
<td><code>Project Group:Group1</code></td>
</tr>
<tr>
<td>Properties for Group</td>
<td>The <code>&lt;&lt; Properties</code> message did not return any value when sent to a</td>
<td>The group item now returns item properties like other project items.</td>
</tr>
<tr>
<td>items</td>
<td>Group item.</td>
<td></td>
</tr>
<tr>
<td>Navigation functions</td>
<td>Project navigation messages (for example, <code>&lt;&lt; GetFirstChild</code>, `&lt;&lt;</td>
<td>Project navigation messages return empty, which can be tested for with</td>
</tr>
<tr>
<td>return empty</td>
<td>GetNextSibling, <code>&lt;&lt; GetPreviousSibling</code>, and <code>&lt;&lt; GetParent</code>) display an error in the</td>
<td><code>IsEmpty()</code>).</td>
</tr>
<tr>
<td></td>
<td>log if no more items are available.</td>
<td>This eliminates the need for a <code>Try()</code> statement when navigating through the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>project tree in JSL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Existing scripts that use <code>Try()</code> should still work.</td>
</tr>
<tr>
<td>New <code>&lt;&lt; Open</code> message</td>
<td>To open project items, specific messages (for example, <code>&lt;&lt; Open Window</code>) are</td>
<td>The specific messages still work. A new and more general <code>&lt;&lt; Open</code> message</td>
</tr>
<tr>
<td></td>
<td>required.</td>
<td>available, to which items respond appropriately. For example, a window</td>
</tr>
<tr>
<td></td>
<td></td>
<td>opens just as if <code>&lt;&lt; Open Window</code> were sent.</td>
</tr>
</tbody>
</table>
### Table 11.9 Changes in Project Scripting (Continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>JMP 8 Behavior</th>
<th>JMP 9 Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output for the &lt;&lt; Properties message</td>
<td>The output of the &lt;&lt; Properties message is a single, large string.</td>
<td>An associative array of properties is returned allowing direct access to the desired property value.</td>
</tr>
<tr>
<td>Example</td>
<td>ProjectName &lt;&lt; Properties;</td>
<td>Example</td>
</tr>
<tr>
<td>SAS Stored Processes</td>
<td>There is no JSL method of adding a stored process.</td>
<td>You can now add stored processes to projects.</td>
</tr>
<tr>
<td></td>
<td>p &lt;&lt; Add Stored Process(&lt;metadataPath&gt;);</td>
<td>p &lt;&lt; Add Stored Process(&lt;metadataPath&gt;);</td>
</tr>
<tr>
<td>&lt;&lt; Properties message for documents</td>
<td>The Last Accessed Date is returned.</td>
<td>The Last Accessed Date is no longer returned.</td>
</tr>
<tr>
<td>Invisible Option</td>
<td>All opened or created projects are visible.</td>
<td>Open Project() and New Project() accept an optional second parameter (invisible). Invisible projects are not shown in the Project Window. Note: If an invisible project is open when JMP is closed, the user is not prompted to save the project. The script writer must ensure that invisible projects are saved and closed appropriately.</td>
</tr>
</tbody>
</table>
Hexadecimal and BLOB Functions

JMP can also handle binary (large) objects, commonly called BLOBs. The functions below convert between hexadecimal values, numbers, characters, and BLOBs. Some of the functions are covered in more detail following Table 11.10.

These functions are listed in the Syntax Reference for JSL Syntax Reference.

Table 11.10 Hexadecimal and BLOB Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex(&quot;text&quot;)</td>
<td>Returns the hexadecimal codes for the characters in text, number, or blob.</td>
</tr>
<tr>
<td>Hex(num)</td>
<td>Char To Hex is an alias.</td>
</tr>
<tr>
<td>Hex(blob)</td>
<td></td>
</tr>
<tr>
<td>Hex To Blob(&quot;hexstring&quot;)</td>
<td>Returns a BLOB representation of the hexadecimal code supplied as a quoted string.</td>
</tr>
<tr>
<td>Hex To Char(&quot;hexstring&quot;, encoding)</td>
<td>Returns a character string that corresponds to the hexadecimal code supplied as a quoted string.</td>
</tr>
</tbody>
</table>

The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.
Hexadecimal and BLOB Functions

Table 11.10 Hexadecimal and BLOB Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex to Number</td>
<td>Returns the number that corresponds to the hexadecimal code supplied as a quoted string.</td>
</tr>
<tr>
<td>Char To Blob(string)</td>
<td>Converts a string of characters into a binary (blob). The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.</td>
</tr>
<tr>
<td>Char To Blob(string,</td>
<td></td>
</tr>
<tr>
<td>encoding)</td>
<td></td>
</tr>
<tr>
<td>Blob To Char(blob)</td>
<td>Converts binary data to a Unicode string. The default encoding supported for the hex code is UTF-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.</td>
</tr>
<tr>
<td>Blob To Char(blob,</td>
<td></td>
</tr>
<tr>
<td>encoding)</td>
<td></td>
</tr>
<tr>
<td>Blob Peek(blob, offset,</td>
<td>Returns a new BLOB that is a subset of the given BLOB that is length bytes long and begins at the offset. Note that the offset is 0-based.</td>
</tr>
<tr>
<td>length)</td>
<td></td>
</tr>
</tbody>
</table>

hex (string) returns the hexadecimal codes for each character in the argument. For example,

```
Hex("Abc")
```

returns

```
"416263"
```

since 41, 62, and 63 are the hexadecimal codes (in Ascii) for “A”, “b”, and “c”.

Hex to Char (string) converts hexadecimal to characters. The resulting character string might not be valid display characters. All the characters must be in pairs, in the ranges 0-9, A-Z, and a-z. Blanks and commas are allowed, and skipped. For example,

```
Hex To Char ("4142")
```

returns

```
"AB"
```

since 41 and 42 are the hexadecimal equivalents of “A” and “B”.

Hex and Hex To Char are inverses of each other, so

```
Hex To Char ( Hex("Abc") )
```

returns

```
"Abc"
```

Hex To Blob(string) takes a string of hexadecimal codes and converts it to a binary object.

```
a = Hex To Blob("6A6B6C"); Show(a);
a = Char To Blob("jk1")
```
Chapter 11

Web

Blob Peek(blob, offset, length) extracts bytes as defined by the parameters from a blob.

\[
\begin{align*}
b &= \text{Blob Peek}(a, 1, 2); \text{ Show}(b); \\
&= \text{Char To Blob}("k1") \\
b &= \text{Blob Peek}(a, 0, 2); \text{ Show}(b); \\
&= \text{Char To Blob}("jk") \\
b &= \text{Blob Peek}(a, 2); \text{ Show}(b); \\
&= \text{Char To Blob}("l")
\end{align*}
\]

Hex(blob) converts a blob into hexadecimal.

\[
\begin{align*}
c &= \text{Hex}(a); \text{ Show}(c); \\
&= "6A6B6C" \\
d &= \text{Hex To Char}(c); \text{ Show}(d); \\
&= "jk1"
\end{align*}
\]

Concat(blob1, blob2) or blob1 || blob2 concatenates two blobs.

\[
\begin{align*}
e &= \text{Hex To Blob}("6D6E6F"); \text{ Show}(e); \\
f &= a||e; \text{ Show}(f); \\
&= \text{Char To Blob}("mno") \\
f &= \text{Char To Blob}("jklmno")
\end{align*}
\]

length(blob) returns the number of bytes in a blob.

\[
\begin{align*}
g &= \text{length}(f); \text{ Show}(g); \\
&= 6
\end{align*}
\]

**Note:** When blobs are listed in the log, they are shown with the constructor function `Char To Blob("...").`

Any hex code outside the ascii range (space to }, or hex 20 - 7D) is encoded as the three-character sequence \(['~'][hexdigit][hexdigit]. For example,

\[
\begin{align*}
h &= \text{Hex To Blob}("19207D7E"); \text{ Show}(h); \\
i &= \text{Hex}(h); \text{ Show}(i); \\
&= \text{Char To Blob}("~19 ~7E") \\
i &= "19207D7E"
\end{align*}
\]

Char To Blob(string) creates a blob from a string, converting ~hex codes. Blob To Char(blob) creates a string with ~hex codes to indicate non-visible and non-ascii codes.

---

Web

Web takes a string as its argument, where the string is a URL. This URL is opened with the default web browser of your system. The URL can be stored as a string variable or quoted directly. For example,

\[
\begin{align*}
\text{url} &= "www.jmp.com"; \\
\text{Web}(\text{url}); \\
\text{does the same thing as} \\
\text{Web}("www.jmp.com")
\end{align*}
\]
Note: On the Windows version of JMP, you have the option of opening the URL in a JMP window instead of your default web browser. To open the url in a JMP window, add JMP Window to the web command:

   Web("www.jmp.com", JMP Window);

### Scheduling Actions

A Schedule function lets you set up a script to be executed some number of seconds later.

   schedule(15, print("hello"));

---

**Figure 11.4 JMP Scheduler**

A Scheduler window shows the time until the next event and has buttons for restarting (Go) or stopping (Stop) the schedule. Its pop-up menu has a command Show Schedule, which echoes the current schedule to the log. For example, if you checked the schedule several times during the “hello” example, you would see something like this:

   Scheduled at 11.55000000000018 :Print("hello")
   Scheduled at 4.716666666666697 :Print("hello")
   Scheduled at 3.083333333333485 :Print("hello")

The script might also be a name referring to a stored expression. For example, try submitting this script, which calls itself:

   quickieScript = expr( show("Hi there"); schedule(15, quickieScript); );
   quickieScript;

This script should show the string “Hi there” in a log window after 15 seconds, and reschedule itself for another 15 seconds, continuing until the Stop button is clicked.

More typically, in a production setting you might want to set up a schedule like this:

   FifteenMinuteCheck = expr(show("Checking data");
   open("my file", options...);
   distribution(column(column1), capability(spec limits));
   schedule(15*60, FifteenMinuteCheck);
   FifteenMinuteCheck;

Schedule initiates an event queue, but once it has the event queued, JMP proceeds with the next statement in the script. For example, the following has results that might surprise you:

   schedule(3, print("one"));
   print("two");
   "two"
   "one"
If you want the script to wait until the scheduled events are finished before proceeding, one solution would be to use `Wait()` with a suitable pause. Another is to embed the subsequent actions into the schedule queue. `Schedule` accepts a series of arguments to queue many events in sequence. Each event is a separate call to the schedule. Each event time is an absolute time relative to “now” (or the instant that Go is clicked). Therefore, the following sequence finishes in five seconds, not in twelve:

```javascript
schedule(3, print("hello"));
schedule(4, print("", world"));
schedule(5, print("--bye"));
```

To cancel all events in a schedule queue, use `Clear Schedule`.

```javascript
scheduler[1] << Clear Schedule();
```

**Note:** It is not possible to create multiple threads using `Schedule`.

### Table 11.11 Schedule commands

<table>
<thead>
<tr>
<th>Message</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>sc=Schedule(n, script)</td>
<td>Queues an event to run the script after n seconds.</td>
</tr>
<tr>
<td>Clear Schedule</td>
<td>sc&lt;&lt; Clear Schedule()</td>
<td>Cancels all events in a schedule queue.</td>
</tr>
</tbody>
</table>

### Additional Numeric Operators

JSL also offers several categories of operations that do not make much sense in the context of the formula editor: matrix operations and numeric derivatives of functions. Algebraic derivatives are also available.

The basic arithmetic operators can also be used with matrix arguments for matrix-wise addition, subtraction, and so on. Matrices also have a few special operators for elementwise multiplication and division, concatenation, and indexing. See the chapter “Matrices,” p. 313, for details.

### Derivatives

JSL has three internal operators (not all available in the calculator) for taking derivatives.

`Derivative` takes the first derivative of an expression with respect to names you specify in the second argument. A single name might be entered as this second argument; or multiple values can be specified in a list, in other words, surrounded by braces.

**Note:** `Derivative` is also available as an editing command inside the formula editor (calculator), located on the drop-down list in the top center of the formula editor (above the keypad). To use it, highlight a single variable in the expression (to designate which variable the derivative should be taken with respect to), then select the `Derivative` command from the menu. The whole formula is replaced by its derivative with respect to the highlighted name.
In scripts, the easiest way to use the function is with a single name. In this example, we first show the mathematical notation and then the JSL equivalent.

For \( f(x) = x^3 \), the first derivative is \( f'(x) \) or \( \frac{d}{dx} x^3 = 3x^2 \).

\[
\text{result} = \text{derivative}(x^3, x); \text{show(result);} \\
\text{result} = 3 \times x \times 2
\]

If you want an efficient expression to take the derivative with respect to several variables, then the variables are specified in a list. The result is a list containing a threaded version of the original expression, followed by expressions for the derivatives. The expression is threaded by inserting assignments to temporary variables of expressions that are needed in several places for the derivatives.

Here is an example involving an expression involving three variables. Listing all three variables returns the first derivatives with respect to each. The result is a list with the original expression and then the derivatives in the order requested. However, note here that JMP creates a temporary variable `T#1` for storing the subexpression \( x^2 \), and then uses that subexpression subsequently to save calculations.

\[
\text{result2} = \text{derivative}(3 \times y \times x^2 + z^3, \{x, y, z\}); \text{show(result2);} \\
\{3 \times y \times (T#1 = x \times 2) + z \times 3, 6 \times x \times y, 3 \times T#1, 3 \times z \times 2\}
\]

To take second derivatives, specify the variable as a third argument. Both the second and third arguments must be lists. JMP returns a list with the original expression, the first derivative(s), and then the second derivative(s) in the order requested.

\[
\text{second} = \text{derivative}(3 \times y \times x^2, \{x\}, \{x\}, \{x\}); \text{show(second);} \\
\text{second} = \{3 \times y \times x \times 2, 6 \times x \times y, 6 \times y\}
\]

\[
\text{second} = \text{derivative}(3 \times y \times x^2, \{y\}, \{y\}); \text{show(second);} \\
\text{second} = \{3 \times y \times (T#1 = x \times 2), 3 \times T#1, 0\}
\]

\[
\text{second} = \text{derivative}(3 \times y \times x^2, \{y\}, \{x\}); \text{show(second);} \\
\text{second} = \{3 \times y \times (T#2 = x \times 2), 3 \times T#2, 6 \times x\}
\]

\text{NumDeriv} takes the first numeric derivative of an operator or function with respect to the value of the first argument by calculating the function at that value and at that value plus a small delta (\( \Delta \)) and dividing the difference by the delta. \text{NumDeriv2} computes the second numeric derivative in a similar fashion. These are used internally for nonlinear modeling but are not frequently useful in JSL. Note that these functions do not differentiate using a variable, but only with respect to arguments to a function. In order to differentiate with respect to \( x \), you have to make \( x \) one of the immediate arguments, not a symbol buried deep into the expression.

Suppose to differentiate \( y = 3x^2 \) at the value of \( x = 3 \). The incorrect way would be to submit

\[
x=3; \\
n=\text{NumDeriv}(3\times x^2);
\]

The correct way is to make \( x \) an argument in the function.

\[
x=3; \\
f=\text{function}(\{x\}, 3\times x^2); \\
n=\text{NumDeriv}(f(x), 1);
\]
Consider both the mathematical notation and the JSL equivalent for another example:

For \( f(x) = x^2 \), it calculates \( \frac{d}{dx} x^2 = \frac{(x+\Delta)^2 - x^2}{\Delta} \). At \( x_0 = 3 \), \( \frac{d}{dx} x^2 = 6.0001 \).

\[
x=3; \ y=numderiv(x^2); \quad // \text{or equivalently: } y = numDeriv(3^2);
\]

6.0000099999513

And here are a few more examples:

\[
x = numderiv(sqrt(7));
y = numDeriv(Normal Distribution(1));
z = Num deriv2(normal distribution(1));
\]

### Table 11.12 Derivative functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derivative</td>
<td><code>Derivative(expr, {name, ...})</code></td>
<td>Returns the derivative of the <code>expr</code> with respect to <code>name</code>. Note that the second argument can be specified in a list with braces <code>[]</code> or simply as a variable if there is only one. Give two lists of names to take second derivatives.</td>
</tr>
<tr>
<td>NumDeriv</td>
<td><code>NumDeriv(expr)</code></td>
<td>Returns the first numeric derivative of the <code>expr</code> with respect to the first argument in the expression.</td>
</tr>
<tr>
<td>NumDeriv2</td>
<td><code>NumDeriv2(expr)</code></td>
<td>Returns the second numeric derivative of the <code>expr</code> with respect to the first argument in the expression.</td>
</tr>
</tbody>
</table>

#### Algebraic Manipulations

JSL provides a way of algebraically unwinding an expression (essentially, solving for a variable). It is accomplished through the `Invert Expr()` function.

\[
\text{Invert Expr}(\text{expression, name, y})
\]

where

- **expression** is the expression to be inverted, or the name of a global containing the expression
- **name** is the name inside expression to unwind the expression around
- **y** is what the expression was originally equal to

For example,

\[
\text{Invert Expr}(\sqrt{\log(x)}, x, y)
\]

is wound around the name x (which should appear in the expression only once), and results in

\[
\exp(y^2)
\]

It is performed exactly as you would when doing the algebra by hand

\[
y = \sqrt{\log(x)}
\]
Invert Expr supports most basic operations that are invertible, and makes assumptions as necessary, such as assuming you are interested only in the positive roots, and that the trigonometric functions are in invertible areas so that the inverse functions are legal.

$F$, Beta, Chi-square, $t$, Gamma, and Weibull distributions are supported for the first arguments in their Distribution and Quantile functions. If it encounters an expression that it cannot convert, Invert Expr() returns Empty().

JSL provides a Simplify Expr command that takes a messy, complex formula and tries to simplify it using various algebraic rules. To use it, submit

```
result = Simplify Expr(expr(expression));
```

or

```
result = Simplify Expr(nameExpr(global));
```

For example,

```
Simplify Expr(expr(2*3*a+b*(a+3-c)-a*b));
```

results in

```
6*a + 3*b + -1*b*c
```

Simplify Expr also unwinds nested If expressions. For example:

```
r = simplifyExpr(expr(If(cond1,result1,if(cond2,result2,if(cond3,result3,resultElse)))))
```

results in

```
If(cond1, result1, cond2, result2, cond3, result3, resultElse);
```

Maximize and Minimize

The Maximize and Minimize functions find the factor values that optimize an expression. The expression is assumed to be a continuous function of the factor values.

The form of the call is

```
result = Maximize(objectiveExpression,{list of factor names}, <<option(value))
result = Minimize(objectiveExpression,{list of factor names}, <<option(value))
```

`objectiveExpression` is the expression whose value is to be optimized, and can either be the expression itself, or the name of a global containing a stored expression.

`{list of factor names}` is an expression yielding a list of names involved in `objectiveExpression`.

The name can be followed by limits that bound the permitted values, for example

```
name(LowerBound,upperBound)
```

If you want to limit the values on one side, make the other side a missing value, for example:
Advanced Concepts
Additional Numeric Operators

\{\text{beta}\} // unconstrained
\{\text{beta (0,1)}\} // constrained between 0 and 1
\{\text{beta (.1)}\} // upper limit of 1
\{\text{beta (0,.)}\} or \{\text{beta (0)}\} // lower limit of 0

Factor values can be either numbers or matrices.

Options available, shown with their default value, include:

<< tolerance(.00000001) // convergence criterion
<< maxIter( 250) // maximum number of iterations
<< limits() //
<< Method(NR|BFGS) // Specify either the Newton-Raphson method or the
Quasi-Newton method with BFGS update.

Initial values are assumed to be already supplied the factor values before calling the function.

These functions are not expected to find global optima for functions that have multiple local optima; they
are useful only for taking an initial value and moving it to either a local or global optimum.

The return value is currently the value of the objective function, if the optimization was successful, or
Empty() if not.

\textbf{Least Squares Example}

The following example uses \texttt{Minimize} to find the least squares estimates of this exponential model, with
data taken from the Nonlinear Example/US Population.jmp sample data table.

\begin{verbatim}
xx = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900,
50.155, 62.947, 75.994, 91.972, 105.71, 122.775, 131.669, 151.325, 179.323,
203.211, 226.5, 248.7];
b0 = 3.9;
b1 = .022;
ssExpr = Expr( Sum( (yy - (b0 * Exp( b1 * (xx - 1790) )))^2 ) );
ss = Minimize(ssExpr, {b0, b1}, <<tolerance(.00001));
Show(b0, b1, ss);
\end{verbatim}
Appendix A

JSL Syntax Reference
Summary of Operators, Functions, And Messages

This appendix gives abbreviated descriptions for all of JMP’s functions, operators, and general object messages. For platform messages, see the chapter, “Scripting Platforms,” p. 153.
Contents

Arithmetic Functions .......................................................... 441
Assignment Functions ....................................................... 443
Character Functions .......................................................... 447
Character Pattern Functions ............................................... 458
Comments ................................................................. 466
Comparison Functions ....................................................... 467
Conditional and Logical Functions ...................................... 472
Constants ................................................................. 477
Date and Time Functions .................................................... 477
Discrete Probability Functions ........................................... 483
Display ................................................................. 486
Financial Functions .......................................................... 497
Graphic Functions ........................................................... 502
Matrix Functions .............................................................. 514
Numeric Functions ............................................................ 528
Probability Functions ......................................................... 529
R Integration Functions ....................................................... 541
Random Functions ............................................................. 546
Row Functions ................................................................. 550
Row State Functions ............................................................. 551
SAS Integration Functions ................................................... 554
Statistical Functions ............................................................ 568
Transcendental Functions ..................................................... 572
Trigonometric Functions ....................................................... 575
Utility Functions ................................................................. 578
Platform Scripting Notes ..................................................... 623
Arithmetic Functions

Here are the six essential arithmetic operators and their function equivalents. You can add numbers, matrices, and variables that contain either. Each operator has two forms: an operator (for example, +) and a functional equivalent (for example, Add()).

---

**Add(a, b, ...)**

*a+b+...*

**Function**

Adds the values of the listed arguments. No arguments are changed.

**Returns**

The sum.

**Arguments**

For Add(), a comma-separated list of variables, numbers, or matrices.

For *a+b*, any number of variables, numbers, or matrices.

**Notes**

Any number of arguments is permitted. If no argument is specified, Add() returns 0.

**See Also**

The Matrix Algebra chapter in the Scripting Guide.

---

**Divide(a, b)**

*Divide(x)*

*a/b*

**Function**

Divides *a* by *b*. If only one argument is given (divide(*x*)), divides 1 by *x*.

**Returns**

The quotient of *a/b*; or the reciprocal of *x* (1/*x*) if only one argument is provided.

**Arguments**

*a*, *b*, *x* Can be a variable, number, or matrix.

**Notes**

If both arguments are matrices, it does matrix division.

**See Also**

The Matrix Algebra chapter in the Scripting Guide.
Minus(a)

- a

Function
Reverses the sign of a.

Returns
- a if a is positive (a=3; -a=-3; Minus(a)=-3).
a if a is negative (a=-3; -a=3; Minus(a)=3).
0 if a is 0 (a=0; -a=0; Minus(a)=0).
Missing if a is missing.

Argument
a Can be variable or a number. A variable must contain a number or a matrix.

Multiply(a, b, ...)

a*b*...

Function
Multiplies all values. No arguments are changed.

Returns
The product.

Arguments
Any number of variables, numbers, or matrices.

Notes
Any number of arguments is permitted. If no argument is specified, Multiply() returns 1.

See Also
The Matrix Algebra chapter in the Scripting Guide.

Power(a, <b>)

a^b

Function
Raises a to the power of b.

Returns
The product of a multiplied by itself b times.

Arguments
a Can be a variable, number, or matrix.
b Optional. Can be a variable or a number.

Notes
For Power(), the second argument (b) is optional, and the default value is 2. Power(a) returns a^2.
### Subtract(a, b)

```plaintext
a-b-...
```

**Function**
Subtracts the values of the listed arguments, left to right. No arguments are changed.

**Returns**
The difference.

**Arguments**
Two or more variables, numbers, or matrices.

**Notes**
Two or more arguments are permitted. Specifying fewer than two arguments produces an error.

**See Also**
The Matrix Algebra chapter in the *Scripting Guide*.

---

### Assignment Functions

JSL also provides operators for in-place arithmetic, or *assignment operators*. These operations are all done in place, meaning that the result of the operation is assigned to the first argument. The most basic assignment operator is the `=` operator (or the equivalent function `Assign`). For example, if `a` is 3 and you do `a+=4`, then `a` becomes 7.

The first argument to an assignment function must be something capable of being assigned (an *L-value*). You cannot do something like `3+=4`, because 3 is just a value and cannot be reassigned. You can, however, do something like `a+=4`, because `a` is a variable whose value you can set.

### Add To(a, b)

```plaintext
a+=b
```

**Function**
Adds `a` and `b` and places the sum into `a`.

**Returns**
The sum.

**Arguments**

- `a`  Must be a variable.
b  Can be a variable, number, or matrix.

Notes
The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument produces an error.

For \texttt{Add To()}: Only two arguments are permitted. If one or no argument is specified, \texttt{Add To()} returns a missing value. Any arguments after the first two are ignored.

For \texttt{a+=b}: More than two arguments can be strung together. JMP evaluates pairs from right to left, and each sum is placed in the left-hand variable. All arguments except the last must be a variable.

Example
\texttt{a+=b+=c}

JMP adds \(b\) and \(c\) and places the sum into \(b\). Then JMP adds \(a\) and \(b\) and places the sum into \(a\).

See Also
The Matrix Algebra chapter in the \textit{Scripting Guide}.

\begin{verbatim}
Assign(a, b)
\end{verbatim}

\texttt{a=b}

Function
Places the value of \(b\) into \(a\).

Returns
The new value of \(a\).

Arguments
\begin{itemize}
  \item \texttt{a}: Must be a variable.
  \item \texttt{b}: Can be a variable, number, or matrix.
\end{itemize}

Notes
\(a\) must be a variable, because it must be able to accept a value change. A number as the first argument produces an error. If \(b\) is some sort of expression, it's evaluated first and the result is placed into \(a\).

\begin{verbatim}
Divide To(a, b)
\end{verbatim}

\texttt{a/=b}

Function
Divides \(a\) by \(b\) and places the result into \(a\).

Returns
The quotient.

Arguments
\begin{itemize}
  \item \texttt{a}: Must be a variable.
  \item \texttt{b}: Can be a variable, number, or matrix.
\end{itemize}

See Also
The Matrix Algebra chapter in the \textit{Scripting Guide}.
Multiply To(a, b)

\[ a^*=b \]

**Function**
Multipies \( a \) and \( b \) and places the product into \( a \).

**Returns**
The product.

**Arguments**
- \( a \) Must be a variable.
- \( b \) Can be a variable, number, or matrix.

**Notes**
The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument produces an error.
For \( \text{Multiply To}() \): Only two arguments are permitted. If one or no argument is specified, \( \text{Multiply To}() \) returns a missing value. Any arguments after the first two are ignored.
For \( a^*=b \): More than two arguments can be strung together. JMP evaluates pairs from right to left, and each product is placed in the left-hand variable. All arguments except the last must be a variable.

**Example**
\[ a^*=b^*=c \]
JMP multiplies \( b \) and \( c \) and places the product into \( b \). Then JMP multiplies \( a \) and \( b \) and places the product into \( a \).

**See Also**
The Matrix Algebra chapter in the *Scripting Guide*.

PostDecrement(a)

\[ a-- \]

**Function**
Post-decrement. Subtracts 1 from \( a \) and places the difference into \( a \).

**Returns**
\( a-1 \)

**Argument**
- \( a \) Must be a variable.

**Notes**
If \( a-- \) or \( \text{Post Decrement}(a) \) is inside another expression, the expression is evaluated first, and then the decrement operation is performed. This expression is mostly used for loop control.

**See Also**
The Conditional Expressions and Loops section of the *Scripting Guide*. 
PostIncrement(a)

a++

Function
Post-increment. Adds 1 to a and places the sum into a.

Returns
a+1

Argument
a  Must be a variable.

Notes
If a++ or PostIncrement(a) is inside another expression, the expression is evaluated first, and then the increment operation is performed. Mostly used for loop control.

See Also
The Conditional Expressions and Loops section of the Scripting Guide.

SubtractTo(a, b)

a-=b

Function
Subtracts b from a and places the difference into a.

Returns
The difference.

Arguments
a  Must be a variable.
b  Can be a variable, number, or matrix.

Notes
The first argument must be a variable, because its value must be able to accept a value change. A number as the first argument produces an error.
For SubtractTo(): Only two arguments are permitted. If fewer than two or more than two arguments is specified, SubtractTo() returns a missing value.
For a-=b: More than two arguments can be strung together. JMP evaluates pairs from right to left, and each difference is placed in the left-hand variable. All arguments except the last must be a variable.

Example
a-=b-=c
JMP subtracts c from b and places the difference into b. Then JMP subtracts b from a and places the difference into a.

See Also
The Matrix Algebra chapter in the Scripting Guide.
Character Functions

Most character functions take character arguments and return character strings, although some take numeric arguments or return numeric data. Arguments that are literal character strings must be enclosed in quotation marks.

The basic character functions are briefly described below. Further details of some of these functions appear after the table.

Other related functions are discussed in “Hexadecimal and BLOB Functions,” p. 430 in the “Advanced Concepts” chapter. For more information about using patterns and regular expressions, see “Pattern Matching and Regular Expressions,” p. 424 in the “Advanced Concepts” chapter.

Blob To Char(blob, <encoding>)

Function
Reinterpret binary data as a Unicode string.

Returns
A string.

Arguments
blob a binary large object.
encoding Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.

Notes
The optional argument ascii is intended to make conversions of blobs containing normal ASCII data simpler when the data might contain CR, LF, or TAB characters (for example) and those characters do not need any special attention.

Blob To Matrix(blob, "type", bytes, "endian", <nCols>)

Function
Creates a matrix by converting each byte in the blob to numbers.

Returns
A matrix that represents the blob.

Arguments
blob A blob or reference to a blob.
"type" A quoted string that contains the named type of number. The options are int, uint, or float.
bytes Byte size of the data in the blob. Options are 1, 2, 4, or 8.
"endian" A quoted string that contains a named type that indicates whether the first byte is the most significant. Options are as follows:
- big indicates that the first byte is the most significant.
- little indicates that the first byte is the least significant.
native indicates that the machine’s native format should be used.

nCols Optional. Specify the number of columns in the matrix. The default value is 1.

\texttt{Char(x, <width>, <decimal>)}

\textbf{Function}

Converts an expression or numeric value into a character string.

\textbf{Returns}

A string.

\textbf{Arguments}

\begin{itemize}
  \item \texttt{x} an expression or a numeric value. An expression must be quoted with \texttt{Expr()}.
  \item \texttt{width} optional number that sets the maximum number of characters in the string.
  \item \texttt{decimal} optional number that sets the maximum number of places after the decimal that is included in the string.
\end{itemize}

\textbf{Note}

The \texttt{width} argument overrides the \texttt{decimal} argument.

\textbf{Example}

\begin{verbatim}
char(pi(), 10, 4)  \hspace{1cm} "3.1416"
char(pi(), 3, 4)  \hspace{1cm} "3.1"
\end{verbatim}

\texttt{Char To Blob(string, <encoding>)}

\textbf{Function}

Converts a string of characters into a binary (blob).

\textbf{Returns}

A binary object.

\textbf{Arguments}

\begin{itemize}
  \item \texttt{string} a quoted string or a reference to a string.
  \item \texttt{encoding} Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.
\end{itemize}
Char To Hex(value, <"integer" | encoding>)

Hex(value, <"integer" | encoding>)

Function
Converts the given value and encoding into its hexadecimal representation.

Returns
A hexadecimal string.

Arguments
value  Any number, quoted string, or blob.
integer Switch that causes the value to be interpreted as an integer.
encoding Optional quoted string that specifies an encoding. Supported encodings are: utf-8, utf-16le, utf-16be, us-ascii, iso-8859, and ascii-hex.

Concat(a, b)

Concat(A, B)

a||b
A||B

Function
For strings: Appends the string b to the string a. Neither argument is changed.
For matrices: Horizontal concatenation of two matrices, A and B.

Returns
For strings: A string composed of the string a directly followed by the string b.
For matrices: A matrix.

Arguments
Two or more strings, string variables, or matrices.

Notes
More than two arguments can be strung together. Each additional string is appended to the end, in left to right order. Each additional matrix is appended in left to right order.

Example
a="Hello"; b=""; c="World"; a||b||c;  
"Hello World"

Concat Items({string1, string2, ...}, <delimiter>)

Function
Converts a list of string expressions into one string, with each item separated by a delimiter. The delimiter is a blank, if unspecified.

Returns
The concatenated string.

Arguments
string any string
delimiter  an optional string that is placed in front of each component string. The delimiter can be more than one character long.

Example

```julia
str1 = "one";
str2 = "two";
str3 = "three";

comb = concat Items ({str1, str2, str3});
"one two three"
comb = concat Items ({str1, str2, str3}, " : ");
"one : two : three"
del = ",";
comb = concat Items ({str1, str2, str3}, del);
"one,two,three"
```

Contains(whole, part, <start>)

Function
Determine whether part is contained within whole.

Returns
If part is found: For lists and strings, the numeric position where the first occurrence of part is located. For associative arrays, 1.
If part is not found, 0 is returned in all cases.

Arguments
whole A string, list, or associative array.
part For a string, a string that can be part of the string whole. For a list, an item that can be an item in the list whole. For an associative array, a key that can be one of the keys in the map whole.
start An optional numeric argument that specifies a starting point within whole. If start is negative, contains searches whole for part backwards, beginning with the position specified by the length of whole – start. Note that start is meaningless for associative arrays and is ignored.

Example

```julia
nameList= {"Katie", "Louise", "Jane", "Jaclyn"};
r = Contains(nameList, "Katie");
The example returns a 1 because the item "Katie" exists in the list.
```

Ends With(string, substring)

Function
Determines whether substring appears at the end of string.

Returns
1 if string ends with substring, otherwise 0.

Arguments
string A quoted string or a string variable. Can also be a list.
substring A quoted string or a string variable. Can also be a list.

Equivalent Expression
Right(string, Length(substring)) == substring

Hex(value, <"integer" | encoding="enc">)
See “Char To Hex(value, <"integer" | encoding>),” p. 449.

Hex To Blob(string)
Function
Converts the quoted hexadecimal string (including whitespace characters) to a blob (binary large object).

Example
Hex To Blob("4A4D50");
Char To Blob("JMP")

Hex To Char(string, <encoding>)
Function
Converts the quoted hexadecimal string to its character equivalent, using the specified encoding, if provided. The default value for encoding is "utf-8".

Example
Hex To Char ("30") results in "0".

Hex To Number(string)
Function
Converts the quoted hexadecimal string to its integer equivalent.

Example
Hex To Number("80");
128

Note
16-digit hexadecimal numbers are converted as IEEE 754 64-bit floating point numbers.

Insert(source, item, <position>)
Insert(source, key, value)
Function
Inserts a new item into the source at the given position. If position is not given, item is added to the end. For an associative array: Adds the key into the source associative array and assigns value to it. If the key exists in source already, its value is overwritten with the new value.

Arguments
source A string, list, expression, or associative array.
item or key Any value to be placed within source. For an associative array, key might or might not be present in source.
position  Optional numeric value that specifies the position in source to place the item into.

value  A value to assign to the key.

**Insert Into(source, item, <position>)**

*Function*

Inserts a new item into the source at the given position in place. The source must be an L-value.

*Arguments*

- **source**  A variable that contains a string, list, expression, or associative array.
- **item or key**  Any value to be placed within source. For an associative array, key might or might not be present in source.
- **position**  Optional numeric value that specifies the position in source to place the item into.
- **value**  A value to assign to the key.

**Item(n, string, <delimiters>)**

*Function*

Extracts the n-th word from a quoted string according to the quoted string delimiters given. The default delimiter is space. If you include a third argument, any and all characters in that argument are taken to be delimiters.

*Note*

Item() is the same as Word() except that Item() treats each delimiter character as a separate delimiter, and Word() treats several adjacent delimiters as a single delimiter.

```plaintext
Item(4,"the  quick brown fox");
"brown"
Word(4,"the  quick brown fox");
"fox"
```

**Left(string, n, <filler>)**

*Function*

Returns a truncated or padded version of the original string or list. The result contains the left n characters or list items, padded with any filler on the right if the length of string is less than n.

**Length(string)**

*Function*

Calculates the number of characters (length) of the quoted string.

**Lowercase(string)**

*Function*

Converts any upper case character found in quoted string to the equivalent lowercase character.
Munger(string, offset, find|length)
Munger(string, offset, find, replace)

Function
Computes new character strings from the quoted string by inserting or deleting characters. It can also produce substrings, calculate indexes, and perform other tasks depending on how you specify its arguments.
Offset is a numeric expression indicating the starting position to search in the string. If the offset is greater than the position of the first instance of the find argument, the first instance is disregarded. If the offset is greater than the search string's length, Munger uses the string's length as the offset.

Num(string)
Function
Converts a character string into a number.

Regex(source, text, <format>, <IGNORECASE>)
Function
Searches for the text within the source string.
Returns
The matched text as a string or numeric missing if there was no match.
Arguments
source A quoted string.
text A quoted regular expression.
format Optional. The default is \0, which is the entire matched string. \n returns the nth match.
IGNORECASE Optional. The search is case sensitive, unless you specify IGNORECASE.

Remove(source, position, <n>)
Remove(source, {items})
Remove(source, key)

Function
Deletes the n item(s), starting from the indicated position. If n is omitted, the item at position is deleted. If position and n are omitted, the item at the end is removed. For an associative array: Deletes the key and its value.
Returns
A copy of the source with the items deleted.
Arguments
source A string, list, expression, or associative array.
position or key An integer (or list of integers) that points to a specific item (or items) in the list or expression.
Optional. An integer that specifies how many items to remove.

Remove From(source, position, <n>)

Remove From(source, key)

Function
Deletes the n item(s) in place, starting from the indicated position. If n is omitted, the item at position is deleted. If position and n are omitted, the item at the end is removed. For an associative array: Deletes the key and its value. The source must be an L-value.

Returns
The original source with the items deleted.

Arguments
source A string, list, expression, or associative array.

position or key An integer (or list of integers) that points to a specific item (or items) in the list or expression.

n Optional. An integer that specifies how many items to remove.

Repeat(source, a)
Repeat(matrix, a, b)

Function
Returns a copy of source concatenated with itself a times. Or returns a matrix composed of a row repeats and b column repeats. The source can be text, a matrix, or a list.

Reverse(source)

Function
Reverse order of elements or terms in the source.

Argument
source A string, list, or expression.

Reverse Into(source)

Function
Reverse order of elements or terms in list or expr in place.

Argument
source A string, list, or expression.
Right(string, n, <filler>)
Right(list, n, <filler>)

Function
Returns a truncated or padded version of the original string or list. The result contains the right n characters or list items, padded with any filler on the left if the length of string is less than n.

Shift(source, <n>)

Function
Shifts an item or n items from the front to the back of the source.

Arguments
source A string, list, or expression.

n Optional. An integer that specifies the number of items to shift. Positive values shift items from the beginning of the source to the end. Negative values shift items from the end of the source to the beginning. The default value is 1.

Shift Into(source, <n>)

Function
Shifts items in place.

Arguments
source A string, list, or expression.

n Optional. An integer that specifies the number of items to shift. Positive values shift items from the beginning of the source to the end. Negative values shift items from the end of the source to the beginning. The default value is 1.

Starts With(string, substring)

Function
Determines whether substring appears at the start of string.

Returns
1 if string starts with substring, otherwise 0.

Arguments
string A quoted string or a reference to one. Can also be a list.
substring A quoted string or a reference to one. Can also be a list.

Equivalent Expression
Left(string, Length(substring)) = = substring
Substitute(string, substring, replacementString, ...)
Substitute(list, listItem, replacementItem, ...)
Substitute(Expr(sourceExpr), Expr(findExpr), Expr(replacementExpr), ...)

Function
This is a search and replace function. It searches for a specific portion (second argument) of the source (first argument), and replaces it (third argument).
If a string, finds all matches to substring in the source string, and replaces them with the replacementString.
If a list, finds all matches to listItem in the source list, and replaces them with the replacementItem.
If an expression, finds all matches to the findExpr in the sourceExpr, and replaces them with the replacementExpr. Note that all expressions must be enclosed within an Expr() function.

Arguments
string, list, sourceExpr A string, list, or expression in which to perform the substitution.
substring, listItem, findExpr A string, list item, or expression to be found in the source string, list, or expression.
replacementString, replacementItem, replacementExpr A string, list item, or expression to replace the found string, list item, or expression.

Substitute Into(string, substring, replacementString, ...)
Substitute Into(list, listItem, replacementItem, ...)
Substitute Into(Expr(sourceExpr), Expr(findExpr), Expr(replacementExpr), ...)

Function
This is a search and replace function, identical to Substitute() except in place. It searches for a specific portion (second argument) of the source (first argument), and replaces it (third argument). The first argument must be an L-value.
If a string, finds all matches to substring in the source string, and replaces them with the replacementString.
If a list, finds all matches to listItem in the source list, and replaces them with the replacementItem.
If an expression, finds all matches to the findExpr in the sourceExpr, and replaces them with the replacementExpr. Note that all expressions must be enclosed within an Expr() function.

Arguments
string, list, sourceExpr A string, list, or expression in which to perform the substitution.
substring, listItem, findExpr A string, list item, or expression to be found in the source string, list, or expression.
replacementString, replacementItem, replacementExpr A string, list item, or expression to replace the found string, list item, or expression.
Substr(string, start, length)

Function
Extracts the characters that are the portion of the first argument beginning at the position given by the second argument and ending based on the number of characters specified in the third argument. The first argument can be a character column or value, or an expression evaluating to same. The starting argument and the length argument can be numbers or expressions that evaluate to numbers.

Example
This example extracts the first name:
Substr("Katie Layman", 1, 5);

The function starts at position 1, reads through position 5, and ignores the remaining characters, which yields "Katie."

Trim("text")

Function
Produces a new character string from its argument, removing any trailing blanks.

Uppercase(string)

Function
Converts any lower case character found in the quoted string to the equivalent uppercase character.

Word(n, "text", <"delimiters">

Function
Extracts the \text{n}th word from a character string according to the delimiters given. The default delimiter is space. If you include a third argument, any and all characters in that argument are taken to be delimiters.

Examples
This example returns the last name:
Word(2, "Katie Layman");

Note
Word() is the same as Item(), except that Item() treats each delimiter character as a separate delimiter, and Word() treats several adjacent delimiters as a single delimiter.

Item(4,"the quick brown fox");
"brown"
Word(4,"the quick brown fox");
"fox"
Words("text", <"delimiters">)

**Function**
Extracts the words from text according to the delimiters given. The default delimiter is space. If you include a second argument, any and all characters in that argument are taken to be delimiters.

**Examples**
Words("the quick brown fox");
   {"the","quick","brown","fox"}
Words("Doe, Jane P.",",".");
   {"Doe","Jane","P"}

---

**Character Pattern Functions**

**Pat Abort()**

**Function**
Constructs a pattern that immediately stops the pattern match. The matcher does not back up and retry any alternatives. Conditional assignments are not made. Immediate assignments that were already made are kept.

**Returns**
0 when a match is stopped.

**Argument**
none

**Pat Altern(pattern1, <pattern 2, ...>)**

**Function**
Constructs a pattern that matches any one of the pattern arguments.

**Returns**
A pattern.

**Argument**
One or more patterns.

**Pat Any("string")**

**Function**
Constructs a pattern that matches a single character in the argument.

**Returns**
A pattern.

**Argument**
string a string.
Pat Arb()

Function
Constructs a pattern that matches an arbitrary string. It initially matches the null string. It then matches one additional character each time the pattern matcher backs into it.

Returns
A pattern.

Argument
none

Example
p = "the beginning" + Pat Arb() >? stuffInTheMiddle + "the end";
Pat Match("in the beginning of the story, and not near the end, there are three bears", p);
show(stuffInTheMiddle);
   stuffInTheMiddle = " of the story, and not near 

Pat Arb No(pattern)

Function
Constructs a pattern that matches zero or more copies of pattern.

Returns
A pattern.

Argument
pattern
a pattern to match against.

Example
adjectives = "large" | "medium" | "small" | "warm" | "cold" | "hot" | "sweet";
rc = Pat Match("I would like a medium hot, sweet tea please",
          Pat Arbono(adjectives | Pat Any("", ")) >> adj +
          ("tea" | "coffee" | "milk"));
show(rc, adj);
   adj = " medium hot, sweet "

Pat At(varName)

Function
Constructs a pattern that matches the null string and stores the current position in the source string into the specified JSL variable (varName). The assignment is immediate, and the variable can be used with expr() to affect the remainder of the match.

Returns
A pattern.

Argument
varName
the name of a variable to store the result in.

Example
p = ":" + Pat At(listStart) + expr(if(listStart==1, Pat Immediate(Pat Len(3), early), Pat Immediate(Pat Len(2), late)));
early = ""; late = "";
Pat Match(":123456789", p);
show(early,late); early = ""; late = "";
Pat Match("   :123456789", p);
show(early,late);

First this is produced:
  early = "123"
  late = ""

and later this:
  early = ""
  late = "12"

---

Pat Break("string")

Function
  Constructs a pattern that matches zero or more characters that are not in its argument; it stops or breaks
  on a character in its argument. It fails if a character in its argument is not found (in particular, it fails to
  match if it finds the end of the source string without finding a break character).

Returns
  A pattern.

Argument
  string a string.

---

Pat Concat(pattern1, pattern2 <pattern 3, ...>)

Pattern1 + Pattern2 + ...

Function
  Constructs a pattern that matches each pattern argument in turn.

Returns
  A pattern.

Argument
  Two or more patterns.

---

Pat Conditional(pattern, type)

Function
  Saves the result of the pattern match, if it succeeds, to a variable named as the second argument (type)
  after the match is finished.

Returns
  A pattern.

Arguments
  pattern a pattern to match against.
varName the name of a variable to store the result in.

Example
```javascript
type = "undefined";
rc = Pat Match("green apples", Pat Conditional("red"|"green", type) + " apples");
show(rc, type);

    rc = 1
    type = "green"
```

**Pat Fail()**

Function
Constructs a pattern that fails whenever the matcher attempts to move forward through it. The matcher backs up and tries different alternatives. If and when there are no alternatives left, the match fails and Pat Match returns 0.

Returns
0 when a match fails.

Argument
none

**Pat Fence()**

Function
Constructs a pattern that succeeds and matches the null string when the matcher moves forward through it, but fails when the matcher tries to back up through it. It is a one-way trap door that can be used to optimize some matches.

Returns
1 when the match succeeds, 0 otherwise.

Argument
none

**Pat Immediate(pattern, varName)**

Function
Saves the result of the pattern match to a variable named as the second argument (varName) immediately.

Returns
A pattern.

Arguments
- pattern a pattern to match against.
- varName the name of a variable to store the result in.

Example
```javascript
type = "undefined";
rc = Pat Match("green apples", ("red"|"green") >> type + " pears");```
show(rc, type);

rc = 0  
type = "green"
Even though the match failed, the immediate assignment was made.

Pat Len(int)

Function
Constructs a pattern that matches n characters.

Returns
A pattern.

Argument
int an integer that specifies the number of characters.

Pat Match(SourceText, Pattern, <ReplacementText>, <ANCHOR>, <FULLSCAN>)

Function
Pat Match executes the Pattern against the SourceText. The pattern must be constructed first, either inline or by assigning it to a JSL variable elsewhere.

Returns
1 if the pattern is found, 0 otherwise.

Arguments
SourceText A string or string variable that contains the text to be searched.
Pattern A pattern or pattern variable that contains the text to be searched for.
ReplacementText Optional string that, if present, defines text to replace the pattern in the source text.
ANCHOR Optional command to anchor the pattern match to the beginning of the string.
FULLSCAN Optional command to force Pat Match to try all alternatives.

Pat Not Any("string")

Function
Constructs a pattern that matches a single character that is not in the argument.

Returns
A pattern.

Argument
string a string.
**Pat Pos(int)**

**Function**
Constructs patterns that match the null string if the current position is `int` from the left end of the string, and fail otherwise.

**Returns**
A pattern.

**Argument**

`int` an integer that specifies a position in a string.

---

**Pat R Pos(int)**

**Function**
Constructs patterns that match the null string if the current position is `int` from the right end of the string, and fails otherwise.

**Returns**
A pattern.

**Argument**

`int` an integer that specifies a position in a string.

---

**Pat R Tab(int)**

**Function**
Constructs a pattern that matches up to position n from the end of the string. It can match 0 or more characters. It fails if it would have to move backwards or beyond the end of the string.

**Returns**
A pattern.

**Argument**

`int` an integer that specifies a position in a string.

---

**Pat Regex(string)**

**Function**
Constructs a pattern that matches the regular expression in the quoted `string` argument.

**Returns**
A pattern.

**Argument**

`string` a string.
Pat Rem()

Function
Constructs a pattern that matches the remainder of the string. It is equivalent to Pat R Tab(0).

Returns
A pattern.

Argument
none

Pat Repeat(pattern, minimum, maximum, GREEDY|RELUCTANT)

Function
Matches pattern between minimum and maximum times.

Returns
A pattern.

Arguments
pattern a pattern to match against.
minimum An integer that must be smaller than maximum.
maximum An integer that must be greater than minimum.
GREEDY|RELUCTANT If GREEDY is specified, it tries the maximum first and works back to the minimum. If RELUCTANT is specified, it tries the minimum first and works up to the maximum.

Notes
Pat Arbno(p) is the same as Pat Repeat(p, 0, infinity, RELUCTANT)
Pat Repeat(p) is the same as Pat Repeat(p, 1, infinity, GREEDY)
Pat Repeat(p, n) is the same as Pat Repeat(p, n, infinity, GREEDY)
Pat Repeat(p, n, m) is the same as Pat Repeat(p, n, m, GREEDY)

Pat Span("string")

Function
Constructs a pattern that matches one or more (not zero) occurrences of characters in its argument. It is greedy; it always matches the longest possible string. It fails rather than matching zero characters.

Returns
A pattern.

Argument
string a string.
Character Pattern Functions

**Pat String("string")**

**Function**
Constructs a pattern that matches its string argument.

**Returns**
A pattern.

**Argument**
string  a string.

**Pat Succeed()**

**Function**
Constructs a pattern that always succeeds, even when the matcher backs into it. It matches the null string.

**Returns**
1 when the match succeeds.

**Argument**
none

**Pat Tab(int)**

**Function**
Constructs a pattern that matches forward to position int in the source string. It can match 0 or more characters. It fails if it would have to move backwards or beyond the end of the string.

**Returns**
A pattern.

**Argument**
int  an integer that specifies a position in a string.

**Pat Test(expr)**

**Function**
Constructs a pattern that succeeds and matches the null string if expr is not zero and fails otherwise.

**Returns**
A pattern.

**Argument**
expr  An expression.

**Note**
Usually the argument is wrapped with expr() because the test needs to be made on the current value of variables set by Pat Immediate, Pat Conditional, and Pat At. Without expr, the test is based on values that were known when the pattern was constructed, which means the test always succeeds or always fails at pattern execution time, which is probably not what you want.

**Example**

nCats = 0;
whichCat = 3;
string = "catch a catnapping cat in a catsup factory";
rc = Pat Match(string, "cat" + Pat Test(expr(nCats=nCats+1; nCats ==
whichCat)), "dog");
show(rc, string, nCats);

rc = 1
string = "catch a catnapping dog in a catsup factory"
nCats = 3

---

**Regex Match(source, pattern)**

**Function**
Executes the pattern match in *pattern* against the quoted string *source*.

**Returns**
A pattern.

**Argument**
- `source` a string.
- `pattern` a pattern.

---

**Comments**

// comment

**Function**
Comments to end of line.

**Notes**
Everything after the `//` is ignored when running the script.

/* comment */

**Function**
A comment that can appear in the middle of a line of script.

**Notes**
Anything between the beginning tag `/*` and the end tag `*/` is ignored when running the script. This comment style can be used almost anywhere, even inside lists of arguments. If you place a comment inside a double-quoted string, the comment is treated merely as part of the string and not a comment. You cannot place comments in the middle of operators.

**Examples**

`+/*comment*/=`
`://*comment*/name`
are invalid and produce errors. The first comment interrupts `+=` and the second interrupts `:`name.

`sums = {(a+b /*comment*/), /*comment*/ (c^2)}`
is valid JSL; the comments are both ignored.
//!

Function
If placed on the first line of a script, this comment line causes the script to be run when opened in JMP
without opening into the script editor window.

Notes
You can over-ride this comment when opening the file. Select File > Open. Hold the CTRL key while
you select the JSL file and click Open. The script opens into a script window instead of being executed.

/*debug step*/
/*debug run*/

Function
If placed on the first line of a script, the script is opened into the debugger when it is run.

Notes
All letters must be lower case. There must be one space between debug and step or run, and there must
be no other spaces present. Only one of these lines can be used, and it must be the first line of the script;
a first line that is blank followed by this comment negates the debug command.

Comparison Functions

The comparison operators (<, <=, >, >=) work for numbers, strings, and matrices. For matrices, they
produce a matrix of results. If you compare mixed arguments, such as strings with numbers or matrices, the
result is a missing value. Comparisons involving lists are not allowed and also return missing values.

The equality operators (== and !=) work for numbers, strings, matrices, and lists. For matrices, they
produce a matrix of results; for lists, they produce a single result. If you test equality of mixed results (for
example, strings with numbers or matrices) the result is 0 or unequal.

Range check operators let you check whether something falls between two specified values:

```
a=1; show(1<=a<3);
b=2; show(2<b<=3);
```

```
1<=a<3 = 1
2<b<=3 = 0
```

Expressions with comparison operators are evaluated all at once, not in sequence

All the comparison operators are eliding operators. That means JMP treats arguments joined by comparison
operators as one big clause, as opposed to the way most expressions are evaluated one operator at a time.
Evaluating as a single clause produces different results than the more usual method of evaluating in pieces.
For example, the following two statements are different:

```
12<a<13;
(12<a)<13;
```

The first statement checks whether \( a \) is between 12 and 13, because all three arguments and both operators
are read and evaluated together. The second statement uses parentheses to regroup the operations explicitly.
to evaluate from left to right, which would be the normal way to evaluate most expressions. Thus it first checks whether 12 is less than \( a \), returning 1 if true or 0 if false. Then it checks whether the result is less than 13, which is always true because 0 and 1 are both less than 13.

All the comparison operators are elided when they are used in matched pairs or in the unmatched pairs \(<... \leq ... <\) and \(<=... <\). What this means is that if you want a comparison statement to be evaluated one comparison operator at a time, you should use parentheses ( ) to control the order of operations explicitly.

\[ \text{Equal}(a, b, ...) \]
\[ a==b==... \]

**Function**
- Compares all the listed values and tests if they are all equal to each other.

**Returns**
- 1 (true) if all arguments evaluate to the same value.
- 0 (false) otherwise.

**Arguments**
- Two or more variables, references, matrices, or numbers.

**Notes**
- If more than two arguments are specified, a 1 is returned only if all arguments are exactly the same. This is typically used in conditional statements and to control loops.
- The comparison is case-sensitive for string comparisons.

**See Also**
- The Conditional Expressions and Loops section of the *Scripting Guide*.

\[ \text{Greater}(a, b, ...) \]
\[ a>b>... \]

**Function**
- Compares all the list values and tests if, in each pair, the left value is greater than the right.

**Returns**
- 1 (true) if \( a \) evaluates strictly greater than \( b \) (and \( b \) evaluates strictly greater than \( c \), and so on).
- 0 (false) otherwise.

**Arguments**
- Two or more variables, references, matrices, or numbers.

**Notes**
- If more than two arguments are specified, a 1 is returned only if each argument is greater than the one that follows it. This is typically used in conditional statements and to control loops.
- \text{Greater}, \text{Less}, \text{GreaterOrEqual}, and \text{LessOrEqual} can also be strung together. If you do not group with parentheses, JMP evaluates each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also**
- The Conditional Expressions and Loops section of the *Scripting Guide*. 
### Greater or Equal(a, b, ...)

\[ a \geq b \geq ... \]

**Function**
- Compares all the list values and tests if, in each pair, the left value is greater than or equal to the right.

**Returns**
- 1 (true) if \( a \) evaluates strictly greater than or equal to \( b \) (and \( b \) evaluates strictly greater than or equal to \( c \), and so on).
- 0 (false) otherwise.

**Arguments**
- Two or more variables, references, matrices, or numbers.

**Notes**
- If more than two arguments are specified, a 1 is returned only if each argument is greater than or equal to the one that follows it. This is typically used in conditional statements and to control loops.

**Greater, Less, GreaterOrEqual, and LessOrEqual** can also be strung together. If you do not group with parentheses, JMP evaluates each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also**
- The Conditional Expressions and Loops section of the Scripting Guide.

### Is Missing(expr)

**Function**
- Returns 1 if the expression yields a missing value and 0 otherwise.

### Less(a, b, ...)

\[ a < b < ... \]

**Function**
- Compares all the list values and tests if, in each pair, the left value is less than the right.

**Returns**
- 1 (true) if \( a \) evaluates strictly less than \( b \) (and \( b \) evaluates strictly less than \( c \), and so on).
- 0 (false) otherwise.

**Arguments**
- Two or more variables, references, matrices, or numbers.

**Notes**
- If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops.

**Greater, Less, GreaterOrEqual, and LessOrEqual** can also be strung together. If you do not group with parentheses, JMP evaluates each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also**
- The Conditional Expressions and Loops section of the Scripting Guide.
**Less LessEqual(a, b, c, ...)**

\[ a < b \leq c \leq \ldots \]

**Function**
Range check, exclusive below and inclusive above.

**Returns**
1 (true) if \( b \) is greater than \( a \) and less than or equal to \( c \).
0 (false) otherwise.

**Arguments**
a, b, c variables, references, matrices, or numbers.

**Notes**
If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops.

**See Also**
The Conditional Expressions and Loops section of the *Scripting Guide*.

---

**Less or Equal(a, b, ...)**

\[ a \leq b \leq \ldots \]

**Function**
Compares all the list values and tests if, in each pair, the left value is less than or equal to the right.

**Returns**
1 (true) if \( a \) evaluates strictly less than or equal to \( b \) (and \( b \) evaluates strictly less than or equal to \( c \), and so on).
0 (false) otherwise.

**Arguments**
Two or more variables, references, matrices, or numbers.

**Notes**
If more than two arguments are specified, a 1 is returned only if each argument is less than or equal to the one that follows it. This is typically used in conditional statements and to control loops. **Greater**, **Less**, **GreaterOrEqual**, and **LessOrEqual** can also be strung together. If you do not group with parentheses, JMP evaluates each pair left to right. You can also use parentheses to explicitly tell JMP how to evaluate the expression.

**See Also**
The Conditional Expressions and Loops section of the *Scripting Guide*.
**LessEqual** \( \text{Less}(a, b, c, \ldots) \)

\[ a \leqslant b < c \ldots \]

**Function**
Range check, inclusive below and exclusive above.

**Returns**
1 (true) if \( b \) is greater than or equal to \( a \) and less than \( c \).
\[ 0 \text{ (false) otherwise.} \]

**Arguments**
\( a, b, c \) variables, references, matrices, or numbers.

**Notes**
If more than two arguments are specified, a 1 is returned only if each argument is less than the one that follows it. This is typically used in conditional statements and to control loops.

**See Also**
The Conditional Expressions and Loops section of the *Scripting Guide*.

**Not Equal** \( \text{NotEqual}(a, b) \)

\[ a \neq b \]

**Function**
Compares \( a \) and \( b \) and tests if they are equal.

**Returns**
0 (false) if \( a \) and \( b \) evaluate to the same value.
1 (true) otherwise.

**Argument**
\( a, b \) Any variable or number.

**Notes**
Mostly used for conditional statements and loop control.

**See Also**
The Conditional Expressions and Loops section of the *Scripting Guide*. 
Conditional and Logical Functions

And(a, b)

Function
Logical And.

Returns
1 (true) if both $a$ and $b$ are true.
0 (false) if either $a$ or $b$ is false or if both $a$ and $b$ are false.
Missing if either $a$ or $b$ is a missing value or if both $a$ and $b$ are missing values.

Arguments
Two or more variables or expressions.

Notes
More than two arguments can be strung together. $a\&b$ returns 1 (true) only if all arguments evaluate to true.

See Also
The Conditional Expressions and Loops section of the Scripting Guide.

AndMZ(a, b)

AndV3(a, b)

Function
Logical And with JMP 3 behavior, which treats missing values as 0.

Returns
1 (true) if both $a$ and $b$ are true.
0 (false) if either $a$ or $b$ is false or if both $a$ and $b$ are false.
0 (false) if either $a$ or $b$ is a missing value or if both $a$ and $b$ are missing values.

Arguments
Two or more variables or expressions.

Notes
More than two arguments can be strung together. $a:\&b$ returns 1 (true) only if all arguments evaluate to true. When opening a JMP 3 data table, this function is automatically used for any And function.

See Also
The Conditional Expressions and Loops section of the Scripting Guide.
Appendix A

Conditional and Logical Functions

---

**Break()**

**Function**

Stops execution of a loop completely and continues to the statement following the loop.

**Note**

Break works with For and While loops, and also with For Each Row.

---

**Choose(expr, r1, r2, r3, ..., rElse)**

**Function**

Evaluates expr. If the value of expr is 1, r1 is returned; if 2, the value of r2 is returned, and so on. If no matches are found, the last argument (rElse) is returned.

**Returns**

The value whose index in the list of parameters matches expr, or the value of the last parameter.

**Arguments**

expr  an expression or a value.

r1, r2, r3, ...  an expression or a value.

---

**Continue()**

**Function**

Ends the current iteration of a loop and begins the loop at the next iteration.

**Note**

Continue works with For and While loops, and also with For Each Row.

---

**For(init, while, increment, body)**

**Function**

Repeats the statement(s) in the body as long as the while condition is true. Init and increment control iterations.

**Returns**

Null.

**Arguments**

init  Initialization of loop control counter.

while  Condition for loop to continue or end. As long as the conditional statement while is true, the loop is iterated one more time. As soon as while is false, the loop is exited.

increment  Increments (or decrements) the loop counter after while is evaluated every time the loop is executed.

body  Any number of valid JSL expressions, glued together if there are more than one.

**Example**

```js
mysum = 0; myprod = 1;
For( i = 1, i <= 10, i++, mysum += i; myprod *= i; );
Show( mysum, myprod );
mysum = 55
```
myprod = 3628800

**If(condition, result, condition, ..., <elseResult>)**

*Function*

Returns *result* when *condition* evaluates true. Otherwise returns next *result* when that *condition* evaluates as true. The optional *elseResult* is used if none of the preceding conditions are true. If no *elseResult* is given, and none of the conditions are true, then nothing happens.

**IfMax(expr1, result1, expr2, result2, ... <all missing result>)**

*Function*

Evaluates the first of each pair of arguments and returns the evaluation of the result expression (the second of each pair) associated with the maximum of the expressions. If more than one expression is the maximum, the first maximum is returned. If all expressions are missing and a final result is not specified, missing is returned. If all expressions are missing and a final result is specified, that final result is returned. The test expressions must evaluate to numeric values, but the result expressions can be anything.

*Returns*

The result expression associated with the maximum of the expressions

**IfMin(expr1, result1, expr2, result2, ... <all missing result>)**

*Function*

Evaluates the first of each pair of arguments and returns the evaluation of the result expression (the second of each pair) associated with the minimum of the expressions. If more than one expression is the minimum, the first minimum is returned. If all expressions are missing and a final result is not specified, missing is returned. If all expressions are missing and a final result is specified, that final result is returned. The test expressions must evaluate to numeric values, but the result expressions can be anything.

*Returns*

The result expression associated with the minimum of the expressions

**IfMZ(condition, result, condition, ..., <elseResult>)**

**IfV3(condition, result, condition, ..., <elseResult>)**

*Function*

Logical *If* with version 3.x behavior, which treats missing values as 0; used automatically when opening v3 data tables.

**Interpolate(x, x1, y1, x2, y2)**

**Interpolate(x, xmatrix, ymatrix)**

*Function*

Linearly interpolates the *y*-value corresponding to a given *x*-value between two points (*x1, y1*), and (*x2, *y2*) or by matrices *xmatrix* and *ymatrix*. The points must be in ascending order.
Match(a, value1, result1, value2, result2, ...)

Function

If $a$ is equal to $value1$, then $result1$ is returned. If $a$ is equal to $value2$, $result2$ is returned, and so on.

MatchMZ(a, value1, result1, value2, result2, ...)
MatchV3(a, value1, result1, value2, result2, ...)

Function

Match with version 3.x behavior, which treats missing values as 0; used automatically when opening v3 data tables.

Not(a)

!a

Function

Logical Not.

Returns

0 (false) if $a$>0.
1 (true) if $a$<=0.
Missing value if $a$ is missing.

Argument

a   Any variable or number. The variable must have a numeric or matrix value.

Notes

Mostly used for conditional statements and loop control.

See Also

The Conditional Expressions and Loops section of the Scripting Guide.
Or(a, b)

Function
Logical Or.

Returns
1 (true) if either of or both \( a \) and \( b \) are true.
0 (false) otherwise.
Missing if either are missing.

Arguments
\( a, b \) Any variable or number.

Notes
Mostly used for conditional statements and loop control.

See Also
The Conditional Expressions and Loops section of the Scripting Guide.

OrMZ(a, b)

OrV3(a, b)

Function
Logical Or with version 3.x behavior, which treats missing values as 0.

Returns
1 (true) if either of or both \( a \) and \( b \) are true.
0 (false) otherwise.

Arguments
\( a, b \) Any variable or number.

Notes
Mostly used for conditional statements and loop control. When opening a JMP 3 data table, this function is automatically used for any Or function.
Or() returns missing if any evaluated argument is missing. OrMZ() returns 0 if any evaluated argument is missing.

See Also
The Conditional Expressions and Loops section of the Scripting Guide.

Step(x, x1, y1, x2, y2, ...)

Function
Finds corresponding \( y \) for a given \( x \) from a step-function fit. The points must be in ascending order.
Stop()

Function
Immediately stops a script that is running.

While(expr, body)

Function
Repeatedly tests the expr condition and executes the body until the expr is no longer true.

---

Constants

JMP provides functions for two useful constants.

Note: These functions do not take an argument, but the parentheses are required.

\( e() \)

Function
Returns the constant \( e \), which is 2.7182818284590451...

\( \Pi() \)

Function
Returns the constant \( \pi \), which is 3.1415926535897931...

---

Date and Time Functions

Datetime values are handled internally as numbers of seconds since midnight, January 1, 1904.

The expression \( x=01\text{Jan}1904 \) sets \( x \) to zero, since the indicated date is the base date or “zero date” in JMP. If you examine the values of dates, they should be appropriately large numbers. For example, \( 5\text{Oct}98 \) is 2990390400.
Abbrev Date(date)

Function
Converts the provided date to a string.

Returns
A string representation of the date.

Argument
date Can be the number of seconds since the base date (midnight, January 1, 1904), or any date-time operator.

Example
Abbrev Date(29Feb2004)
Feb 29, 2004

See Also
Section on Date-Time formats in the Scripting Guide.

Date Difference(datetime1, datetime2, "interval_name", <"alignment">)

Function
Returns the difference in intervals of two date-time values.

Returns
A number.

Arguments
datetime1, datetime2 Datetime values.
"interval_name" A quoted string that contains a date-time interval, such as "Day" or "Hour".
"alignment" An optional string. Options are as follows:
• start counts the number of times the interval starts.
• actual counts whole intervals.
• fractional counts fractional intervals.

Date DMY(day, month, year)

Function
Constructs a date value from the arguments.

Returns
The specified date, expressed as the number of seconds since midnight, 1 January 1904.

Arguments
day number, day of month, 1-31. Note that there is no error-checking, so you can enter February 31.
month number of month, 1-12.
year number of year.
Date Increment(datetime, interval, <increment>, <alignment>)

Function
Adds 1 or more intervals to a starting datetime value.

Returns
Returns the new datetime value.

Arguments
- datetime The starting datetime value.
- interval A quoted string that contains the name of a datetime interval. For example, "Month".
- increment An optional number that specifies the number of intervals. The default value is 1.
- alignment An optional quoted string that contains a keyword:
  - "start" truncates lesser field values.
  - "actual" keeps the lesser field values.
  - "fractional" honor fractional increment values.

Date MDY(month, day, year)

Function
Constructs a date value from the arguments.

Returns
The specified date, expressed as the number of seconds since midnight, 1 January 1904.

Arguments
- month number of month, 1-12.
- day number, day of month, 1-31. Note that there is no error-checking, so you can enter February 31.
- year number of year.

Day(datetime)

Function
Determine the day of the month supplied by the datetime argument.

Returns
Returns an integer representation for the day of the month of the date supplied.

Arguments
- datetime Number of seconds since midnight, 1 January 1904. This can also be an expression.

Example
```plaintext
d1 = datedmy(12, 2, 2003);
   3127852800

day(3127852800);
   12

day(d1);
   12
```
Day Of Week(datetime)

Function
Determine the day of the week supplied by the datetime argument.

Returns
Returns an integer representation for the day of the week of the date supplied.

Arguments
datetime Number of seconds since midnight, 1 January 1904. This can also be an expression.

Day Of Year(datetime)

Function
Determine the day of the year supplied by the datetime argument.

Returns
Returns an integer representation for the day of the year of the date supplied.

Arguments
datetime Number of seconds since midnight, 1 January 1904. This can also be an expression.

Format(x, "format", <"currency code">, <decimal>)

Function
Converts the value of date into the format that you specify in the second argument.

Returns
Returns the converted date in the format specified.

Arguments
x Can be a column, a number, or a datetime.
format Any valid format, as a quoted string: Best, Fixed Decimal, Percent, PValue, Scientific, Currency, or any of the Date/Time formats.
currency code An optional ISO 4217 code for a specific currency. For example, GBP for Great Britain, Pound. This argument is valid only if format is specified as Currency.
decimal An optional integer that specifies the number of decimal places to be shown.

Hour(datetime, <12|24>)

Function
Determines the hour supplied by the datetime argument.

Returns
Returns an integer representation for the hour part of the date-time value supplied.

Arguments
datetime Number of seconds since midnight, 1 January 1904. This can also be an expression.
12|24 Changes the mode to 12 hours (with am and pm). The default is 24-hour mode.
Appendix A

JSL Syntax Reference

Date and Time Functions

In Days(n)

Function
Returns the number of seconds per n days.

In Hours(n)

Function
Returns the number of seconds per n hours.

In Minutes(n)

Function
Returns the number of seconds per n minutes.

In Weeks(n)

Function
Returns the number of seconds per n weeks.

In Years(n)

Function
Returns the number of seconds per n years.

Informat(string, format)

Parse Date(string, format)

Function
Parses a quoted string of a given quoted format and returns a date/time value expressed as number of seconds since 12am, 1 January 1904.

Example
Informat("07152000", "MMDDYYYY")
15Jul2000

Long Date(date)

Function
Returns a locale-specific string representation for the date supplied, formatted like "Sunday, February 29, 2004".

MDYHMS(date)

Function
Returns a string representation for the date supplied, formatted like "2/29/04 00:02:20".
Minute(datetime)

Function
Determines the minute supplied by the datetime argument, 0-59.

Returns
Returns an integer representation for the minute part of the date-time value supplied.

Month(date)

Function
Returns an integer representation for the month of the date supplied.

Parse Date()

See “Informat(string, format),” p. 481.

Quarter(datetime)

Function
Returns the annual quarter of a datetime value as an integer 1-4.

Second(datetime)

Function
Determines the second supplied by the datetime argument.

Returns
Returns an integer representation for the second part of the date-time value supplied.

Argument
datetime Number of seconds since midnight, 1 January 1904. This can also be an expression.

Short Date(date)

Function
Returns a string representation for the date supplied, in the format mm/dd/yy. For example, "2/29/04" for the next Leap Day.

Time Of Day(date)

Function
Returns an integer representation for the time of day of the datetime supplied

Today()

Function
Returns the current date and time expressed as the number of seconds since midnight, 1 January 1904.
Discrete Probability Functions

**Week Of Year**(date)

  Function
  Returns an integer representation of week of year for date.

**Year**(date)

  Function
  Returns an integer representation for the year of date.

**Discrete Probability Functions**

**Beta Binomial Distribution**(k, p, n, delta)

  Function
  Returns the probability that a beta binomially distributed random variable is less than or equal to k.

**Beta Binomial Probability**(k, p, n, delta)

  Function
  Returns the probability that a beta binomially distributed random variable is equal to k.

**Beta Binomial Quantile**(p, n, delta, cumprob)

  Function
  Returns the smallest integer quantile for which the cumulative probability of the Beta Binomial(p, n, delta) distribution is larger than or equal to cumprob.

**Binomial Distribution**(p, n, k)

  Function
  The probability that a binomially distributed random variable is less than or equal to k.

  Returns
  The cdf for the binomial distribution with n trials, probability p of success for each trial, and k successes.

  Arguments
  p  probability of success for each trial
  n  number of trials
  k  number of successes

  See Also
  The section on probability functions in the *Using JMP*. 
Binomial Probability(p, n, k)

**Function**
The probability that a binomially distributed random variable is equal to \( k \).

**Returns**
The probability of getting \( k \) successes out of \( n \) trials if the probability is \( p \).

**Arguments**
- \( p \) probability of success for each trial
- \( n \) number of trials
- \( k \) number of successes

**See Also**
The section on probability functions in the *Using JMP*.

Binomial Quantile(p, n, cumprob)

**Function**
The quantile function of binomial distribution.

**Returns**
The smallest integer quantile for which the cumulative probability of the Binomial(p, n) distribution is larger than or equal to \( \text{cumprob} \).

**Arguments**
- \( p \) probability of success for each trial
- \( n \) number of trials
- \( \text{cumprob} \) cumulative probability

Gamma Poisson Distribution(k, lambda, sigma)

**Function**
Returns the probability that a gamma-Poisson distributed random variable is less than or equal to \( k \).

**Arguments**
- \( k \) The count of interest.
- \( \lambda \) The mean parameter.
- \( \sigma \) The overdispersion parameter.

Gamma Poisson Probability(k, lambda, sigma)

**Function**
Returns the probability that a gamma-Poisson distributed random variable is equal to \( k \).

**Arguments**
- \( k \) The count of interest.
- \( \lambda \) The mean parameter.
- \( \sigma \) The overdispersion parameter.
**Gamma Poisson Quantile**\( (\lambda, \sigma, \text{cumprob}) \)

Function

Returns the smallest integer quantile for which the cumulative probability of the Gamma Poisson(\( \lambda, \sigma \)) distribution is larger than or equal to \( \text{cumprob} \).

**Hypergeometric Distribution**\( (N, K, n, x, <r>) \)

Function

Returns the cumulative distribution function at \( x \) for the hypergeometric distribution with population size \( N \), \( K \) items in the category of interest, sample size \( n \), count of interest \( x \), and optional odds ratio \( r \).

**Hypergeometric Probability**\( (\text{pop}, k, n, x, <r>) \)

Function

Returns the probability that a hypergeometrically distributed random variable is equal to \( x \).

Arguments

- \( \text{pop} \) Population size.
- \( k \) The Number of items in the category of interest.
- \( n \) Sample size.
- \( x \) Count of interest.
- \( r \) Optional odds ratio.

**Neg Binomial Distribution**\( (p, n, k) \)

Function

Returns the probability that a negative binomially distributed random variable is less than or equal to \( k \), where the probability of success is \( p \) and the number of successes is \( n \).

**Neg Binomial Probability**\( (p, n, k) \)

Function

Returns the probability that a negative binomially distributed random variable is equal to \( k \), where the probability of success is \( p \) and the number of successes is \( n \).

**Poisson Distribution**\( (\lambda, k) \)

Function

Returns the cumulative distribution function at \( k \) for the Poisson distribution with mean \( \lambda \).

**Poisson Probability**\( (\lambda, k) \)

Function

Returns the probability that a Poisson distributed random variable with mean \( \lambda \) is equal to \( k \).
Poisson Quantile(lambda, cumprob)

Function
Returns the smallest integer quantile for which the cumulative probability of the Poisson(lambda) distribution is larger than or equal to `cumprob`.

Display

Border Box(<Left(pix)>, <Right(pix)>, <Top(Pix)>, <Bottom(Pix)>, <Sides(0)>, db)

Function
Constructs a display box that holds contains another display box. Optional arguments (Left, Right, Top, Bottom) add space between the border box and what it contains. The other optional argument (Sides) draws borders around the border box on any single side or combination of sides; draws the border in black or the highlight color; makes the background transparent or white; erases the background of a display box that contains it; draws the border in an embossed style or flat.

Returns
The display box.

Arguments

- **Left** An integer that measures pixels.
- **Right** An integer that measures pixels.
- **Top** An integer that measures pixels.
- **Bottom** An integer that measures pixels.
- **Sides** An integer that maps to settings for the display box.
- **db** A display box object (for example, a text box or another border box).

Notes
The formula for deriving the integer for Sides is: \(1*\text{top} + 2*\text{left} + 4*\text{bottom} + 8*\text{right} + 16*\text{highlightcolor} + 32*\text{whitebackground} + 64*\text{erase} + 128*\text{emboss}\). Thus, if you want to just draw a black border on the top and bottom, \(1+4 = 5\). If you want that same box with a white background, \(5+32 = 37\). If you want the top and bottom border embossed, \(37+128 = 165\).

Button Box(title, <<Set Icon("path"), script)

Function
Constructs a button with the text `title` that executes `script` when clicked.

Returns
The display box (button box).

Arguments

- **title** A quoted string or a string variable.
- **script** A quoted string or a reference to a string where the string is a valid JSL script.
<<Set Icon("path") Displays the image in the pathname on the button. Most common graphic formats are supported, such as GIF, JPG, PNG, BMP, TIF. Since the title argument is optional, you can create a button with only a text title, with only an icon, or with both a text title and an icon. In the last case, the icon is placed next to the text title.

Check Box(list, <script>)

Function
  Constructs a display box to show one or more check boxes.

Returns
  The display box (Check Box).

Arguments
  list  a list of quoted strings or a reference to a list of strings.
  script an optional JSL script.

Messages
  <<Get(n) Returns 1 if the check box item specified by n is selected, or 0 otherwise.
  <<Set(n, 0|1) Sets the check box item specified by n as either selected (1) or cleared (0).
  <<Get Selected Returns a list of strings that contain the names of the check box items that are selected.
  <<Enable Item(n, 0|1) Sets the check box item specified by n as either enabled (1) or disabled (0). The state of a disabled check box cannot be changed.
  <<Item Enabled(check box item) Returns 0 or 1 depending on whether the specific check box item is enabled.

Example
  Create three check boxes labeled “one”, “two”, and “three”. The first and third check boxes are checked.
  Check Box({"one", "two", "three"}, <<set(1, 1), <<set(3, 1))

Col List Box(<all>, <width(pix)>, <maxSelected(n)>, <nlines(n)>, <script>, <MaxItems(n)>, <MinItems(n)>, <character|numeric>, <On Change(expr)>)

Function
  Constructs a display box to show a list box that allows selection of data table columns.

Returns
  The display box (ColListBox).

Arguments
  all  an optional command that adds all columns of the current data table into the list.
  width(pix) an optional command that sets the width of the list box to pix. pix is a number that measures pixels.
  maxSelected(n) an optional command that sets whether only one item can be selected. For n>1, n is ignored.
  nlines(n) an optional command that sets the length of the list box to n number of lines. n is an integer.
**script** an optional script.

**MaxItems(n)** An optional number that only allows \( n \) columns to be added to the list.

**MinItems(n)** An optional number that only requires at least \( n \) columns for the list. If \( n=2 \), the top two slots in the **Col List Box** an initial display of “required numeric” (or whatever you set the data type to be).

**character | numeric** If you do not specify **All**, adding either **character** or **numeric** results in an empty collistbox that lists “optional character” or “optional numeric”.

**On Change(expression)** An option command that ensure that each time a change is made to the contents of a column list box, the expression is evaluated. Drag and drop operations between two column list boxes that both have this parameter set results in both expressions being evaluated. When both are evaluated, the expression for the target of the drag and drop operation is evaluated first, then the one for the source.

**Note**
The **maxSelected** argument only affects whether one or more than one item can be selected. It does not enforce a limit greater than 1.

---

**ComboBox(list, <script>)**

**Function**
Constructs a display box to show a combo box with a menu.

**Returns**
The display box (ComboBox).

**Arguments**
- **list** a list of quoted strings or a reference to a list of strings.
- **script** an optional JSL script.

---

**Context Box(displayBox, ...)**

**Function**
Defines a scoped evaluation context. Each Context Box is executed independently of each other.

**Returns**
A display box.

**Arguments**
Any number of display boxes.

---

**Current Journal()**

**Function**
Gets the display box at the top of the current (topmost) journal.

**Returns**
Returns a reference to the display box at the top of the current journal.
Current Report()

Function
Gets the display box at the top of the current (topmost) report window.

Returns
Returns a reference to the display box at the top of the current report window.

Current Window()

Function
Returns a reference to the current window.

Excerpt Box(report, subscripts)

Function
Returns a display box containing the excerpt designated by the report held at number report and the list of display subscripts subscripts. The subscripts reflect the current state of the report, after previous excerpts have been removed.

Expr As Picture(expr(…))

Function
Converts expr() to a picture as it would appear in the Formula Editor.

Returns
Reference to the picture.

Argument
expr(…) Place any valid JSL expression that can be displayed as a picture inside expr().

Global Box(global)

Function
Constructs a box for editing global value directly.

Graph()

See “Graph Box(properties, script),” p. 490.
Graph 3D Box(properties)

Function
Constructs a display box with 3-D content.

Returns
The display box.

Arguments
Properties can include: framesize(x, y), Xname("title"), Yname("title"), Zname("title").

Note
This display box constructor is experimental.

Graph Box(properties, script)
Graph(properties, script)

Function
Constructs a graph with axes.

Returns
The display box (Graph Box).

Arguments
properties Named property arguments: title("title"), XScale(low, high), YScale(low, high), FrameSize(h, v), XName("x"), YName("y"), DoubleBuffer.

script Any script to be run on the graph box.

H Center Box(display box)

Function
Returns a display box with the display box argument centered horizontally with respect to all other sibling display boxes.

H List Box(display box, ...)

Function
Creates a display box that contains other display boxes and displays them horizontally.

H Sheet Box(<<Hold(report), display boxes)

Function
Returns a display box that arranges the display boxes provided by the arguments in a horizontal layout. The <<Hold() message tells the sheet to own the report(s) that is excerpted.
Hier Box("text", Hier Box(...), ...)  
Function  
Constructs a node of a tree (similar to Diagram output) containing text. Hier Box can contain additional Hier Boxes, allowing you to create a tree. The text can be a Text Edit Box.

Icon Box(name)  
Function  
Constructs a display box containing an icon, where the quoted string name can be the name of any JMP icon or a path to a custom icon. For example, Icon Box("Nominal") constructs a display box that contains the Nominal icon.

Argument  
name Quoted string that is the name of a JMP icon or the path to an icon.

If Box(Boolean, display boxes)  
Function  
Constructs a display box whose contents are conditionally displayed.

Arguments  
Boolean 1 displays the display boxes inside the If Box, does not display them.  
display boxes Any display box tree.

Note  
If you use a variable to contain the Boolean argument, you can display and remove the display box arguments merely by changing the variable's value.

Journal Box("Journal Text")  
Function  
Constructs a display box that displays the quoted string journal box. We recommend that you do not generate the journal text by hand.

Lineup Box(<NCol(n)>, <Spacing(pixels)>, display boxes, ...)  
Function  
Constructs a display box to show an alignment of boxes in n columns.

ListBox(OperItem("item", ...), <width(pixels)>, <maxSelected(n)>, <nLines(n)>, <script>)  
Function  
Constructs a display box to show a list box of selection items.
Matrix Box(matrix)

Function
Displays the matrix given in the usual array form.

New Window("title", display box tree)

Function
Makes a new window with the indicated title (a required argument) and a display box tree.

Additional Arguments

<<Script, <"script"> Creates a new script window. The optional quoted string script is placed inside the script window.
<<Journal Creates an empty journal.
<<Modal Makes the new window a modal window, which prevents any other actions in JMP until the window is closed. If you do not include an OK or Cancel button, one is added automatically for you.
Note: If used, this argument must be the second argument, directly after the window title.
Show Toolbars(0|1) Show or hide the toolbar. The default value is 1. (Windows only.)
Show Menu(0|1) Show or hide the menu bar. The default value is 1. (Windows only.)
Suppress Auto Hide(0|1) Suppress or use the auto-hide feature for menus and toolbars. The default value is 1.

Notes
Dialog is deprecated. Use New Window() with the Modal argument instead.

Number Col Box("title", numbers)

Function
Creates a column named title with numeric entries given in list or matrix form.

Number Col Edit Box("title", numbers)

Function
Creates a column named title with numeric entries given in list or matrix form. The numbers can be edited.

Number Edit Box(value)

Function
Creates an editable number box that initially contains the value argument.

Returns
The display box object.

Argument
value Any number.
Outline Box("title", display box, ...)  
Function  
Creates a new outline named title containing the listed display boxes.

Page Break Box()  
Function  
Creates a display box that forces a page break when the window is printed.

Panel Box("title", display box)  
Function  
Creates a display box labeled with the quoted string title that contains the listed display boxes.

Picture Box(picture)  
Function  
Creates a display box that contains a graphics picture object.  
Returns  
A reference to the display box.  
Argument  
picture The pathname for the picture to include.

Plot Col Box(title, numbers)  
Function  
Returns a display box labeled with the quoted string title to graph the numbers. The numbers can be either a list or a matrix.

Popup Box({"command1", script1, "command2", script2, ...})  
Function  
Creates a red triangle menu. The single argument is an expression yielding a list of an even number of items alternating between the command string and the expression that you want evaluated when the command is selected. If the command is an empty string, a separator line is inserted.

Radio Box({"item", ...}, <script>)  
Function  
Constructs a display box to show a set of radio buttons. The optional script is run every time a radio button is selected.

Report(obj)  
Function  
Returns the display tree of a platform obj. This can also be sent as a message to a platform:
Scene Box(x size, y size)

Function
Creates an $x$ by $y$-sized scene box for 3-D graphics.

Script Box(<script>, <width>, <height>)

Function
Constructs an editable box that contains the quoted string $script$. The editable box is a script window and can both be edited and run as JSL.

Arguments
- **script** An optional quoted string that appears in the script box.
- **width** An optional integer that sets the width of the script box.
- **height** An optional integer that sets the height of the script box.

Scroll Box(<size(h,v)>, <flexible(Boolean)>, displayBox, ...)

Function
Creates a display box that positions a larger child box using scroll bars.

Returns
A reference to the scroll box object.

Arguments
- **size(h,v)** Optional. The $h$ and $v$ arguments specify the size of the box in pixels.
- **flexible(Boolean)** Optional. True (1) sets the box to be resizable with the window. False (0) sets the box to remain the same size when the window is resized.
- **displayBox** Any number of display box arguments can be contained in the scroll box.

Note
You can send a scroll box object a message to set the background color:
```
<<Set Background Color( {R, G, B} | "color" )
```

Sheet Part(title, display box)

Function
Returns a display box containing the display box argument with the quoted string title as its title.
Slider Box(min, max, global, script, <set width(n)>, <rescale slider(min, max)>)

Function
   Creates an interactive slider control.

Returns
   The display box (SliderBox).

Arguments
   min, max  Numbers that set the minimum and maximum value the slider represents.
   global  the global variable whose value is set and changed by the slider box.
   script  Any valid JSL commands that is run as the slider box is moved.
   set width(n) specify the width of the slider box in pixels.
   rescale slider(l, u) resets the max and min values for the slider box.

Notes
   You can send Set Width and Rescale Slider as commands to a slider object. For example:
   ex = .6;
   New Window("Example", mybox=Slider Box(0, 1, ex, Show(ex)));
   mybox<<Set Width(200)<<Rescale Slider(0, 5);

Spacer Box(<size(h,v)>, <color(color)>)

Function
   Creates a display box that can be used to maintain space between other display boxes, or to fill a cell in a LineUp Box.

Returns
   A reference to the display box.

Arguments
   size(h,v)  Optional. The h and v arguments specify the size of the box in pixels.
   color(color)  Optional. Sets the color of the box to the JSL color argument.

String Col Box("title", {"string", ...})

Function
   Creates column in the table containing the string items listed.

String Col Edit Box("string", ...)

Function
   Creates column in the table containing the string items listed. The string boxes are editable.

Note
   To retrieve the data, use this message:
   data = obj << Get;
**Tab Box**("page title1", contents of page 1, "page title 2", contents of page 2, ...)

**Function**
Creates a tabbed window pane. The arguments are an even number of items alternating between the name of a tab page and the contents of the tab page.

**Table Box**(display box, ...)

**Function**
Creates a report table with the *display boxes* listed as columns.

**Text Box**("text", <arguments>)

**Function**
Constructs a box that contains the quoted string *text*.

**Text Edit Box**("text", <script>)

**Function**
Constructs an editable box that contains the quoted string *text*. The optional script argument attaches a script to the text box, either by adding the script as a second argument, or by sending the Set Script message.

**V Center Box**(display box)

**Function**
Returns a display box with the display box argument centered vertically with respect to all other sibling display boxes.

**V List Box**(display box, ...)

**Function**
Creates a display box that contains other display boxes and displays them vertically.

**V Sheet Box**(<<Hold(report), display boxes)

**Function**
Returns a display box that arranges the display boxes provided by the arguments in a vertical layout. The <<Hold() message tells the sheet to own the report(s) that is excerpted.
Web Browser Box(url)

Function
Creates a display box that contains a web page.

Returns
A reference to the web browser box object.

Argument
url A quoted string containing the URL to the web page to display.

Window(<string|int>)

Returns
Either a list of references to all open windows, or a reference to an explicitly named window.

Arguments
string A quoted string containing the name of a specific open window.
int the number of a specific open window.

Notes
If no argument is provided, a list of all open windows is returned.
If the argument (either a window name or number) does not exist, an empty list is returned.

Financial Functions

Double Declining Balance(cost, salvage, life, period, <factor>)

Function
Returns the depreciation of an asset for a specified period of time. The function uses the
double-declining balance method or some other depreciation factor.

Arguments
cost The initial cost.
salvage The value at the end of the depreciation.
life The number of periods in the depreciation cycle.
period The length of the period, in the same units as life.
factor An optional number that is the rate at which the balance declines. The default value is 2.

Note
This function is equivalent to the Excel function DDB.
Future Value(rate, nper, pmt, <pv>, <type>)

Function
Returns the future value of an investment that is based on periodic, constant payments and a constant interest rate.

Arguments
rate  The interest rate.
nper  The number of periods.
pmt  The constant payment.
pv  An optional number that is the present value. The default value is 0.
type  An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function FV.

Interest Payment(rate, per, nper, pv, <fv>, <type>)

Function
Returns the interest payment for a given period for an investment that is based on periodic, constant payments and a constant interest rate.

Arguments
rate  The interest rate.
per  The period for which you want the interest.
nper  The total number of periods.
pv  The present value.
fv  An optional number that is the future value. The default value is 0.
type  An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function IPMT.

Interest Rate(nper, pmt, pv, <fv>, <type>, <guess>)

Function
Returns the interest rate per period of an annuity.

Arguments
nper  The total number of periods.
pmt  The constant payment.
pv  The present value.
fv  An optional number that is the future value. The default value is 0.
type An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

guess An optional number that is what you think the rate will be. The default value is 0.1 (10%).

Note
This function is equivalent to the Excel function RATE.

**Internal Rate of Return(values, <guess>)**

**Internal Rate of Return(guess, value1, value2, ...)**

Function
Returns the internal rate of return for a series of cash flows in the *values* argument.

Arguments
values A one-dimensional matrix of values. If the second form of the function is used, list each value separately.
guess The number that you think is near the result. The default value is 0.1 (10%).

Note
This function is equivalent to the Excel function IRR.

**Modified Internal Rate of Return(values, finance rate, reinvest rate)**

**Modified Internal Rate of Return(finance rate, reinvest rate, value1, value2, ...)**

Function
Returns the modified internal rate of return for a series of periodic cash flows. The cost of investment and the interest received on reinvested cash is included.

Arguments
values A one-dimensional matrix of values. If the second form of the function is used, list each value separately.
finance rate The interest rate that you pay on the money in the cash flows.
reinvest rate The interest rate that you receive on the cash flows when you reinvest them.

Note
This function is equivalent to the Excel function MIRR.

**Net Present Value(rate, values)**

**Net Present Value(rate, value1, value2, ...)**

Function
Returns the net present value of an investment by using a discount rate and a series of future payments (negative values) and income (positive values).

Arguments
rate The discount rate.
values A one-dimensional matrix of values. If the second form of the function is used, list each value separately.

Note
This function is equivalent to the Excel function NPV.

Number of Periods(rate, pmt, pv, <fv>, <rate>)
Function
Returns the number of periods for an investment that is based on periodic, constant payments and a constant interest rate.
Arguments
rate The interest rate.
pmt The constant payment.
pv The present value.
fv An optional number that is the future value. The default value is 0.
type An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function NPER.

Payment(rate, nper, pv, <fv>, <type>)
Function
Returns the payment for a loan that is based on constant payments and a constant interest rate.
Arguments
rate The interest rate.
nper The total number of periods.
pv The present value.
fv An optional number that is the future value. The default value is 0.
type An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function PMT.

Present Value(rate, nper, pmt, <fv>, <type>)
Function
Returns the present value of an investment.
Arguments
rate The interest rate per period.
nper The total number of periods.
pmt The constant payment.
fv  An optional number that is the future value. The default value is 0.
type  An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function PV.

Principal Payment(rate, per, nper, pv, <fv>, <type>)

Function
Returns the payment on the principal for a given period for an investment that is based on periodic, constant payments and a constant interest rate.

Arguments
rate  The interest rate per period.
per  The period for which you want the interest.
nper  The total number of periods.
pv  The present value.
fv  An optional number that is the future value. The default value is 0.
type  An optional switch. 0 specifies end-of-period payments, and 1 specifies beginning-of-period payments. The default value is 0.

Note
This function is equivalent to the Excel function PPMT.

Straight Line Depreciation(cost, salvage, life)

Function
Returns the straight-line depreciation of an asset for one period.

Arguments
cost  The initial cost of the asset.
salvage  The value at the end of the depreciation.
life  The number of periods in the depreciation cycle.

Note
This function is equivalent to the Excel function SLN.

Sum Of Years Digits Depreciation(cost, salvage, life, per)

Function
Returns the sum-of-years' digits depreciation of an asset for a specified period.

Arguments
cost  The initial cost of the asset.
salvage  The value at the end of the depreciation.
life  The number of periods in the depreciation cycle.
per  The length of the period, in the same units as life.

Note
This function is equivalent to the Excel function SYD.

Graphic Functions

Arc(x1, y1, x2, y2, startangle, endangle)

Function
Inscribes an arc in the rectangle described by the arguments.

Returns
Null.

Arguments
x1, y1  The point at the top left of the rectangle
x2, y2  The point at the bottom right of the rectangle
startangle, endangle  The starting and ending angle in degrees, where 0 degrees is 12 o'clock and
the arc or slice is drawn clockwise from startangle to endangle.

Arrow(<pixellength>, {x1, y1}, {x2, y2})

Function
Draws an arrow from the first point to the second point. The optional first argument specifies the length
of the arrow's head lines (in pixels).

Returns
Null.

Arguments
pixellength  Optional: specifies the length of the arrowhead in pixels.
{x1, y1}, {x2, y2}  Two lists of two numbers that each specify a point in the graph.

Notes
The two points can also be enclosed in square brackets: Arrow(<pixellength>, [x1, x2], [y1, y2]).

Back Color(name)

Function
Sets the color used for filling the graph's background.

Returns
Null.

Argument
name  A name color or a color index.
Char To Path(path)

Function
Converts a path specification from a string to a matrix.

Returns
A matrix.

Arguments
path A string that contains the path specification.

Circle({x, y}, radius|PixelRadius(n), <...>, <fill>)

Function
Draws a circle centered at {x, y} with the specified radius.

Returns
Null.

Arguments
{x, y} A number that describes a point in the graph
radius A number that describes the length of the circle's radius in relation to the vertical axis. If the vertical axis is resized, the circle is also resized.
PixelRadius(n) A number that describes the length of the circle's radius in pixels. If the vertical axis is resized, the circle is not resized.
fill Optional: A color that is used to fill the circle.

Note
The center point and the radius can be placed in any order. You can also add additional center point and radius arguments and draw more than one circle in one statement. One point and several radii results in a bull's-eye. Adding another point still draws all previous circles, and then adds an additional circle with the last radius specified. This means that this code:

graphbox(circle({20, 30}, 5, {50, 50}, 15))

results in three circles, not two. First, a circle with radius 5 is drawn at 20, 30. Second, a circle with radius 5 is drawn at 50, 50. Third, a circle with radius 15 is drawn at 50, 50. Note that fill must be the last argument, and it is applied to all circles defined in this function.

Color To HLS(color)

Function
Converts the color argument to a list of HLS values.

Returns
A list of the hue, lightness, and saturation components of color. The values range between 0 and 1.

Argument
color a number from the JMP color index.

Example
The output from ColorToHLS() can either be assigned to a single list variable or to a list of three scalar variables:
hls = colortohls(8);
{h, l, s} = colortohls(8);
show(hls, h, l, s);

hls = {0.75, 0.529411764705883, 1}
h = 0.75
l = 0.529411764705883
s = 1

---

**Color To RGB(color)**

*Function*
Converting the *color* argument to a list of RGB values.

*Returns*
A list of the red, green, and blue components of *color*. The values range between 0 and 1.

*Argument*
*color* a number from the JMP color index.

*Example*
The output from `ColorToRGB()` can either be assigned to a single list variable or to a list of three scalar variables:

```plaintext
rgb = colortorgb(8);
{r, g, b} = colortorgb(8);
show(rgb, r, g, b)

rgb = {0.67843137254902, 0.247058823529412, 0.972549019607843}
r = 0.67843137254902
g = 0.247058823529412
b = 0.972549019607843
```

---

**Contour(xVector, yVector, zGridMatrix, zContour, <zColors>)**

*Function*
Draws contours given a grid of values.

*Returns*
None.

*Arguments*
`xVector` The *n* values that describe `zGridMatrix`.
`yVector` The *m* values that describe `zGridMatrix`.
`zGridMatrix` An *nxm* matrix of values on some surface.
`zContour` Optional: Definition of values for the contour lines.
`zColors` Optional: Definition of colors to use for the contour lines.
Contour Function(expr, xName, yName, z, < <<XGrid(min, max, incr)>, < <<YGrid(min, max, incr)>, < <<zColor(color)>, < <<zLabeled>, < <<Filled>, < <<FillBetween>, < <<Ternary>, < <<Transparency(alpha|vector))>

Function

Draws sets of contour lines of the expression, a function of the two symbols. The z argument can be a single value or an index or matrix of values.

Returns

None.

Required Arguments

expr  Any expression. For example, Sine(y)+Cosine(x).
xName, yName  Values to use in the expression.
z  A z-value or a matrix of z-values.

Optional Arguments

<<XGrid, <<YGrid  Defines a box, beyond which the contour lines are not drawn.
<<zColor  Defines the color in which to draw the contour lines. The argument can be either a scalar or a matrix, but must evaluate to numeric.
<<zLabeled  Labels the contours.
<<Filled  Fills the contour levels using the current fill color.
<<FillBetween  Fills only between adjacent contours using the current fill color. For nz contours specified, this option fills nz-1 regions for the intervals between the nz values. Using this option is recommended over using the <<Filled option.
<<Ternary  Clips lines to be within the ternary coordinate system inside ternary plots.
<<Transparency  sets the transparency level of the fill. A vector of numbers between 0 and 1 are sequenced through and cycled for the z contours. This option should be used only in conjunction with the <<FillBetween option.

Drag Line(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

Function

Draws line segments between draggable vertices at the coordinates given by the matrix arguments.

Returns

None.

Arguments

xMatrix  A matrix of x-coordinates.
yMatrix  A matrix of y-coordinates.

dragScript  Any valid JSL script; it is run at drag.
mouseupScript  Any valid JSL script; it is run at mouseup.
Drag Marker(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

**Function**
Draws draggable markers at the coordinates given by the matrix arguments.

**Returns**
None.

**Arguments**
- **xMatrix** A matrix of x-coordinates.
- **yMatrix** A matrix of y-coordinates.
- **dragScript** Any valid JSL script; it is run at drag.
- **mouseupScript** Any valid JSL script; it is run at mouseup.

Drag Polygon(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

**Function**
Draws a filled polygon with draggable vertices at the coordinates given by the matrix arguments.

**Returns**
None.

**Arguments**
- **xMatrix** A matrix of x-coordinates.
- **yMatrix** A matrix of y-coordinates.
- **dragScript** Any valid JSL script; it is run at drag.
- **mouseupScript** Any valid JSL script; it is run at mouseup.

Drag Rect(xMatrix, yMatrix, <dragScript>, <mouseupScript>)

**Function**
Draws a filled rectangle with draggable vertices at the first two coordinates given by the matrix arguments.

**Returns**
None.

**Arguments**
- **xMatrix** A matrix of two x-coordinates.
- **yMatrix** A matrix of two y-coordinates.
- **dragScript** Any valid JSL script; it is run at drag.
- **mouseupScript** Any valid JSL script; it is run at mouseup.

**Note**
- **xMatrix** and **yMatrix** should each contain exactly two values. The resulting coordinate pairs should follow the rules for drawing a `rect()`: the first point (given by the first value in **xMatrix** and the first value in **yMatrix**) must describe the top, left point in the rectangle, and the second point (given by the second value in **xMatrix** and the second value in **yMatrix**) must describe the bottom, right point in the rectangle.
Drag Text(xMatrix, yMatrix, "text", <dragScript>, <mouseupScript>)

Function
Draws the text (or all the items if a list is specified) at the coordinates given by the matrix arguments.

Returns
None.

Arguments
- xMatrix A matrix of x-coordinates.
- yMatrix A matrix of y-coordinates.
- text A quoted string to be drawn in the graph.
- dragScript Any valid JSL script; it is run at drag.
- mouseupScript Any valid JSL script; it is run at mouseup.

Fill Color(n)

Function
Sets the color used for filling solid areas.

Returns
None.

Argument
n Index for a color or a quoted color name.

Fill Pattern()

Function
Obsolete. Patterns have been removed. Use Transparency() instead.

Gradient Function(zexpr, xname, yname, [zlow, zhight], zcolor([colorlow, colorhigh]), < <<XGrid(min, max, incr)>, < <<YGrid(min, max, incr)> < <<Transparency(alpha|vector))

Function
Fills a set of rectangles on a grid according to a color determined by the expression value as it crosses a range corresponding to a range of colors.

Example
Gradient Function(Log(a * a + b * b), a, b, [2 10], Z Color([4, 6])
Zexpr is a function in terms of the two following variables (a and b), whose values range from zlow to zhight (2 to 10). Zcolor defines the two colors that are blended together (4 is green, 6 is orange).
H Line(<x1, x2>, y)

Function
Draws a horizontal line at y across the graph. If you specify start and end points on the x-axis (x1 and x2), the line is drawn horizontally at y from x1 to x2. You can also draw multiple lines by using a matrix of values in the y argument.

H Size()

Function
Returns the horizontal size of the graphics frame in pixels.

Handle(a, b, dragScript, mouseupScript)

Function
Places draggable marker at coordinates given by a, b. The first script is executed at drag and the second at mouseup.

Heat Color(n, <theme>)

Function
Returns the JMP color that corresponds to n in the color theme.

Returns
An integer that is a JMP color.

Arguments
n  A number between 0 and 1.
theme  Any color theme that is supported by Cell Plot. The default value is "Blue to Gray to Red".

HLS Color(h, l, s)
HLS Color({h, l, s})

Function
Converts hue, lightness, and saturation values into a JMP color number.

Returns
An integer that is a JMP color number.

Arguments
Hue, lightness, and saturation, or a list containing the three HLS values. All values should be between 0 and 1.
In `Path(x, y, path)`

**Function**
Determines if the point described by `x` and `y` falls in `path`.

**Returns**
True (1) if the point `(x, y)` is in the given path, False(0) otherwise.

**Arguments**
- `x` and `y` The coordinates of a point.
- `path` Either a matrix or a string describing a path.

In `Polygon(x, y, xx, yy)`
In `Polygon(x, y, xyPolygon)`

**Function**
Returns 1 or 0, indicating whether the point `(x, y)` is inside the polygon that is defined by the `xx` and `yy` vector arguments.

The vector arguments `(xx, yy)` can also be combined into a 2-column matrix (`xyPolygon`), allowing you to use three arguments instead of four. Also, `x` and `y` can be conformable vectors, and then a vector of 0s and 1s are returned based on whether each `(x, y)` pair is inside the polygon.

**Line(({x1, y1}, {x2, y2}, ...))**
**Line(([x1, x2, ...], [y1, y2, ...]))**

**Function**
Draws a line between points.

**Arguments**
- Can be any number of lists of two points, separated by commas; or a matrix of `x`s and a matrix of `y`s.

**Line Style(n)**

**Function**
Sets the line style used to draw the graph.

**Argument**
- `n` Can be either a style name or the style's number:
  - 0 or `Solid`
  - 1 or `Dotted`
  - 2 or `Dashed`
  - 3 or `DashDot`
  - 4 or `DashDotDot`
Marker(<markerState>, {x1, y1}, {x2, y2}, ...)  
Marker(<markerState>, [x1, x2, ...], [y1, y2, ...])

**Function**
Draws one or more markers at the points described either by lists or matrices. The optional `markerState` argument sets the type of marker.

Marker Size(n)

**Function**
Sets the size used for markers.

Mousetrap(dragscript, mouseupscript)

**Function**
Captures click coordinates to update graph parameters. The first script is executed at drag and the second at `mouseup`.

Normal Contour(prob, meanMatrix, stdMatrix, corrMatrix, <colorsMatrix>, <fill=x>)

**Function**
Draws normal probability contours for \(k\) populations and two variables.

**Arguments**

- `prob` A scalar or matrix of probabilities.
- `meanMatrix` A matrix of means of size \(k\) by 2.
- `stdMatrix` A matrix of standard deviations of size \(k\) by 2.
- `corrMatrix` A matrix of correlations of size \(k\) by 1.
- `colorsMatrix` Optional. Specifies the color(s) for the \(k\) contour(s). The colors must be specified as JSL colors (either JSL color integer values or return values of JSL Color functions such as RGB Color or HLS Color).
- `fill=x` Optional. Specifies the amount of transparency for the contour fill color.

Oval(x1, y1, x2, y2, fill)
Oval({x1, y1}, {x2, y2}, fill)

**Function**
Draws an oval inside the rectangle whose diagonal has the coordinates \((x1, y1)\) and \((x2, y2)\). The oval is filled with the color `fill`. 
Path(path, <fill>)
  Function
  Draws a stroke along the given path. If a fill is specified, the interior of the path is filled with the current
  fill color.
  Argument
  path  Can be either an Nx3 matrix or a string that contains SVG syntax.
  fill  An optional, Boolean argument that specifies whether a line is drawn (0) or the path is filled (1).
          The default value is 0.
  Note
  A path matrix has three columns, for x and y, and a flag. The flag value for each point can be 0 for
  control, 1 for move, 2 for line segment, 3 for cubic Bézier segment, and any negative value to close the
  path.

Path To Char(path)
  Function
  Converts a path specification from a matrix to a string.
  Returns
  A string.
  Argument
  path  An Nx3 path matrix.
  Note
  A path matrix has three columns, for x and y, and a flag. The flag value for each point can be 0 for
  control, 1 for move, 2 for line segment, 3 for cubic Bézier segment, and any negative value to close the
  path.

Pen Color(n)
  Function
  Sets the color used for the pen.

Pen Size(n)
  Function
  Sets the thickness of the pen in pixels.

Pie(x1, y1, x2, y2, startangle, endangle)
  Function
  Draws a filled pie slice. The two points describe a rectangle, within which is a virtual oval. Only the slice
  described by the start and end angles is drawn.
Pixel Line To(x, y)

Function
Draws a one-pixel-wide line from the current pixel location to the location given in pixel coordinates. Set the current pixel location using the Pixel Origin and Pixel Move To commands.

Pixel Move To(x, y)

Function
Moves the current pixel location to a new location given in pixel coordinates.

Pixel Origin(x, y)

Function
Sets the origin, in graph coordinates, for subsequent Pixel Line To or Pixel Move To commands.

Polygon({x1, y1}, {x2, y2}, ...)
Polygon(xmatrix, ymatrix)

Function
Draws a filled polygon defined by the listed points.

Rect(x1, y1, x2, y2, <fill>)
Rect({x1, y1}, {x2, y2}, <fill>)

Function
Draws a rectangle whose diagonal has the coordinates (x1, y1) and (x2, y2). Fill is Boolean. If fill is 0, the rectangle is empty. If fill is nonzero, the rectangle is filled with the current fill color. The default value for fill is 0.

RGB Color(r, g, b)
RGB Color({r, g, b})

Function
Converts red, green, and blue values into a JMP color number.

Returns
An integer that is a JMP color number.

Arguments
Red, green, and blue, or a list containing the three RGB values. All values should be between 0 and 1.

Text(<properties>, {x, y}, text)

Function
Draws the quoted string text at the given point. Properties can be any of several named arguments: Center Justified, Right Justified, Erased, Boxed, Counterclockwise, Position, and named arguments. The
position, named arguments, and strings can be added in any order. The position and named arguments apply to all the strings.

**Text Color(n)**

Function
Sets the color for Text strings.

**Text Size(n)**

Function
Sets the font size in points for Text strings.

**Transparency(alpha)**

Function
Sets the transparency of the current drawing, with alpha between 0 and 1 where 0 is clear (no drawing) and 1 is completely opaque (the default).

Note
Not all operating systems support transparency.

**V Line(x, <y1, y2>)**

Function
Draws a vertical line at x across the graph. If you specify start and end points on the y-axis (y1 and y2), the line is drawn vertically at x from y1 to y2. You can also draw multiple lines by using a matrix of values in the x argument.

**V Size()**

Function
Returns the vertical size of the graphics frame in pixels.

**X Function(expr, symbol, <Min(min), Max(max), Fill(value), Inc(bound), Show Details(n)>)**

Function
Draws a plot of the function as the symbol is varied over the y-axis of the graph.

**X Origin()**

Function
Returns the x-value for the left edge of the graphics frame.
Matrix Functions

X Range()

Function
Returns the distance from the left to right edge of the display box. For example, X Origin() + X Range() is the right edge.

X Scale(xmin, xmax)

Function
Sets the range for the horizontal scale. The default value for xmin is 0, and the default value for xmax is 100.

Y Function(expr, symbol, <Min(min), Max(max), Fill(value), Inc(bound), Show Details(n)>)

Function
Draws a plot of the function as the symbol is varied over the x-axis of the graph.

Y Origin()

Function
Returns the y-value for the bottom edge of the graphics frame.

Y Range()

Function
Returns the distance from the bottom to top edges of a display box. For example, Y Origin() + Y Range() is the top edge.

Y Scale(ymin, ymax)

Function
Sets the range for the vertical scale. If you do not specify a scale, it defaults to (0, 100).

Matrix Functions

All(A...)

Returns
1 if all arguments are nonzero; 0 otherwise.

Any(A...)

Returns
1 if one or more elements of the matrix are nonzero; 0 otherwise.
CDF(YVec)

Function
Returns values of the empirical cumulative probability distribution function for YVec. Cumulative probability is the proportion of data values less than or equal to the value of QuantVec.

Syntax
{QuantVec, CumProbVec} = CDF(YVec)

Chol Update(L, V, C)

Function
If L is the Cholesky root of an n x n matrix A, then after calling cholUpdate L is replaced with the Cholesky root of A+V*C*V' where C is an m x m symmetric matrix and V is an n x m matrix.

Cholesky(A)

Function
Finds the lower Cholesky root (L) of a symmetric matrix. L*L' = A.

Returns
L (the Cholesky root).

Arguments
A a symmetric matrix.

Correlation(matrix)

Function
Calculates the correlation matrix of the data in the matrix argument.

Returns
The correlation matrix for the specified matrix.

Argument
matrix A matrix that contains the data. If the data has m rows and n columns, the result is an m-by-m matrix.

Notes
Rows are discarded if they contain missing values.
This function uses multithreading if available, so it is recommended for large problems with many rows.
When a column is constant, the correlations for it are 0, and the diagonal element is also 0.
Covariance(matrix)

Function
Calculates the covariance matrix of the data in the matrix argument.

Returns
The covariance matrix for the specified matrix.

Argument
matrix A matrix that contains the data. If the data has m rows and n columns, the result is an m-by-m matrix.

Notes
Rows are discarded if they contain missing values.
This function uses multithreading if available, so it is recommended for large problems with many rows.

Design(vector, < <<levels >>)

Function
Creates design columns for a vector of values.

Returns
A design matrix or a list that contains the design matrix and a list of levels.

Argument
vector A vector.
<<levels An optional argument that changes the return value to a list that contains the design matrix and a list of levels.

Design Nom(vector, < <<levels >>)
DesignF(vector, < <<levels >>)
**Design Ord(vector, <<levels>>)***

**Function**
A version of Design for making full-rank versions of design matrices for ordinal effects.

**Returns**
A full-rank design matrix or a list that contains the design matrix and a list of levels.

**Argument**
- **vector** A vector.
- **levels** An optional argument that changes the return value to a list that contains the design matrix and a list of levels.

**DesignF()**
See “Design Nom(vector, <<levels>>),” p. 516.

**Det(A)**

**Function**
Determinant of a square matrix.

**Returns**
The determinant.

**Argument**
- **det** A square matrix.

**Diag(A, <B>)**

**Function**
Creates a diagonal matrix from a square matrix or a vector. If two matrices are provided, concatenates the matrices diagonally.

**Returns**
The matrix.

**Argument**
- **A** A matrix or a vector.

**Direct Product(A, B)**

**Function**
Direct (Kronecker) product of square matrices or scalars A[i,j]*B.

**Returns**
The product.

**Arguments**
- **A, B** Square matrices or scalars.
Distance(x1, x2, <scales>, <powers>)

Function
Produces a matrix of distances between rows of x1 and rows of x2.

Returns
A matrix.

Arguments
x1, x2 Two matrices.
scales Optional argument to customize the scaling of the matrix.
powers Optional argument to customize the powers of the matrix.

E Div(A, B)

A/B

Function
Element-by-element division of two matrices.

Returns
The resulting matrix.

Arguments
A, B Two matrices.

E Mult(A, B)

A*B

Function
Element-by-element multiplication of two matrices.

Returns
The resulting matrix.

Arguments
A, B Two matrices.

Eigen(A)

Function
Eigenvalue decomposition.

Returns
A list \{M, E\} such that \( E \cdot \text{Diag}(M) \cdot E = A' \).

Argument
A A symmetric matrix.
Matrix Functions

\texttt{G Inverse(A)}

\textbf{Function}

Generalized (Moore-Penrose) matrix inverse.

\texttt{H Direct Product(A, B)}

\textbf{Function}

Horizontal direct product of two square matrices of the same dimension or scalars.

\texttt{Identity(n)}

\textbf{Function}

Creates an \( n \)-by-\( n \) identity matrix with ones on the diagonal and zeros elsewhere.

\textbf{Returns}

The matrix.

\textbf{Argument}

\( n \) An integer.

\texttt{Index(i, j, <increment>)} \\
\( i::j \)

\textbf{Function}

Creates a column matrix whose values range from \( i \) to \( j \).

\textbf{Returns}

The matrix.

\textbf{Arguments}

\( i, j \) Integers that define the range: \( i \) is the beginning of the range, \( j \) is the end.

\( \text{increment} \) Optional argument to change the default increment, which is +1.

\texttt{Inv()} \\

See “Inverse(A),” p. 520.

\texttt{Inv Update(A, X, 1|-1)}

\textbf{Function}

Efficiently update an \( X'X \) matrix.

\textbf{Arguments}

\( A \) The matrix to be updated.

\( X \) One or more rows to be added to or deleted from the matrix \( A \).

\( 1|-1 \) The third argument controls whether the row or rows defined in the second argument, \( X \), are added to or deleted from the matrix \( A \). 1 means to add the row or rows and -1 means to delete the row or rows.
Inverse(A)

Inv(A)

Function
Returns the matrix inverse. The matrix must be square non-singular.

Is Matrix(x)

Function
Returns 1 if the evaluated argument is a matrix, or 0 otherwise.

J(nrows, <ncols>, <value>)

Function
Creates a matrix of identical values.

Returns
The matrix.

Arguments
nrows Number of rows in matrix. If <ncols> is not specified, nrows is also used as <ncols>.
<ncols> Number of columns in matrix.
value The value used to populate the matrix. If value is not specified, 1 is used.

Loc(A)

Loc(list, item)

Function
Creates a matrix of subscript positions where A is nonzero and nonmissing. For the two-argument function, Loc returns a matrix of positions where item is found within list.

Returns
The new matrix.

Argument
A a matrix
list a list
item the item to be found within the list

Loc Max(A)

Function
Returns the position of the minimum element in a matrix.

Returns
An integer that is the specified position.

Argument
A a matrix
**Loc Min(A)**

**Function**
Returns the position of the minimum element in a matrix.

**Returns**
An integer that is the specified position.

**Argument**

A a matrix

---

**Loc Sorted(A, B)**

**Function**
Creates a matrix of subscript positions where the values of A have values less than or equal to the values in B. A must be a matrix sorted in ascending order.

**Returns**
The new matrix.

**Argument**

A, B matrices

---

**Matrix({{x11, x12, ..., x1m}, {x21, x22, ..., 2m}, {...}, {xn1, xn2, ..., xnm}})**

**Function**
Constructs an n-by-m matrix from a list of n lists of m row values.

**Returns**
The matrix.

**Arguments**

A list of lists in which each list forms a row with the specified values.

**Example**

mymatrix = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}});

```
[ 1 2 3,
 4 5 6,
 7 8 9,
10 11 12]
```

**Equivalent Expression**

```
[x11 x12 ... x1m,
 ..., xnm]
```
Matrix Mult(A, B)
C=A*B, ...

Function
Matrix multiplication.

Arguments
Two or more matrices, which must be conformable (all matrices after the first one listed must have the same number of rows as the number of columns in the first matrix).

Note
Matrix Mult() allows only two arguments, while using the * operator enables you to multiply several matrices.

Maximize(expr, {x1, x2, ...}, messages)

Function
Finds the values for the listed x's that maximize the expr.

Messages
<<Max Iter(int) An integer that specifies the maximum number of iterations to be performed.
<<Tolerance(p) p is a number between 0 and 1.
<<Method(NR|BFGS) Specify either the Newton-Raphson method or the Quasi-Newton method with BFGS update.
<<Limits(arguments) The arguments specify ranges for each x (for example, x1<3)

Minimize(expr, {x1, x2, ...}, messages)

Function
Finds the values for the listed x's that minimize the expr.

Messages
<<Max Iter(int) An integer that specifies the maximum number of iterations to be performed.
<<Tolerance(p) p is a number between 0 and 1.
<<Method(NR|BFGS) Specify either the Newton-Raphson method or the Quasi-Newton method with BFGS update.
<<Limits(arguments) The arguments specify ranges for each x (for example, x1<3)

Multivariate Normal Impute(y, mean, covariance)

Function
Transforms the responses to the principal component space.

Arguments
y The vector of responses.
mean The vector of response means.
covariance A symmetric matrix containing the response covariances.
N Col(x)
N Cols(x)

Function
Returns the number of columns in either a data table or a matrix.

Argument
x Can be a data table or a matrix.

Ortho(A, <Centered(0)>, <Scaled(1)>)

Function
Orthonormalizes the columns of matrix A using the Gram Schmidt method. Centered(0) makes the columns to sum to zero. Scaled(1) makes them unit length.

Ortho Poly(vector, order)

Function
Returns orthogonal polynomials for a vector of indices representing spacings up to the order given.

Print Matrix(M, <named arguments>)

Function
Prints a well-formatted matrix to the log.

Returns
Returns a string that contains the well-formatted matrix.

Argument
M A matrix.

Named Arguments
Note that the following named arguments are all optional.
<<ignore locale(Boolean) Set to false (0) to use the decimal separator for your locale. Set to true (1) to always use a period (.) as a separator. The default value is false (0).
<<decimal digits(n) An integer that specifies the number of digits after the decimal separator to print.
<<style("style name") Use one of three available styles: Parseable is a reformatted JSL matrix expression. Latex is formatted for LaTex. If you specify Other, you must define the following three arguments.
<<separate("character") Define the separator for concatenated entries.
<<line begin("character") Define the beginning line character.
<<line end("character") Define the ending line character.
**JSL Syntax Reference**  
Matrix Functions

---

**QR(A)**  
**Function**  
Returns the QR decomposition of A. Typical usage is \( \{Q, R\} = \text{QR}(A) \).

---

**Random Index(n, k)**  
**Function**  
Returns a \( k \) by 1 matrix of random integers between 1 and \( n \) with no duplicates.

---

**Random Shuffle(matrix)**  
**Function**  
Returns the matrix with the elements shuffled into a random order.

---

**Rank Index(vector)**  
**Rank(vector)**  
**Function**  
Returns a vector of indices that, used as a subscript to the original vector, sorts the vector by rank. Excludes missing values.

---

**Ranking(vector)**  
**Function**  
Returns a vector of ranks of the values of vector, low to high as 1 to \( n \), ties arbitrary.

---

**Ranking Tie(vector)**  
**Function**  
Returns a vector of ranks of the values of vector, but ranks for ties are averaged.

---

**Shape(A, nrow, <ncol>)**  
**Function**  
Reshapes the matrix \( A \) across rows to the specified dimensions. Each value from the matrix \( A \) is placed row-by-row into the re-shaped matrix.

**Returns**  
The reshaped matrix.

**Arguments**  
\( A \) a matrix  
\( nrow \) the number of rows that the new matrix should have.
\texttt{ncol} \hspace{2em} \text{optional. The number of columns the new matrix should have.}

**Notes**

- If \texttt{ncol} is not specified, the number of columns is whatever is necessary to fit all of the original values of the matrix into the reshaped matrix.
- If the new matrix is smaller than the original matrix, the extra values are discarded.
- If the new matrix is larger than the original matrix, the values are repeated to fill the new matrix.

**Examples**

```julia
a = matrix([ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} ]); 
[ 1 2 3,  
 4 5 6,  
 7 8 9 ]

shape(a, 2); 
[ 1 2 3 4 5,  
 6 7 8 9 1 ]

shape(a, 2, 2); 
[ 1 2 ,  
 3 4 ]

shape(a, 4, 4); 
[ 1 2 3 4,  
 5 6 7 8,  
 9 1 2 3,  
 4 5 6 7 ]
```

**Solve\((A, b)\)**

**Function**

Solves a linear system. In other words, \(x = \text{inverse}(A) \ast b\).

**Sort Ascending\((source)\)**

**Function**

Returns a copy of a list or matrix \texttt{source} with the items in ascending order.

**Sort Descending\((source)\)**

**Function**

Returns a copy of a list or matrix \texttt{source} with the items in descending order.

**Spline Coef\((x, y, \text{lambda})\)**

**Function**

Returns a five column matrix of the form \texttt{knots||a||b||c||d} where \texttt{knots} is the unique values in \texttt{x}.
Spline Eval(x, coef)

Function
Evaluates the spline predictions using the \texttt{coef} matrix, where \texttt{coef} is in the same form as Spline Coef().

Spline Smooth(x, y, lambda)

Function
Returns the smoothed predicted values from a spline fit.

SVD(A)

Function
Singular value decomposition.

Sweep(A, <indices>)

Function
Sweeps, or inverts a matrix a partition at a time.

Trace(A)

Function
The trace, or the sum of the diagonal elements of a square matrix.

Transpose(A)

Function
Transposes the rows and columns of the matrix \texttt{A}.

Returns
The transposed matrix.

Arguments
\texttt{A} \hspace{1em} \texttt{A} matrix.

Equivalent Expression
\texttt{A}'

V Concat(A, B, ...)

Function
Vertical concatenation of two or more matrices.

Returns
The new matrix.

Arguments
Two or more matrices.
Matrix Functions

V Max(matrix)

Function
Returns a row vector containing the maximum of each column of matrix.

V Mean(matrix)

Function
Returns a row vector containing the mean of each column of matrix.

V Min(matrix)

Function
Returns a row vector containing the minimum of each column of matrix.

V Std(matrix)

Function
Returns a row vector containing the standard deviations of each column of matrix.

V Sum(matrix)

Function
Returns a row vector containing the sum of each column of matrix.

Vec Diag(A)

Function
Creates a vector from the diagonals of a square matrix A.

Returns
The new matrix.

Arguments
A square matrix.

Note
Using a matrix that is not square results in an error.

Vec Quadratic(symmetric matrix, rectangular matrix)

Function
Constructs an n-by-m matrix. Used in calculation of hat values.

Returns
The new matrix.

Arguments
Two matrices. The first must be symmetric.

Equivalent Expression
Vec Diag(X*Sym*X')
Numeric Functions

Abs(n)

Function
Calculates the absolute value of \( n \).

Returns
Returns a positive number of the same magnitude as the value of \( n \).

Argument
\( n \) Any number, numeric variable, or numeric expression.

Ceiling(n)

Function
If \( n \) is not an integer, rounds \( n \) to the next highest integer.

Returns
Returns the smallest integer greater than or equal to \( n \).

Argument
\( n \) any number

Floor(n)

Function
If \( n \) is not an integer, rounds \( n \) to the next lowest integer.

Returns
Returns the largest integer less than or equal to \( n \).

Argument
\( n \) any number

Examples

Floor(2.7)
\[ 2 \]

Floor(−.5)
\[ −1 \]

Mod()

See “Modulo(number, divisor),” p. 529
Appendix A

JSL Syntax Reference

Probability Functions

Modulo(number, divisor)

Mod(number, divisor)

Function
Returns the remainder when number is divided by divisor.

Examples
Modulo(6, 5)
1

Round(n, places)

Function
Rounds n to number of decimal places given.

Probability Functions

Beta Density(x, alpha, beta, <theta>, <sigma>)

Function
Calculates the beta probability density function (pdf).

Returns
The density function at quantile x for the beta distribution for the given arguments.

Arguments
x  A quantile between theta and theta + sigma. Theta’s default value is 0. Sigma’s default value is 1.
alpha, beta  Shape parameters that must both be greater than 0.
theta  optional threshold. The allowable range is $-\infty < \theta < \infty$. The default is 0.
sigma  optional scale parameter, which must be greater than 0. The default is 1.

See Also
The section on probability functions in the Using JMP.

Beta Distribution(x, alpha, beta, <theta>, <sigma>)

Function
Calculates the cumulative distribution function for the beta distribution.

Returns
Returns the cumulative distribution function at quantile x for the beta distribution with shape arguments alpha and beta.

Arguments
x  A quantile between theta and theta + sigma.
alpha, beta  Shape parameters that must both be greater than 0.
theta  optional threshold. The allowable range is $-\infty < \theta < \infty$. The default is 0.
sigma  optional scale parameter, that must be greater than 0. The default is 1.

See Also
The section on probability functions in the *Using JMP*.

**Beta Quantile**(*p*, *alpha*, *beta*, <theta>, <sigma>)

**Function**
Calculates the requested quantile for the beta distribution.

**Returns**
Returns the *p*th quantile from the beta distribution with shape arguments *alpha* and *beta*.

**Arguments**
- *p*  The probability of the quantile desired; *p* must be between 0 and 1.
- *alpha*, *beta*  Shape parameters that must both be greater than 0.
- *theta*  optional threshold. The allowable range is \(-\infty < \theta < \infty\). The default is 0.
- *sigma*  optional scale parameter, which must be greater than 0. The default is 1.

See Also
The section on probability functions in the *Using JMP*.

**ChiSquare Density**(*q*, *df*, <center>)

**Function**
The chi-square density at *q* of the chi-square with *df* degrees of freedom and optional non-centrality parameter *center*.

**Returns**
The chi-square density.

**Arguments**
- *q*  quantile
- *df*  degrees of freedom.
- *center*  non-centrality parameter

**ChiSquare Distribution**(*q*, *df*, <center>)

**Function**
Returns cumulative distribution function at quantile *x* for chi-square with *df* degrees of freedom centered at center.

**ChiSquare Quantile**(*q*, *df*, <center>)

**Function**
Returns the *p*th quantile from the chi-square distribution with *df* degrees of freedom, centered at center.
Dunnett P Value(q, nTrt, dfe, lambdaVec)

Function
Returns the p-value from Dunnett’s multiple comparisons test.

Returns
A number that is the p-value.

Arguments
q A number that is the test statistic.
nTrt The number of treatments in the study.
dfe The error degrees of freedom.
lambdaVec A vector of parameters. If lambdaVec is missing (.), the parameters are set to 1/Sqrt(2).

Dunnett Quantile(1-alpha, nTrt, dfe, lambdaVec)

Function
Returns quantile needed in from Dunnett’s multiple comparisons test.

Returns
A number that is the quantile.

Arguments
1-alpha A number that is the confidence interval.
nTrt The number of treatments in the study.
dfe The error degrees of freedom.
lambdaVec A vector of parameters. If lambdaVec is missing (.), the parameters are set to 1/Sqrt(2).

F Density(x, dfnum, dfden, <center>)

Function
Returns the F density at x for the F distribution with numerator and denominator degrees of freedom dfnum and dfden, with optional noncentrality parameter center.

F Distribution(x, dfnum, dfden, <center>)

Function
Returns cumulative distribution function at quantile x for F distribution with numerator and denominator degrees of freedom dfnum and dfden and noncentrality parameter center.

F Power(alpha, dfh, dfm, d, n)

Function
Calculates the power from a given situation involving an F test or a t test.
F Quantile(x, dfnum, dfden, <center>)

Function
Returns the \( p \)th quantile from the F distribution with numerator and denominator degrees of freedom \( dfnum \) and \( dfden \) and noncentrality parameter center.

F Sample Size(alpha, dfh, dfm, d, power)

Function
Calculates the sample size from a given situation involving an \( F \) test or a \( t \) test.

Frechet Density(x, mu, sigma)

Function
Returns the density at \( x \) of a Frechet distribution with location \( mu \) and scale \( sigma \).

Arguments
- \( x \) A number.
- \( mu \) A location.
- \( sigma \) The scale.

Frechet Distribution(x, mu, sigma)

Function
Returns the probability that the Fréchet distribution with location \( mu \) and scale \( sigma \) is less than \( x \).

Arguments
- \( x \) A number.
- \( mu \) A location.
- \( sigma \) The scale.

Frechet Quantile(p, mu, sigma)

Function
Returns the quantile associated with a cumulative probability \( p \) for a Frechet distribution with location \( mu \) and scale \( sigma \).

Arguments
- \( p \) The probability of the quantile desired; \( p \) must be between 0 and 1.
- \( mu \) A location.
- \( sigma \) The scale.
**Gamma Density**($q$, $alpha$, $<scale>$, $<threshold>$)

Function
Calculates the density at $q$ of a Gamma probability distribution.

Returns
The density function at quantile $q$ for the gamma density distribution for the given arguments.

Arguments
- $q$ A quantile.
- $alpha$ Shape parameters that must be greater than 1.
- $scale$ Optional scale, which must be greater than 0. The default is 1.
- $threshold$ Optional threshold parameter. The allowable range is $-\infty < 0 < \infty$. The default is 0.

See Also
The section on probability functions in the Using JMP.

**Gamma Distribution**($x$, $<shape, scale, threshold>$)

IGamma($x$, $<shape, scale, threshold>$)

Function
Returns cumulative distribution function at quantile $x$ for the gamma distribution with $shape$, $scale$, and $threshold$ given.

**Gamma Quantile**($p$, $<shape, scale, threshold>$)

Function
Returns the $p$th quantile from the gamma distribution with the $shape$, $scale$, and $threshold$ parameters given.

**GLog Density**($q$, $mu$, $sigma$, $lambda$)

Function
Returns the density at $q$ of a generalized logarithmic distribution with location $mu$, scale $sigma$, and shape $lambda$.

**GLog Distribution**($q$, $mu$, $sigma$, $lambda$)

Function
Returns the probability that a generalized logarithmically distribution random variable is less than $q$.

**GLog Quantile**($p$, $mu$, $sigma$, $lambda$)

Function
Returns the quantile for whose value the probability is $p$ that a random value would be lower.
IGamma()  
See “Gamma Distribution(x, <shape, scale, threshold>),” p. 533.

**Johnson Sb Density(q, gamma, delta, theta, sigma)**  
*Function*  
Returns the density at *q* of a Johnson Sb distribution.  
*Arguments*  
- **q** A value that is in the interval *theta* to *theta* + *sigma*.  
- **gamma** Shape parameter that can be any value.  
- **delta** Shape parameter that must be greater than 0.  
- **theta** Location parameter that can be any value.  
- **sigma** Scale parameter that must be greater than 0.

**Johnson Sb Distribution(q, gamma, delta, theta, sigma)**  
*Function*  
Returns the probability that a Johnson Sb-distributed random variable is less than *q*.  
*Arguments*  
- **q** A value that is in the interval *theta* to *theta* + *sigma*.  
- **gamma** Shape parameter that can be any value.  
- **delta** Shape parameter that must be greater than 0.  
- **theta** Location parameter that can be any value.  
- **sigma** Scale parameter that must be greater than 0.

**Johnson Sb Quantile(p, gamma, delta, theta, sigma)**  
*Function*  
Returns the quantile whose value for which the probability is *p* that a random value would be lower.  
*Arguments*  
- **p** The probability of the quantile desired; *p* must be between 0 and 1.  
- **gamma** Shape parameter that can be any value.  
- **delta** Shape parameter that must be greater than 0.  
- **theta** Location parameter that can be any value.  
- **sigma** Scale parameter that must be greater than 0.
**Johnson Sl Density***(q, gamma, delta, theta, sigma)***

**Function**
Returns the density at *q* of a Johnson Sl distribution.

**Arguments**
- q  A value that is in the interval *theta* to +infinity.
- gamma  Shape parameter that can be any value.
- delta  Shape parameter that must be greater than 0.
- theta  Location parameter that can be any value.
- sigma  Parameter that defines if the distribution is skewed positively or negatively. *Sigma* must be equal to either +1 (skewed positively) or -1 (skewed negatively).

**Johnson Sl Distribution***(q, gamma, delta, theta, sigma)***

**Function**
Returns the probability that a Johnson Sl-distributed random variable is less than *q*.

**Arguments**
- q  A value that is in the interval *theta* to +infinity.
- gamma  Shape parameter that can be any value.
- delta  Shape parameter that must be greater than 0.
- theta  Location parameter that can be any value.
- sigma  Parameter that defines if the distribution is skewed positively or negatively. *Sigma* must be equal to either +1 (skewed positively) or -1 (skewed negatively).

**Johnson Sl Quantile***(p, gamma, delta, theta, sigma)***

**Function**
Returns the quantile whose value for which the probability is *p* that a random value would be lower.

**Arguments**
- p  The probability of the quantile desired; *p* must be between 0 and 1.
- gamma  Shape parameter that can be any value.
- delta  Shape parameter that must be greater than 0.
- theta  Location parameter that can be any value.
- sigma  Parameter that defines if the distribution is skewed positively or negatively. *Sigma* must be equal to either +1 (skewed positively) or -1 (skewed negatively).

**Johnson Su Density***(q, gamma, delta, theta, sigma)***

**Function**
Returns the density at *q* of a Johnson Su distribution.

**Arguments**
- q  A value that is between -infinity and +infinity.
gamma  Shape parameter that can be any value.
delta  Shape parameter that must be greater than 0.
theta  Location parameter that can be any value.
sigma  Scale parameter that must be greater than 0.

**Johnson Su Distribution**(q, gamma, delta, theta, sigma)

**Function**

Returns the probability that a Johnson Su-distributed random variable is less than q.

**Arguments**

- q  A value that is between -infinity and +infinity.
- gamma  Shape parameter that can be any value.
- delta  Shape parameter that must be greater than 0.
- theta  Location parameter that can be any value.
- sigma  Scale parameter that must be greater than 0.

**Johnson Su Quantile**(p, gamma, delta, theta, sigma)

**Function**

Returns the quantile whose value for which the probability is p that a random value would be lower.

**Arguments**

- p  The probability of the quantile desired; p must be between 0 and 1.
- gamma  Shape parameter that can be any value.
- delta  Shape parameter that must be greater than 0.
- theta  Location parameter that can be any value.
- sigma  Scale parameter that must be greater than 0.

**LEV Density**(x, mu, sigma)

**Function**

Returns the density at x of the largest extreme value distribution with location mu and scale sigma.

**LEV Distribution**(x, mu, sigma)

**Function**

Returns the probability that the largest extreme value distribution with location mu and scale sigma is less than x.

**LEV Quantile**(p, mu, sigma)

**Function**

Returns the quantile associated with a cumulative probability p of the largest extreme value distribution with location mu and scale sigma.
Logistic Density(x, mu, sigma)

Function
Returns the density at x of a logistic distribution with location mu and scale sigma.

Logistic Distribution(x, mu, sigma)

Function
Returns the probability that the logistic distribution with location mu and scale sigma is less than x.

Logistic Quantile(x, mu, sigma)

Function
Returns the quantile associated with a cumulative probability p of the logistic distribution with location mu and scale sigma.

Loglogistic Density(x, mu, sigma)

Function
Returns the density at x of a loglogistic distribution with location mu and scale sigma.

Loglogistic Distribution(x, mu, sigma)

Function
Returns the probability that the loglogistic distribution with location mu and scale sigma is less than x.

Loglogistic Quantile(x, mu, sigma)

Function
Returns the quantile associated with a cumulative probability p of the loglogistic distribution with location mu and scale sigma.

Lognormal Density(x, mu, sigma)

Function
Returns the density at x of a lognormal distribution with location mu and scale sigma.

Lognormal Distribution(x, mu, sigma)

Function
Returns the probability at x of a lognormal distribution with location mu and scale sigma.

Lognormal Quantile(x, mu, sigma)

Function
Returns the quantile at p of a lognormal distribution with location mu and scale sigma.
Normal Biv Distribution(x, y, r, <mu1>, <s1>, <mu2>, <s2>)

Function
Computes the probability that an observation (X, Y) is less than or equal to (x, y) with correlation coefficient r where X is individually normally distributed with mean mu1 and standard deviation s1 and Y is individually normally distributed with mean mu2 and standard deviation s2. If mu1, s1, mu2, and s2 are not given, the function assumes the standard normal bivariate distribution with mu1=0, s1=1, mu2=0, and s2=1.

Normal Density(x, <mean>, <stddev>)
Function
Returns the value of the density function at quantile x for the normal distribution with mean and stddev. The default mean is 0. the default stddev is 1.

Normal Distribution(x, <mean>, <stddev>)
Function
Returns the cumulative distribution function at quantile x for the normal distribution with mean and stddev. The default mean is 0. the default stddev is 1.

Normal Mixture Density(q, mean, stdev, probability)
Function
Returns the density at q of a normal mixture distribution with group means mean, group standard deviations stdev, and group probabilities probability. The mean, stdev, and probability arguments are all vectors of the same size.

Normal Mixture Distribution(q, mean, stdev, probability)
Function
Returns the probability that a normal mixture distributed variable with group means mean, group standard deviations stdev, and group probabilities probability is less than q. The mean, stdev, and probability arguments are all vectors of the same size.

Normal Mixture Quantile(p, mean, stdev, probability)
Function
Returns the quantile, the values for which the probability is p that a random value would be lower. The mean, stdev, and probability arguments are all vectors of the same size.

Normal Quantile(p, <mean>, <stddev>)
Probit(p, <mean>, <stddev>)
Function
Returns the pth quantile from the normal distribution with mean and stddev. The default mean is 0. the default stddev is 1.
Probit()

See “Normal Quantile(p, <mean>, <stddev>),” p. 538.

SEV Density(x, mu, sigma)

Function
Returns the density at x of the smallest extreme distribution with location mu and scale sigma.

SEV Distribution(x, mu, sigma)

Function
Returns the probability that the smallest extreme distribution with location mu and scale sigma is less than x.

SEV Quantile(p, mu, sigma)

Function
Returns the quantile at p of the smallest extreme distribution with location mu and scale sigma.

Students t Density()

See “t Density(q, df),” p. 539.

Students t Distribution()

See “t Distribution(q, df, <nonCentrality>),” p. 539.

Students t Quantile()

See “t Quantile(p, df, <nonCentrality>),” p. 540.

t Density(q, df)

Students t Density(q, df)

Function
Returns the value of the density function at quantile x for the Student’s t distribution with degrees of freedom df.

t Distribution(q, df, <nonCentrality>)

Students t Distribution(q, df, <nonCentrality>)

Function
Returns the probability that a Student’s t distributed random variable is less than q. NonCentrality defaults to 0.
**t Quantile**(*p*, df, *<nonCentrality>*)

**Students t Quantile**(*p*, df, *<nonCentrality>*)

**Function**
Returns the *p*th quantile from the Student's *t* distribution with degrees of freedom *df*. *NonCentrality* defaults to 0.

**Tukey HSD P Value**(*q*, *n*, *dfe*)

**Function**
Returns the *p*-value from Tukey's HSD multiple comparisons test.

**Arguments**
- *q*  The test statistic.
- *n*  The number of groups in the study.
- *dfe*  The error degrees of freedom, based on the total study sample.

**Tukey HSD Quantile**(*1-alpha*, *n*, *dfe*)

**Function**
Returns the quantile needed in Tukey's HSD multiple comparisons test.

**Arguments**
- *1-alpha*  The confidence level.
- *n*  The number of groups in the study.
- *dfe*  The error degrees of freedom, based on the total study sample.

**Weibull Density**(*x*, shape, *<scale, threshold>*)

**Function**
Returns the value of the density function at quantile *x* for the Weibull distribution with the parameters given.

**Weibull Distribution**(*x*, shape, *<scale, threshold>*)

**Function**
Returns the cumulative distribution function at quantile *x* for the Weibull distribution with the parameters given.

**Weibull Quantile**(*p*, shape, *<scale, threshold>*)

**Function**
Returns the *p*th quantile from the Weibull distribution with the parameters given.
R Integration Functions

R Connect( <named_arguments> )

Function
   Initializes the R integration interfaces and returns an active R integration interface connection as a
   scriptable object.

Returns
   R scriptable object.

Arguments
   Echo(Boolean) Optional. Sends all source lines to the JMP log. This option is global. The default
   value is true.

R Execute( { list of inputs }, { list of outputs }, rCode,
<named_parameters> )

Function
   Submit the specified R code to the active global R connection given a list of inputs. On completion, the
   outputs are returned into the specified list.

Returns
   0 if successful; nonzero otherwise.

Arguments
   { list of inputs } A list of JMP variable names to be sent to R as inputs.
   { list of outputs } A list of JMP variable names to contain the outputs returned from R.
   rCode A quoted string that contains the R code to submit.
   Expand(Boolean) An optional, Boolean, named parameter. Performs an Eval Insert() on the R
   code before submitting to R.
   Echo(Boolean) An optional, Boolean, named parameter. Sends all source lines to the JMP log. This
   option is global. The default value is true.

Example
   Send the JMP variables x and y to R, execute the R statement z <- x * y, and then get the R variable z and
   return it to JMP.
   x = [1 2 3];
   rc = R Send( x );
   y = [4 5 6];
   rc = R Send( y );
   rc = R Execute( {x, y}, {z}, "z <- x * y" );
R Get( variable_name )

Function
Gets the named variable from R to JMP.

Returns
The value of the named variable.

Argument
name Required. The name of an R variable whose value to return to JMP.

Example
Assume that a matrix named qbx and a data frame named df are present in your R connection.

// Get the R variable qbx and placed it into a JMP variable qbx
qbx = R Get( qbx );

// Get the R variable df and placed it into a JMP data table referenced by df
df = R Get( df );

R Get Graphics( format )

Function
Gets the last graphics object written to the R graph display window in the specified format.

Returns
A JMP picture object.

Argument
format Required. Specifies the graphics format to be used. The valid formats are as follows:
- png
- bmp
- jpeg or jpg
- tiff or tif

R Init( named_arguments )

Function
Initializes the R session.

Returns
0 if the initialization is successful; any nonzero value otherwise.

Argument
Echo(Boolean) Optional. Sends all source lines to the JMP log. This option is global. The default value is true.
R Is Connected()

Function
Determines whether a connection to R exists.

Returns
1 if connected; 0 otherwise.

Arguments
None.

R JMP Name to R Name( name )

Function
Maps the specified JMP variable name to the corresponding R variable name using R naming rules.

Argument
name The name of a JMP variable to be sent to R.

Returns
A string that contains the R name.

R Send( name, <R Name(name)>)

Function
Sends named variables from JMP to R.

Returns
0 if the send is successful; any nonzero value otherwise.

Arguments
name required. The name of a JMP variable to be sent to R.
R Name(name) Optional. You can give the variable that you send to R a different name. For example
R Send(Here:x, R Name("localx"))

For data tables only:
Selected(Boolean) optional, named, Boolean. Send only selected rows from the referenced data table to R.
Excluded(Boolean) optional, named, Boolean. Send only excluded rows from the referenced data table to R.
Labeled(Boolean) optional, named, Boolean. Send only labeled rows from the referenced data table to R.
Hidden(Boolean) optional, named, Boolean. Send only hidden rows from the referenced data table to R.
Colored(Boolean) optional, named, Boolean. Send only colored rows from the referenced data table to R.
Markered(Boolean) optional, named, Boolean. Send only markered rows from the referenced data table to R.
Row States(Boolean, <named arguments>) optional, named. Includes a Boolean argument and optional named parameters. Send row state information from the referenced data table to R by adding an additional data column named “RowState”. Multiple row states are created by adding together individual settings. The individual values are as follows:

- Selected = 1
- Excluded = 2
- Hidden = 4
- Labeled = 8
- Colored = 16
- Markered = 32

The named arguments for the Row States() argument are as follows:

Colors(Boolean) optional, named, Boolean. Sends row colors. Adds additional data column named “RowStateColor”.

Markers(Boolean) optional, named, Boolean. Sends row markers. Adds additional data column named “RowStateMarker”.

Examples
Create a matrix, assign it to X, and send the matrix to R:

\[
X = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix};
\]

rc = R Send( X );

Open a data table, assign a reference to it (dt), and send the data table, along with its current row states, to R:

\[
dt = \text{Open( "\$SAMPLE_DATA/Big Class.jmp" )};
\]

rc = R Send( dt, Row States(1) );

R Send File( "pathname", <R Name("name")>)

Function
Sends the specified data file from JMP to R.

Returns
0 if the send is successful; any nonzero value otherwise.

Arguments

- pathname required. A quoted string that contains a pathname for a file.
- R Name(name) Optional. You can give the data file that you send to R a different name.
R Submit( rCode, <named_parameters> )

Function
Submits the specified R code to the active global R connection.

Returns
0 if successful; nonzero otherwise.

Arguments
rCode  A required, quoted string that contains the R code to submit.
Expand(Boolean) An optional, Boolean, named parameter. Performs an Eval Insert() on the R code before submitting to R.
Echo(Boolean) An optional, Boolean, named parameter. Sends all source lines to the JMP log. This option is global. The default value is true.
Async(Boolean) An optional, Boolean, named parameter. If set to true (1), the submit can be canceled either by pressing the ESCAPE key, or by using this message to an R connection: rconn<<Control( Interrupt( 1 ) ). False (0) is the default value.

Example
rc = R Submit("\n  x <- rnorm(5)
  print(x)
  y <- rnorm(5)
  print(y)
  z = plot(x, y)
  ]\")");

R Submit File( pathname )

Function
Submits statements to R using a file pointed in the specified pathname.

Returns
0 if successful; nonzero otherwise.

Argument
Pathname  A quoted string that contains the pathname to the file that contains the R code to be executed.

R Term()

Function
Terminates the currently active R integration interface

Returns
1 if an active R connection exists; otherwise 0.

Arguments
None
Random Functions

Col Shuffle()

Function
  Shuffles the values randomly each time it’s evaluated.

Returns
  The last value placed into the last row.

Argument
  none

Example
  For Each Row(x:=shuffle = colshuffle());
  show(x)
  \( x = 6 \)
  In a table with 40 rows, the above script places the values 1-40 randomly into each row of the column named shuffle. All numbers appear only once. Each time the script is run, the numbers are placed in a different random order. If the value placed into the row 40 of column shuffle is 6, that number is assigned to \( x \), as in the above example.

Random Beta(alpha, beta)

Function
  Generates a pseudo-random number distributed Beta (alpha, beta)

Random Beta Binomial(n, p, delta)

Function
  Returns a random number from a beta binomial distribution for \( n \) trials with probability \( p \) and correlation \( delta \).

Random Binomial(n, p)

Function
  Returns random numbers from a binomial distribution with \( n \) trials and probability \( p \) of the event of interest occurring.

Random Cauchy(alpha, beta)

Function
  Returns a Cauchy distribution with given \( alpha \) and \( beta \).

Random Exp()

Function
  Returns a random number distributed exponentially from 0 to infinity. Equivalent to the negative log of Random Uniform.
Random Frechet(mu, sigma)
  Function
  Returns a random number from a Fréchet distribution with the location mu and scale sigma.

Random Gamma(lambda, <scale> )
  Function
  Gives a gamma distribution for given lambda and optional scale.

Random Gamma Poisson(lambda, sigma)
  Function
  Returns a random number from a gamma Poisson distribution with parameters lambda and sigma.

Random Geometric(p)
  Function
  Returns random numbers from the geometric distribution with probability p that a specific event occurs at any one trial.

Random GLog(mu, sigma, lambda)
  Function
  Returns a random number from a generalized logarithmic distribution.

Random Integer(<k>, n)
  Function
  Returns a random integer from 1 to n or from k to n.

Random Johnson Sb(gamma, delta, theta, sigma)
  Function
  Returns a random number from the Johnson Sb distribution.

Random Johnson Sl(gamma, delta, theta, <sigma=1> )
  Function
  Returns a random number from the Johnson Sl distribution.

Random Johnson Su(gamma, delta, theta, sigma)
  Function
  Returns a random number from the Johnson Su distribution.
Random LEV(mu, sigma)

Function
Returns a random number from an LEV distribution with the location \( mu \) and scale \( sigma \).

Random Logistic(mu, sigma)

Function
Returns a random number from a logistic distribution with the location \( mu \) and scale \( sigma \).

Random Loglogistic(mu, sigma)

Function
Returns a random number from a loglogistic distribution with the location \( mu \) and scale \( sigma \).

Random Lognormal(mu, sigma)

Function
Returns a Lognormal-distributed random number with location \( mu \) and scale \( sigma \).

Random Negative Binomial(r, p)

Function
Generates a negative binomial distribution for \( r \) successes with probability of success \( p \).

Random Normal( )

Function
Generates random numbers that approximate a normal distribution with mean 0 and variance 1. The normal distribution is bell shaped and symmetrical.

Random Normal Mixture(mean, std_dev, probabilities)

Function
Returns a random number from a normal mixture distribution with the specified arguments.

Arguments
- mean A vector that contains group means.
- std_dev A vector that contains the group standard deviations.
- probabilities A vector that contains the group probabilities.

Random Poisson(lambda)

Function
Generates a Poisson variate for given \( lambda \).
### Random Functions

**Random Reset(seed)**

Function

Restarts the random number sequences with *seed*.

**Random SEV(mu, sigma)**

Function

Returns a random number from an SEV distribution with the specified location *mu* and scale *sigma*.

**Random Triangular(midpoint)**

Function

Generates a triangular distribution of numbers between 0 and 1 with the *midpoint* that you specify.

**Random Uniform(<x>)**

**Random Uniform(min, max)**

Function

Generates random numbers uniformly between 0 and 1. *Random Uniform(x)* generates numbers between 0 and *x*. *Random Uniform (min, max)* generates numbers between *min* and *max*. The result is an approximately even distribution.

**Random Weibull(beta, alpha)**

Function

Returns a random number from a Weibull distribution.

**Resample Freq(<rate, <column>>)**

Function

Generates a frequency count for sampling with replacement. If no arguments are specified, the function generates a 100% resample.

**Arguments**

- **rate** Optional. Specifies the rate of resampling. The default value is 1.
- **column** Optional. If you specify column, you must also specify rate. The sample size is calculated by the rate multiplied by the sum of the specified column.

**Note**

A typical use of this generates a column with many 1s, some 0s, some 2s, and so forth, corresponding to which rows were randomly assigned any of *n* randomly selected rows.

A typical use of this with an existing frequency column produces a new frequency column whose values are similar to the old frequency column (have that expected value), but vary somewhat due to random selection at the rates corresponding to the old frequency column.

**See Also**

For an example, see the JSL Functions Index entry.
Row Functions

**Count(from, to, step, times)**

**Function**
Used for column formulas. Creates row by row the values beginning with the *from* value and ending with the *to* value. The number of *steps* specifies the number of values in the list between and including the *from* and *to* values. Each value determined by the first three arguments of the count function occurs consecutively the number of *times* that you specify. When the *to* value is reached, count starts over at the *from* value. If the *from* and *to* arguments are data table column names, *count* takes the values from the first row only. Values in subsequent rows are ignored.

**Returns**
The last value.

**Arguments**
- **from** Number, column reference, or expression. *Count* starts counting with this value.
- **to** Number, column reference, or expression. *Count* stops counting with this value.
- **step** Number or expression. Specifies the number of steps to use to count between *from* and *to*, inclusive.
- **times** Number or expression. Specifies the number of times each value is repeated before the next step.

**Examples**
- For Each Row(:colname[row()] = count(0, 6, 3, 1))
  //The rows in the column named *colname* are filled with the series 0, 3, 6, 0, ...
  //until all rows are filled.

- For Each Row(:colname[row()] = count(0, 6, 3, 2))
  //The rows in the column named *colname* are filled with the series 0, 0, 3, 3,
  //6, 6, 0, ... until all rows are filled.

**Note**
*Count()* is dependent on *Row()*, and is therefore mainly useful in column formulas.

**Dif(col, n)**

**Function**
Calculates the difference of the value of the column *col* in the current row and the value *n* rows previous to the current row.

**Returns**
The difference.

**Arguments**
- **col** A column name (for example, :age).
- **n** A number.
Lag(col, n)

Function
Returns for each row the value of the column \( n \) rows previous.

\[ \text{N Row}(dt); \text{NRow}(\text{matrix}) \]
\[ \text{N Rows}(dt); \text{NRows}(\text{matrix}) \]

Function
Returns the number of rows in the data table given by \( dt \) or in the \( \text{matrix} \).

Row()
Row() = y

Function
Returns or sets the current row number. No argument is expected.

Sequence(from, to, <stepSize>, <repeatTimes>)

Function
Produces an arithmetic sequence of numbers across the rows in a data table. The \( \text{stepSize} \) and \( \text{repeatTimes} \) arguments are optional, and the default value for both is 1.

Subscript(a, b, c)
list[i]
matrix[b, c]

Function
Subscripts for lists extract the \( i \)th item from the \( \text{list} \), or the \( b \)th row and the \( c \)th column from a \( \text{matrix} \).

**Row State Functions**

As Row State(i)

Function
Converts \( i \) into a row state value.

Returns
A row state from the \( i \) given.

Argument
\( i \) an integer
Color Of(rowstate)

Function
Returns or sets the color index.

Returns
The color index of rowstate.

Argument
rowstate a row state argument

Example
Set the color of the fifth row to red.
colorof(rowstate(5)) = 3

Color State(i)

Function
Returns a row state with the color index of i.

Returns
A row state.

Argument
i index for a JMP color

Combine States(rowstate, rowstate, ...)

Function
Generates a row state combination from two or more row state arguments.

Returns
A single numeric representation of the combined row states.

Arguments
rowstate Two or more row states.

Excluded(rowstate)

Function
Returns or sets an excluded index.

Returns
The excluded attribute, 0 or 1.

Argument
rowstate One or more row states.

Excluded State(num)

Function
Returns a row state for exclusion from the num given.
Hidden(rowstate)
   Function
   Returns or sets the hidden index.

Hidden State(num)
   Function
   Returns a row state for hiding from the num given.

Hue State(num)
   Function
   Returns a hue state from the num given.

Labeled(rowstate)
   Function
   Returns or sets the labeled index.

Labeled State(num)
   Function
   Returns a labeled state from the num given.

Marker Of(rowstate)
   Function
   Returns or sets the marker index of a row state.

Marker State(num)
   Function
   Returns a marker state from the num given.

Row State(<n>)
   Function
   Returns the row state changed from the initial condition of the active row or the nth row.

Selected(rowstate)
   Function
   Returns or sets the selected index.
Selected State(num)

  Function
  Returns a selected state from the num given.

Shade State(num)

  Function
  The Shade State function assigns 5 shade levels to a color or hue.

### SAS Integration Functions

**Current Metadata Connection()**

  Function
  Returns the active SAS metadata connection, if any, as a scriptable object.

**Current SAS Connection()**

  Function
  Gets the active global SAS server connection, if any, as a scriptable object.

**Get SAS Version Preference()**

  Function
  Returns the SAS version selected in Preferences as a string.

**Meta Connect(<machine, port>, <authDomain>, <username>, <password>, named arguments)**

**Meta Connect(<Profile(profile name)>, <Password(password)>, named arguments)**

  Function
  Connects to a SAS Metadata Server. If no arguments are specified, an empty connection window appears. If some arguments are specified, a window partially filled in with the argument values appears. If all arguments are specified, the connection is made and no window appears.

  Returns
  1 if connection is successful, 0 if not.

  Arguments
  - `machine` Optional: quoted string that contains the DNS name of the machine.
  - `port` Required if machine is specified. Quoted string or integer that contains the port on which the metadata server listens.
  - `authDomain` Optional: quoted string that contains the authentication domain for the credentials supplied. Not necessary unless `username` and `password` are included.
username  Optional: quoted string that contains the user name for the connection.
password  Optional: quoted string that contains the password for the connection.

Named Arguments
All named arguments are optional.
Profile("profile_name") A quoted string that contains the name of the metadata server connection profile from which connection information should be retrieved.
Password("password") A quoted string that contains the password for the specified profile name.
CheckPreferenceOnly(0|1) If specified, Meta Connect returns the status of the I want to connect to a SAS Metadata Server option in the SAS Integration page of JMP Preferences. If that box is checked, Meta Connect returns 1; if not, 0.
Repository("string") Takes a quoted string that contains the name of the repository to which to connect.
ProfileLookup(0|1) If machine and port are specified rather than a profile name, and ProfileLookup is specified, an attempt is made to find a metadata server connection profile with a machine name and port matching those provided. If one is found, other connection information (such as authentication domain, user name, and password) is obtained from that profile.
Prompt(Always|Never|IfNeeded) Takes one of the keywords Always (always prompt before attempting to connect), Never (never prompt, just fail), or If Needed (the default; prompt if connection with the given parameters fails).

Notes
If no arguments are included and if no profile is saved, the Connect to SAS Metadata Server window appears.

Meta Create Profile(profile, <named arguments>)

Function
Creates a metadata server connection profile and adds it to the current user's set of saved metadata server connection profiles.

Returns
1 if profile was successfully created, otherwise 0.

Arguments
profile A quoted string that contains the name of the created profile. If a profile by the given name already exists, MetaCreateProfile fails unless Replace is specified.

Named Arguments
The following named arguments are all optional.
HostName(name) A quoted string that contains the name of the host computer running the SAS Metadata Server that this profile will connect to.
Port(n) The port number (n) that the SAS Metadata Server is listening for connections on.
AuthenticationDomain("domain") | AuthDomain("domain") A quoted string that sets the authentication domain to use for the connection.
Description("desc") | Desc("desc") A quoted string that sets a description for this profile.
Password("password") A quoted string that contains the password to store in this profile.
Replace(0|1) If name matches a profile that already exists, Replace must be specified for the existing profile to be replaced by the one provided. The default value is False (0).

Repository("repository") A quoted string that contains the name of the repository in the SAS Metadata Server that this profile will connect to. This option is valid only for connections to SAS 9.1.3 Metadata Servers.

UserName("username") A quoted string that contains the user name that this profile uses to connect to the SAS Metadata Server.

UseSingleSignOn(0|1) If specified, this profile attempts to use Single Sign-On (currently also known as Integrated Windows Authentication) to connect to the SAS Metadata Server. This option is valid only for connecting to SAS 9.2 Metadata Servers. If UseSingleSignOn is True(1), UserName and Password cannot be specified. The default value is False (0).

Meta Delete Profile(name)
Function
Deletes the named metadata server connection profile from the current user's set of saved metadata server connection profiles

Returns
1 if profile was successfully deleted, otherwise 0.

Argument
name A quoted string that contains the name of the profile to delete.

Meta Disconnect()
Function
Disconnect the current SAS Metadata Server connection, if any.

Returns
Void.

Meta Get Repositories()
Function
Gets a list of the repositories available on the current SAS Metadata Server connection.

Returns
A list of repository names as strings.

Meta Get Servers()
Function
Get a list of the SAS Servers that are registered in the SAS Metadata Repository to which the session is currently connected.

Returns
A list of server names as strings.
Meta Get Stored Process(path)

Function
Get a stored process object from the currently connected SAS Metadata Repository.

Returns
Stored Process scriptable object.

Arguments
path Quoted string that is the path to the stored process in metadata, starting at the BIP Tree.

Meta Is Connected()

Function
Determines whether a current connection to a SAS Metadata Server exists.

Returns
1 if a connection exists; 0 otherwise.

Arguments
None.

Meta Set Repository(repositoryName)

Function
Set the SAS Metadata Repository to use for metadata searches.

Returns
1 if setting the repository was successful, 0 otherwise.

Arguments
repositoryName Quoted string that contains the name of the repository to make current.

SAS Assign Lib Refs(libref, path, <engine>, <engine options>)

Function
Assign a SAS libref on the active global SAS server connection.

Returns
1 if successful, 0 otherwise.

Arguments
libref Quoted string that contains a library reference (8-character maximum) to assign.
path Quoted string that contains the full path on the SAS server to the library being assigned.
engine Optional, quoted string that contains the engine for the SAS server to use when accessing members of this library.
engine options Optional, quoted string that contains the options needed for the engine being used.
SAS Connect(name>, <port>, <named arguments>)

Function
Connect to a local, remote, or logical SAS server.

Returns
SAS Server scriptable object.

Arguments
name Optional: quoted string that can contain a physical machine name or the name of a
metadata-defined (logical) server. In the first case, the port must be provided. In the second case, a
port must not be provided. If neither name nor port are included, and JMP is running on Windows, a
connection to SAS on the local machine (via COM) is attempted, and all named parameters are
ignored.

port Optional: quoted string or integer. If name is a physical machine name, this is the port on that
machine to connect to. If name is a metadata-defined (logical) server, port must not be included.

Named Arguments
All the named arguments are optional.

UserName(name) A quoted string that contains the user name for the connection.

Password(password) A quoted string that contains the password for the connection.

ReplaceGlobalConnection(0|1) A Boolean. The default value is True. If True, and a successful
SAS server connection is made, this connection replaces the active SAS connection that becomes the
target of other global SAS JSL function calls. If False, the global SAS connection is not changed, and
the returned SASServer scriptable object should be used to send messages to this server connection.

ShowDialog(0|1) A Boolean. The default value is False. If True, other arguments (except
ReplaceGlobalConnection) are ignored and the SAS Server Connection window appears. This
provides the JSL programmer a way to open the SAS Connect window.

Prompt(Always|Never|IfNeeded) A Keyword. Always means always prompt before attempting
to connect. Never means never prompt even if the connection attempt fails (just fail and send an
error message to the log), and IfNeeded (the default value) means prompt if the attempt to connect
with the given parameters fails (or is not possible with the information given).

SAS Deassign Lib Refs(libref)

Function
De-assign a SAS libref on the active global SAS server connection.

Returns
1 if successful; 0 otherwise.

Arguments
libref Quoted string that contains the library reference to de-assign.
**SAS Disconnect()**

Function
 Disconnect the active global SAS connection, if any.

Returns
 Void.

Arguments
 None.

**SAS Export Data(dt, "library", "dataset", <named_arguments>)**

Function
 Exports a JMP data table to a SAS data set in a library on the active global SAS server connection.

Returns
 1 if the data table was exported successfully; 0 otherwise.

Arguments
 dt  data table or a reference to a data table.
 "library"  the library to which to export the data table.
 "dataset"  the name of the new SAS data set.

Named Arguments
 All the named arguments are optional.
 Columns(list)|Columns(col1, col2, ...)  A list of columns or a comma-separated list of columns.
 Password("password")  A string that contains the password to serve as the READ, WRITE, and ALTER password for the exported SAS data set. If the exported data set is replacing an existing data set with an ALTER password, this password is used as the ALTER password for overwriting the data set. If Password is specified, values for ReadPassword, WritePassword, and AlterPassword are ignored.
 ReadPassword("password")  A string that contains the password to serve as the READ password for the exported SAS data set.
 WritePassword("password")  A string that contains the password to serve as the WRITE password for the exported SAS data set.
 AlterPassword("password")  A string that contains the password to serve as the ALTER password for the exported SAS data set. If the exported data set is replacing an existing data set with an ALTER password, this password is used as the ALTER password for overwriting the data set.
 PreserveSASColumnNames(0|1)  A Boolean. If true and the JMP data table originally came from SAS, the original SAS column names are used in the exported SAS data set. The default value is False.
 PreserveSASFormats(0|1)  A Boolean. If true and the JMP data table originally came from SAS, the original SAS formats and informats are applied to the columns in the exported SAS data set. The default value is True.
ReplaceExisting(0|1) A Boolean. If true, an existing SAS data set with the specified name in the specified library is replaced by the exported SAS data set. If false, a data set with the specified name already exists in the specified library; the export is stopped. The default value is false.

HonorExcludedRows(0|1) A Boolean. If true, any rows in the JMP data table that are marked as excluded are not exported. The default value is false.

Note
Information about the export is sent to the log.

**SAS Get Data Sets(libref)**

Function
Returns a list of the data sets defined in a SAS library.

Returns
List of strings.

Arguments
libref Quoted string that contains the SAS libref or friendly library name associated with the library for which the list of defined SAS data sets is returned.

**SAS Get File(source, dest)**

Function
Get a file from the active global SAS server connection.

Returns
1 if successful, 0 otherwise.

Arguments
source Quoted string that contains the full path of file on the server to be downloaded to the client machine.

dest Quoted string that contains the full path on the client machine for where to put the copy of the file downloaded from the server.

**SAS Get File Names("fileref")**

Function
Get a list of filenames found in the given fileref on the active global SAS server connection.

Returns
List of strings.

Arguments
"fileref" Quoted string that contains the name of the fileref from which to retrieve filenames.
SAS Get File Names In Path("path")

Function
Get a list of filenames found in the given path on the active global SAS server connection.

Returns
List of strings.

Arguments
"path" Quoted string that contains the directory path on the server from which to retrieve filenames.

SAS Get File Refs()

Function
Get a list of the currently defined SAS filerefs on the active global SAS server connection.

Returns
List of two lists. The first list is a list of quoted strings of fileref names. The second is a corresponding list of quoted strings of physical names.

SAS Get Lib Refs(<named arguments>)

Function
Get a list of the currently defined SAS librefs on the current global SAS server connection.

Returns
List of strings.

Named Arguments
Friendly Names(0|1) Optional, Boolean. If True, then for any libraries that have friendly names (metadata-defined libraries), the friendly name is returned rather than the 8-character libref.

SAS Get Log()

Function
Retrieve the SAS Log from the active global SAS server connection.

Returns
A string.

SAS Get Output()

Function
Retrieve the listing output from the last submission of SAS code to the current global SAS server connection.

Returns
A string.
SAS Get Results()

Function
Retrieve the results of the previous SAS Submit as a scriptable object, which allows significant flexibility in what to do with the results.

Returns
A SAS Results Scriptable object.

SAS Get Var Names(string, <dataset>, <password("password")>)

Function
Retrieves the variable names contained in the specified data set on the current global SAS server connection.

Returns
List of strings.

Arguments

string
A quoted string that contains one of the following:

• The name of the SAS Library containing the SAS data set to be imported. In that case, the dataset name parameter is required.
• The full member name of the SAS data set to be imported, in the form “libname.membername”.
• The SAS Folders tree path to a logical SAS data table to be imported. This option requires a connection to a SAS 9.2 Metadata Server.

dataset
Optional: quoted string that contains the name of the data set from which to retrieve variable names.

password("password")
Optional, quoted string that contains the read password for the data set. If this is not provided and the data set has a read password, the user is prompted to enter it.

SAS Import Data(string, <dataset>, <named arguments>)

Function
Import a SAS data set from the active global SAS server connection into a JMP table.

Returns
JMP Data Table object.

Arguments

string
A quoted string that contains one of the following:

• The name of the SAS Library containing the SAS data set to be imported. In that case, the dataset name parameter is required.
• The full member name of the SAS data set to be imported, in the form “libname.membername”.
• The SAS Folders tree path to a logical SAS data table to be imported. This option requires a connection to a SAS 9.2 Metadata Server.
dataset  Optional: quoted string that contains the name of the data set.

Named Arguments

All named arguments are optional.

Columns(1ist)|Columns(col1, col2, ...) A quoted string list or multiple strings that contain the names of columns to include in the import.

ConvertCustomFormats(0|1) The default value is True (1). If True and custom formats are found in the SAS data set being imported, an attempt is made to convert the SAS custom formats to JMP value labels for those columns.

Invisible(0|1) The default value is False (0). If true, the JMP data table is created as a hidden table. Hidden data tables that never become visible or linked to an analysis remain in memory until they are explicitly closed, reducing the amount of memory that is available to JMP. To explicitly close the hidden data table, call Close(dt), where dt is the data table reference returned by SASImportData.

Where("filter") A quoted string that contains the filter to use when importing data, as in Where("salary<50000").

Password("password") A quoted string that contains the read password for the data set. If this is not provided and the data set has a read password, the user is prompted to enter it.

UseLabelsForVarNames(0|1) If True, the labels from the SAS data set become the column names in the resulting JMP table. If False, the variable names from the SAS data set become the column names in the JMP table. The default value is False.

Sample(named arguments) optional, named. Allows a random sample of the SAS data set to be imported into JMP. If both Where and Sample are specified, the WHERE clause is used to filter the SAS data set first, and then a random sample of the resulting rows is taken based on the parameters supplied to Sample. Note that Sample uses PROC SURVEYSELECT on the SAS server, which is available only if the SAS/STAT package is licensed and installed on that server. The documentation for PROC SURVEYSELECT might be helpful in understanding how sampling is performed. By default (if no parameters are supplied), a 5% simple random sample is taken. Available parameters (all optional) to Sample are as follows:

- Simple | Unrestricted: If Simple is specified, sampling is performed without replacement. If Unrestricted is specified, sampling is performed with replacement. These two options are mutually exclusive and only one can be specified.

- SampleSize(int) | N(int): Total number of rows for the sample, or number of rows per strata level for stratified sampling

- SampleRate(number) | Rate(number) | Percent(number): Specifies the sampling rate. For stratified sampling, the rate is applied to each strata level. Note that the supplied value is assumed to be a percentage, so SampleRate(3.5) means a 3.5% sampling rate.

- Strata({col1, col2, ...}) | Strata(col1, col2, ...): Perform stratified random sampling using the column names supplied as Strata variables.

- NMin(int): Minimum number of rows (either overall or per strata level for stratified sampling) to return. Only applies to rate-based sampling.

- NMax(int): Maximum number of rows (either overall or per strata level for stratified sampling) to return. Only applies to rate-based sampling.
JSL Syntax Reference
Appendix A

SAS Integration Functions

- **Seed(int)**: Number to use as the seed for sampling. Useful for replicating the same sample. By default, the seed is a random number based on time of day. See PROC SURVEYSELECT documentation for more information.

- **OutputHits(0|1)**: Boolean; the default value is false. When doing Unrestricted sampling, if the same row of the input data set is selected more than once, by default that row still appears only once in the resulting JMP data table, with the NumberHits column indicating the number of times that the row was selected. Setting OutputHits to true causes an input row that is selected multiple times to appear multiple times in the resulting JMP data table.

- **SelectAll(0|1)**: Boolean, the default value is true. If SelectAll is true, PROC SURVEYSELECT selects all stratum rows whenever the stratum sample size exceeds the total number of rows in the stratum. If SelectAll is false and PROC SURVEYSELECT finds a case where the stratum sample size exceeds the total number of rows in a given stratum, an error results and sampling fails. SelectAll only applies to Simple random sampling.

- **SQLTableVariable(0|1)**: If True, an SQL table variable is created in the resulting JMP table that shows the SQL that was submitted to SAS to obtain the data. If False, the SQL table variable is not created. The default value is True. If an SQL table variable is created and the data set required a read password, the password is masked.

**SAS Import Query(″sqlquery″, <named arguments>)**

Function
Execute the requested SQL query on the current global SAS server connection, importing the results into a JMP data table.

Returns
JMP Data Table object.

Arguments

"sqlquery"  Quoted string that contains the SQL query to perform and from which to import the result.

Named Arguments
All named arguments are optional.

- **ConvertCustomFormats(0|1)**: The default value is true. If true and custom formats are found in the SAS data set being imported, an attempt is made to convert the SAS custom formats to JMP value labels for those columns.

- **Invisible(0|1)**: The default value is false. If true, the JMP data table is created hidden. Hidden data tables that never become visible or linked to an analysis remain in memory until they are explicitly closed, reducing the amount of memory that is available to JMP. To explicitly close the hidden data table, call Close(dt), where dt is the data table reference returned by SAS Import Query.

- **UseLabelsForVarNames(0|1)**: The default value is true. If True, the labels from the SAS data set become the column names in the resulting JMP table. If False, the variable names from the SAS data set become the column names in the JMP table.

- **SQLTableVariable(0|1)**: The default value is true. If True, an SQL table variable is created in the resulting JMP table that shows the SQL that was submitted to SAS to obtain the data. If False, the
SQL table variable is not created. If an SQL table variable is created and the data set required a read password, the password is masked.

**SAS Is Connected()**

**Function**

Discovers whether there is an active global SAS server connection.

**Returns**

1 if an active global SAS connection exists, 0 otherwise.

**SAS Load Text File("path")**

**Function**

Download the file specified in path from the active global SAS server connection and retrieve its contents as a string.

**Returns**

String.

**Arguments**

"path"  Quoted string that contains the full path on the server of the file to download and retrieve the contents as a string.

**SAS Name("name")**

**SAS Name({list of names})**

**Function**

Converts JMP variable names to SAS variable names by changing special characters and blanks to underscores and various other transformations to produce a valid SAS name.

**Returns**

A string that contains one or more valid SAS names, separated by spaces.

**Argument**

"name"  A quoted string that represents a JMP variable name; or a list of quoted JMP variable names.

**SAS Open For Var Names("path")**

**Function**

Opens a SAS data set only to obtain the names of its variables, returning those names as a list of strings.

**Returns**

A list of variable names in the file.

**Argument**

"path"  A quoted string that is a pathname of a SAS data set.
SAS Send File(source, dest)

Function
Send a file from the client machine to the active global SAS server connection.

Returns
1 if successful, 0 otherwise.

Arguments
source Quoted string that contains the full path of the file on the client machine to be uploaded to the server.
dest Quoted string that contains the full path on the server that receives the file uploaded from the client machine.

SAS Submit("sas code", <named arguments>)

Function
Submit some SAS code to the active global SAS server connection.

Returns
1 if successful, 0 otherwise.

Arguments
sasCode Quoted string that contains the SAS code to submit.

Named Arguments
All named arguments are optional.

Async(0|1) A Boolean. If True (1), the submit occurs asynchronously (in the background). Use the Get Submit Status() message on the SAS Server Scriptable Object to determine the status of the submit. The default value is False (0).

ConvertCustomFormats(0|1) A Boolean. When SAS data sets generated by submitted SAS code are imported into JMP after the submit completes (see Open Output Datasets), the value of ConvertCustomFormats determines whether an attempt is made to convert any custom formats found on columns in the SAS data to JMP value labels. The default value is True (1).

DeclareMacros(var1, var2, ...) JSL variable names. Provides a simple way to pass the values of JSL variables to SAS as macro variables. Each JSL variable specified should evaluate to a string or numeric value. Fully qualified JSL variables names, only the variable name is sent to SAS. For example, namespace:variable_name becomes variable_name in SAS.

GetSASLog(<Boolean|OnError>, <JMPLog|Window>) A Boolean. If no arguments are supplied, the SAS Log is retrieved and displayed in the location indicated in SAS Integration Preferences. The first argument to GetSASLog can be either a Boolean value or the keyword OnError. If a Boolean value is supplied, true means display the SAS Log, and false means not to display it. OnError instructs JMP to only show the SAS Log if an error occurred in the submit. The second argument to GetSASLog tells JMP where to display the SAS Log. If JMPLog is specified, the SAS Log is appended to the JMP Log. If Window is specified, the SAS Log is opened in a separate window.

GraphicsDevice(string) or GDevice(string) A string that specifies a value for the GDEVICE SAS option to be used for graphics generated by the submitted SAS code. The value must be a valid SAS graphics device. The default value is determined in Preferences.
NoOutputWindow  A Boolean. If True, the SAS Output window containing the listing output from the submission does not appear. The default value is False.

ODS(0|1)  If true, additional SAS code is submitted causing ODS results to be generated for the submitted SAS code. The default value is determined in Preferences.

ODSFormat(string)  A quoted string that determines the format of generated ODS results. Valid values are “HTML”, “RTF”, and “PDF”. The default value is determined in Preferences.

ODSGraphics(0|1)  If true, ODS statistical graphics are generated for the submitted SAS code. Setting ODSGraphics to true causes ODS to also be set to true. The default value is determined in Preferences.

ODSStyle(string)  A quoted string that specifies the ODS Style to use when generating ODS results. String must be a valid SAS Style. The default value is determined in Preferences.

ODSStyleSheet(path)  A quoted string that specifies a local CSS style sheet to use when formatting generated ODS results. Path must be a path to a CSS file valid for the client machine (the machine running JMP). The default value is determined in Preferences.

OnSubmitComplete(script)  A quoted string that specifies a JSL script that should be run when the submit completes. This is especially useful for asynchronous submits. If script is the name of a defined JSL function, that function is executed, with the SAS Server scriptable object passed as the first parameter.

OpenODSResults(0|1)  If true, ODS results that are generated by the submitted SAS code (due to ODS being true) are automatically opened after the submit completes. The default value is True (1).

OpenOutputDatasets(<All|None|dataset1, dataset2, ...>)  JMP detects when submitted SAS code creates new SAS data sets. OpenOutputDatasets (which can be abbreviated OutData) determines what, if anything, is done with those data sets with the SAS Submit completes. If All is specified, all data sets generated by the SAS code are imported into JMP when the SAS Submit completes. If None is specified, none of the generated data sets are imported. If there are specific data sets known to be generated by the submitted SAS code that you want to be imported into JMP when the SAS submit completes, you can alternative provide their names, and only the requested data sets are imported. The default value is determined in Preferences.

Title(string)  A quoted string that specifies the window title to use for the window that displays ODS output from the submit.

SAS Submit File(filename, <named arguments>)

Function
Submit a SAS code file to the active global SAS server connection.

Returns
1 if successful; 0 otherwise.

Arguments
filename  Quoted string that contains the name of file containing SAS code to submit.

Named Arguments
Same as for SAS Submit.
Statistical Functions

Col Maximum(name)

Function
Calculates the maximum value across all rows of the specified column.

Returns
The maximum value that appears in the column.

Argument
name a column name

Col Mean()

Function
Calculates the mean across all rows of the specified column.

Returns
The mean of the column.

Argument
name a column name

Col Minimum(name)

Function
Calculates the minimum value across all rows of the specified column.

Returns
The minimum value that appears in the column.

Argument
name a column name

Col N Missing(name)

Function
Calculates the number of missing values across all rows of the specified column.

Returns
The number of missing values in the column.

Argument
name a column name.
Col Number(name)

Function
Calculates the number of nonmissing values across all rows of the specified column.

Returns
The number of nonmissing values in the column.

Argument
name a column name.

Col Quantile(name, q)

Function
Calculates the specified quantile q of across all rows of the specified column.

Returns
The value of the quantile.

Argument
name a column name.
q a number between 0 and 1 that represents a quantile.

Example
Using Big Class.jmp:

colquantile(:height, .5)
63
63 is the 50th percentile, or the median, of all rows in the height column.

Col Rank(column)

Function
Ranks each row's value, from 1 for the lowest value to the number of columns for the highest value. Ties are broken arbitrarily.

Argument
column The column to be ranked.

Col Standardize(name)

Function
Calculates the column mean divided by the standard deviation across all rows of the specified column.

Returns
The standardized mean.

Argument
name a column name.

Notes
Standardizing centers the variable by its sample standard deviation. Thus, the following commands are equivalent:
dt << New Column("stdht", Formula(Col Standardize(height)));

dt << New Column("stdht2", Formula((height - Col Mean(height)) / Col Std Dev(height)));

**Col Std Dev(name)**

**Function**
Calculates the standard deviation across rows in a column.

**Returns**
The standard deviation.

**Argument**
name a column name.

**Col Sum(name)**

**Function**
Calculates the sum across rows in a column.

**Returns**
The sum.

**Argument**
name a column name.

**Desirability(yVector, desireVector, y)**

**Function**
Fits a function to go through the three points, suitable for defining the desirability of a set of response variables (y’s). yVector and desireVector are matrices with three values, corresponding to the three points defining the desirability function. The actual function depends on whether the desire values are in the shape of a larger-is-better, smaller-is-better, target, or antitarget.

**Returns**
The desirability function.

**Arguments**
yVector Three input values.
desireVector the corresponding three desirability values.
y the value of which to calculate the desirability.

**Max()**
See “Maximum(var1, var2, ...),” p. 571.
Maximum(var1, var2, ...)  
Max(var1, var2, ...)  

Function  
Returns the maximum value of the arguments or of the values within a single matrix or list argument. If multiple arguments are specified, they must be either all numeric values or all strings.

Mean(var1, var2, ...)  

Function  
Rowwise mean of the variables specified.

Min  
See “Minimum(var1, var2,...),” p. 571.

Minimum(var1, var2, ...)  
Min(var1, var2, ...)  

Function  
Returns the minimum value of the arguments or of the values within a single matrix or list argument. If multiple arguments are specified, they must be either all numeric values or all strings.

N Missing(expression)  

Function  
Rowwise number of missing values in variables specified.

Number(var1, var2, ...)  

Function  
Rowwise number of nonmissing values in variables specified.

Product(i=initialValue, limitValue, bodyExpr)  

Function  
Multiplies the results of bodyExpr over all i until the limitValue and returns a single product.

Quantile(p, arguments)  

Function  
Quantile of the arguments, where p is a value between 0 and 1, and the arguments are numbers, lists of numbers, or matrices.
Std Dev(var1, var2, ...)

Function
Rowwise standard deviation of the variables specified.

Sum(var1, var2, ...)

Function
Rowwise sum of the variables specified.

Summation(init, limitvalue, body)

Function
Summation sums the results of the body statement(s) over all i to return a single value.

Transcendental Functions

Arrhenius(n)

Function
Converts the temperature n to the value of explanatory variable in Arrhenius model.

Returns
11605/(n+273.15)

Argument
n  Temperature in Celsius.

Notes
This is frequently used as a transformation.

Arrhenius Inv(n)

Function
The inverse of the Arrhenius function. Converts the value n to the temperature in Celsius.

Returns
11605/(n-273.15)

Argument
n  The value of the converted explanatory variable in Arrhenius model.

Notes
This is frequently used as a transformation.
**Beta(a, b)**

**Function**
\[ \Gamma(a)\Gamma(b) \]
\[ \Gamma(a + b) \]

**Returns**
Returns the beta function.

**Arguments**
a, b numbers

---

**Digamma(n)**

**Function**
The derivative of the log of the gamma function (\( \text{LGamma} \)).

**Returns**
The digamma function evaluated at \( n \).

**Argument**
n A number

---

**Exp(a)**

**Function**
Raises e to the power \( a \).

**Returns**
\( e^a \).

**Argument**
a A number

**Equivalent Expression**
e()^a

---

**Factorial(n)**

**Function**
Multiplies all numbers 1 through \( n \), inclusive

**Returns**
The product.

**Arguments**
\( n \) Any integer

**Notes**
One and only one argument must be specified.
Gamma(t, <limit>)

Function
The gamma function of x, or for each row if x is a column:
\[ \Gamma(t) = \int_{0}^{\infty} x^{t-1} e^{-x} dx \]

Returns
The gamma.

Note
Gamma(t, limit) is the same integral as Gamma(t) but with the limit of integration that is defined instead of infinity.

Arguments
t a number or a column
limit optional limit. The default is infinity.

LGamma(t)

Function
Returns the log gamma function for t, which is the natural log of gamma.

Ln()

See “Log(n, <base>),” p. 574.

Log(n, <base>)

Ln(n)

Function
Returns the natural logarithm (base e logarithm) of n. An optional second argument lets you specify a different base. For example, Log(n, 3) for the base 3 logarithm of n. The Log argument can be any numeric expression. The expression Log(e()) evaluates as 1, and Log(32, 2) is 5.

Log10(n)

Function
Returns the common (base 10) logarithm of n.

Log1P(n)

Function
Same as Log(1 + x), except that it is more accurate when x is very small.
Logit(p)

Function
Returns \( \log(p/(1-p)) \).

N Choose K(n, k)

Function
This function returns the number of \( n \) things taken \( k \) at a time ("\( n \) choose \( k \)") and is computed in the standard way using factorials, as \( n!/(k!(n-k)!\). For example, \( \text{NChooseK}(5, 2) \) evaluates as 10.

Note
This is implemented internally in JMP using \( \Gamma \) functions. The result is not always an integer.

Root(n, <r>)

Function
Returns the \( r \)th root of \( n \), where \( r \) defaults to 2 for square root.

Scheffe Cubic(x1, x2)

Function
Returns \( x1*x2*(x1-x2) \). This function supports notation for cubic mixture models.

Squish(expr)

Function
Equivalent to \( \text{Squash}(-expr) \), or \( 1/(1 + e^{-expr}) \).

Trigamma()

Function
Returns the trigamma function evaluated at \( n \). The trigamma function is the derivative of the digamma function.

Trigonometric Functions

JMP’s trigonometric functions expect all angle arguments in radians.
ArcCosH(x)

Function
Inverse hyperbolic cosine.

Returns
The inverse hyperbolic cosine of x.

Argument
x Any number, numeric variable, or numeric expression.

ArcCosine(x)

ArCos(x)

Function
Inverse cosine.

Returns
The inverse cosine of x, an angle in radians.

Argument
x Any number, numeric variable, or numeric expression.

ArcSine(x)

ArSin(x)

Function
Inverse sine.

Returns
The inverse sine of x, an angle in radians.

Argument
x Any number, numeric variable, or numeric expression.

ArcSinH(x)

Function
Inverse hyperbolic sine.

Returns
The inverse hyperbolic sine of x.

Argument
x Any number, numeric variable, or numeric expression.
ArcTangent(x)
ArcTan(x)
ATan(x)

Function
Inverse tangent.

Returns
The inverse tangent of x, an angle in radians.

Argument
x Any number, numeric variable, or numeric expression.

ArcTanH(x)

Function
Inverse hyperbolic tangent.

Returns
The inverse hyperbolic tangent of x.

Argument
x Any number, numeric variable, or numeric expression.

CosH(x)

Function
Hyperbolic cosine.

Returns
The hyperbolic cosine of x.

Argument
x Any number, numeric variable, or numeric expression.

Cosine(x)
Cos(x)

Function
Cosine.

Returns
The cosine of x.

Argument
x Any number, numeric variable, or numeric expression. The angle in radians.
Utility Functions

Sine(expr)
Sin(expr)

Function
Returns the sine.

SinH(expr)

Function
Returns the hyperbolic sine.

Tangent(expr)
Tan(expr)

Function
Returns the tangent of an argument given in radians.

TanH(expr)

Function
Returns the hyperbolic tangent of its argument.

Utility Functions

Arg(expr, i)
Arg Expr(expr, i)

Function
Finds the argument numbered by $i$ within the given expression.

Returns
The $i$th argument within the expression $expr$.
Empty() if that argument does not exist or is not specified.

Arguments
expr an expression defined previously in the JSL script.
i an integer denoting which argument to return.

Notes
Arg Expr() was deprecated in a previous release of JMP. Use Arg() instead.
ARIMA Forecast(column, length, model, estimates, from, to)

Function
Determines the forecasted values for the specified rows of the specified column using the specified model and estimates.

Returns
A vector of forecasted values for column within the range defined by from and to.

Arguments
- column A data table column.
- length Number of rows within the column to use.
- model Messages for Time Series model options.
- estimates A list of named values that matches the messages sent to ARIMA Forecast(). If you perform an ARIMA Forecast and save the script, the estimates are part of the script.
- from, to Define the range of values. Typically, from is between 1 and to, inclusive. If from is less than or equal to 0, and if from is less than or equal to to, the results include filtered predictions.

As Column(name)
As Column(dt, name) :name
dt:name

Function
This scoping operator forces name to be evaluated as a data table column in the current data table (or the table given by the optional data table reference argument, dt) rather than as a global variable.

Arguments
- name Variable name.
- dt The data table reference

Note
:name refers to a column name in the current data table. You can also specify which data table to refer to by use dt:name.

As Date(x)

Function
Formats the number or expression x so that it shows as a date when displayed in a text window.

Returns
A date that is calculated from the number or expression provided.

Argument
x Number or expression.

See Also
Section on Date-Time formats in the Scripting Guide.
As Global(name)

::name

Function
This scoping operator forces name to be evaluated as a global variable rather than as a data table column.

Arguments
name Variable name.

As List(matrix)

Function
Converts a matrix into a list. Multi-column matrices are converted to a list of row lists.

Returns
A list.

Argument
matrix Any matrix.

As Name(string)

Function
Evaluates argument as a string and changes it into a name.

Returns
A name.

As Namespace(name)

Function
Accesses the specified namespace. An error is thrown if no such namespace exists.

Returns
The namespace.

Arguments
name Unquoted name of a defined namespace.

As Scoped(namespace, variable)

namespace:variable

Function
Accesses the specified variable within the specified namespace.

Returns
The value of the variable, or an error the scoped variable is not found.

Arguments
namespace The name of a defined namespace.
variable A variable defined within namespace.
**As Table(matrix, <matrix 2, ...>, < <<invisible >, < <<Column Names(list) >)**

**Function**
Creates a new data tables from the `matrix`.

**Returns**
The new data table.

**Argument**
matrix Any matrix.
<<invisible Creates an invisible data table.
<<Column Names(list) The list specified column names for the data. The argument is a list of quoted column names.

**Associative Array({key, value}, ...)**

**Associative Array(keys, values)**

**Function**
Creates an associative array (also known as a dictionary or hash map).

**Returns**
An associative array object.

**Arguments**
Either list of key-value pairs; or a list, matrix, or data table column that contains keys followed by a list, matrix, or data table column, respectively, that contains the corresponding values.

**Batch Interactive(Boolean)**

**Function**
Sets the run-time environment for JSL to either batch mode or interactive mode.

**Returns**
The previous setting.

**Example**
Batch Interactive(1); // Set Batch mode one
Batch Interactive(0); // Revert to previous mode

**Note**
This function can be used in journal scripts, to ensure that messages are sent to the log rather than to interactive message windows.

**Beep()**

**Function**
Produces an alert sound.

**Returns**
Null.
Best Partition(xindices, yindices, <<Ordered, <<Continuous Y, <<Continuous X)

    Function
    Experimental function to determine the optimal grouping.
    Returns
    A list.
    Arguments
    xindices, yindices Same-dimension matrices.

Blob MD5(blob)

    Function
    Converts the blob argument into a 16-byte blob.
    Note
    The 16-byte blob is the MD5 checksum, or the hash, of the source blob.

Blob Peek(blob, offset, length)

    Function
    Creates a new blob from a subrange of bytes of the blob argument.
    Returns
    A blob object.
    Arguments
    blob a binary large object.
    offset An integer that specifies how many bytes into the blob to begin construction. The first byte is at offset 0, the second byte at offset 1.
    length An integer that specifies how many bytes to copy into the new blob, starting at the offset.

Build Information()

    Function
    Returns the build date and time, whether it's a release or debug build, and the product name in a comma-delimited string.
Caption({h, v}, text, <Delayed(n)>, <Spoken(Boolean))

Function
Displays a caption window at the location described by {h, v} that displays text. The caption can be delayed before being displayed by n, or can be spoken.

Returns
Null.

Arguments

{h, v} a list with two values. h is the horizontal displacement from the top left corner of the monitor in pixels. v is the vertical displacement from the top left corner in pixels.

text A quoted string or a reference to a string that is to be displayed in the caption.

Delayed(n) n is optional delay before displaying the caption in seconds.

Spoken(Boolean) Causes text to be spoken as well as displayed.

ChiSquare Log CDistribution(x, df, <nc>)

Function
Returns 1 - log (value) of the distribution function at quantile x for the chi-square distribution.

ChiSquare Log Density(x, df, <nc>)

Function
Returns the log of the value of the density function at quantile x for the chi-square distribution.

ChiSquare Log Distribution(x, df, <nc>)

Function
Returns the log of the value of the distribution function at quantile x for the chi-square distribution.

ChiSquare Noncentrality(x, df, prob)

Function
Solves the noncentrality such that prob=ChiSquare Distribution (x, df, nc)

Clear Globals(<name>, <name>, ...)

Function
Clears the values for all global symbols. Symbols in any scope other than global are not affected. If one or more names are specified, only those global symbols are cleared.

Returns
Null.

Arguments

name Optional: any global variable name(s).

See
“Clear Symbols(<name>, <name>, ...),” p. 584
Clear Log()

Function
Empties the log.

Clear Symbols(<name>, <name>, ...)

Function
Clear the values for all symbols in any and all scopes. If one or more names are specified, only those symbols are cleared.

Returns
Null.

Arguments
name  Optional: any global variable name(s).

See
“Clear Globals(<name>, <name>, ...),” p. 583

Close(<dt>, <nosave|save("path")>)

Function
Closes a data table. If no arguments are specified, the current data table is closed. If the data table has been changed, it is automatically saved. All dependent windows are also closed (for example, report windows that are based on the data table).

Returns
Void.

Arguments
dt  an optional reference to a data table.
nosave|save("path")  an optional switch to either save the data table to the specified path before closing or to close the data table without saving it.

Close All(type, <NoSave|Save>)

Function
Closes all open resources of type.

Argument
type  A named argument that defines the type of resources that you want to close. The allowable types are: Data Tables, Reports, and Journals.

NoSave or Save  An optional argument that specifies to close all the windows of type either saving all of them or not saving any of them. No prompts appear if you use one of these two arguments. Otherwise, you are prompted to save any unsaved windows that are to be closed.
**Column(<dt>, name)**

**Column(<dt>, n)**

Function

Gets a reference to the data table column.

Arguments

- **dt** Optional reference to a data table. If this is not supplied, the current data table is used.
- **name** Quoted string that is the name of the column.
- **n** Column number.

**Column Dialog(ColList("rolename"), specifications)**

Function

Draws a dialog box for column role assignments.

Returns

A list of commands that were sent and the button that was clicked.

Arguments

- **ColList** Specifies the name of at list one list to add variables to.
- **specifications** Any additional Dialog items (for example, Max Col Datatype).

**Column Name(n)**

Function

Determines the name of the column specified by number.

Returns

The name of the $n$th column as an expression (not a string).

Argument

- **n** The number of a column.
**Concat To(a, b)**

\[ a \| = b \]

**Function**
Appends the string \( b \) to the string \( a \) and places the new concatenated string into \( a \).

**Returns**
A string composed of the string \( a \) directly followed by the string \( b \).

**Arguments**
Two or more strings or string variables. The first must be a variable whose value can be changed.

**Notes**
More than two arguments can be strung together. Each additional string is appended to the end, in left to right order. This function also works with matrices as the arguments.

**Example**
\[
\begin{align*}
a &= "Hello"; \\
b &= " World"; \\
c &= "World"; \\
\text{ConcatTo}(a, b, c); \\
\text{Show}(a);
\end{align*}
\]

\[ a = "Hello World" \]

**Convert File Path(path, <absolute|relative>, <posix|windows>, <base(path)>)**

**Function**
Converts a file path according to the arguments.

**Returns**
The converted path.

**Arguments**

- **path** A pathname that can be either Windows or POSIX.
- **absolute|relative** Optional, specifies whether the returned pathname is in absolute or relative terms. The default value is absolute.
- **posix|windows** Optional, specifies whether the returned pathname is in Windows or POSIX style. The default is POSIX.
- **base(path)** Optional, specifies the base pathname, useful if relative is specified. The default is the default directory.

**Cumulant Quadrature(vector)**

**Function**
Returns the optimal quadrature abscissas and weights.

**Argument**

- **vector** A vector of cumulants (1 by m) between 0 and m-1 for a distribution.
Current Data Table(<dt>)

Function
Without an argument, gets the current (topmost) data table. With an argument, sets the current data table.

Returns
Reference to the current data table.

Argument
dt  Optional name of or reference to a data table.

Data Table(n)
Data Table("name")

Function
Gets reference to the nth open data table or the table with the given name in a global variable.

Returns
Reference to the specified data table.

Argument
n  Number of a data table.
name  Quoted string, name of a data table.

Datafeed()

See “Open Datafeed(),” p. 610.

Decode64 Double("string")

Function
Creates a floating point number from a base 64 encoded string.

Returns
A floating point number.

Arguments
string  a base 64 encoded string.

Delete Globals(<name>, <name>, ...)

Function
Deletes all global symbols. Symbols in any scope other than global are not affected. If one or more names are specified, only those global symbols are deleted.

Arguments
name  Optional: any global variable name(s).

See
“Delete Symbols(<name>, <name>, ...),” p. 588
Delete Symbols(<name>, <name>, ...)

Function
Deletes all symbols in any and all scopes. If one or more names are specified, only those symbols are deleted.

Arguments
- name: Optional: any global variable name(s).

See
“Delete Globals(<name>, <name>, ...),” p. 587

Derivative(expr, {name, ...}, ...)

Function
Calculates the derivative of the expr with respect to name.

Returns
Returns the derivative.

Arguments
- expr: Any expression. Indirect arguments (for example, Name Expr, Expr, Eval) are supported.
- name: Can be a single variable or a list of variables.

Note
Adding an additional variable (derivative(expr, name, name2)) takes the second derivative.

Dialog(contents)

Function
Draws a dialog box composed of contents given.

Returns
A reference to the dialog box.

Arguments
- contents: Any number of possible JSL commands for a Dialog.

Notes
Dialog is deprecated. Use New Window() with the Modal argument instead.

Empty()

Function
Does nothing. Used in the formula editor for making empty boxes.

Returns
Missing.

Arguments
None.
**Encode64 Double(n)**

*Function*

Creates a base 64 encoded string from a floating point number.

*Returns*

A base 64 encoded string.

*Arguments*

- **n** a floating point number.

**Eval(expr)**

*Function*

Evaluates `expr`, and then evaluates the result of `expr` (unquoting).

*Returns*

The result of the evaluation.

*Argument*

- **expr** Any JSL expression.

**Eval Expr(expr)**

*Function*

Evaluates any expressions within `expr`, but leaves the outer expression unevaluated.

*Returns*

An expression with all the expressions inside `expr` evaluated.

*Argument*

- **expr** Any JSL expression.

**Eval Insert(string, <startDel>, <endDel>)**

*Function*

Allows for multiple substitutions.

*Returns*

The result.

*Arguments*

- **string** A quoted string with embedded expressions.
- **startDel** Optional starting delimiter. The default value is `^`.
- **endDel** Optional ending delimited. The default value is the starting delimiter.
Eval Insert Into(string, <startDel>, <endDel>)

Function
Allows for multiple substitutions in place. The same operation as in Eval Insert is performed, and the result is placed into string.

Returns
The result.

Arguments
string A string variable that contains a string with embedded expressions.
startDel Optional starting delimiter. The default value is ^.
endDel optional ending delimited. The default value is the starting delimiter.

Eval List(list)

Function
Evaluates expressions inside list.

Returns
A list of the evaluated expressions.

Arguments
list A list of valid JSL expressions.

Exit(<NoSave>)

Quit(<NoSave>)

Function
Exits JMP.

Returns
Void.

Arguments
NoSave Optional, named command; exits JMP without prompting to save any open files. This command is not case-sensitive, and spaces are optional.

Expr(x)

Function
Returns the argument unevaluated (expression-quoting).

Returns
The argument, unevaluated.

Argument
x Any valid JSL expression.
Extract Expr(expr, pattern)

Function
Find expr matching pattern.

Returns
A pattern that matches the specified pattern.

Arguments
expr Any expression.
pattern Any pattern.

F Log CDistribution(x, dfn, dfd, <nc>)

Function
Returns 1 - log (value) of the normal distribution function at quantile x for the F distribution.

F Log Density(x, dfn, dfd, <nc>)

Function
Returns the log of the value of the density function at quantile x for the F distribution.

F Log Distribution(x, dfn, dfd, <nc>)

Function
Returns the log of the value of the distribution function at quantile x for the F distribution.

F Noncentrality(x, ndf, ddf, prob)

Function
Solves the noncentrality such that prob=F Distribution (x, ndf, ddf, nc)

FFT(list, <named arguments>)

Function
Conducts a Fast Fourier Transformation (FFT) a list of matrices.

Returns
The function takes a matrix of complex numbers and returns the dimension matrix of complex numbers.

Argument
List A list of one or two matrices. If one is provided, it is considered to be the real part. If two are provided, the first is the real part and the second is the imaginary part. Both matrices must have the same dimensions, and both must have more than one row.

Named Arguments
All named arguments are optional.
<<inverse(Boolean) If true (1), an inverse FFT is conducted.
<<multivariate(Boolean) If true (1), a multivariate FFT is conducted. If false(0), a spatial FFT is conducted.
<<scale(number) Multiplies the return values by the specified number.

**Files In Directory(path, <recursive>)**

*Function*
- Returns a list of filenames in the *path* given.

*Returns*
- List of filenames.

*Arguments*
- *path* A valid pathname.
- *recursive* An optional keyword that causes all folders in the path (and all folders that they contain, and so on) to be searched for files.

**First(expr, <expr>, ...)**

*Function*
- Evaluates all expressions provided as arguments.

*Returns*
- Only the result of the first evaluated expression.

*Arguments*
- *expr* Any valid JSL expression.

**Fit Censored(Distribution(name), YLow(vector) | Y(VECTOR), <YHigh(vector)>, <Weight(vector)>, <X(matrix)>, <Z(matrix)>, <HoldParm(vector)>)**

*Function*
- Fits a distribution using censored data.

*Returns*
- A list that contains parameter estimates, the covariance matrix, the loglikelihood, the AIC, the BIC, and a convergence message.

*Required Arguments*
- *Distribution(name)* The name of the distribution to fit.
- *YLow(vector) | Y(VECTOR)* If you do not have censoring, then use *Y* and an array of your data, and do not specify *YHigh*. If you do have censoring, then specify *YLow* and *YHigh* as the lower and upper censoring values, respectively.

*Optional Arguments*
- *YHigh(vector)* A vector that contains the upper censoring values. Specify this only if you have censoring and also specify *YLow*.
- *Weight(vector)* A vector that contains the weight values.
- *X(matrix)* The regression design matrix for location.
- *Z(matrix)* The regression design matrix for scale.
HoldParm(vector)  An array of specified parameters. The parameters should be nonmissing where they are to be held fixed, and missing where the are to be estimated. This is primarily used to test hypotheses that certain parameters are zero or some other specific value.

Fit Transform To Normal(Distribution(name), Y(vector), <Freq(vector))
  Function
  Fits a transformation to normality for a vector of data. This includes the Johnson Sl, Johnson Sb, Johnson Su, and GLog distributions.
  Returns
  A list that contains parameter estimates, the covariance matrix, the log-likelihood AICc, a convergence message, and the transformed values.

For Each Row(script)
  Function
  Repeats the script on each row of the data table.
  Returns
  Null.
  Argument
  script  Any valid JSL expressions.

Function({arguments}, <{local variables}>, script)
  Function
  Stores the body script with arguments as local variables.
  Returns
  The function as defined.
  When called later, it returns the result of the script given the specified arguments.
  Arguments
  {arguments}  A list of arguments to pass into the function.
  {local variables}  A list of variables that are local to the function. You can declare local variables in three ways:
    {var1, var2}
    {var1=0, var1="a string"}
    {Default Local}
    The last option declares that all unscoped variables used in the function are local to the function.
  script  Any valid JSL script.

Gamma Log CDistribution(x, alpha, <scale = 1>, <threshold = 0>)
  Function
  Same as Log (1 - Gamma Distribution(x, alpha)) except that it has a much greater range.
Gamma Log Density\((x, \alpha, <\text{scale} = 1>, <\text{threshold} = 0>)\)

Function
Same as \(\text{Log}(\text{Gamma Density}(x, \alpha))\) except that it has a much greater range.

Gamma Log Distribution\((x, \alpha, <\text{scale} = 1>, <\text{threshold} = 0>)\)

Function
Same as \(\text{Log}(\text{Gamma Distribution}(x, \alpha))\) except that it has a much greater range.

Get Addin("id")

Function
Retrieves a registered add-in by \(id\).

Returns
A scriptable object for the add-in. Returns empty if no add-in with the specified ID was found.

Argument
"id" The ID of an installed add-in.

Get Addins()

Returns
A list of all registered add-ins.

Get Addr Info(address, <port>)

Function
Converts a name to its numeric address.

Returns
A list of strings. The first element is the command (Get Addr Info). The second is the results (for example, “ok” if the command was successful). The third is a list of strings of information. Included in that information is the address that corresponds to the name that was supplied.

Arguments
address A string that specifies the name (for example, www.sas.com).
port The port of the address.

Get Current Directory()

Function
Retrieves the current directory, which is used in the Open File window.

Returns
The absolute pathname as a string.

Arguments
none
Get Default Directory()
  Function
  Retrieves the JMP default directory, which is used as a base for subsequent relative paths.
  Returns
  The absolute pathname as a string.
  Arguments
  none

Get Environment Variable("variable")
  Function
  Retrieves the value of an operating system environment variable.
  Returns
  A string that contains the value of the specified environment variable. If the specified variable is not
  found, an empty string is returned.
  Arguments
  "variable" A string that contains the name of an environment variable.
  Notes
  On Macintosh, environment variable names are case-sensitive. On Windows, the names are
  case-insensitive.

Get File Search Path()
  Function
  Retrieves the current list of directories to search for opening files.
  Returns
  A list of pathnames as strings.

Get Log(<n>)
  Function
  Returns a list of lines from the log.
  Returns
  A list of strings. Each string contains one line from the log.
  Argument
  n  Optional, integer. If no argument is specified, all the lines are returned. If a positive number is
  specified, the first n lines are returned. If a negative number is specified, the last n lines are returned. If
  n=0, no lines are returned (an empty list). If the log is empty, an empty list is returned.
Get Name Info(address, <port>)

Function
Converting a numeric address to its name.

Returns
A list of strings. The first element is the command \((\text{NameGetInfo})\). The second is the results (for example, “ok” if the command was successful). The third is a list of strings of information. Included in that information is the port name that corresponds to the address that was supplied.

Arguments
- **address** A string that specifies the numeric address (for example, 149.173.5.120).
- **port** The port of the address.

Get Path Variable()

Function
Retrieves the value of name, a path variable.

Returns
The absolute pathname as a string.

Argument
- **name** A quoted string that contains a path variable. (Examples: SAMPLE_DATA, SAMPLE_SCRIPTS)

Get Platform Preferences(<platform_name(<option_name, ...)>), ...)

Function
Returns the preferences for the specified platforms.

Returns
A list of platform preferences.

Argument
- **platform_name** Optional. If no platform name is specified, all platform preferences are returned. Otherwise, the preferences for the specified platforms are returned. You can specify one or more preferences for a platform. Only the preference settings for the named options are returned.
Get Preferences(<preference_name>)
Get Preference(<preference_name>)

Function
Returns the settings for the specified preferences.

Returns
A list of preference settings.

Argument
preference_name  Optional. If no preference is specified, all preferences are returned. Otherwise, the settings for the specified preference are returned.

Notes
The preferences for the following areas are not accessible in JSL: Text Data Files, Windows Specific, Mac OS Settings, Fonts, Communications, File Locations, Script Editor, SAS Integration, and JMP Updates.

Get Project(<name>)

Function
Retrieves either the first project listed in the Project window or the project named.

Returns
The project.

Argument
name  An optional project name as a quoted string.

Get Project List()

Function
Returns a list of all open projects.

Returns
A list quoted strings that each contains the name of a project. If no projects are open, the list is empty.

Glue(expr1, expr2, ...)

;  

Function
Evaluates each argument in turn.

Returns
The result of the last argument evaluated.

Arguments
One or more valid JSL expressions.
Head(exprArg)

Head Expr(exprArg)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the head of the evaluated expression, without its arguments.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head Expr() is deprecated. Use Head() instead.</td>
</tr>
</tbody>
</table>

Head Name(expr)

Head Name Expr(expr)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the head of the evaluated expression as a string.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head Name Expr() is deprecated. Use Head Name() instead.</td>
</tr>
</tbody>
</table>

Hier Clust(x)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the clustering history for a hierarchical clustering using Ward's method (without standardizing data).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>x  A data matrix.</td>
</tr>
</tbody>
</table>

Host Is(argument)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determines whether the host environment is the specified OS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>True (1) if the current host environment matches the argument, false (0) otherwise.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument Windows or Mac tests for the specified operating system. Bits32 or Bits64 tests for the specified 32-bit or 64-bit machine.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only one argument can be tested at a time. Invalid arguments return false (0).</td>
</tr>
</tbody>
</table>
Include(pathname, <named arguments>)

Function
Opens the script file identified by the quoted string pathname, parses the script in it, and executes it.

Returns
Whatever the included script returns. If you use the <<Parse Only option, Include returns the contents of the script.

Named Arguments

<<Parse Only Parses the script but does not execute the script.
<<New Context Causes the included script to be run its own unique namespace.
<<Allow Include File Recursion Lets the included script include itself.

Include File List()

Function
Returns a list of files that are included at the point of execution.

Invert Expr(expr, name)

Function
Attempts to unfold expr around name.

IRT Ability(Q1, <Q2, Q3, ...,> parameter matrix)

Function
Solves for ability in an Item Response Theory model.

Is Alt Key()

Function
Returns 1 if the Alt key is being pressed, or 0 otherwise.

Note
On a Macintosh, Is Alt Key() tests for the Option key.

Is Associative Array(name)

Function
Returns 1 if the evaluated argument is an associative array, or 0 otherwise.

Is Command Key()

Function
Returns 1 if the Command key is being pressed, or 0 otherwise.
Is Context Key()
Function
Returns 1 if the Context key is being pressed, or 0 otherwise.

Is Control Key()
Function
Returns 1 if the Control key is being pressed, or 0 otherwise.

Note
On a Macintosh, Is Control Key() tests for the Command (⌘) key.

Is Empty(global)
Is Empty(dt)
Is Empty(col)
Function
Returns 1 if the global variable, data table, or data column does not have a value (is uninitialized), or 0 otherwise.

Is Expr(x)
Function
Returns 1 if the evaluated argument is an expression, or 0 otherwise.

Is List(x)
Function
Returns 1 if the evaluated argument is a list, or 0 otherwise.

Is Name(x)
Function
Returns 1 if the evaluated argument is a name-only expression, or 0 otherwise.

Is Namespace(namespace)
Function
Returns 1 if the namespace argument is a namespace; returns 0 otherwise.

Is Number(x)
Function
Returns 1 if the evaluated argument is a number or missing numeric value, or 0 otherwise.
Is Option Key()
  Function
  Returns 1 if the Option key is being pressed, or 0 otherwise.

Is Scriptable(x)
  Function
  Returns 1 if the evaluated argument is a scriptable object, or 0 otherwise.

Is Shift Key()
  Function
  Returns 1 if the Shift key is being pressed, or 0 otherwise.

Is String(x)
  Function
  Returns 1 if the evaluated argument is a string, or 0 otherwise.

JMP Product Name()
  Function
  Returns either "Standard" or "Pro", depending on which version of JMP is licensed.

JMP Version()
  Function
  Returns the version number of JMP that you are running.
  Returns
  release.revision<.fix>
  Arguments
  none

JMP6 SAS Compatibility Mode(Boolean)
  Function
  Setting this to 1 (true) causes SAS operators to operate in a mode compatible with JMP 6 capabilities.

KDE(vector, <named arguments>)
  Function
  Returns a kernel density estimator with automatic bandwidth selection.
  Argument
  vector A vector.
  Named Arguments
  All the named arguments are optional.
**JSL Syntax Reference**

**Utility Functions**

**<<weights**
Must be a vector of the same length as `vector`, and can contain any nonnegative real numbers. `Weights` represents frequencies, counts, or similar concepts.

**<<bandwidth(n)**
A nonnegative real number. Enter a value of 0 to use the bandwidth selection argument.

**<<bandwidth scale(n)**
A positive real number.

**<<bandwidth selection(n)**
`n` must be 0, 1, 2, or 3, corresponding to Sheather and Jones, Normal Reference, Silverman rule of thumb, or Oversmoother, respectively.

**<<kernel(n)**
`n` must be 0, 1, 2, 3, or 4, corresponding to Gaussian, Epanechnikov, Biweight, Triangular, or Rectangular, respectively.

---

**KDTable(matrix)**

**Function**

Returns a table to efficiently look up near neighbors.

**Returns**

A KDTable object.

**Argument**

`matrix` A matrix of k-dimensional points. The number of dimensions or points is not limited. Each column in the matrix represents a dimension to the data, and each row represents a data point.

**Messages**

**<<Distance between rows(row1, row2)** Returns the distance between two the two specified rows in the KDTable. The distance applies to removed and inserted rows as well.

**<<K nearest rows(stop, <position>)** Returns a matrix. If `position` is not specified, returns the `n` nearest rows and distances to all rows. If `position` is specified, returns the `n` nearest rows and distances to either a point or a row. `Stop` is either `n` or `{n, limit}`. Position is a point that is described as a (1xK) matrix, as the number of dimensions, or as the number of a row.

**<<Remove rows(number | vector)** Remove either the row specified by `number` or the rows specified by `vector`. Returns the number of rows that were removed. Rows that were already removed are ignored.

**<<Insert rows(number | vector)** Re-insert either the row specified by `number` or the rows specified by `vector`. Returns the number of rows that were inserted. Rows that were already inserted are ignored.

**Note**

When rows are removed or inserted, the row indices do not change. You can remove and re-insert only rows that are in the KDTable object. If you need different rows, construct a new KDTable.

---

**LenthPSE(x)**

**Function**

Returns Lenth’s pseudo-standard error of the values within a vector.

**Argument**

`x` A vector.
List(a, b, c, ...)  
{a, b, c, ...}  

Function  
Constructs a list from a set of items.

LnZ(x)  

Function  
Returns the natural logarithm of x, except when x=0. In that case, 0 is returned. This function is intended for internal use by derivatives.

Load DLL("path")  

Function  
Loads the DLL in the specified path.

See Also  
Once a DLL is loaded, you send the DLL object messages to interact with it. See “Dynamic Link Libraries (DLLs),” p. 675 in the “Messages” appendix for details about these messages. See “Dynamic Link Libraries (DLLs),” p. 349 in the “Extending JMP” chapter for some examples.

Load Text File(path, <XMLParse|SASODSXML>)  

Function  
Opens the text file stored at path.

Returns  
The text contained in the file.

Arguments  
  path  A pathname that points to a text file. This can be a URL.
  XMLParse  Optional argument that causes an XML text file to be converted into JSL.
  SASODSXML  Optional argument that causes the text file to be parsed as SAS ODS default XML.

Local({name=value, ...}, script)  

Function  
Resolves names to local expressions.

Lock Globals(name1, name2, ...)  

Function  
Locks one or more global variables to prevent it or them from being changed.
Lock Symbols(<name>, <name>, ...)  
Function  
Locks the specified symbols, which prevents them from being modified or cleared. If no symbols are provided, all global symbols are locked. If no symbols are provided and the script has the Names Default To Here mode turned on, then all local symbols are locked.

LogCapture(expr)  
Function  
Evaluates the expr, captures the output that would normally be sent to the log, and instead returns it.  
Returns  
A string that contains the log output.  
Argument  
Any valid JSL expression.  
Note  
No output appears in the log.

LPSolve(A, b, c, L, U, neq, nle, nge, <slackVars(Boolean)>)  
Function  
Returns a list containing the decision variables (and slack variables if applicable) in the first list item and the optimal objective function value (if one exists) in the second list item.  
Arguments  
A  A matrix of constraint coefficients.  
b  A matrix that is a column of right hand side values of the constraints.  
c  A vector of cost coefficients of the objective function.  
L, U  Matrices of lower and upper bounds for the variables.  
neq  The number of equality constraints.  
nle  The number of less than or equal inequalities.  
gge  The number of greater than or equal inequalities.  
slackVars(Boolean)  Optional. Determines whether the slack variables are returned in addition to the decision variables. The default value is 0.  
Note  
The constraints must be listed as equalities first, less than or equal inequalities next, and greater than or equal inequalities last.

Mail("host.id", "subject", "message", "attachment")  
Function  
(Windows only) Sends e-mail (using MAPI) to the host.id with the specified subject and message texts. Sends an attachment specified by the optional attachment parameter.
Main Menu(string, <string>)

Function
Execute the command found on JMP's menu named by the quoted string. The optional second quoted string specifies the name of the window to send the command to.

Mixture Moment(powers, <L>, <U>)

Function
Computes the exact moments of monomials over bounded subsets of the face of the unit simplex.

Arguments
powers The vector of polynomial powers in the monomial.
L, U Optional lower and upper bounds that must be between 0 and 1. The default values for L and U are 0 and 1, respectively.

Moment Quadrature(vector)

Function
Returns the optimal quadrature abscissas and weights.

Argument
vector A vector of moments (1 by m) between 0 and m-1 for a distribution.

Multivariate Gaussian Quadrature(nDim, nStrata, nSim)

Function
Returns a list of two elements: values and weights for radial-spherical integration for smooth functions of independent standard normal variables.

Arguments
nDim Number of dimensions.
nStrata Number of strata.
nSim Number of simulations.

N Arg(exprArg)

Function
Returns the number of arguments of the evaluated expression head.

N Arg Expr(exprArg)

Function
Returns the number of arguments of the expression head.
N Items(source)

Function
Determines the number of elements in the source specified. Can be assigned to a variable.

Returns
For a list, the number of items in the list. For an associative array, the number of keys.

Arguments
source  A list or an associative array.

N Table()

Function
Returns the number of open data tables.

Name Expr(x)

Function
Returns the unevaluated expression of x rather than the evaluation of x.

Names Default To Here(Boolean)

Function
Determines where unresolved names are stored, either as a global or local (if Boolean is 0) or in the Here scope (if Boolean is 1).

Namespace(name)

Function
Returns a reference to the named namespace (name).

Argument
Name  A namespace name string, a reference to a namespace, or a reference to an object that owns a namespace (such as a data table, a window, or a platform).

Namespace Exists(name)

Function
Returns 1 if a namespace with the specified name exists; otherwise, returns 0.

New Column("name", attributes)

Function
Adds a new column named name after the last column in dt.

Note
Can also be used as a message: dt<<New Column("name", attributes).
New Image()
New Image(width, height)
New Image("filepath")
New Image(picture)
New Image(matrix)
New Image("rgb"|"r"|"rgba", {matrix, ...})

Function
Creates a new image in a report.

Returns
An image.

Arguments
All argument sets are optional, but all arguments within each set are required.
width, height Sets the width and height of the image in pixels.
"filepath" A filepath to an image.
picture A JSL picture object.
matrix The image as a matrix of JSL color pixels.
"rgb"|"r"|"rgba", {matrix, ...} Specify the channels (rgb, r, or rgba) and provide a matrix
of values (0.0-1.0) for each channel. Examples:
New Image("r", [r matrix]);
New Image("rgb", [[r matrix], [g matrix], [b matrix]]);
New Image("rgba", [[r matrix], [g matrix], [b matrix], [a matrix]]);

New Namespace(<name>, <{expr, ...}>)

Function
Creates a new namespace with the specified name. If a name is not provided, an anonymous name is
provided.

Returns
A reference to the namespace.

Arguments
name An optional, quoted string that contains the name of the new namespace.
{list of expressions} An optional list of expressions within the namespace.
New Project("name", <invisible>)

Function: Creates a new project with the specified name.
Returns: The project.
Argument:
  "name" A quoted string that contains the name of the new project.
  invisible Optional keyword that creates an invisible project.

New Table("name")

Function: Creates a new data table with the specified name.

Normal Integrate(muVector, sigmaMatrix, expr, x, nStrata, nSim)

Function: Returns the result of radial-spherical integration for smooth functions of multivariate, normally distributed variables.
Arguments:
  muVector A vector.
  sigmaMatrix A matrix.
  expr An expression in terms of the variable x.
  x The variable used in the expression expr.
  nStrata Number of strata.
  nSim Number of simulations.

Normal Log CDistribution(x)

Function: Returns 1 - log (value) of the distribution function at quantile x for the normal distribution.

Normal Log Density(x)

Function: Returns the log of the value of the density function at quantile x for the normal distribution.

Normal Log Distribution(x)

Function: Returns the log of the value of the distribution function at quantile x for the normal distribution.
Appendix A
JSL Syntax Reference

Utility Functions

Num Deriv(expr)

Function
Returns the first numeric derivative of the expr with respect to the first argument in the expression.

Num Deriv2(expr)

Function
Returns the second numeric derivative of the expr with respect to the first argument in the expression.

Open("path", <arguments>)

Function
Opens the data table named by the path.

Arguments
Force Refresh Closes the file specified if it’s already open before trying to open it again.
Invisible Opens the table as invisible.
Charset("option") The available character set options are "Best Guess", "utf-8", "utf-16", "us-ascii", "windows-1252", "x-max-roman", "x-mac-japanese", "shift-jis", "euc-jp", "utf-16be", and "gb2312".
HTML Table(n) To import a table from a web page, use the URL as the filepath. This optional argument specifies which table (n) on the web page to open.
Use Labels for Var Names(0|1) For SAS data sets, this option specifies to use SAS labels as JMP columns names. The default value is 0.
"flag" An optional quoted string that specifies the type of file that you are opening. For example, "text", "journal", "sas", "script", "jmp". This can be useful if your file does not have a file extension.
These arguments apply to opening text files:
Strip Quotes(Boolean) If the fields in the text file are quoted, setting this to true removes the quotes, and setting it to false does not remove the quotes. The default value is True.
End of Line(CR|LF|CRLF|Semicolon) Specifies the end-of-line character.
End of Field(Tab|Space|Spaces|Comma) Specifies the end-of-field character.
EOLOther("char") Specifies a custom end-of-line character.
EOFOther("char") Specifies a custom end-of-field character.
Labels(Boolean) Specifies whether the first line in the text file contains column names. The default value is True.
Zip The path argument should be for the zip archive. The extension (.zip) is not required. See “Zip Archives,” p. 703 in the “Messages” appendix for the messages that you can send to a zip archive. The basic functionality is to get a list of files in the zip archive, to read a file in the zip archive into either a string or a blob, and to write files into the zip archive. Note that reading a zip archive temporarily puts the contents into memory. Reading very large zip archives can cause errors.
Open Database("connectInfo", "sqlStatement", "outputTableName")

Function
Opens the database indicated by connectInfo with the sqlStatement, returning a data table named outputTableName.

Open Datafeed()

Datafeed()

Function
Creates a Datafeed object and window.

Returns
A reference to the Datafeed object.

Arguments
No arguments are required. You usually set up the basic operation of the data feed within the Open Datafeed() command, however.

Open Project(pathname, <invisible>)

Function
Opens the project identified by the quoted string pathname.

Returns
The project.

Arguments
pathname A pathname that points to a project file.
invisible Optional keyword that creates an invisible project.

Parameter({name=value, ...}, model expression)

Function
Defines formula parameters for models for the Nonlinear platform.

Parse(string)

Function
Converts a character string into a JSL expression.

Parse XML(string, On Element("tagname", Start Tag(expr), End Tag(expr)))

Function
Parses an XML expression using the On Element expressions for specified XML tags.
Pick Directory(<"prompt string">
)

Function
Prompts the user for a directory, returning the directory path as a string. If a quoted string is provided, that string is included in the window.

Returns
The path for the directory that the user selects.

Arguments
prompt An optional quoted string. If provided, that string is used for the window title.

Pick File(<"caption">, <"initial directory">, <{filter list}>, <first filter>, <save flag>, <"default file">
)

Function
Prompts the user to select a file in the Open File window.

Returns
The path of the file that the user selects.

Arguments
caption An optional quoted string. If provided, that string is used for the window title, providing a prompt for the user.

initial directory An optional quoted string that is a valid filepath to a folder. If provided, it specifies where the Open window begins. If not provided, or if it’s an empty string, the JMP Default Directory is used.

filter list An optional list of quoted strings that define the filetypes to show in the Open window. See the following example for syntax.

first filter An optional integer that specifies which of the filters in the filter list to use initially. If you use an integer that is too large or small for the list (for example, 4 for a list of 3), the first filter in the list is used.

save flag An optional Boolean that specifies whether the Open window or Save window is used. 0 causes the Open window to appear, where a user can select a file to open in JMP. 1 causes the Save As window to appear, where a user can save a new, empty file of the selected type in the selected folder. The default value is 0.

default file The name of the file that appears in the window by default.

Note
Although all arguments are optional, they are also positional. For example, you cannot specify a filter list without also specifying the caption and the initial directory.

Example
Pick File("Select JMP File", "$DOCUMENTS", {"JMP Files|jmp;jsl;jrn", "All Files|*"}
Platform Preferences(platform(option(value)), ...)  
Platform Preference(platform(option(value)), ...)  
Set Platform Preferences(platform(option(value)), ...)  
Set Platform Preference(platform(option(value)), ...)  

Function  
Sets preferences for platforms.

Polytope Uniform Random(samples, A, b, L, U, neq, nle, nge, <nwarm=200>, <nstride=25>)  

Function  
Generates random uniform points over a convex polytope.

Arguments  
Samples  The number of random points to be generated.  
A  The constraint coefficient matrix.  
B  The right hand side values of constraints.  
L, U  The lower and upper bounds for the variables.  
neq  The number of equality constraints.  
nle  The number of less than or equal inequalities.  
nge  The number of greater than or equal inequalities.  
nwarm  Optional: The number of warm-up repetitions before points are written to the output matrix.  
nstride  Optional: The number of repetitions between each point that is written to the output matrix.  

Note  
The constraints must be listed as equalities first, less than or equal inequalities next, and greater than or equal inequalities last.

Preferences(pref1(value1), ...)  
Preference(pref1(value1), ...)  
Pref(pref1(value1), ...)  
Prefs(pref1(value1), ...)  
Set Preferences(pref1(value1), ...)  
Set Preference(pref1(value1), ...)  

Function  
Sets preferences for JMP.

Arguments  
Analysis Destination(window)  Specifies where to route new analyses.
Appendix A

JSL Syntax Reference

Utility Functions

Annotation Font(font, size, style) Font choice for annotations in reports.
Axis Font(font, size, style) Font choice for axis labels.
Axis Title Font(font, size, style) Font choice for axis titles.
Background Color(color) Sets the background color for windows.
Calculator Boxing(Boolean) Turns on boxing to show hierarchy of expressions.
Data Table Font(font, size, style) Font choice for data tables.
Data Table Title on Output(Boolean) Titles reports with name of data table.
Date Title on Output(Boolean) Titles reports with current date.
Excel Has Labels(Boolean) When on, forces JMP to interpret the first row of data as column labels.
Excel Selection(Boolean) When on, the user is prompted for which non-blank Excel worksheets should be imported from an Excel workbook.
Foreground Color(color) Sets the foreground color for windows.
Formula Font(font, size, style) Font choice for the formula editor.
Graph Background Color(color) Sets the color for the background area inside the graph frame.
Graph Marker Size(size) Default size for drawing markers.
Heading Font(font, size, style) Font choice for table column headings in reports.
Initial JMP Starter Window(Boolean) Specifies whether the JMP Starter window is shown at launch.
Initial Splash Window(Boolean) Enables you to show or suppress the initial splash screen.
Marker Font(font, size, style) Font choice for markers used in plots.
Monospaced Font(font, size, style) Font choice for monospaced text.
Outline Connecting Lines(Boolean) Draws lines between titles for same-level outline nodes.
Report Table Style(style) Specify how columns in report tables are bordered. Choices are Plain, Bordered, Embossed.
Save Table In Extended File Format(Boolean) Enables saving data tables with more than 32,766 columns.
Show Explanations(Boolean) Some analyses have optional text that explains the output.
Show Menu Tips(Boolean) Turns menu tips on or off.
Show Status Bar(Boolean) Turns display of the status bar on or off.
Small Font(font, size, style) Font choice for small text.
Text Font((font, size, style) Font choice for general text in reports.
Thin Postscript Lines(Boolean) Macintosh only. Specifies that line widths drawn to a Postscript printer be narrower than otherwise.
Utility Functions

Title Font(font, size, style)  
Font choice for titles. Arguments are name of font (for example, Times), size in points, and style (bold, plain, underline, italic).

Notes  
The preferences for the following areas are not accessible in JSL: Text Data Files, Windows Specific, Mac OS Settings, Fonts, Communications, File Locations, Script Editor, SAS Integration, and JMP Updates.

Print(expr, expr, ...)  
Function  
Prints the values of the specified expressions to the log.

Quit()  
See “Exit(<NoSave>),” p. 590.

Recurse(function)  
Function  
Makes a recursive call of the defining function.

Register Addin(unique_id, home_folder, <named_parameters>)  
Function  
Register a JMP Add-In and load the add-in if it registers successfully.

Returns  
If successful, returns a scriptable object representing the registered add-in. If unsuccessful, returns Empty.

Arguments  
unique_id A quoted string that contains the unique identifier for the add-in. The string can contain up to 64 characters. The string must begin with a letter and contain only letters, numbers, periods, and underscores. Reverse-DNS names are recommended to increase the likelihood of uniqueness.

home_folder A quoted string that contains the filepath for the folder containing the add-in files. The filepath must conform to the valid pathname requirement for the host operating system.

FriendlyName( "friendly name" ) An optional, quoted string that contains a name that can be displayed in the JMP user interface wherever add-in names are displayed, instead of the unique ID.

JMPVersion("version") An optional string that contains a specific version of JMP. The default value is "All", which enables the add-in to be loaded and run in any version of JMP that supports add-ins. "Current" restricts the use of the add-in to only the current version. Any quoted version number (for example, "7" or "9") restricts the add-in to a single specific version of JMP.

LoadsAtStartup(Boolean) An optional Boolean. The default value is True (1), which causes the add-in to be loaded when JMP is started. If the value is False (0), the add-in is not loaded automatically.
LoadNow(Boolean)  Loads the add-in immediately.

Note
If a file named addin.def is found in the specified home folder, values from that file are used for any optional parameters that are not included in the Register Addin() function.

Revert Menu()
Function
Resets your JMP menus to factory defaults.

Save Log(pathname)
Function
Writes the contents of the log to the specified file location.

Save Text File(path, text)
Function
Saves the JSL variable text into the file specified by path.

SbInv(z, gamma, delta, theta, sigma)
Function
Johnson Sb inverse transformation. If argument is normal, the result is Johnson Sb.

SbTrans(x, gamma, delta, theta, sigma)
Function
Johnson Sb transformation from a doubly bound variable to a standard normal (0, 1) distribution.

Schedule(n, script)
Function
Queues an event to run the script after n seconds.

Send(obj, message)
obj << message
Function
Sends a message to a platform object.

Set Clipboard(string)
Function
Evaluates the string argument looking for a character result, and then places the string on the clipboard.
Set Current Directory( )

Function
Sets the current directory for Open File operations.

Set Default Directory( )

Function
Sets the default directory, which is used for resolving relative paths.

Set File Search Path({path or list of paths})

Function
Sets the current list of directories to search for opening files. "." means the current directory.

Set Menu(path)

Function
Sets your JMP menus to the JMPMENU file specified by the quoted string path.

Set Path Variable(name)

Function
Sets the path stored in the variable.

Set Platform Preference()
Set Platform Preferences()

See “Platform Preferences(platform(option(value)), ...),” p. 612.

Set Preference()
Set Preferences()

See “Preferences(pref1(value1), ...),” p. 612.

Show(expr, expr, ...)

Function
Prints the name and value of each expression to the log.

Show Addins Dialog()

Function
Opens the Add-In Status window (View > Add-Ins).
Arguments:
None.
Show Commands()
Function
Lists scriptable objects and operators. Arguments are All, DisplayBoxes, Scriptables, Scriptable Objects, StatTerms, Translations.

Show Globals()
Function
Shows the values for all global symbols. Symbols in any scope other than global are not shown.
See
“Show Symbols( ),” p. 617

Show Namespaces()
Function
Shows the contents of all user-defined namespaces, both named and anonymous.

Show Preferences(<all>)
Function
Shows current preferences. If no argument is specified, preferences that have been changed are shown. If all is given as the argument, all preferences are shown.

Show Properties(object)
Function
Shows the messages that the given object can interpret, along with some basic syntax information.

Show Symbols()
Function
Shows the values for all symbols in any and all scopes.
See
“Show Globals( ),” p. 617

Simplify Expr(expr(expression))
Simplify Expr(nameExpr(global))
Function
Algebraically simplifies an expression

SlInv(z, gamma, delta, theta, sigma)
Function
Johnson Sl inverse transformation. If argument is normal, the result is Johnson Sl.
SlTrans(x, gamma, delta, theta, sigma)

Function
Johnson Sl transformation from a doubly bound variable to a standard normal (0, 1) distribution.

Socket(<STREAM | DGRAM>)

Function
Creates a socket.

Returns
The socket that was created.

Arguments
STREAM | DGRAM Optional argument to specify whether the socket is a stream or datagram socket. If no argument is supplied, a stream socket is created.

Sort List(list|expr)

Function
Sort the elements or terms of list or expr.

Sort List Into(list|expr)

Function
Sort the elements or terms of list or expr in place.

Speak(text, <wait(Boolean)>)

Function
Calls system’s speech facilities to read aloud the text. If Wait is turned on, script execution pauses until speaking is done.

Sqrt(n)

Function
Returns the square root of n.

Squash(expr)

Function
An efficient computation of the function 1/ [1 + exp(expr)].

Status Msg("message")

Function
 Writes the message string to the status bar.
SuInv(z, gamma, delta, theta, sigma)

Function
Johnson Su inverse transformation. If argument is normal, the result is Johnson Su.

Summarize(<by>, <count>, <sum>, <mean>, <min>, <max>, <stddev>, <quantile>)

Function
Gathers summary statistics for a data table and stores them in global variables.

Returns
None.

Arguments
All arguments are optional and can be included in any order. Typically, each argument is assigned to a variable so you can display or manipulate the values further.

name=By(col) Using a BY variable changes the output from single values for each statistic to a list of values for each group in the BY variable.
c=Count The number of values in the column (or for each BY group).
sum=Sum(col) The sum of values in the column (or for each BY group).
mean=Mean(col) The mean of the column (or of each BY group).
min=Min(col) The minimum value in the column (or in each BY group).
max=Max(col) The maximum value in the column (or in each BY group).
stddev=StdDev(col) The standard deviation of the column (or of each BY group).
quantile=Quantile(col, q) The quantile specified by q for the column (or for each BY group).

Note
If all data are excluded or missing, missing values are returned by the summary statistics commands.

Suppress Formula Eval(Boolean)

Function
Turns off automatic calculation of formulas for all data tables.

SuTrans(x, gamma, delta, theta, sigma)

Function
Johnson Su transformation from a doubly bound variable to a standard normal (0, 1) distribution.

t Log CDistribution(x, df, <nc>)

Function
Returns 1 - log (value) of the normal distribution function at quantile x for the t distribution.
t Log Density(x, df, <nc>)

Function
Returns the log of the value of the density function at quantile x for the t distribution.

t Log Distribution(x, df, <nc>)

Function
Returns the log of the value of the distribution function at quantile x for the t distribution.

T Noncentrality(x, df, prob)

Function
Solves the noncentrality such that
prob=T Distribution(x, df, nc)

Throw("text")

Function
Returns a Throw. If you include text, throwing stores text in a global exception_msg. If text begins with "!", throwing creates an error message about where the exception was caught.

Tick Seconds()

Function
Measures the time taken for a script to run, measured down to the 60th of a second.

Tolerance Limit(1-alpha, p, n)

Function
Constructs a 1-alpha confidence interval to contain proportion p of the means with sample size n.

Try(expr1, expr2)

Function
Evaluates expr1. If that returns a Throw, stops, returns nothing, evaluates expr2 and returns result.

Type(x)

Function
Returns a string that names the type of object x is. The list of possible types is: Unknown, List, DisplayBox, Picture, Column, TableVar, Table, Empty, Pattern, Date, Integer, Number, String, Name, Matrix, RowState, Expression, Associative Array, Blob.
Unlock Symbols(name1, name2, ...)
Unlock Globals(name1, name2, ...)

Function
Unlocks the specified symbols that were locked with a Lock Symbols() or Lock Globals() command.

Unregister Addin(unique_id)

Function
Unregisters (removes) a previously registered add-in.

Argument
unique_id A quoted string that contains the unique identifier for the add-in to unregister.

V Concat To(A, B, ...)

Function
Vertical concatenation in place. This is an assignment operator.

Returns
The new matrix.

Arguments
Two or more matrices.

Wait(n)

Function
Pauses n seconds before continuing the script.

Watch(global, global, ...)

Function
Displays globals’ names and values in a window. If “all” is provided as the argument, all globals are placed into the window.

Note
New globals are not added to the window list.

Web(string, <JMP Window>)

Function
Opens the URL stored in string in the default system web browser. Under Microsoft Windows, you can add JMP Window as the second argument to have the HTML open in a JMP browser.
Wild()

Function
Only used with Extract Expr() for expression matching to denote a wildcard position that matches any expression.

Wild List()

Function
Only used with Extract Expr() for expression matching to denote a series of wildcard arguments that match any expression.

Write("text")

Function
Prints text to the log without surrounding quotation marks.

XML Attr("attr name")

Function
Extracts the string value of an xml argument in the context of evaluating a Parse XML command.

XML Decode("xml")

Function
Decodes symbols in XML to ordinary text. For example, &amp; becomes &, and &lt; becomes <.

Argument
xml A quoted string containing XML.

XML Encode("text")

Function
Prepares text for embedding in XML. For example, & becomes &amp;, and < becomes &lt;.

Argument
xml A quoted string containing plain text.

XML Text()

Function
Extracts the string text of the body of an XML tag in the context of evaluating a Parse XML command.

Zero Or Missing(expr)

Function
Returns 1 if expr yields a missing value or zero, 0 otherwise.
Platform Scripting Notes

Scripting Options for All Platforms

The following messages are available for any platform object only in JSL:

- `object <<Get Script`
- `object <<Get Script With Data Table`
- `object <<Get Data Table`
- `object <<Get Timing`
- `object <<Report`
- `object <<Top Report`
- `object <<Title`
- `object <<SendToReport( Dispatch(...) )`
- `object <<SendToByGroup( ... )`
- `object <<Action`

Individual Platform Options

Distribution

The following message for a Distribution object is available only in JSL:

- `dist_object <<(Fit Handle[n] << ...)`

The following JSL option for `Fit Distribution()` is equivalent to the `Make into Data Table` option on the right-click menu for the Parameter Estimates table:

- `Fit Distribution(
  <distribution>( Make Parms Table )
);`

Bivariate

The following message for a Bivariate object is available only in JSL:

- `biv_object <<Fit Where( WHERE_clause )`

Fit Model: Standard Least Squares

The following messages for a Fit Model object are available only in JSL:

- `fit_model_object <<Get Variance Components( );`
- `fit_model_object <<Get Effect Names( );`
- `fit_model_object <<Get Effect PValues( );`
- `fit_model_object <<Get Estimates( );`
- `fit_model_object <<Get Parameter Names( );`
- `fit_model_object <<Get Random Effect Names( );`
- `fit_model_object <<Get Std Errors( );`
- `fit_model_object <<Get X Matrix( );`
- `fit_model_object <<Get XPX Inverse( );`
Fit Model: MANOVA

The following message for a Fit Model object is available only in JSL:

```javascript
fit_model_object << (Response[1] << (Effect[1] << ... ));
```

Fit Model: Generalized Linear Models

The following message for a Fit Model object is available only in JSL:

```javascript
Parametric Formula( )
```

Neural and Neural Net

JMP introduces a new Neural platform that replaces Neural Net. Scripts that use Neural Net are deprecated and might not work in the future. We recommend that you use the Neural platform for new scripts and change your old scripts that use Neural Net to Neural.

The following option for `Neural()` is available only in JSL:

```javascript
Multithreading( Boolean )
```

The following message for a Neural object is available only in JSL:

```javascript
neural_object <<Get NBoost
```

The following messages for a Neural Fit object are available only in JSL:

```javascript
neural_object <<(Fit[n] <<Get RSquare Training)
neural_object <<(Fit[n] <<Get RSquare Validation)
neural_object <<(Fit[n] <<Get RSquare Test)
neural_object <<(Fit[n] <<Get Gen RSquare Training)
neural_object <<(Fit[n] <<Get Gen RSquare Validation)
neural_object <<(Fit[n] <<Get Gen RSquare Test)
neural_object <<(Fit[n] <<Get Misclassification Rate Training)
neural_object <<(Fit[n] <<Get Misclassification Rate Validation)
neural_object <<(Fit[n] <<Get Misclassification Rate Test)
neural_object <<(Fit[n] <<Get Average Log Error Training)
neural_object <<(Fit[n] <<Get Average Log Error Validation)
neural_object <<(Fit[n] <<Get Average Log Error Test)
neural_object <<(Fit[n] <<Get RMS Error Training)
neural_object <<(Fit[n] <<Get RMS Error Validation)
neural_object <<(Fit[n] <<Get RMS Error Test)
neural_object <<(Fit[n] <<Get Average Absolute Error Training)
neural_object <<(Fit[n] <<Get Average Absolute Error Validation)
neural_object <<(Fit[n] <<Get Average Absolute Error Test)
neural_object <<(Fit[n] <<Get ROC Area Training)
neural_object <<(Fit[n] <<Get ROC Area Validation)
neural_object <<(Fit[n] <<Get ROC Area Test)
neural_object <<(Fit[n] <<Get Confusion Matrix Training)
neural_object <<(Fit[n] <<Get Confusion Matrix Validation)
neural_object <<(Fit[n] <<Get Confusion Matrix Test)
```
neural_object<<(Fit[n]<<(Get Confusion Rates Training))
neural_object<<(Fit[n]<<(Get Confusion Rates Validation))
neural_object<<(Fit[n]<<(Get Confusion Rates Test))
neural_object<<(Fit[n]<<(Get Seconds))

Partition

The following options for Partition() are available only in JSL:

  Missing Value Categories( Boolean )
  Set Random Seed( n )
  Multithreading( Boolean )
  Time Limit ( n )

The following messages for a Partition object are available only in JSL:

  partition_object<<(Get RSquare Training)
  partition_object<<(Get RSquare Validation)
  partition_object<<(Get RSquare Test)
  partition_object<<(Get Gen RSquare Training)
  partition_object<<(Get Gen RSquare Validation)
  partition_object<<(Get Gen RSquare Test)
  partition_object<<(Get Misclassification Rate Training)
  partition_object<<(Get Misclassification Rate Validation)
  partition_object<<(Get Misclassification Rate Test)
  partition_object<<(Get ROC Area Training)
  partition_object<<(Get ROC Area Validation)
  partition_object<<(Get ROC Area Test)
  partition_object<<(Get Average Log Error Training)
  partition_object<<(Get Average Log Error Validation)
  partition_object<<(Get Average Log Error Test)
  partition_object<<(Get RMS Error Training)
  partition_object<<(Get RMS Error Validation)
  partition_object<<(Get RMS Error Test)
  partition_object<<(Get Average Absolute Error Training)
  partition_object<<(Get Average Absolute Error Validation)
  partition_object<<(Get Average Absolute Error Test)
  partition_object<<(Get Seconds)
  partition_object<<(Get SAS Data Step)
  partition_object<<(Get Tolerant SAS Data Step)
  partition_object<<(Get MM SAS Data Step)
  partition_object<<(Get MM Tolerant SAS Data Step)

Time Series

The following options for Time Series() are available only in JSL:

  Autocorrelation Lags( n )
  Forecast Periods( n )

The following messages for a Time Series object are available only in JSL:

  time_series_object<<(Maximum Iterations( n ))
Choice

The following option for Choice() is available only in JSL:

\[\text{Convergence Criterion( fraction )}\]

Multivariate

The following message for a Multivariate object is available only in JSL:

\[\text{multivariate_object << Create SAS Job( )}\]

Hierarchical Clustering

The following messages for a Hierarchical cluster object are available only in JSL:

\[\begin{align*}
\text{hier_cluster_object << Get Column Names} \\
\text{hier_cluster_object << Get Clusters} \\
\text{hier_cluster_object << GetDisplayOrder} \\
\text{hier_cluster_object << GetColumnDisplayOrder} \\
\text{hier_cluster_object << Get Distance Matrix}
\end{align*}\]

KMeans Clustering

The following message for a KMeans cluster object is available only in JSL:

\[\text{kmeans_cluster_object << Get Statistics}\]

Life Distribution

The following messages for a Life Distribution object are available only in JSL:

\[\begin{align*}
\text{life_dist_object << Suppress Plot(plot_name)} \\
\text{life_dist_object << Get Results} \\
\text{life_dist_object << Get Estimates} \\
\text{life_dist_object << Get Formula}
\end{align*}\]

Fit Life by X

The following messages for a Fit Life by X object are available only in JSL:

\[\begin{align*}
\text{fit_life_object << Set Scriptables( ... )} \\
\text{fit_life_object << Get Results}
\end{align*}\]

Fit Parametric Survival

The following messages for a Fit Parametric Survival object are available only in JSL:

\[\begin{align*}
\text{survival_object << Get Parameter Names} \\
\text{survival_object << Get Estimates} \\
\text{survival_object << Get Std Errors} \\
\text{survival_object << Get Effect Names} \\
\text{survival_object << Get Effect PValues}
\end{align*}\]
Control Chart
The following option for Control Chart() is available only in JSL:

Use Excluded Points on MR( Boolean )

Pareto Plot
The following message for a Pareto Plot object is available only in JSL:

No Plot(Boolean)

Variability Chart
The following message for a Variability Chart object is available only in JSL:

variability_object <<Show Box Plot Whisker Bars( Boolean )

Profiler
The following messages for a Profiler object are available only in JSL:

profiler_object <<Get Objective
profiler_object <<Objective Formula
profiler_object <<Get Objective Formula

Surface Plot
The following messages for a Surface Plot object are available only in JSL:

surface_plot_object <<Mode( Isosurface | Sheet )
surface_plot_object <<Set Z Axis( col, Current Value( n ) )
surface_plot_object <<Set Variable Axis( col, Current Value( n ) )
surface_plot_object <<Response( ... )
surface_plot_object <<Formula( ... )
Symbols

! 475
– 442–443
-- 445
!= 471
* 442
*= 445
+ 441
    patterns 460
++ 446
+= 443
/* */ 466
// 441, 466
//! 467
//= 444
: 579
:: 580
:| 476
; 597
< 469
< ... <= 470
<= 470
<= ... < 471
= 444
-= 446
== 468
> 468
>= 469
^ 442
| 476
|| 449
||= 586

A
Abbrev Date 478
Abs 528
Add 441

Add To 443
All 514
And 472
AndMZ 472
AndV3 472
Any 514
Arc 502
Arc Cos H 576
Arc Cosine 576
ArCos 576
ArcSine 576
ArcSinH 576
ArcTan 577
ArcTangent 577
ArcTanH 577
Arg 578
Arg Expr 578
ARIMA Forecast 579
Arrhenius 572
Arrhenius Inv 572
Arrow 502
ArSin 576
As Column 579
As Date 579
As Global 580
As List 580
As Name 580
As Namespace 580
As Scoped 580
As Table 581
Assign 444
Associative Array 581
ATan 577

B
Back Color 502
Batch Interactive 581
Beep 581
Best Partition 582
Beta 573
Beta Binomial Distribution 483
Beta Binomial Probability 483
Beta Binomial Quantile 483
Beta Density 529
Beta Distribution 529
Beta Quantile 530
Binomial Distribution 483
Binomial Probability 484
Binomial Quantile 484
Blob MDS 582
Blob Peek 582
Blob to Char 447
Blob To Matrix 447
Border Box 486
Break 473
Build Information 582
Button Box 486

C
Caption 583
CDF 515
Ceiling 528
Char 448
Char to Blob 448
Char to Hex 449
Char To Path 503
Check Box 487
ChiSquare Density 530
ChiSquare Distribution 530
ChiSquare Log CDF 583
ChiSquare Log Density 583
ChiSquare Log Distribution 583
ChiSquare Noncentrality 583
ChiSquare Quantile 530
Chol Update 515
Cholesky 515
Choose 473
Circle 503
Clear Globals 583
Clear Log 584
Clear Symbols 583
Close 584
Close All 584
Col List Box 487
Col Max 568
Col Maximum 568
Col Mean 568
Col Min 568
Col Minimum 568
Col N Missing 568
Col Number 569
Col Quantile 569
Col Rank 569
Col Shuffle 546
Col Standardize 569
Col Std Dev 570
Col Sum 570
Color to HLS 503
Color to RGB 504
Column 585
Column Dialog 585
Column Name 585
Combo Box 488
Concat 449
Concat Items 449
Concat To 586
Contains 450
Context Box 488
Continue 473
Contour 504
Contour Function 505
Convert File Path 586
Correlation 515
Cos 577
CosH 577
Cosine 577
Count 550
Covariance 516
Cumulant Quadrature 586
Current Data Table 587
Current Journal 488
Current Metadata Connection 554
Current Report 489
Current SAS Connection 554
Current Window 489

D
Data Table 587
Datafeed 610
Date Difference 478
Date DMY 478
Date Increment 479
Date MDY 479
Day 479
Day of Week 480
Day of Year 480
debug run 467
debug step 467
Decode64 Double 587
Delete Globals 588
Delete Symbols 588
Derivative 588
Design 516
Design Nom 516
Design Ord 517
DesignF 516
Desirability 570
Det 517
Diag 517
Dialog 588
Dif 550
Digamma 573
Direct Product 517
Distance 518
Divide 441
Divide To 444
Double Declining Balance 497
Drag Line 505
Drag Marker 506
Drag Polygon 506
Drag Rext 506
Drag Text 507
Dunnett P Value 531
Dunnett Quantile 531

E
e 477
EDiv 518
EMult 518
Eigen 518
Empty 588
Encode64 Double 589
Ends With 450
Equal 468
Eval 589

Eval Expr 589
Eval Insert 589
Eval Insert Into 590
Eval List 590
Excerpt Box 489
Exit 590
Expr 573
Expr 590
Expr As Picture 489
Extract Expr 591

F
F Density 531
FDistribution 531
F Log CDistribution 591
F Log Density 591
F Log Distribution 591
F noncentrality 591
F Power 531
F Quantile 532
F Sample Size 532
Factorial 573
FFT 591
Files in Directory 592
Fill Color 507
Fill Pattern 507
First 592
Fit Censored 592
Fit Transform To Normal 593
Floor 528
For 473
For Each Row 593
Format 480
Format Date 480
Frechet Density 532
Frechet Distribution 532
Frechet Quantile 532
Functions 593
Future Value 498

G
G Inverse 519
Gamma 574
Gamma Density 533
Gamma Distribution 533
Gamma Log CDF
Gamma Log Density
Gamma Log Distribution
Gamma Poisson Distribution
Gamma Poisson Probability
Gamma Poisson Quantile
Gamma Quantile
Get Addin
Get Addins
Get Addr Info
Get Current Directory
Get Default Directory
Get Environment Variable
Get File Search Path
Get Log
Get Name Info
Get Path Variable
Get Platform Preference
Get Platform Preferences
Get Preference
Get Preferences
Get Project
Get Project List
Get SAS Version Preference
Global Box
GLog Density
GLog Distribution
GLog Quantile
Glue
Gradient Function
Graph
Graph 3D Box
Graph Box
Greater
Greater or Equal

H
H Center Box
H Direct Product
H Line
H List Box
H Sheet Box
H Size
Handle
Head
Head Expr

Head Name
Head Name Expr
Heat Color
Hex
Hex to Blob
Hex to Char
Hex to Number
Hier Box
Hier Clust
HLS Color
Host Is
Hour
Hypergeometric Distribution
Hypergeometric Probability
Icon Box
Identity
If
If Box
IfMax
IfMin
IfMZ
IfV3
IGamma
In Days
In Format
In Hours
In Minutes
In Path
In Polygon
In Weeks
In Years
Include
Include File List
Index
Insert
associative arrays
Insert Into
associative arrays
Interest Payment
Interest Rate
Internal Rate of Return
Interpolate
Index

Inverse 520
Invert Expr 599
IRT Ability 599
Is Alt Key 599
Is Associative Array 599
Is Command Key 599
Is Context Key 600
Is Control Key 600
Is Empty 600
Is Expr 600
Is List 600
Is Matrix 520
Is Missing 469
Is Name 600
Is Namespace 600
Is Number 600
Is Option Key 601
Is Scriptable 601
Is Shift Key 601
Is String 601
Item 452

J
J 520
JMP Product Name 601
JMP Version 601
JMP6 SAS Compatibility Mode 601
Johnson Sb Density 534
Johnson Sb Distribution 534
Johnson Sb Quantile 534
Johnson SI Density 535
Johnson SI Distribution 535
Johnson SI Quantile 535
Johnson Su Density 535
Johnson Su Distribution 536
Johnson Su Quantile 536
Journal Box 491

K
KDE 601
KDTable 602

L
Lag 551
Left 452
Length 452
LenthPSE 602, 608
Less 469
Less LessEqual 470
Less or Equal 470
LessEqual Less 471
LEV Density 536
LEV Distribution 536
LEV Quantile 536
LGamma 574
Line 509
Line Style 509
Lineup Box 491
List 603
List Box 491
Ln 574
LnZ 603
Load DLL 603
Load Text File 603
Loc 520
Loc Max 520
Loc Min 521
Loc Sorted 521
Local 603
Lock Globals 603
Lock Symbols 604
Log 574
Log10 574
Log1P 574
LogCapture 604
Logistic Density 537
Logistic Distribution 537
Logistic Quantil 537
Logit 575
Loglogistic Density 537
Loglogistic Distribution 537
Loglogistic Quantile 537
Lognormal Density 537
Lognormal Distribution 537
Lognormal Quantile 537
Long Date 481
Lowercase 452
LPSolve 604
M
Mail 604
Main Menu 605
Marker 510
Marker Size 510
Match 475
MatchMZ 475
MatchV3 475
Matrix 521
Matrix Box 492
Matrix Mult 522
Max 571
Maximize 522
Maximum 571
MDYHMS 481
Mean 571
Meta Connect 554
Meta Create Profile 555
Meta Delete Profile 556
Meta Disconnect 556
Meta Get Repositories 556
Meta Get Servers 556
Meta Is Connected 557
Meta Set Repository 557
Meta Stored Process 557
Min 571
Minimize 522
Minimum 571
Minus 442
Minute 482
Mixture Moment 605
Mod 529
Modified Internal Rate of Return 499
Modulo 529
Moment Quadrature 605
Month 482
Mousetrap 510
Multiply 442
Multiply To 445
Multivariate Gaussian Quadrature 605
Multivariate Normal Impute 522
Munger 453

N
N Arg 605
N Arg Expr 605
N Choose K 575
N Col 523
N Cols 523
N Items 606
N Missing 571
N Row 551
N Rows 551
N Table 606
Name Expr 606
Names Default To Here 606
Namespace 606
Namespace Exists 606
Neg Binomial Distribution 485
Neg Binomial Probability 485
Net Present Value 499
New Column 606
New Image 607
New Namespace 607
New Project 608
New Table 608
New Window 492
Normal Biv Distribution 538
Normal Contour 510
Normal Density 538
Normal Distribution 538
Normal Log CDistribution 608
Normal Log Density 608
Normal Log Distribution 608
Normal Mixture Density 538
Normal Mixture Distribution 538
Normal Mixture Quantile 538
Normal Quantile 538
Not 475
Not Equal 471
Num 453
Num Deriv 609
Num Deriv2 609
Number 571
Number Col Box 492
Number Col Edit Box 492
Number Edit Box 492
Number of Periods 500

O
Open 609
Index

Open Database 610
Open Datafeed 610
Open Project 610
Or 476
OrMZ 476
Ortho 523
Ortho Poly 523
OrV3 476
Outline Box 493
Oval 510

P
Page Break Box 493
Panel Box 493
Parameter 610
Parse 610
Parse Date 481
Parse XML 610
Pat Abort 458
Pat Altern 458
Pat Any 458
Pat Arb 459
Pat Arb No 459
Pat At 459
Pat Break 460
Pat Concat 460
Pat Conditional 460
Pat Fail 461
Pat Fence 461
Pat Immediate 461
Pat Len 462
Pat Match 462
Pat Not Any 462
Pat Pos 463
Pat R Pos 463
Pat R Tab 463
Pat Regex 463
Pat Rem 464
Pat Repeat 464
Pat Span 464
Pat String 465
Pat Succeed 465
Pat Tab 465
Pat Test 465
Path 511
Path To Char 511
Payment 500
Pen Color 511
Pen Size 511
Pi 477
Pick Directory 611
Pick File 611
Picture Box 493
Pie 511
Pixel Line 512
Pixel Move 512
Pixel Origin 512
Platform Preferences 612
Plot Col Box 493
Poisson Distribution 485
Poisson Probability 485
Poisson Quantile 486
Polygon 512
Polytope Uniform Random 612
Popup Box 493
Post Decrement 445
Post Increment 446
Power 442
Pref 612
Preference 612
Preferences 612
Present Value 500
Principal Payment 501
Print 614
Print Matrix 523
Probit 538
Product 571

Q
QR 524
Quantile 571
Quarter 482
Quit 590

R
R Connect 541
R Execute 541
R Get 542
R Get Graphics 542
R Init 542
R Is Connected 543
R JMP Name to R Name 543
R Send 543
R Send File 544
R Submit 545
R Submit File 545
R Term 545
Radio Box 493
Random Beta 546
Random Beta Binomial 546
Random Binomial 546
Random Cauchy 546
Random Exp 546
Random Frechet 547
Random Gamma 547
Random Gamma Poisson 547
Random Geometric 547
Random GLog 547
Random Index 524
Random Integer 547
Random Johnson Sb 547
Random Johnson Sl 547
Random Johnson Su 547
Random LEV 548
Random Logistic 548
Random Loglogistic 548
Random Lognormal 548
Random Negative Binomial 548
Random Normal 548
Random Normal Mixture 548
Random Poisson 548
Random Reset 549
Random SEV 549
Random Shuffle 524
Random Triangular 549
Random Uniform 549
Random Weibull 549
Rank 524
Rank Index 524
Ranking 524
Ranking Tie 524
Rect 512
Recurse 614
Regex 453
Regex Match 466
Register Addin 614
Remove 453
associative arrays 453
Remove From 454
associative arrays 454
Repeat 454
Report 493
Resample Freq 549
Reverse 454
Reverse Into 454
Revert Menu 615
RGB Color 512
Right 455
Round 529
Row 551
run script without opening 467

S
SAS Assign Lib Refs 557
SAS Connect 558
SAS Design Lib Refs 558
SAS Disconnect 559
SAS Export Data 559
SAS Get Data Sets 560
SAS Get File 560
SAS Get File Names 560
SAS Get File Names In Path 561
SAS Get File Refs 561
SAS Get Log 561
SAS Get Output 561
SAS Get Results 562
SAS Get Var Names 562
SAS Import Data 562
SAS Import Query 564
SAS Is Connected 565
SAS Load Text File 565
SAS Name 565
SAS Open For Var Names 565
SAS Send File 566
SAS Submit 566
SAS Submit File 567
Save Log 615
Save Text File 615
SBInv 615
SbTrans 615
Scene Box 494
Index

Schedule 615
Scheffe Cubic 575
Script Box 494
Scroll Box 494
Second 482
Send 615
Sequence 531
Set Current Directory 616
Set Default Directory 616
Set File Search Path 616
Set Menu 616
Set Path Variable 616
SEV Density 539
SEV Distribution 539
SEV Quantile 539
Shape 524
Sheet Part 494
Shift 455
Shift Into 455
Short Date 482
Show 616
Show Addins Dialog 616
Show Commands 617
Show Globals 617
Show Namespaces 617
Show Preferences 617
Show Properties 617
Show Symbols 617
Simplify Expr 617
Sin 578
Sine 578
SinH 578
Slider Box 495
SlInv 617
SlTrans 618
Socket 618
Solve 525
Sort Ascending 525
Sort Descending 525
Sort List 618
Sort List Into 618
Spacer Box 495
Speak 618
Spline Coef 525
Spline Eval 526
Spline Smooth 526
Sqrt 618
Squash 618
Squish 575
Starts With 455
Status Msg 618
Std Dev 572
Step 476
Stop 477
Straight Line Depreciation 501
String Col Box 495
String Col Edit Box 495
Students t Density 539
Students t Distribution 539
Students t Quantile 540
Subscript 531
Substitute 456
Substitute Into 456
Substr 457
Subtract 443
Subtract To 446
SuInv 619
Sum 572
Sum Of Years Digits Depreciation 501
Summarize 619
Summation 572
Suppress Formula Eval 619
SuTrans 619
SVD 526
Sweep 526

T

t Density 539
t Distribution 539
t Log CDistribution 619
t Log Density 620
t Log Distribution 620
T Noncentrality 620
t Quantile 540
Tab Box 496
Table Box 496
Tan 578
Tangent 578
TanH 578
Text 512
Text Box 496
Text Color 513
Text Edit Box 496
Text Size 513
Throw 620
Tick Seconds 620
Time of Day 482
Today 482
Tolerance Limit 620
Trace 526
Transparency 513
Transpose 526
Trigamma 575
Trim 457
Try 620
Tukey HSD P Value 540
Tukey HSD Quantile 540
Type 620

Unlock Symbols 621
Unregister Addin 621
Uppercase 457

V
V Center Box 496
V Concat 526, 621
V Line 513
V List Box 496
V Max 527
V Mean 527
V Min 527
V Sheet Box 496
V Size 513
V Std 527
V Sum 527
Vec Diag 527
Vec Quadratic 527

W
Wait 621
Watch 621
Web 621
Web Browser 497
Week of Year 483
Weibull Density 540
Weibull Distribution 540
Weibull Quantile 540
While 477
Wild 622
Wild List 622
Window 497
Word 457
Words 458
Write 622

X
X Function 513
X Origin 513
X Range 514
X Scale 514
XML Attr 622
XML Decode 622
XML Encode 622
XML Text 622

Y
Y Function 514
Y Origin 514
Y Range 514
Y Scale 514
Year 483

Z
Zero or Missing 622
This appendix gives abbreviated descriptions for all of JMP’s general object messages. For platform messages, see the Scripting Platforms chapter. The tables are arranged in chapter order, and items within each table are arranged in alphabetical order, rather than topical order as shown in the chapters.

For an alphabetical index to all JSL functions, see the Index to the Syntax Reference.
Contents

Associative Arrays ......................................................... 641
Data Tables ................................................................. 642
Columns ................................................................. 653
Rows ................................................................. 657
Data Filter ............................................................ 658
Databases ............................................................... 660
Datafeed ............................................................... 660
Display Boxes .............................................................. 661
  Windows ............................................................ 661
  All Display Boxes .................................................. 663
  Outline Boxes ...................................................... 667
  Frame Boxes ......................................................... 667
  Table Boxes ......................................................... 668
  Tab Boxes .......................................................... 669
  Text Boxes ........................................................ 669
  Axis Boxes ........................................................ 670
  Number Col Boxes ................................................ 672
  Slider Boxes ......................................................... 673
  String Col Boxes .................................................... 673
  Matrix Boxes ........................................................ 674
  Nom Axis Boxes ..................................................... 674
  Panel Boxes ........................................................ 674
  Border Boxes ........................................................ 674
Dynamic Link Libraries (DLLs) ............................................. 675
Platforms ................................................................. 676
R Integration Messages .................................................... 679
SAS Integration Messages ................................................... 683
Messages for Schedule .................................................... 699
Sockets ................................................................ 700
Other Objects ................................................................ 703
Associative Arrays

For the following messages, \texttt{map} stands for an associative array or a reference to one.

\texttt{map<<First}

Returns the first key within \texttt{map}, or \texttt{Empty()} if \texttt{map} has no keys. Note that keys are returned in lexicographical order.

\texttt{map<<Get Contents}

Returns a list of all key-value pairs within \texttt{map}.

\texttt{map<<Get Keys}

Returns a list of all the keys within \texttt{map}.

\texttt{map<<Get Default Value()}

Returns the implicit value of all absent keys, or \texttt{Empty()} if none has been set.

\texttt{map<<Get Value(key)}

Returns the value for the \texttt{key} within \texttt{map}.

\texttt{map<<Get Values(<keylist>)}

If no argument is provided, a list of all values within \texttt{map} is returned.

If a list of keys is provided, a list of the values corresponding to only those keys is returned.

\texttt{map<<Insert(key, value)}

Inserts the \texttt{key} into \texttt{map} and assigns \texttt{value} to it. If \texttt{key} already exists in \texttt{map}, its value is replaced by the new \texttt{value} given. This message is equivalent to the function \texttt{Insert Into}.

\texttt{map<<Next(key)}

Returns the key following the given \texttt{key} within the \texttt{map}, or \texttt{Empty()} if \texttt{map} has no keys. Note that keys are returned in lexicographical order.

\texttt{map<<Remove(key)}

Removes the \texttt{key} and \texttt{value} from \texttt{map}. This message is equivalent to the function \texttt{Remove From}.

\texttt{map<<Set Default Value(v)}

Sets the implicit default value of all absent keys. Any key added without a value is assigned this value by default.
Data Tables

dt<<Add Column Properties(property argument, ...)
   Add the specified properties to the selected column.

dt<<Add From Row States
   Updates a row state column with any currently used row state changes that are not the default state.

dt<<Add Multiple Columns("prefix", n, position, attributes)
   Adds n columns to dt at the position indicated.

dt<<Add Rows(count, <rownum>)
dt<<Add Rows(assignment list)
   Add the number of rows specified to the bottom of the data table or starting at the rownum specified.

dt<<Add Scripts to Table(script, ...)
   Pastes the specified scripts to the data table.

dt<<Add To Row States
   Copies all row state values in a column that are not the default state to the currently used row state in the data table.

dt<<Begin Data Update
   Turns off display updating to allow for quick updating of a data table. Use End Data Update in conjunction with this command to turn display updating back on.

dt<<Clear Column Selection
   Deselects all selected columns.

dt<<Clear Row States
   Cancels any row states in effect.

dt<<Clear Select
   Turns off the current selection.
**Messages**

**Data Tables**

- `dt<<Clone Formula Column(column, n, Substitute Column Reference(column1, {list}))`
  
  Creates `n` new formula columns, substituting references to `column1` with columns from the `list` into the formula from the original `column`.

- `dt<<Color or Mark by Column(col, <optional arguments>)`

  **Function**
  
  Assigns colors or markers according to the values of a data table column. If no optional arguments are provided, row states are used.

  **Arguments**
  
  - **Color Number(n)** Uses the specified JMP color.
  - **Color Theme(theme)** Uses the specified color theme.
  - **Marker Theme(named argument)** Uses the specified marker theme: standard, hollow, paired, classic, or alphanumeric.

- `dt<<Color Rows by Row State`

  Colors the rows in the data table grid using the color assignments by row states. Send the message again to turn off the row colors.

- `dt<<Compress Selected Columns({column1, ...})`

  Compresses the listed columns into the most compact form that is possible. Columns with character data are compressed to 1 byte if there are fewer than 255 levels. Columns with numeric data are compressed to 1 byte if the numeric values are between -127 and 127.

- `dt<<Concatenate(dt2, ..., Keep Formulas, Output Table Name("name"))`

  Creates a new table (`name`) from the rows of `dt` and `dt2`.

- `dt<<Copy Table Script`

  Copies the script to recreate the data table onto the clipboard so that it can be pasted somewhere else.
dt<<Data Filter(<Mode(...)>), <Add Filter (...)>)

Constructs a data filter. If no arguments are specified, the Add Filter Columns window appears. Arguments for Mode() include Select(), Show(), and Include(). They are all Boolean. Select defaults to true(1), and Show and Include default to false (0). Arguments for Add Filter() include Columns(), and Where(). Columns() takes one or more column names separated by commas. You can add one or more WHERE clauses to define the filter. There are several additional arguments. For more information, see the Data Filter chapter in the Using JMP and the Data Tables chapter in the Scripting Guide.

dt<<Data View(<options>)

Duplicates the data table in a new window. If you specify one of the following arguments, the new data table includes only the corresponding rows.

excluded The new data table includes only the rows that are marked as excluded in the original data table.

labeled | labelled The new data table includes only the rows that are marked as labeled in the original data table.

hidden The new data table includes only the rows that are marked as hidden in the original data table.

selected The new data table includes only the rows that are selected in the original data table.

dt<<Delete Columns(col, col2, ...)

dt<<Delete Column

Deletes column(s) from the data table dt. Specify which column or columns to delete. Without an argument, deletes the selected columns, if any. Delete Column is a synonym.

dt<<Delete Rows(<n>)

dt<<Delete Rows({n, o, p, ...})

dt<<Delete Rows({n::q})

Deletes the currently selected rows or rows specified.

dt<<Delete Table Property(name)

Delete a table property.

dt<<Delete Table Variable(name)

Delete a table variable.
dt<<End Data Update

Resumes display updating after a Begin Data Update message. These commands are used for quick updates of the data table when many changes have to be made. Speed is gained by turning off display updating.

dt<<Exclude
dt<<Unexclude

Toggles selected rows in dt from excluded to unexcluded or vice versa.

dt<<Get All Columns As Matrix

Returns the values from all columns of dt in a matrix. Character columns are numbered according to the levels, starting at 1.

dt<<Get As Matrix

Returns values from the numeric columns of dt in a matrix.

dt<<Get As Report

Returns the data table as a report.

dt<<Get Column Names(arguments)

Returns a list of column names in a data table. The arguments restrict the names retrieved as follows: Numeric, Ordinal, Rowstate, Continuous, Ordinal, and Nominal get only the specified types of columns. More than one can be specified. String returns a list of strings rather than a list of column references.

dt<<Get Excluded Rows

Returns the currently excluded rows in the data table.

dt<<Get Hidden Rows

Returns the currently hidden rows in the data table.

dt<<Get Labeled Rows
dt<<Get Labelled Rows

Returns the currently labeled rows in the data table.
dt<<Get Name()
Returns the name of the table.

dt<<Get Path
Returns the absolute path for the data table.

dt<<Get Property("name")
Returns the script in the property name.

dt<<Get Rows Where(WHERE clause)
Returns the rows in the data table that match the specified where criteria. Some examples are as follows:
  dt<<Get Rows Where(:sex == "M");
  dt<<Get Rows Where(:sex == "M" & :age < 15);

dt<<Get Script
Returns as an expression a script to recreate the data table.

dt<<Get Selected Columns()
Returns a list of selected columns.

dt<<Get Selected Rows()
Returns the selected rows.

dt<<Get Table Script Names()
Returns a list of the names of all the scripts and properties in the data table.

dt<<Get Table Variable("name")
Returns the value from the variable or script name.

dt<<Get Table Variable Names()
Returns a list of the names of all the variables in the data table.

dt<<Go To Row(n)
Locates and selects row number n in dt.
dt<<Group Columns({col1, col1, ...})

Groups the columns specified. You can provide either a list of columns to group, or a column name and the number of columns to group. In the latter case, the number n specifies to group the column given with the n-1 columns that follow.

dt<<Hide

Toggles selected rows in dt from hidden to unhidden or vice versa.

dt<<Unhide

Toggles selected rows in dt from unhidden to hidden or vice versa.

dt<<Invert Row Selection

Selects any row currently deselected and deselects any row currently selected.

dt<<Join(With(table), Select(columns), Select With(columns), method, Drop Multiples(Boolean, Boolean), Include Non Matches(Boolean, Boolean), Output Table Name("name"))

Combines data tables dt and table side to side. For details about joining tables, see the Using JMP book.

dt<<Journal

Makes a journal from the data table. Only the data grid is included, not notes, variables, or scripts.

dt<<Journal Link

Adds a link to the data table in the current journal. If a journal does not exist, a new one is created.

dt<<Label
dt<<Unlabel

Toggles selected rows in dt from labeled to unlabeled or vice versa.

dt<<Last Modified

Returns the date on which the data table was last saved.

dt<<Layout

Makes a layout window from the data table. Only the data grid is included, not notes, variables, or scripts.
dt<<Lock Data Table

Locks the data table so that values cannot be changed.

dt<<Make RowState Handler

Creates a row state handler function. The argument of the function holds the rows whose row states get changed.

dt<<Make SAS Data Step

Returns the data table as a SAS Data Step.

dt<<Make SAS Data Step Window

Returns the data table as a SAS Data Step and places it in a SAS script window.

dt<<Marker by Column(col)

Assigns markers according to the values of a data table column.

dt<<Markers(n)

Assigns marker n to the selected rows.

dt<<Maximize Display

Forces the data table to remeasure all of its columns and zoom to the best-sized window.

dt<<Move Selected Columns(To First|To Last|After(col))

Moves the selected columns in the data table to the specified position.

dt<<Move Rows(At Start|At End|After(n))

Moves the selected rows in the data table to the specified position.

dt<<New Column("name", attributes)

Adds a new column named name after the last column in dt. Can also be used as a command: New Column("name", attributes).
dt<<New Data View

Opens a duplicate of the data table. The second data table is identical to and linked to the original data table, so that any changes made in one are reflected in the other. Closing either data table also closes the other and all references to the data tables are deleted. This can be useful to show an invisible data table.

dt<<New Property("name", script)
dt<<New Script("name", script)
dt<<New Table Property("name", script)
dt<<Set Property("name", script)

 Creates a new table property using the specified name that stores the specified script.

dt<<New Table Variable("name", value)
dt<<Set Table Variable("name", value)

 Creates a new table variable with the specified name and value.

dt<<Next Selected

Scrolls data table down to show the next selected row that is not already in view.

dt<<Original Order

Restores saved order of columns in dt.

dt<<Previous Selected

Scrolls data table up to show the previous selected row that is not already in view.

dt<<Print Window(<Show Dialog>)

Prints the window. If the optional argument Show Dialog is specified, the print window is displayed. Otherwise, the window is printed to the default printer using the current settings, and no print window is displayed.

dt<<Reorder By Data Type

Reorders columns in dt, row state first, then character, then numeric.

dt<<Reorder By Modeling Type

Reorders columns in dt to continuous, then ordinal, then nominal.
dt<<Reorder By Name
Reorders columns in \textit{dt} to alphanumeric order by name.

dt<<Reverse Order
Reverses columns in \textit{dt} from current order.

dt<<Revert
Reverts to the most recently saved version of \textit{dt}.

dt<<Run Formulas
Evaluates all formulas in the data table. All the evaluations are finished before the next JSL command is run.

dt<<Run Script("name")
Finds the table property \textit{name} and runs it as a JSL script.

dt<<Save("path")
dt<<Save As("path")
Saves the table under the \textit{path} given.

dt<<Save Database(connection\_information, table\_name)
Save the data table to the database named using the connection and table name specified.

dt<<Save Script to Script Window
Saves a script to reproduce the data table in a Script Journal window.

dt<<Select All Rows
Selects all rows in the data table.

dt<<Select Excluded
Selects only those rows in the data table that are currently excluded.

dt<<Select Hidden
Selects only those rows in the data table that are currently hidden.
dt<<Select Labeled

Selects only those rows in the data table that are currently labeled.

dt<<Select Randomly(p|n)

Randomly selects the given percentage \( p \) of the rows in the data table, or the number of rows \( n \).

dt<<Select Rows({list})

Selects the rows given in the list of row numbers.

dt<<Select Where(condition)

Selects the rows in \( dt \) where the condition evaluates as true.

dt<<Set Dirty

Marks the data table as changed, even if no changes have been made.

dt<<Set Label Columns(column_1, ...)

Assigns the specified columns as label columns.

dt<<Set Matrix(matrix)

Inserts matrix into a data table, adding new columns and rows as necessary.

dt<<Set Name("name")

Gives a name to the table.

dt<<Set Property("name", script)


dt<<Set Label Columns("name", ...)

Turns on the Label attribute for the specified columns. If no columns are listed, turns Label attribute off.

dt<<Set Row States(matrix)

Assigns row states to each row of a data table. matrix contains the codes corresponding to the desired state and contains one entry per row.
dt<<Set Scroll Lock Columns("name", ...)  
    Locks scrolling for the specified columns. If no columns are listed, unlocks scrolling.

dt<<Set Table Variable("name", value)  
    See “dt<<New Table Variable("name", value),” p. 649.

dt<<Sort(By(columns), order(Descending or Ascending), Output Table Name("name"))  
    Creates a new table (name) by rearranging the rows of dt according to the values of one or more columns.

dt<<Split(Split(columns), Group(gcol), Col ID(idcol), Remaining Columns(choice), Output Table Name("name"))  
    Creates a new table (name) by breaking one or more columns from dt into several.

dt<<Stack(Stack(columns), ID(columns), Stacked(newcol), Output Table Name("name"))  
    Creates a new table (name) by combining the values from several columns in dt into one column newcol.

dt<<Subscribe(condition)  
    Subscribes to a data table to get messages regarding changes in the data table.

dt<<Subset(Columns(columns), Rows(matrix), Linked, Table Name("name"), Copy Formula(1|0), Suppress Formula Evaluation(1|0), Sampling Rate(rate))  
    Creates a new table (name) from the rows that you specify in dt.

dt<<Summary(Group(col), Subgroup(col), statistic(col))  
    Creates a new table of summary statistics for the col that you specify, according to groups and subgroups.

dt<<Suppress Formula Eval(Boolean)  
    Turns off automatic calculation of formulas for data table dt.

dt<<Transpose(Columns(columns), Rows(matrix), Output Table Name("name"))  
    Creates a new table (name) from the rows and columns that you specify.
dt<<Ungroup Columns({col1, col1, ...})
   Ungroups the columns defined in the list argument.

dt<<Unsubscribe(condition)
   Releases any previous subscriptions to the data table.

Columns

col<<Add Column Properties ("name", expression)
   Adds the property name with the expression given. You can add any standard column property by name or a user-specified property.

col<<Add From Row States
   Updates a row state column with any currently used row state changes that are not the default state.

col<<Add To Row States
   Copies all row state values in a column that are not the default state to the currently used row state in the data table.

col<<Color Cells(color)
   Colors the cells of the column within the data table grid. Use any named color or 0 to clear the color.

col<<Color Cell by Value(Boolean)
   Colors the cells of the column in the data table grid using the value color property.

col<<Copy Column Properties
   Copies the column properties into the buffer.

col<<Copy From Row States
   Copies all row state values currently used in the data table to a column.

col<<Copy to Row States
   Copies all row state values in the column to the currently used row state in the data table.

col<<Data Type(type)
   Sets the type for col; choices are Numeric, Character, Rowstate.
**Messages**

Data Tables

---

**col<<Delete Formula**

Delete the formula from a column.

**col<<Delete Property(name)**

**col<<Delete Column Property(name)**

Delete the property name from a column.

---

**col<<Eval Formula**

Forces the formula to evaluate (perhaps again).

---

**col<<Exclude(0|1)**

Turns the excluded or unexcluded state on, depending on the Boolean argument.

---

**col<<Format("format", <width>, <decimal>, <"Use Thousands Separator">)**

Sets the numeric display format specified.

---

**col<<Formula(expression)**

Sets the formula for the variable and evaluates it.

---

**col<<Get Column Field Width**

Returns the field width used for displaying data in the column.

---

**col<<Get Data Type**

Returns the data type of col.

---

**col<<Get Format**

Returns the format of the column.

---

**col<<Get Formula**

Returns the formula.

---

**col<<Get Input Format**

Returns the format used for input and storing of data for the column.
col<<Get List Check
   Returns the list check definition. If list check is not defined for the column, a message is sent to the log stating so.

col<<Get Lock
   Returns the current Lock setting.

col<<Get Modeling Type
   Returns the modeling type of the column ("Continuous" or "Ordinal" or "Nominal").

col<<Get Name
   Returns the name of the column.

col<<Get Property
   Returns the specified property definition. If the specified property is not defined for the column, a message is sent to the log stating so.

col<<Get Range Check
   Returns the range check definition. If range check is not defined for the column, a message is sent to the log stating so.

col<<Get Role
   Returns the preselected role of col.

col<<Get Script
   Returns the script to reproduce the column.

col<<Get Selected
   Returns 1 if the column is selected, or 0 otherwise.

col<<Get Value Labels
   Returns the value labels definition. If value labels is not defined for the column, a message is sent to the log stating so.

col<<Get Use Value Labels
   Returns 1 if the value labels are set to be used for the column, or 0 otherwise.
Messages
Data Tables

col<<Get Values
Returns the values in the column.

col<<Hide(Boolean)
Turns the Hide attribute on or off according to the Boolean argument given.

col<<Input Format(format)
Sets the format used for input and storage for the column. The argument is the name of any JMP format (for example, dmmmyyyy for a data column).

date_col<<Is Transformed On SAS Export
Returns true if the data in the resulting SAS data set for the date column will be changed when it is exported to SAS.

col<<Label(Boolean)
Turns the Label attribute on or off according to the Boolean argument given.

col<<Lock(Boolean)
col<<Set Lock(Boolean)
Turns the Lock attribute on or off according to the Boolean argument given.

col<<Preselect Role(role)
Preselects the role for the column. Choices are Y, X, Weight, Freq, and None or No Role.

col<<Scroll Lock(Boolean)
Turns the Scroll Lock attribute on or off according to the Boolean argument given.

col<<Set Each Value(n)
Sets all the values in the column to n.

col<<Set Field Width(n)
Sets the field width for the column to n.

col<<Set Modeling Type(type)
Sets the modeling type for the variable. Choices are Continuous, Ordinal, or Nominal.
Appendix B  
Messages  
Data Tables  

\[\text{col} \triangleleft \text{Set Name}("name")\]
Sets the name for the column.

\[\text{col} \triangleleft \text{Set Property} \left(\text{"name"}, \{\text{argument list}\}\right)\]
Sets the property \textit{name} to the \textit{expression} given. You can set any standard column property by name or a user-specified property.

\[\text{col} \triangleleft \text{Set Selected}(\text{Boolean})\]
Sets the column to be selected (\text{true} or 1) or not selected (\text{false} or 0).

\[\text{col} \triangleleft \text{Set Values}([\text{matrix}] \text{ or } \{\text{list}\})\]
\[\text{col} \triangleleft \text{Values}([\text{matrix}] \text{ or } \{\text{list}\})\]
Sets values from the matrix (for numeric variables) or list (for character variables).

\[\text{col} \triangleleft \text{Suppress Eval}(\text{Boolean})\]
Turns off automatic calculation of formulas for the column.

\[\text{Rows}\]

\[\text{row} \triangleleft \text{Colors}(n)\]
Assigns the color \textit{n} to the selected rows.

\[\text{row} \triangleleft \text{Exclude}(0|1)\]
\[\text{row} \triangleleft \text{Unexclude}(0|1)\]
Turns the excluded or unexcluded state on for the selected rows according to the Boolean argument given.

\[\text{row} \triangleleft \text{Hide}(0|1)\]
\[\text{row} \triangleleft \text{Unhide}(0|1)\]
Turns the Hide attribute on or off according to the Boolean argument given.

\[\text{row} \triangleleft \text{Label}(0|1)\]
\[\text{row} \triangleleft \text{Unlabel}(0|1)\]
Turns the Label attribute on or off according to the Boolean argument given.
Messages
Data Tables

---

row<<Markers(marker)

Assigns the `marker` to the selected rows.

---

row<<Next Selected

Causes the next selected row in the data table to blink.

---

row<<Previous Selected

Causes the previous selected row in the data table to blink.

---

row<<Row Editor

Opens the Row Editor window for the selected rows.

---

Data Filter

---

dtf<<Auto Clear(0|1)

Clears all currently selected rows before setting a new selection.

---

dtf<<Clear

Clears the currently selected rows.

---

dtf<<Close

Closes the data filter window.

---

dtf<<Columns(col1, col2, ...)

Sets the columns to use in the data filter.

---

dtf<<Data Table Window

Shows the data table that the data filter window is using.

---

dtf<<Delete All

Removes all filters that are set.

---

dtf<<Delete(col1, col2, ...)

Removes the specified columns from the data filter.
dtf<<Display(col, <Size(x, y)>, Blocks Display|List Display|Single Category Display)
Sets how the specified categorical column levels are displayed in the filter. The specified column must be categorical.

dtf<<Location(x, y)
Moves the data filter window to the specified location. X and y are measured in pixels. 0,0 is the top left of the monitor.

dtf<<Make Subset
Creates a new subset data table that contains the rows that are selected in the data filter.

dtf<<Match(Filter Columns(:col1, :col2, ...), Where(WHERE clause))
Sets the filter conditions for each column. The WHERE clause is used for all the columns listed. To use different WHERE clauses for different columns, send the Match message separately for each column.

dtf<<Mode(Select(0|1)|Show(0|1)|Include(0|1))
Sets the action, or mode, that is used when rows are selected using the data filter.

dtf<<Save and restore current row states
Saves the current row states for the data table, and then restores those states when the data filter is closed.

dtf<<Show Columns Selector(0|1)
Displays or hides the column selector after completing a filter.

dtf<<to Clipboard
Creates a WHERE clause from the current state of the data filter and places it on the clipboard, where it can be pasted elsewhere.

dtf<<to Data Table
Creates a WHERE clause from the current state of the data filter and saves it as a property to the data table.

dtf<<to Journal
Creates a WHERE clause from the current state of the data filter and appends it to the current journal. If there is no current journal, a new journal is opened and the WHERE clause is added to it.
**dtf<<to Row State Column**

Creates a row state column whose formula is the WHERE clause.

**dtf<<to Script Window**

Creates a WHERE clause from the current state of the data filter and appends it to the current script window. If there is no current script window, a new script window is opened and the WHERE clause is added to it.

**dtf<<Use Floating Window(0|1)**

Sets whether the data filter window floats on top of its associated data table or behaves as a normal window.

**dtf<<Where(WHERE clause)**

Sets a condition for selecting rows.

---

**Databases**

**dt<<Save Database("connectInfo", "TableName")**

Saves a data table to a database.

---

**Datafeed**

**feed<<Close**

Closes the Datafeed object and its window.

**feed<<Connect(portSettings)**

(Windows only) Sets up port settings for the connection to the device.

**feed<<Disconnect**

(Windows only) Disconnects the device from the Datafeed queue but leaves the Datafeed object active.

**feed<<Get Line**

Returns and removes one line from the Datafeed queue.
**Messages**

**Display Boxes**

**Windows**

```plaintext
feed<<Get Lines
   Returns as a list and removes all lines from the Datafeed queue.

feed<<Queue Line(string)
   Sends one line to the end of the Datafeed queue.

feed<<Restart
   Restarts processing queued lines.

feed<<Set Script(script)
   Assigns the script that is run each time a line of data is received.

feed<<Show Window(Boolean)
   Specifies whether to show a status window for the data feed.

feed<<Stop
   Stops processing queued lines.
```

```plaintext
window<<Bring Window to Front
   Brings the window to the front.

window<<Close Window(<nosave>)
   Closes the window. If the optional argument nosave is specified, the window (journal, project, and so forth) is closed without saving or prompting.

window<<Get Content Size
   Returns the size of the window's contents.

window<<Get Window Icon
   Returns the name of the window's icon.
```
window<<Get Window Position
  Returns the position of the window.

window<<Get Window Size
  Returns the size of the window.

window<<Get Window Title
  Returns the title of the window.

window<<Maximize Display
  Maximizes the window.

window<<Maximize Window
  Maximizes the window.

window<<Minimize Window
  Minimizes the window.

window<<Move Window(x, y)
  Moves the window to the specified position.

window<<On Close(script)
  Runs the script when the window is closed.

window<<Pad Window(Boolean)
  Turns padding around a window's contents on (1) or off (0). The default value is off.

window<<Print Window
  Prints the window to the default printer. Note that the Print window is not opened and user input is not required.

window<<Set Content Size(x, y)
  Sets the size of the window's contents.

window<<Set Main Window
  Sets the specified window as the default window that appears when JMP is run.
window<<Set Window Icon("icon name")
Sets the window’s icon as specified.

window<<Set Window Size(x, y)
Resizes the window.

window<<Show Window(0|1)
1 shows the window (brings it to the front). 0 hides the window. If the window is also minimized (on Windows) or docked (on Macintosh), showing the window restores it to the normal state and brings it to the front.

window<<Size Window(x, y)
Resizes the window.

window<<Zoom Window
Resizes the window to be large enough to show all its contents.

All Display Boxes

db<<Add Text Annotation(Text("string"), Text Box(x1, y1, x2, y2))
Draws a text annotation box that contains the string. The Text Box argument controls where the text annotation box is drawn in relation to the display box, from the upper left corner to the lower right corner.

db<<Append(db2)
Add db2 to the display tree after db.

db<<Child
Returns the child of the box.

db<<Class Name
Returns the name of the display class for the box.

db<<Clone Box
Makes a new copy of the display box.
Messages
Display Boxes

**db<<Close Window**
Closes the containing window.

**db<<Copy Picture**
Puts a picture of the box on the clipboard.

**db<<Delete**
Delete the display box

**db<<Get HTML**
Returns a string containing HTML source for the box.

**db<<Get Journal**
Returns a string containing journal source for the box.

**db<<Get Picture**
Captures \( db \) as a Picture Object.

**db<<Get RTF**
Returns a string containing RTF source for the box.

**db<<Get Script**
Returns the script for recreating the display box.

**db<<Get Size**
Returns either \( \{ x, y \} \) or \( \{ h, v \} \) in pixels:
\[
xy = \text{DisplayBox} \ll \text{Get Size};
\]
Returns \( x \) and \( y \) in pixels:
\[
\{ x, y \} = \text{DisplayBox} \ll \text{Get Size};
\]

**db<<Get Text**
Returns a string containing the text of the box.

**db<<Journal**
Appends the box to the journal.
Appendix B

Messages

Display Boxes

---

**db<<Journal Window**

Appends the containing window of the display box to the journal; compare with *Journal*.

---

**db<<Move Window(x, y)**

Moves the window to the (x,y) location on your screen.

---

**db<<Page Break**

Inserts a page break before the box.

---

**db<<Parent**

Returns the parent of this display box.

---

**db<<Prepend(db2)**

Add *db2* to the display tree before *db*.

---

**db<<Reshow**

Forces a screen-refresh.

---

**db<<Save Capture("path", <format>, <Add Sibling(n)>)**

Saves the display box as a graphic to the specified *path* in the specified *format*. The optional Add Sibling argument adds the number of sibling display boxes to include in the capture. The default value is 1, which captures only the specified display box. Note that the specified portion of the report is not guaranteed to be scrolled into view or unobstructed by other windows. If the display box is not visible, the saved graphic will not contain the contents that you expect.

---

**db<<Save HTML("" | "path", format)**

Saves HTML source and folder of graphics in *format* specified.

---

**db<<Save Journal("" | "path")**

Saves journal source for the box.

---

**db<<Save MSWord("pathname", Native)**

Saves the display box as a Microsoft Word document. (Windows Only)

---

**db<<Save Picture("" | "pathname", format)**

Saves a picture of the box.
**Messages**

Display Boxes

```
db<<Save RTF("" | "pathname", format)
   Saves RTF source with graphics in format specified.
```

```
db<<Save Text("" | "pathname", format)
   Saves a file containing the text of the box.
```

```
db<<Scroll Window(x, y)
   Scrolls the containing window.
```

```
db<<Select
   db<<Deselect
      Selects (highlights) or deselects the box.
```

```
db<<Set Report Title("title")
   Sets a new title.
```

```
Show Properties(db)
   Shows the messages a given display box can interpret.
```

```
db<<Sib
   Returns the sibling of the box.
```

```
db<<Sib Append(db2)
   Appends a display as a sibling to this one. The argument must evaluate to a display box owner or reference.
```

```
db<<Size Window(x, y)
   Resizes the containing window.
```

```
db<<Zoom Window
   Zooms the containing window.
```
Outline Boxes

`outline box<<Close(Boolean)`
Closes the outline box.

`outline box<<Close All Below`
Closes all the node's child nodes.

`outline box<<Close All Like This`
Closes all nodes similar to this outline box.

`outline box<<Close Where No Outlines`
Closes all nodes that do not have children.

`outline box<<Get Title`
Gets the title of the outline box.

`outline box<<Horizontal(Boolean)`
Horizontally aligns the node's children.

`outline box<<Open All Below`
Opens all the node's child nodes.

`outline box<<Open All Like This`
Opens all nodes similar to this outline box.

`outline box<<Set Title("text")`
Changes the title of the outline box.

Frame Boxes

`frame box<<Add Graphics Script(<order>, script)`
Adds a script to draw graphics in the framebox. The optional `order` argument specifies in what order to draw the graphics element. `Order` can be either the keyword Back or an integer that specifies drawing order for a number of graphics elements.
Messages

Display Boxes

frame box<<Background Color(color)
   Changes the background color.

frame box<<Edit Graphics Script
   Brings up a dialog box to view, edit, or delete the current graphics scripts.

frame box<<Frame Size(x, y)
   Resets the size of the frame, in pixel units.

frame box<<Marker Size(size)
   Changes the marker size.

frame box<<Row Colors(color)
frame box<<Row Markers(marker)
frame box<<Row Exclude(Boolean)
frame box<<Row Hide(Boolean)
frame box<<Row Label(Boolean)
   Forwards commands to the data table associated with the report, so that the row states of selected rows can be manipulated.

frame box<<X Axis(min, max, inc, named arguments)
   Scales the X coordinate system.

frame box<<Y Axis(min, max, inc, named arguments)
   Scales the Y coordinate system.

Table Boxes

table box<<Get
   Gets the entries of the table in list form.

table box<<Get As Matrix
   Gets the numeric entries of the table in matrix form.
**Table Boxes**

`table box<<Make Combined Data Table`

Same as `Make Data Table`, but also searches the report for report tables with the same columns and combine all of these into the new data table.

`table box<<Make Data Table(name)`

Turns the table entries into a new data table.

`table box<<Table Style(style)`

Sets the presentation style.

**Tab Boxes**

`tab box<<Get Tab Margin()`

Returns a list of the current margins in pixels for the tab box in this order: `{left, top, right, bottom}`.

`tab box<<Set Tab Margin(n|{...})`

Sets the tab margin for the tab box. If a single number is specified, all four margins are set to that number of pixels. If a list of two numbers is specified, the left and right margins are set to the first number, and the top and bottom margins are set to the second number. If a list of four numbers is specified, the margins are set in this order: `{left, top, right, bottom}`.

`tab box<<Show Tabs(0|1)`

Shows or hides the tabs for tab boxes. If you hide the tabs, you need to provide another way to select and show tabs. For example, a list box that contains a list of references to the tabs. The default value is 1.

**Text Boxes**

`text box<<Font Color(n)`

Sets the color for `Text` strings.

`text box<<Get Text`

Returns the string content of the box.

`text box<<Rotate Text(direction)`

Rotates the text 90 degrees left or right, or returns it to the horizontal.
**Messages**

**Display Boxes**

- `text box<<Set Font(fontname)`
  Sets the font for text strings.

- `text box<<Set Font Size(n)`
  Sets the font size in points for text strings.

- `text box<<Set Font Style(style)`
  Sets the font style for text strings.

- `text box<<Set Script(script)`
  Associate a script to a text box. The script executes when the user hits enter (or the text edit box otherwise loses focus).

- `text box<<Set Text("string")`
  Changes the string content of the box.

- `text box<<Set Wrap(n)`
  Set the wrap point, in pixels, in pixels ($n$).

**Axis Boxes**

- `axis box<<Add Axis Label("string")`
  Adds an axis label with the string given.

- `axis box<<Add Ref Line(number, linestyle, color)`
  Adds a reference line at $number$ in the $linestyle$ and $color$ specified.

- `axis box<<Axis Settings(choices)`
  Applies numerous axis changes in a single message. Choose from the messages below for $choices$ arguments.

- `axis box<<Decimal(width, decimalplaces)`
  Changes the numeric format for axis values.

- `axis box<<Format("name")`
  Changes to the numeric format given by $name$. 
axis box<<Inc(n)
   Set the increment between ticks.

axis box<<Interval(choice)
   Specifies the units used for Inc() with date/time data formats.

axis box<<Max(max)
   Changes just the maximum of the axis scale.

axis box<<Min(min)
   Changes just the minimum of the axis scale.

axis box<<Minor Ticks(number)
   Specifies the number of minor tick marks between major tick marks.

axis box<<Remove Axis Label
   Removes any label added with Add Axis Label.

axis box<<Revert Axis
   Restores the axis' original settings (from time of creation).

axis box<<Rotated Labels(Boolean|named_argument)
   If you specify 0, labels are not rotated. If you specify 1, the labels are rotated vertically.
   You can also use a keyword to specify the rotation angle more precisely: horizontal, vertical, perpendicular, parallel, and angled.

axis box<<Scale(type)
   Changes the scale of the axis to type (Log or Linear).

axis box<<Show Labels(Boolean)
   Shows or hides labels for the axis values.

axis box<<Show Major Grid(Boolean)
   Adds or removes grid lines at the major tick values.
axis box<<Show Major Ticks(Boolean)
    Shows or hides major tick marks.

axis box<<Show Minor Grid(Boolean)
    Adds or removes grid lines at the minor tick values.

axis box<<Show Minor Ticks(Boolean)
    Shows or hides minor tick marks.

**Number Col Boxes**

number col box<<Add Element(item)
    Adds the item to the Number Col Box. Item can be a single number, a list of numbers, or a matrix of numbers.

number col box<<Get
number col box<<Get(i)
    Gets the values in a list, or the i\textsuperscript{th} value.

number col box<<Get As Matrix
    Gets the values in a matrix, specifically a column vector.

number col box<<Get Format
    Returns the current format (width, decimalplaces). A decimalplaces > 100 indicates date/time values.

number col box<<Get Heading
    Returns the column heading text.

number col box<<Set Format(width, decimalplaces)
number col box<<Set Format("format", width)
    Sets the format. Set decimalplaces > 100 for date/time values. Set decimalplaces = 97 for p-value format.
    For example,
    \begin{verbatim}
    << Set Format( 10, 2)
    << Set Format("Scientific", 10 )
    \end{verbatim}
number col box<<Set Heading("string")
Changes the column heading text.

Slider Boxes

slider box<<Get()
Returns the current value of the slider.

slider box<<Set(n)
Sets the value of the slider.

slider box<<Get Min()
Returns the minimum value possible for the slider.

slider box<<Get Max()
Returns the maximum value possible for the slider.

slider box<<Set Script(<script>)
Sets a script to be run when the slider is updated.

String Col Boxes

string col box<<Add Element(item)
Adds the item to the String Col Box. Item can be a single quoted string, a list of quoted strings, or a matrix of quoted strings.

string col box<<Get
string col box<<Get(i)
Gets the values in a list, or the i'th value.

string col box<<Get Heading
Returns the column heading text.

string col box<<Set Heading("text")
Changes the column heading text.
Matrix Boxes

matrix box<<Get

Returns the matrix contents.

matrix box<<Make Into Data Table

Turns the matrix into a new data table.

matrix box<<Set Format(width, decimalplaces)

Sets the numeric format for matrix elements.

Nom Axis Boxes

nom axis box<<Divider Lines(Boolean)

Adds or removes divider lines between labels in the axis box.

nom axis box<<Lower Frame(Boolean)

Adds or removes a lower frame around the axis.

nom axis box<<Rotated Tick Labels(Boolean)

Rotates or unrotates the labels at each tick value.

Panel Boxes

panel box<<Get Title

Gets the title of the panel box.

panel box<<Set Title("text")

Changes the title of the panel box.

Border Boxes

border box<<Set Background Color(color)

Sets the background color for a border box.
Dynamic Link Libraries (DLLs)

```plaintext
dll object<<Call DLL(function_name, signature, arguments)
Calls the specified function in the DLL with the specified signature and arguments.

dll object<<Declare Function("name", <named_arguments>)
Declares the return type and argument types for the specified function so that it can be successfully
invoked. You can use one of the named arguments for
Convention: STDCALL or PASCAL, or CDECL.
The type argument for Returns takes the same named arguments as Arg.

Named Arguments

Alias("name")

Arg(type, <"description">, <access_mode>, <array>) Arg is optional and can appear
multiple times, once for each argument to be sent to the function.
Type is one of these keywords that specifies the argument type: Int8, UInt8, Int16, UInt16,
Int32, UInt32, Int64, UInt64, Float, Double, AnsiString, UnicodeString, Struct,
IntPtr, UIntPtr, or ObjPtr.
"description" is an optional string that describes the argument for reference.
access_mode is an optional keyword that specifies how the argument is passed. input specifies that
the argument is passed by value. output specifies that the argument is passed by address with the
initial value undefined. update specifies that the argument is passed by reference and the value of the
JSL variable is set as the initial value. The default value is input.
array is an optional keyword. It is valid only if the Type is specified as Double and the access_mode
is specified as either input or update. Specifies that the exported function expects an array of
doubles.

Convention(calling_convention) An optional keyword that specifies the calling convention:
STDCALL or PASCAL, or CDECL. The default value is STDCALL. STDCALL and PASCAL are equivalent.
MaxArgs(n) An optional integer that specifies the maximum number of arguments that can be
supplied.
MinArgs(n) An optional integer that specifies the minimum number of arguments that can be
supplied.

Returns(type) Optional. Specifies the data type that the function returns: Int8, UInt8, Int16,
UInt16, Int32, UInt32, Int64, UInt64, Float, Double, AnsiString, UnicodeString,
Struct, IntPtr, UIntPtr, or ObjPtr.

StackOrder(order) Optional keyword that specifies the order in which arguments are to be placed
on the stack when calling the function. Valid values are L2R (left-to-right) and R2L (right-to-left).
The default value is R2L.

StackPop(pop) Optional keyword that specifies how the exported function expects the stack to be
cleared after the function returns. Valid values are CALLER and CALLEE. The default value is CALLEE.
```
StructArg(Arg(...), <Arg(...)>, ..., <access_mode>, <pack_mode>, 
<"description">) Optional and can appear multiple times. If an exported DLL function requires 
that a structure argument be passed in as a parameter, use StructArg to declare the structure 
members. The parameters to StructArg are Arg arguments using the same syntax as above for Arg 
arguments to DeclareFunction (one for each structure member), an access_mode indicator and a 
pack_mode indicator.

access_mode is an optional keyword that indicates whether the struct argument should be passed by 
value (input) or by reference (update).

pack_mode is an optional integer that determines how the structure is packed. Valid values are 1, 2, 
4, 8, and 16. The default value is 8.

"description" is an optional, quoted string that contains a description of the structure for 
reference.

dll object<<Get Declaration JSL

Sends the declaration JSL from the DLL object to log.

dll object<<Show Functions

Sends the declared functions for the DLL object to the log.

dll object<<Unload DLL

Unloads the DLL.

Platforms

obj<<Bring Window To Front

Brings the current window to the front.

obj<<Close Window

Closes window identified by obj, typically a platform surface.

obj<<Data Table Window

Makes the associated data table window active (front-most).

obj<<Get Script

Returns script to reproduce the analysis as an expression.
Appendix B

Messages
Platforms

---

**obj<<Get Window Position**

Gets the position of the window. Returns an ordered pair.

---

**obj<<Get Window Size**

Gets the window size, in pixels. Returns an ordered pair.

---

**obj<<Journal Window**

Appends the contents of the window to the journal.

---

**obj<<Maximize Window**

Maximizes the window. Equivalent to pushing the maximize button in the corner of the window. This message takes an optional Boolean argument:

```c
// maximize the window:
obj<<Maximize Window(1)
// restore the window:
obj<<Maximize Window(0)
```

---

**obj<<Minimize Window**

Minimizes the window. Equivalent to pushing the minimize button in the corner of the window. This message takes an optional Boolean argument:

```c
// minimize the window:
obj<<Minimize Window(1)
// restore the window:
obj<<Minimize Window(0)
```

---

**obj<<Move Window(x, y)**

Moves the window to the (x,y) location on your screen.

---

**obj<<Print Window**

Sends the selected window to the printer.

---

**obj<<Redo Analysis**

Launches the platform again with the same options.

---

**obj<<Report**

Report(obj)

Returns a display box reference for the report in the platform window.
obj<<Save Script for All Objects
  Saves script to reproduce all analyses found within the object’s window in the Script Journal window.

obj<<Save Script to Datatable
  Saves script to reproduce analysis as a property in the associated data table.

obj<<Save Script to Report
  Saves script to reproduce analysis as a text box at the top of the report.

obj<<Save Script to Script Window
  Saves a script to reproduce analysis in the Script Journal.

obj<<Scroll Window(x, y)
obj<<Scroll Window({x, y})
  Scrolls the window x pixels to the left and y pixels down from the current position.

obj<<Show Window(0|1)
  1 shows the window (brings it to the front). 0 hides the window. If the window is also minimized (on Windows) or docked (on Macintosh), showing the window restores it to the normal state and brings it to the front.

obj<<Size Window(x, y)
  Resizes the window to x pixels wide by y pixels high.

obj<<Zoom Window
  Resizes the window to the maximum size of its contents.
Appendix B

Messages

R Integration Messages

The R interfaces are also scriptable using an R connection object. A scriptable R connection object can be obtained using the R `Connect()` JSL function.

```
rconn<<Control(Interrupt(1))
```

If `Async` is set to true (1) for `R Submit()`, this message immediately stops the execution of the R code that was submitted.

```
rconn<<Disconnect()
```

Disconnects this R connection.

```
rconn<<Is Connected()
```

Returns 1 if the R connection is active, 0 otherwise.

```
rconn<<Send("name", named_arguments)
```

Send the specified JMP variable to R.

Returns

0 if successful, nonzero otherwise.

Argument

"name" required. The name of a JMP variable to be sent to R.

Arguments for Data Tables

- `Selected(0|1)` Optional, Boolean. If true, sends only the selected rows from the referenced data table to R.
- `Excluded(0|1)` Optional, Boolean. If true, sends only the excluded rows from the referenced data table to R.
- `Labeled(0|1)` Optional, Boolean. If true, sends only labeled rows from the referenced data table to R.
- `Hidden(0|1)` Optional, Boolean. If true, sends only hidden rows from the referenced data table to R.
- `Colored(0|1)` Optional, Boolean. If true, sends only colored rows from the referenced data table to R.
- `Markered(0|1)` Optional, Boolean. If true, sends only markered rows from the referenced data table to R.
- `Row States(0|1, <named arguments>)` Optional. Includes a Boolean argument and optional named arguments. Sends row state information from the referenced data table to R by adding an additional data column named "RowState". The row state value consists of individual settings with the values shown in Table B.1.
Send the specified JMP data file to R.

Gets the specified variable from R.

Returns
The value of the specified variable.

Arguments
"name" A quoted string that contains the name of a JMP variable to be retrieved from R.

Gets the last graphics object written to the R graph display window. The graphics object can be returned in different graphic formats.

Returns
A JMP picture object.

Arguments
type The format the R graph display window contents are to be converted to. Valid formats are as follows:

- png
- bmp
- jpeg, jpg
- tiff, tif

Table B.1 Row States

| Multiple row states are created by adding together individual settings. | Selected = 1
| Excluded = 2
| Labeled = 4
| Hidden = 8
| Colored = 16
| Markered = 32 |

Arguments

Colors(Boolean) Optional, Boolean. If true, sends row colors and adds an additional data column named "RowStateColor".

Markers(Boolean) Optional, Boolean. If true, sends row markers and adds an additional data column named "RowStateMarker".
rconn<<Submit("code", named_arguments)

Submits the R code to the active R connection.

Returns
0 if successful, nonzero otherwise.

Arguments
"code" A string that contains the R code to submit.
Expand(0|1) Optional, Boolean. Performs an Eval Insert() on the R code before submitting the code.
Echo(0|1) Optional, Boolean. Echoes the R source lines to the JMP log. The default value is true.

Rconn<<Submit File("filepath")

Submits statements to R using the file in the filepath.

Arguments
"filepath" A string that contains the pathname to the file that contains R code to be executed.

rconn<<Execute({ list of inputs }, { list of outputs }, "code", named_arguments)

Submits the R code to the active R connection given a list of inputs. Upon completion, a list of outputs are retrieved.

Returns
0 if successful, nonzero otherwise.

Arguments
{ list of inputs } List of JMP variable names to be sent to R as inputs.
{ list of outputs } List of JMP variable names to be retrieved from R as outputs.
"code" A string that contains the R code to submit.
named_arguments See rconn<<Submit("code", named_arguments) on p. 681.

rconn<<Control(named_arguments)

Controls the execution of R.

Returns
Void.

Argument
Echo(Boolean) Optional, Boolean. Echoes the R source lines to the JMP log.
**Messages**

R Integration Messages

---

**rconn<<Get Version()**

Gets the current version of R that is installed.

**Returns**

A vector of length 3 containing the R version number.

---

**rconn<<JMP Name To R Name("name")**

Maps a JMP Name to its corresponding R Name using R variable name naming rules.

**Returns**

A string that contains the R name.

**Arguments**

"name" A string that contains the name of a JMP variable to be sent to R.
SAS Integration Messages

Metadata Server Objects

metaserver<<Disconnect()

Function
   Disconnects the metadata server.

Returns
   Void.

metaserver<<Get Display Name()

Function
   Gets the display name of the metadata server.

Returns
   A string.

metaserver<<Get Host Name()

Function
   Gets the host (machine) name of the metadata server.

Returns
   A string.

metaserver<<Get Port()

Function
   Gets the port used for the metadata server connection.

Returns
   An integer.

metaserver<<Get User Identity()

Function
   Gets the identity of the connected user as defined in metadata.

Returns
   A string.
SAS Integration Messages

metaserver<<Get User Name()

Function
   Gets the user name (login ID) that was used for the metadata server connection.

Returns
   A string.

SAS Server Objects

sasconn<<Assign Libref(libref, path, engine, engine options)

Function
   Assign a SAS libref on this SAS server connection.

Returns
   Void.

Arguments
   See SAS Assign Lib Refs(libref, path, <engine>, <engine options>) on p. 557 in the
   "JSL Syntax Reference".

sasconn<<Cancel Submit()

Function
   Cancels the currently running SAS Submit for this server that is presumably running asynchronously.

Returns
   1 if a running submit was found and canceled; 0 otherwise.

sasconn<<Clear Log History()

Function
   Clears the SAS Log history for this server.

Returns
   Void.

sasconn<<Clear Output History()

Function
   Clears the SAS Output history for this server.

Returns
   Nothing
Appendix B

Messages

SAS Integration Messages

sasconn<<Connect(<named arguments>)

Function
Attempt to reconnect a SAS server connection object that has become disconnected.

Returns
1 if the connection was successful, 0 otherwise.

Named Arguments
All named arguments are optional.

UserName(name) A quoted string that contains the user name for the connection.

Password(password) A quoted string that contains the password for the connection.

Prompt(Always|Never|IfNeeded) A keyword. Always means always prompt before attempt to
connect. Never means never prompt even if the connection attempt fails (just fail with an error
message going to the log), and IfNeeded (the default) means prompt if the attempt to connect with
the given arguments fails (or is not possible with the information given).

sasconn<<Deassign Libref(libref)

Function
De-assign a SAS libref on this SAS server connection.

Returns
Void.

Arguments
libref Quoted string that contains the library reference.

sasconn<<Disconnect()

Function
Disconnect this SAS server connection.

Returns
Void.

sasconn<Does Module Exist(moduleName)

Function
Determines whether the specified SAS module exists in the SAS installation represented by the SAS
connection. This can be helpful in determining whether certain SAS products are installed. The SAS
DATA Step function MODEXIST is used to determine module existence. Because MODEXIST is new
for SAS 9.2, this function throws an exception if it is called for a SAS connection that is not version SAS
9.2 or later.

Returns
1 if the specified module is found to exist, 0 if it does not exist.

Argument
moduleName The SAS module the existence of which should be checked. Do not include any
extension.
**Messages**

SAS Integration Messages

---

`sasconn<<Export Data(dt, library, dataset, <named arguments>)`

**Function**
Exports a JMP data table to a SAS data set in a library on the current global SAS server connection.

**Returns**
1 if the data table was exported successfully; 0 otherwise.

**Arguments**
See SAS Export Data(dt, "library", "dataset", <named_arguments>) on p. 559 in the "JSL Syntax Reference".

---

`sasconn<<Get Data Sets(libref)`

**Function**
Returns a list of the data sets defined in a SAS library on this SAS server connection.

**Returns**
List of strings.

**Arguments**

- `libref` Quoted string that contains the SAS libref or friendly library name associated with the library for which the list of defined SAS data sets will be returned.

---

`sasconn<<Get Error Count()`

**Function**
Gets the count of the number of errors encountered in the previous SAS Submit.

**Returns**
An integer.

---

`sasconn<<Get File(source, dest)`

**Function**
Download a file from this SAS server connection.

**Returns**
Void.

**Arguments**
See SAS Get File(source, dest) on p. 560 in the "JSL Syntax Reference".

---

`sasconn<<Get File Names(fileref)`

**Function**
Get a list of filenames found in the given fileref on this SAS server connection.

**Returns**
List of strings.

**Arguments**

- `fileref` Quoted string that contains the name of fileref from which to retrieve filenames.
sasconn<<Get File Names In Path(path)

Function
Get a list of filenames found in the given path on this SAS server connection.

Returns
List of strings.

Arguments
path  Quoted string that contains the directory path on the server from which to retrieve filenames.

sasconn<<Get File Refs()

Function
Get a list of the currently defined SAS filerefs on this SAS server connection.

Returns
List of strings.

sasconn<<Get Librefs(<named arguments>)

Function
Get a list of the currently defined SAS librefs on this SAS server connection.

Returns
List of strings.

Arguments
See SAS Get Lib Refs(<named arguments>) on p. 561 in the "JSL Syntax Reference".

sasconn<<Get Log()

Function
Retrieve the SAS Log from the last SAS Submit from this SAS server connection.

Returns
String.

sasconn<<Get Output()

Function
Retrieve the listing output from the last submission of SAS code to this SASServer object.

Returns
String.
**sasconn<<Get Results()**

**Function**
Retrieve the results of the previous SAS Submit as a scriptable object, which allows significant flexibility in what to do with the results.

**Returns**
A SAS Results Scriptable Object.

**sasconn<<Get Submit Status()**

**Function**
Gets the current status of a SAS Submit for this server that is presumably running asynchronously.

**Returns**
1 if the submit has not started; 2 if the submit is running; 3 if the submit has been canceled; 10 if the submit has completed successfully; 11 if the submit has completed with errors.

**sasconn<<Get Var Info("libref", "dataset", <password>)**

**Function**
Returns information about the variables the specified SAS data set.

**Arguments**
"libref"  Quoted string that contains the library reference to de-assign.
"dataset"  Quoted string that contains the name of the data set from which to retrieve variable names.
Password("password")  A quoted string that contains the password for the connection.

**sasconn<<Get Var Names(libref, dataset, <named arguments>)**

**Function**
Retrieves the variable names contained in the specified data set on this SAS server connection.

**Returns**
List of strings.

**Arguments**
See SAS Get Var Names(string, <dataset>, <password("password")>) on p. 562 in the "JSL Syntax Reference".
sasconn<<Get Version(<long>)

Function
Returns the SAS version as a string such as “9.1” or “9.2”.

Returns
A string that contains the SAS version.

Argument
long An optional keyword that specifies to return the long SAS version, which corresponds to the
SYSVLONG SAS macro. For example, "9.02.02M0P01152009".

sasconn<<Get Work Folder()

Function
Returns the full path of the folder corresponding to the WORK library for this server.

Returns
A string that contains the work folder path.

sasconn<<Import Data(library, dataset, <named arguments>)

Function
Import a SAS data set from this SAS server connection into a JMP table.

Returns
JMP Data Table object.

Arguments
See SAS Import Data(string, <dataset>, <named arguments>) on p. 562 in the "JSL Syntax Reference".

sasconn<<Import Query(sqlquery, <named arguments>)

Function
Execute the requested SQL query on this SAS server connection, importing the results into a JMP data
table.

Returns
JMP data table object.

Arguments
See SAS Import Query("sqlquery", <named arguments>) on p. 564 in the "JSL Syntax Reference".

sasconn<<Is Connected()

Function
Determine whether this SAS Server object is currently connected to SAS.

Returns
1 if sasconn is connect, 0 otherwise.
sasconn<<Is Product Available(productName)

Function
Determine whether the specified SAS product is both licensed and installed in the session represented by the SAS connection. The SAS DATA Step functions SYSPROD and MODEXIST are used to determine the licensed and installed status of the product.

Returns
1 if the specified product is licensed, 0 if the product is not licensed, or -1 if the specified product is not recognized by SAS. This function throws an exception if the requested product is not one for which JMP knows how to check the installed status.

Argument
productName The SAS product for which licensing should be checked. The product name can be specified with or without the “SAS/” prefix.

Note
The MODEXIST function is new in SAS 9.2. For SAS 9.1.3, this function only checks the license, not the installed status. In other words, for SAS 9.1.3, this function operates the same way as Is Product Licensed().

sasconn<<Is Product Licensed(productName)

Function
Determines whether the specified SAS product is licensed in the session represented by the SAS connection. The SAS DATA Step function SYSPROD is used to determine the licensing status of the product.

Returns
1 if the specified product is licensed, 0 if the product is not licensed, or -1 if the specified product is not recognized by SAS.

Argument
productName The SAS product for which licensing should be checked. The product name can be specified with or without the “SAS/” prefix.

sasconn<<Kill Session(<n>)

Function
Immediately terminates the SAS connection.

Returns
Void.

Arguments
n An optional number. The system waits n seconds for a normal shut-down before immediately terminating the SAS connection.
sasconn<<Load Text File(path, <named arguments>)

Function
Download the file specified in path from the active global SAS server connection and retrieve its contents as a string.

Returns
String.

Arguments
See SAS Load Text File("path") on p. 565 in the "JSL Syntax Reference".

sasconn<<Open Log Window()

Function
Opens (or brings to the front) the SAS Log window for this server.

Returns
Void.

sasconn<<Open Output Window()

Function
Opens (or brings to the front) the SAS Output window for this server.

Returns
Void.

sasconn<<Open SAS Results()

Function
Open the results from the previous SAS Submit. Intended to be used with asynchronous SAS submits or the use of the OnSubmitComplete option to SAS Submit to give the JSL author a way to conditionally open the results of a submit.

Returns
Void.

sasconn<<Open Submit Results()

Function
Opens all the results from the last SAS Submit command.

Returns
Void.
Messages
SAS Integration Messages

sasconn<<Send File(source, dest)

Function
Upload a file to this SAS server connection.

Returns
Void.

Arguments
See SAS Send File(source, dest) on p. 566 in the "JSL Syntax Reference".

sasconn<<Submit(sasCode, <named arguments>)

Function
Submit some SAS code to this SAS server connection.

Returns
Void.

Arguments
See SAS Submit("sas code", <named arguments>) on p. 566 in the "JSL Syntax Reference".

sasconn<<Submit File(filename, <named arguments>)

Function
Submit a SAS code file to this SAS server connection.

Returns
Void.

Arguments
See SAS Submit File(filename, <named arguments>) on p. 567 in the "JSL Syntax Reference".

Stored Processes

stp<<Begin Run(<named arguments>)

Function
Start this stored process executing in the background. This message is paired with End Run, which should also be called at some point after Begin Run to wait for the stored process to complete.

Returns
-1 = execution failed.
1 = not started.
2 = running.
3 = canceled.
10 = completed successfully.
11 = completed with errors.

Arguments
Same as Run, except AutoOpenResults and NoAlerts are not supported. They are available on EndRun.
AutoResume(<filename>) Optional, quoted string. If specified with no argument, it specifies that the stored process results should be auto-opened when the stored process completes. If filename is specified, filename is opened rather than all results of the stored process being auto-opened.

AutoResumeScript(script) Optional, quoted string that specifies that after stored process execution completes, script should be evaluated. If the script is a function taking at least one argument, the function is evaluated with the scriptable stored process object passed as the first (and only) argument. AutoResume and AutoResumeScript are mutually exclusive.

stp<<Delete Results(<named arguments>)

Function
Delete all results from the execution of this stored process.

Returns
1 if deletion is successful, 0 otherwise (error message to JMP log).

Arguments
NoAlerts(0|1) Optional, Boolean. If True, the user is not prompted for confirmation before the attempt is made to delete results.
DeleteDirectory(0|1) Optional, Boolean. If true, deletes the directory containing the stored process results along with the result files themselves. The default value is true.

stp<<Edit Param Values()

Function
Opens the stored process parameter window for interactively setting parameter values.

Returns
1 if the user clicks OK to dismiss the window, 0 if the user clicks Cancel.

stp<<End Run(<named arguments>)

Function
Waits a specified amount of time (or forever) for a stored process started with Begin Run to complete. If the stored process is complete, retrieves the results, and opens them.

Returns
-1 = execution failed.
1 = not started.
2 = running.
3 = canceled.
10 = completed successfully.
11 = completed with errors.

Arguments
AutoOpenResults(0|1) Optional, Boolean. If True, results are automatically opened if the stored process completes in the time specified by MaxWait. If False, results are not automatically opened, and can be manually opened via the object returned by the Get Results message. Default is True.
MaxWait(milliseconds)  Optional, an integer. Specifies the maximum amount of time in milliseconds to wait for the stored process to complete. If MaxWait is not specified, End Run waits forever for the stored process to complete.

NoAlerts(0|1)  Optional, Boolean. If True, error messages are sent to the JMP log rather than message boxes. The default value is False.

stp<<Get Metadata Id()
Function
  Returns the metadata ID of the stored process.
Returns
  String.

stp<<Get Metadata Path()
Function
  Returns the full metadata path of the stored process.
Returns
  String.

Stp<<Get Name()
Function
  Returns the name of the stored process.
Returns
  String.

stp<<Get Param Enum Labels(name)
Function
  Get the enumeration labels for a parameter.
Returns
  List of strings.
Arguments
  name  Quoted string that contains the name of the parameter whose enumeration labels to retrieve.

stp<<Get Param Enum Values(name)
Function
  Get the possible enumerated values for a parameter.
Returns
  List of strings.
Arguments
  name  Quoted string that contains the name of the parameter whose possible enumerated values to retrieve.
**Appendix B**  
**Messages**  
SAS Integration Messages

---

**stp<<Get Param Names(<named arguments>)**

**Function**
Get a list of parameter names for this stored process of specific types.

**Returns**
List of strings.

**Arguments**

- **Visible(0|1)** Optional, Boolean. If *True*, get only visible parameters. If *False*, get only non-visible parameters. If not specified, get both visible and non-visible parameters.
- **Modifiable(0|1)** Optional, Boolean. If *True*, get only modifiable parameters. If *False*, get only non-modifiable parameters. If not specified, get both modifiable and non-modifiable parameters.
- **Required(0|1)** Optional, Boolean. If *True*, get only required parameters. If *False*, get only non-required parameters. If not specified, get both required and non-required parameters.
- **Expert(0|1)** Optional, Boolean. If *True*, get only expert parameters. If *False*, get only non-expert parameters. If not specified, get both expert and non-expert parameters.

---

**stp<<Get Param Value(name)**

**Function**
Get the current value of the specified parameter.

**Returns**
String.

**Arguments**

- **name** Quoted string that contains the name of the parameter whose value to retrieve.

---

**stp<<Get Results()**

**Function**
Get the results generated by the execution of this stored process as a scriptable object.

**Returns**
SAS Results scriptable object.

---

**stp<<Get Status()**

**Function**
Get the execution status of the stored process.

**Returns**

- `-1` = execution failed.
- `1` = not started.
- `2` = running.
- `3` = canceled.
- `10` = completed successfully.
- `11` = completed with errors.
Messages
SAS Integration Messages

stp<<Get Status Message()

Function
Get the message associated with the failure of the stored process, if any.

Returns
String.

stp<<Reset Param Values()

Function
Reset all parameter values to their metadata-defined default values.

Returns
Void.

stp<<Run(<named arguments>)

Function
Execute this stored process object in the foreground.

Returns
-1 = execution failed.
1 = not started.
2 = running.
3 = canceled.
10 = completed successfully.
11 = completed with errors.

Arguments
AutoOpenResults(0|1) Optional, Boolean. If True, results are automatically opened when the
stored process completes. If False, results are not auto-opened, and can be manually opened via the
object returned by the GetResults message. The default value is True.

UserName(username) Optional, quoted string that contains the user name under which to run the
stored process.

Password(password) Optional, quoted string that contains the password for Username.

AuthDomain(authDomain) Optional, quoted string that contains the authentication domain of the
credentials (username, password) given.

ODSDest(dest) Optional, quoted string that contains an ODS destination (HTML, PDF,
tagsets.SASReport12) for any ODS-generated results from the stored process. This requires the stored
process SAS code to call %STPBEGIN. The default value is HTML.

GraphicsDevice(device) Optional, quoted string that contains the SAS graphics device to use
when generating graphics in ODS results. This requires the stored process SAS code to call
%STPBEGIN. The default value is GIF.

ODSStyle(styleName) Optional, quoted string that contains an ODS style to apply to the results.
This requires the stored process SAS code to call %STPBEGIN. There is no default value.
ODSStyleSheet(cssFile)  Optional, quoted string that contains the full path to a CSS file on the client machine that is to be applied to generated ODS results. This requires the stored process SAS code to call %STPBEGIN. There is no default value.

NoAlerts(0|1)  Optional, Boolean. If True, error messages are sent to the JMP log rather than message boxes. The default value is False.

stp<<Set Param Value(name, value)
Function
Sets the value of the specified stored process parameter to the specified value.

Returns
1 if successful, 0 otherwise (value can violate the parameter's constraints).

Arguments
name  Quoted string that contains the name of the parameter whose value to set.
value  Quoted string that contains the value to which to set the parameter.

stp<<Set Results Directory(directory)
Function
Sets the directory on the client machine where stored process results are placed.

Returns
String.

Arguments
directory  Quoted string that contains the full path of the directory where results of the stored process execution should be placed. The directory must exist or be creatable. If the results directory is not set, a temporary location appropriate for the operating system will be used, and that directory can be retrieved from the stored process Results scriptable object after the stored process executes.

SAS Results

results<<Delete All Result Files()  
Function
Deletes all files created by the SAS Submit or Stored Process execution. Note that any result files that are still in use are not deleted.

Returns
1 if the deletion was successful; 0 if some of the files could not be deleted.

results<<Get Directory()
Function
Gets the directory where the results generated by the stored process or SAS submit are located.

Returns
String.
results<<Get Log()

Function
Get the SAS Log from the execution of the stored process or SAS submit.

Returns
String.

results<<Get Main Result File Name(<named arguments>)

Function
Gets the full path of the main result file generated by the stored process or SAS submit.

Returns
String.

Arguments
FullPath(0|1) Optional, Boolean. If True, the main result filename is returned as a full path. The default value is False.

results<<Get Output()

Function
Gets the SAS Listing output from the execution of the stored process or SAS submit.

Returns
String.

results<<Get Output Datasets()

Function
Get a list of output data set generated by the SAS Submit that created this SAS Results object.

Returns
A list of data set names in the form “libname.membername”.

results<<Get Result File Info(<MIMType("mime-type")>, <FullPath>)

Function
Get information about result files that were generated by the execution of the stored process or SAS submit.

Returns
List of two lists of strings. The first list is filenames, and the second list is the MIME-type of the corresponding file from the first list.

Arguments
MIMType(mime_type) Optional, quoted string that restricts the set of files for which information is returned to only those files with the specified MIME-type. If not specified, information about all generated files is returned.
FullPath Optional, Boolean. If True, the filename returned for each result file is returned as a full path; if False, only the name of the file is returned. The default value is False.
**results<<Make JMP Report()**

**Function**
Parses the ODS XML results and creates a JMP report.

**Returns**
The display box for the report.

**results<<Open All Results()**

**Function**
Opens all results generated by the execution of the stored process or SAS submit.

**Returns**
Void.

**results<<Open Result File(filename, <named arguments>)**

**Function**
Attempts to open the result file with the given name.

**Returns**
JMP Data Table if one was opened.

**Arguments**

- **filename** Quoted string that contains the name of the file from the generated results to open.
  - filename should just be the name of the file, not the full path. If `filename` is a filename with no extension, both JMP data tables and JSL scripts in the results are searched for a match, and if both exist, both are opened.

- **RunScript(0|1)** Optional, Boolean. If `True`, and if `filename` is a JSL script, the script is executed. If `False`, `filename` is just opened, even if it is JSL.

**results<<Run Script(filename)**

**Function**
Looks for a JSL file in the results with the given filename and runs it if it finds it.

**Returns**
Void.

**Arguments**

- **filename** Quoted string that contains the name of the JSL file from the generated results to open.
  - `Filename` should just be the name of the file, not the full path, and it does not need to include the `.jsl` extension.

---

**Messages for Schedule**

See also `Schedule(n, script)` on p. 615 in the "JSL Syntax Reference".
Messages

For more information about scheduling actions, see “Scheduling Actions,” p. 433 in the “Advanced Concepts” chapter.

sch<<Clear Schedule()
---
Cancels all events in a schedule queue.

sch<<Close()
---
Cancels all events in a schedule queue.

sch<<Restart()
---
Cancels all events in a schedule queue.

sch<<Show Schedule()
---
Cancels all events in a schedule queue.

sch<<Stop()
---
Cancels all events in a schedule queue.

Sockets

skt<<Accept(<callback, timeout>)

Function
Tells the server socket to accept a connection and return a new connected socket.

Returns
A list of up to four items. The first is a string that echoes the command ("accept"). The second is a string, either "ok" or an error. The third is a string that specifies the name of the machine that just connected. The fourth is a reference to the socket that you can send more messages.

Arguments

  callback an optional argument that specifies the name of a function to receive the data.

  timeout if you use a callback, timeout specifies how long the function should wait for an answer. For a server socket, 0 is an acceptable value because a server should not shut down because no one has connected to it recently.
Appendix B

Messages

Sockets

---

skt<<bind("localhost", port)

Function
Associates a port on the local machine with the socket.

Returns
A list of two strings. The first string is the command name ("bind") and the second is “ok” if successful or an error.

Argument
- localhost Specifies the local machine. You cannot bind to another machine.
- port The port that should be used.

---

skt<<Close()

Function
Closes a socket.

Returns
A list of two strings. The first string is the command name ("close") and the second is “ok” if successful.

---

skt<<Connect(socketname, port)

Function
Connects to a listening socket.

Returns
A list of two strings. The first string is the command name ("connect") and the second is “ok” for a successful connection or an error sent back by the other socket.

Arguments
- socketname the name of the other socket. If you are connecting to a web server, this is the web address (the name is preferred to the IP address).
- port The port of the other socket to connect through.

---

skt<<GetPeerName()

Function
Retrieves the address and port of the socket at the other end of the connection.

Returns
A list of four strings. The first echoes the command ("getpeername"). The second is either “ok” or an error. The third and fourth are the address and the port.
skt<<GetSockName()

Function
Retrieves the address and port of the socket at this end of the connection.

Returns
A list of four strings. The first echoes the command ("getsockname"). The second is either "ok" or an error. The third and fourth are the address and the port.

skt<<ioctl(FIONBIO, 1)

Function
Controls the socket's blocking behavior.

Returns
A list of two strings. The first string is the command name ("ioctl") and the second is "ok" if successful or an error.

Arguments
FIONBIO, 1 FIONBIO means Non-Blocking I/O. 1 turns on the behavior and the argument.

skt<<Listen()

Function
Tells the server socket to listen for connections.

Returns
A list of two strings. The first echoes the command ("listen") and the second is "ok" or an error message.

skt<<recv(n, <callback, timeout>)
skt<<recvfrom(n, <callback, timeout>)

Function
Receives either a stream message (recv) or a datagram message (recvfrom) from the other socket. If the two optional arguments are used, the data is not received immediately. Instead, the data is received when the function callback is called.

Returns
A list of three strings. The first string is the command name ("recv" or "recvfrom"). The second is "ok" if successful or an error message if not. The third string is the data that was received. If a callback function is used, a fourth element is the socket that was used in the original recv or recvfrom message.

Arguments
n specifies the number of bytes to receive from the other socket.
<callback> an optional argument that specifies the name of a function to receive the data.
timeout if you use a callback, timeout specifies how long the function should wait for an answer.
Appendix B

Messages

Other Objects

skt<<Send(stream)

skt<<SendTo(dgram)

Function
Sends the data in the argument to the other socket. Send sends a stream and sendto sends a datagram.

Returns
A list of three strings. The first string is the command name ("send" or "sendto"). The second is "ok" if successful or an error message if not. The third string is any portion of the stream that could not be sent, or empty if all the data was sent correctly.

Arguments

stream the command to send to the other socket.

dgram the command to send to the other socket.

Note
Either argument might need to contain binary data. JMP represents non-printable ASCII characters with a tilde (~) followed by the hexadecimal number. For example,

```
skt<<send("GET / HTTP/1.0~0d~0a~0d~0a");
```
sends a "get request" to an HTTP server.

Zip Archives

list = za<<dir

Returns a list of member names

data = za<<read(membername, <format(blob)>)

Returns a string that contains the entire member data. If the optional argument is specified, the message returns a blob.

actualname = za<<write(membername, textstring|blobdata)

Adds the text or blob data to the zip archive. Actualname and membername are the same. To avoid writing over an existing membername, you can vary the actualname by adding or changing a numeric suffix to it. For example,

```
name1 = za<<write(name, blobdata)
```
Journals

\texttt{jnl<<Save HTML("path")}

Saves the journal as HTML. "Path" is a quoted string that contains the filepath for the saved HTML file. For example, "c:/myFile.html".

\texttt{jnl<<Save RTF("path")}

Saves the journal as an RTF file. "Path" is a quoted string that contains the filepath for the saved RTF file. For example, "c:/myFile.rtf".

\texttt{jnl<<Save PDF("path")}

Saves the journal as a PDF file. "Path" is a quoted string that contains the filepath for the saved PDF file. For example, "c:/myFile.pdf".
Compatibility Notes
Changes in Scripting from JMP 8 to JMP 9

Missing Value Preference

The Missing Value Rule preference setting has been removed in JMP9. If you had the preference set to Closest or Random in a script, you now see an error that says that the option “Missing Value Rule” for the Partition platform is not defined.

Opening SAS Data Sets with the Use Labels as Column Names Setting

In JMP 8.0.2, this setting was not honored. This problem has been fixed for JMP 9. You might see SAS files imported differently into JMP because of this change.

Pattern Strings

When strings are saved to a list, they are now saved as strings instead of converted to numbers when possible. This has been changed because information can be lost if strings are converted to numbers. For example, “1” and “1.0” would be converted to the same number. Also, a script that eventually encounters numbers in a match, but was tested only with characters, would probably not work correctly.

KMeans Cluster

The KMeans Cluster platform has been largely redesigned for JMP 9. Scripts using this platform in JMP 8 and earlier might not work correctly, or produce unexpected results. See the *Modeling and Multivariate Methods* book for details about the Cluster platform.

Range Check for Rows Added by JSL

JMP 8 ignored the Range Check property when rows were added through JSL. The problem has been fixed for JMP 9. If you add a row and specify invalid values for a Range Check column, the cells contain missing values instead of the invalid values.

Scatterplot 3D

Scatterplot 3D requires at least two variables. JMP 8 incorrectly ran Scatterplot 3D scripts that specified only one variable. JMP 9 produces an error in that case.

Default Confidence Intervals for Comparing Distributions

For the Life Distribution platform, the default confidence interval for the Compare Distributions plot is now Nair’s Simultaneous confidence interval and produces different results.

If you want to see the confidence intervals used in JMP 8 (pointwise confidence intervals), use the `Interval Type(Pointwise)` option.
Compatibility Notes

Opening Projects on Macintosh

Opening projects on Macintosh now honors the **When the project is opened** project setting and the **Restore this window when project is opened** item property setting.

Fit Model Contrast Test Report

The Contrast Test Report for Fit Model (Standard Least Squares) has been reorganized. Scripts written for JMP 8 and earlier might not work.

Fit Model Interaction Effects

If a main effect is transformed, any interactions involving that effect are also transformed. A script produces different results in JMP 8 and JMP 9.

Initial Log Appearance

The **Initial Log Window** preference was set internally in JMP 8 and earlier, depending on how the JMP Log was configured (floating or docked). That preference is deprecated for JMP 9 in favor of the **Log Open Strategy** (Windows only). This preference is available in either the Preferences window on the Windows Specific page, or by using the Preferences JSL function.

Display Box Numbering

An EvalContextBox has been added at the root of all display trees, in JMP 9. Scripts written in JMP 8 and earlier that use display box subscripting into the display tree might result in errors in JMP 9 because the hierarchy has changed.

Closing Data Tables

**Close All(Data Tables, No Save)** now closes all report windows that are attached to the data tables. JMP 8 incorrectly left the reports open.

Retrieving Formulas

**Get Property( "Formula" )** now returns **Empty()** for a column that does not have a formula, instead of producing the **Scriptable[]** message in the log. This change makes it easier to loop through data tables and find columns that have formulas. However, scripts written in JMP 8 and earlier might not work correctly. Possible workarounds are as follows:

- Use **Get Formula()** instead of **Get Property("Formula")**.
- Detect the bad return value. For example, **Type( Name Expr( f ) ) != "Expression"**.

New Window and Dialog

**Dialog()** is deprecated and might not work in future versions of JMP. **New Window()** now takes an argument that creates a modal window:

```javascript
New Window("Title", <<Modal, ...);
```

We recommend that you change your scripts that use **Dialog()** to use **New Window()** instead.
Associative Arrays

Associative Array<First> now returns Empty() instead of NULL or Missing if there are no elements in the associative array.

Variables that Contain Empty()

The following code returned false in JMP 8 and earlier because the evaluation or assignment of Empty() evaluated to missing.

```julia
x = empty();
result = isEmpty( x );
```

In JMP 9, the evaluation returns missing for calculations and empty for the assignments.

Scoping Global Variables

The addition of more advanced scoping and namespaces to JMP 9 means that not explicitly scoping a global variable (::var) might cause scripts written in JMP 8 and earlier to not work in JMP 9.

Graph Builder

In JMP 8, the Graph Builder platform did not resolve column names as other platforms do. Instead, the column name variable had to be presented as a character string for Graph Builder to recognize the column name. For example:

```julia
xx = "height";
yy = "weight";
wrp = "age";
ovly = "sex";
Graph Builder(
    Variables( X( xx ), Y( yy ), Wrap( wrp ), Overlay( ovly ) ),
    Elements( Points( X, Y, Legend( 4 ) ) )
);
```

In JMP 9, the Graph Builder platform does now resolve column names as other platforms do. This means that scripts written for JMP 8 will not work in JMP 9. You must either use Eval() around the strings, or use Column() instead. For example, any of the following three examples work for JMP 9:

```julia
// Example 1: Eval()
xx = "height";
yy = "weight";
wrp = "age";
ovly = "sex";
Graph Builder(
    Variables( X( Eval(xx) ), Y( Eval(yy) ), Wrap( Eval(wrp) ), Overlay( Eval(ovly) ) ),
    Elements( Points( X, Y, Legend( 4 ) ) )
);

// Example 2: Column()
xx = "height";
yy = "weight";
```
wrp = "age";
ovly = "sex";
Graph Builder(
  Variables( X( Column(xx) ), Y( Column(yy) ), Wrap( Column(wrp) ),
  Overlay( Column(ovly) ) ),
  Elements( Points( X, Y, Legend( 4 ) ) )
);

// Example 3: Column()
xx = Column( "height" );
yy = Column( "weight" );
wrp = Column( "age" );
ovly = Column( "sex" );
Graph Builder(
  Variables( X( xx ), Y( yy ), Wrap( wrp ), Overlay( ovly ) ),
  Elements( Points( X, Y, Legend( 4 ) ) )
);

Set Format and Get Format with the Use Thousands Separator Option

Because of the addition of the "Use Thousands Separator" option, the Get Format and Set Format messages for display boxes have different list lengths. For example:

Open( "$SAMPLE_DATA/big class.jmp" );
p = Bivariate( Y( :weight ), X( :height ), Fit Line( {Line Color( "Red" )} ) );
r = Report( p );
r["Linear Fit"]["Analysis of Variance"][Number Col Box( "Sum of Squares" )]
  << Set Format( 11, 4, "Fixed Dec", "Use thousands separator" );
r["Linear Fit"]["Analysis of Variance"][Number Col Box( "Sum of Squares" )]
  << Get Format();
  {10, 4, "Fixed Dec", "Use thousands separator"}
Appendix D

Glossary
Terms, Concepts, and Placeholders

| In syntax summaries, | means “or” and separates possible choices. Usually choices separated by | are mutually exclusive. In other words, you have to pick one and cannot list several.

**argument** An argument is something specified inside the parentheses of a JSL operator, function, message, and so forth. For example, the 10 in Log(10) or the color name "red" in pen color("red"). See also named argument.

**Boolean** A Boolean is a yes/no value, something that is on or off, shown or hidden, true or false, 1 or 0, yes or no. An operator listed as being a Boolean operator is one that evaluates to true or false (or missing).

**col** In syntax summaries, a placeholder for any reference to a data table column. For example, Column("age").

**command** A generic description for a JSL statement that performs an action. This book prefers the more specific terms operator, function, and message when they are applicable.

**current data table** The current data table is the data table that Current Data Table() either returns or is assigned.

**current row** The current row for scripting is defined to be zero (no row) by default. You can set a current row with Row() or For Each Row, and so forth.

**database** Although the term is much more general, for JMP’s purposes, the word “database” describes any external data source (such as SQL) accessed through ODBC with JSL’s Open Database command.

**Datafeed** A Datafeed is a method to read real-time data continuously, such as from a laboratory measurement device connected to a serial port.

**db** In syntax summaries, a placeholder for any reference to a display box. For example, report(Bivariate[1]).

**dt** In syntax summaries, a placeholder for any reference to a data table. For example, Current Data Table() or Data Table("Big Class.jmp").

**eliding operator** An eliding operator is one that causes arguments on either side to combine and evaluate differently than if the statement were evaluated strictly left to right. For example, 12<a<13 is a range check to test whether a is between 12 and 13; JMP reads the whole expression before evaluating. If < did not elide, the expression would be evaluated left to right as (12<a)<13. In other words, it would check whether the result of the comparison (1 or 0, for false or true) is below 13, which of course would always yield 1 for true. The << operator (for object<<message, which is equivalent to Send(object, message)) is another example of an eliding operator.

**function** A function takes an argument or series of arguments inside parentheses after the function name. For example, the infix operator + has a function equivalent Add(). The statements 3 + 4 and Add(3, 4) are equivalent. All JSL’s operators have function equivalents, but not all functions have
operator equivalents. For example, $\sqrt{a}$ can be represented only by the function. Also see the Function operator for storing a function under a name.

**global variable** A global variable is a name to hold values that exists for the remainder of a session.Globals can contain many types of values, including numbers, strings, lists, or references to objects. They are called globals because they can be referred to almost anywhere, not just in some specific context.

**infix operator** In infix operator takes one argument on each side, such as $+$ in arithmetic, $3 + 4$, or the $=$ in an assignment, $a=7$.

**L-value** Something that can be the destination of an assignment. In this manual, L-value describes an expression that normally returns its current value but that can alternatively receive an assignment to set its value. For example, you would ordinarily use a function such as Row() to get the current row number and assign it to something else. For example, $x=$Row(). However, since Row is an L-value, you can also place it on the left side of an assignment to set its value. For example, Row()=10.

**list** A list is a multiple-item data type entered in special brace { } notation or with the List operator. Lists enable scripts to work with many things at once, often in the place of a single thing.

**matrix** A matrix is a JMP data type for a rectangular array of rows and columns of number. In JSL, matrices are entered in bracket [ ] notation or with the Matrix operator.

**message** A message is a JSL statement that is directed to an object, which knows how to execute the message.

**metadata** In JMP data tables, metadata are data about the data, such as the source of the data, comments about each variable, scripts for working with the data, and so on.

**mousedown** An event generated by pressing down the mouse button. See “Handle,” p. 266 and “MouseTrap,” p. 269.

**mouseup** An event generated by releasing the mouse button. See “Handle,” p. 266 and “MouseTrap,” p. 269.

**name** A name is a reference to a JSL object. For example, when you assign the numeric value 3 to a global variable in the statement $a=3$, “a” is a name.

**namespace** A namespace is a collection of unique names and corresponding values. Namespaces are useful for avoiding name collisions between different scripts.

**named argument** A named argument is an argument with a given context-sensitive effect chosen from a finite set of predefined choices for a particular JSL operator, function, message, and so forth. For example, in a Graph Box command, Frame Size is one of numerous possible named arguments that is defined to have a specific meaning that is unique to Graph Box. Frame Size happens to expect (non-named) arguments of its own, which are pixel dimensions. JSL could potentially have named arguments with the same names to many different functions, and each function would define the behavior of the named argument for its own purposes.

**ODBC database** The Microsoft standard for Open DataBase Connectivity. JSL supports access to any ODBC-enabled data source through the Open Database command.

**obj** In syntax summaries, a placeholder for any reference to an analysis platform. For example, Bivariate[1].
**object**  
An object is a dynamic entity in JMP, such as a data table, a data column, a platform results window, a graph, and so forth. Most objects can receive **messages** telling them to act on themselves in some way.

**operator**  
Usually operator refers to a one- or two-character symbol such as + for addition or <= for less than or equal to.

**POSIX**  
POSIX is an acronym for Portable Operating System Interface and is a registered trademark of the IEEE. POSIX pathnames enable you to use one syntax for paths for any operating system, instead of having to use a different syntax for each.

**postfix operator**  
A postfix operator takes on argument on its left side (before the operator), such as a++ for postincrement or a-- for postdecrement.

**pre-evaluated statistics**  
Statistics that are calculated once and used as constants thereafter.

**prefix operator**  
A prefix operator takes one argument on its right side (after the operator), such as !a for negation.

**reference**  
A way to address a scriptable **object** in order to send it **messages**. For example,. column("age") or Current Data Table() or Bivariate[1]. Typically a reference is stored in a **global variable** for convenience.

**row state**  
A data element type to store any combination of the following attributes for data rows: excluded, hidden, labeled, selected, color, marker, hue, shade.

**scalar**  
A simple non-matrix numeric value.

**scoping operator**  
A scoping operator forces a name to be interpreted as a particular type of data element, for example the : operator in :name forces name to be resolved as a column; the :: operator in ::name forces name to be resolved as a global variable.

**toggle**  
If you omit the argument (1 or 0) for a **Boolean** command, the command toggles the setting. That is, it flips the setting from on to off, or from off to on. Sending such a command repeatedly flips back and forth between on and off. If you include the Boolean argument, the command sets an absolute on or off state, and sending the command repeatedly has no further effect.

**vector**  
A matrix with only one column or row.
Symbols

" 5, 88
( ) 5, 27
* 318
+ 424
, 5
../ 87
/ 318
// 32
//! 6, 32
: 38–40, 99–100, 270
:* 318
:/ 318
:: 38–41, 270, 317
; 33, 42
<= 468
<<, 91, 160, 162, 166, 182, 184
<= < 468
= 33
== 149
 and Is Missing 83
>> 424
>7 424
[ ] 29, 72, 181, 209, 235, 250, 315–317, 321
[... ] 29
[Action Choice] 163
[Action] 94, 163
[Boolean] 163
[Column Names] 163
[Enums] 163
[New Entity] 163
[Scripting Only] 94
[Subtable] 94, 163
"] 29
\\ 29
\10 29
\1b 29
\1f 29
\IN 29, 55
\IN 29
\r 29
\t 29
— 421
\ 320
{ } 30
glossary 709
|/ 319, 337
|= 319
|| 319, 337, 425
||= 319

A

Action 164
Add From Row States 140
Add Multiple Columns 98
Add Rows 110
Add To Row States 140
Add Watch (debugger) 18
All 81, 323
analysis platform
by group 158
scripting 155
And 82
missing values 82
AndMZ 66, 82
ANOVA 336
Any 81, 323
Append 193
Arc 254
ArcBall
definition 284
ArcBall 281, 284
Arg 409
argument 155
glossary 709
arithmetic
matrix 317
arithmetic operators 441
Arrow 251
As Column 40
As Global 40
As Row State 140, 149, 551
As Table 110, 335
Assign 33
assignment operators 443
associative array 30
adding keys and values 74
assign values 72
constructors 73
create 72
data type 72
default value 73
deleting keys and values 74
graph theory 76
N Items 73
traversing 76
associative arrays
messages 641
sets 76
attenuating light source 301
audit trail 6
Auto-complete parentheses and braces, in script editor 16
autoexec script 107
automatic recall 166
auto-submit 6, 32
axis
property parameters 131
Axis Box 229
axis boxes
messages 670

B
Back Color 261
Background Color 261
back-quote 320
backslash 29
Beep 56
begin 285
Begin Data Update 109
Bézier curves 306
Big Class.jmp 57, 165
Bivariate 163, 179, 181, 185, 247
blank 29
BlendFunc 304
border boxes
messages 674
Boolean 157–158
glossary 709
Border Box 203
braces
script editor, auto-complete 16
bracket see [ ], matrix, Subscript
breakpoint 19
Bring Window to Front 167
Bullet Point 195
Button 226
Button Box 178, 195, 199, 229, 266, 272
By 158–159, 168

C
Call List 281, 294
CallList 284
capability analysis 210
Caption 55–57
carriage return 29, 89
Char 77, 122, 411–412
character functions 447
character values
missing 83
Check Box 198, 227
child object 162, 166
Cholesky 315, 330
Choose 65–66
Circle 255
clear 283
Clear Column Selection 125
Clear Globals 44–45
Clear Select 115
Close 92–93, 182
Close Window 167
coding (DOE)
property parameters 131
col 123, 164
col
glossary 709
Col List 226
Col List Box 202
Get Items 202
Col Maximum 150
Col Mean 150
Col Minimum 150
Col N Missing 150
Col Number 150
Col Quantile 150
Col Standardize 150
Col Std Dev 150
Col Sum 150
colors 38–41, 100, 270, 317
color 276–277, 294
color By Column 113
color gradient
  property parameters 131
Color Of 137, 139–140, 143–144, 148, 352
Color State 139–140, 144, 147–148, 352
Colors 261
Colors 113
colors 261–263
Column 33, 40, 123
column 164
Column Dialog 218, 222
Column Name 122
column names, special characters 124
column properties
  delete (JSL) 129
  get (JSL) 129
  set (JSL) 129
column property 129
column references 100
column vector 315
Columns 223
columns
  grouping 124
  messages 653
  ungrouping 124
columnwise functions 151
Combine States 138–140, 143–144, 253, 552
Combo Box 198, 227
comma 5, 26, 61
command 155
glossary 709
  vs. message 93
comments 32
comparison operator
  matrix 323
comparison operators 467
  examples 81
  missing values 82
Concat 77, 319, 337
Concat Items 77
Concatenate 119
conditional function 64
confusion alert
  | (pipe symbol) 23
Col Mean versus Mean 150
column names 124
current row versus selected row 41, 138
data values for datafeed 344
display boxes 175
evaluating formulas 128
Format message versus Format function 127
If versus IFMZ 82
  introduction 1
missing values for logical operators 66, 82
names default to here 391
OpenGL, transposed matrix 297
path variables 31
platforms versus reports 163, 181
pre-evaluated statistics versus Summarize arguments 150
row state operators, number-to-state versus state-to-number 140
semicolons and commas in JSL versus C 61
Set and Get messages 125
subscripting column references 100
subscripting to a row in a column 100
Week of Year 47
connections
  SAS metadata server objects 683
construct display tree 190
context 36, 127–128
Contour 259
Contour Function 245
control limits
  property parameters 132
Convert File Path 31
Copy From Row States 140
Copy Picture 664
Copy To Row States 140
creating JMP windows 175
currency 53
current context data table 40
Current Data Table 33, 40, 91, 99, 107
glossary 709
current row number 37, 41, 99, 101
glossary 709
custom graph 241
custom marker 265
custom platform 212
custom property
  property parameters 133
custom window 190
Customize 239
CV 150
Cylinder 294
D

Data Filter
  data filter commands 115
  syntax reference 644
data table
  calculation scripting 150
  importing 87
  matrix 334–335
  message 94
  opening 87
  row state scripting 133
data table column scripting 121
data table scripting 85
Data Table Window 157, 167
data tables
  invisible 90
  messages 642
database
  glossary 709
  open 354
database access 354
  messages 660
datafeed
  control chart example 346
datafeed object 343
data-reading example 346
glossary 709
  messages 344, 660
  real-time data capture 343
date and time
  notation 28
date/time
  formats 48, 127
  operators 477
  Y2K 47
datafeed
  messages 348
Day 46
Day Of Week 46
Day Of Year 46
db
  display box reference 180
  glossary 709
debug 44, 272
debug step 17
decomposition 329
default value, associative array 73
Delete (Display box) 194
Delete Column Property 107
Delete Columns 98
Delete Formula 107
Delete Property 107, 129
Delete Rows 111
Delete Table Property 107
Delete Table Variable 107
delimiter 89
Derivative 434, 436
derivatives 434, 436
Deselect 182
Design 327
design matrices 327
Design Nom 328
  design role (DOE)
  property parameters 132
DesignNom 328, 337
DesignOrd 328
Det 327
Diag 319
Dialog 222, 224–225
Dialog 241
dialog boxes
  constructing 218
data column 222
  pick dialogs 189
  using JMP built in dialogs 189
Dif 102
Direct Product 329
Disable 311
Disk 294
Dispatch 183
display box 19, 177, 180
display box tree 179
display boxes
  messages 661
display scripting 175
display trees 175
  navigating 177
distribution
  property parameters 132
Divide 318
DLLs 349
DOCUMENTS 30
DOE Bayes Diagonal 173
DOE K Exchange Value 172
DOE Mixture Sum 172
DOE Search Points Per Factor 172
DOE Sphere Radius 173
DOE Starting Design 172
DOE Starts 172
double quote 29
double underscore and variable names 421
Drag Line 270
Drag Marker 270
Drag Polygon 270
Drag Rect 270
Drag Text 270
dt data table reference 180
glossary 709
referencing a data table 91
dynamic link libraries 349

E
e 27
eager operator 406
Edge Flag 294
Edit Number 226
Edit Text 227
Editable 241
EDiv 318
Eigen 315, 329
eigenvalue decomposition 330
elementwise matrix operator 318
elapping operator 162, 467
glossary 709
EMult 318
Enable 289, 293, 300, 303, 311
encryption
  encrypting a script 420
  hiding variables 421
End 285
End Data Update 109
equality operators 467
escape sequences 28, 55
eval and Formulas 407
Eval Coord 294
Eval Expr 411–412
Eval Formula 128
Eval Insert 410
Eval List 72, 225, 411–412
Eval Point 294
EvalFormula 109
EvalMesh1 307
EvalMesh2 309
Evaluators
  one-dimensional 306
two-dimensional 309
exceptions 387–388
Exclude 125, 134
Excluded 139–140, 143, 552
Excluded State 139–140, 143, 552
Exp 27
Expr As Picture 218
expression 414, 419
Extra space at bottom of document, in script editor 16

F
factor changes (DOE)
  property parameters 132
Factorial 389
fence matching 27
file path format 30
Files In Directory 190
Fill Color 261
Fill Pattern 264
find a window 183
First 42
Fit Model 37, 338
Fitness.jmp 249
fitting personality 170
flow control 59
Fog 305
Font Color 261
fonts, setting in script editor 15
For 4, 61, 188, 209, 243–244, 334
For Each Row 41, 99–101, 137–138
Format 49, 52, 127
  currency 53
formatting scripts 15
formfeed 29
Formula 97, 151, 407
formula
  and eval 407
  evaluation 108
picture 218
Frame Box 230
frame boxes
  messages 667
freeze all 96
freeze frames 97
freeze frames with scripts 97
freeze pictures 97
Frustum 280
Function 45, 388
function glossary 709
functions local variables 389

G

Get 188
Get All Columns As Matrix 110
Get As Matrix 110, 126, 334–335
Get Column Names 122
Get Format 127
Get Formula 125, 128
Get Items
   and Col List Box 202
Get List Check 129
Get Lock 133
Get Name 125
Get Picture 217
Get Property 106, 129–130
Get Range Check 129
Get Rows Where 113, 334
Get Script 108, 133, 167
Get Selected Columns 99, 123
Get Selected Rows 113, 334
Get Table Variable 106
Get Text 201
Get Values 126
Get Window Position 168
Get Window Size 168
GInverse 325
Global Box 178, 195–196, 230, 266
global variable 37–40
   and expr 407
   and functions 389
global variables 3, 25, 27
   and expr 416
   and Global Box 230
   and in-place operators 415
   and matrices 330
   and Summarize 102
   and Summary 117
column names 99
column reference 37
glossary 710
hiding 421
interactive display elements 196
prefix operator 99
referencing a column 123
referencing a data table 91
referencing an object 160
scoping 40, 317
testing for empty value 53
Glue 33, 42
glue 42
Go To Line 3
Go To Row 112
Gradient Function 247
Graph Box 241, 243
graph theory and associative arrays 76
graphic of formula 218
graphics primitives 285
grouping columns 124

H

Handle 195, 266–270, 272
HDirect Product 329
HeadName 410
Hex to Char 431
Hidden 139–140, 143, 553
Hidden State 139–140, 143, 553
Hide 125, 134
hiding global variables 421
HierBox 179, 231
HLine 251
HList 223, 226
HList Box 177, 191
HOME 30
Host Is 54
Hue State 139–140, 147–148, 553

I

Icon Box 231
Identity 316
If 64
If versus IfMZ 66
IfMZ 66, 82
import 87, 89
In Days 47
In Format 52
In Hours 47
In Minutes 47
In Polygon 259
In Weeks 47
In Years 47
Include 390
   Parse Only 390
indent guides, showing in script editor 16
Index 316–317, 322
infix operator 30, 40, 100, 317
glossary 710
in-place operators 415–416
Insert 415, 418
Insert Into 415, 417–418
InsertInto 415
interactive graph 195, 266–272
Interpolate 66
Invalid Row Number error 100
Inverse 325–326
Inverse Matrices, updating 333
Invert Expr 436
invert matrix 324–329
Invert Row Selection 112
invisible 90, 165–166
invisible data tables 90
Invisible Reports 165
Is Empty 53
Is List 71
Is Matrix 317
Is Missing 82
and == 83
iterate 61, 100

J
J 316, 337
JMP Version() 54
JMP Window 433
jmpStart.jsl 107
Join 119
Journal Box 204
Journal Window 168, 217
Journals 217
JSL 19, 23, 87
JSL browser 19
JSL Debugger 17

L
Label 125
Labeled 139–140, 143, 553
Labeled State 139–140, 143, 553
Lag 102
leaf 179
legend, adding 249
LELE 129
LELT 129
lexical rules 26
Light 300
light
   attenuating 301
Light Model 302
Line 249, 252
line numbers, in script editor 16
Line Stipple 289
Line Style 264
Line Up 226
Line Up Box 202
Line Width 289
LINE_LOOP 286
LINE_STRIP 286
linefeed 29, 89
LINES 286
List 30, 68, 71
list 30, 419
glossary 710
List Box 198, 231
List Check 128
list check
   property parameters 130
list checking 128
Load Matrix 298
Load Text File 390
Loc 71, 322
Loc Max 323
Loc Min 323
Loc Sorted 323
Local 38, 45
local
   parameters 388
   variables 45
local variables
   in functions 389
Lock 133
Lock Globals 45
Log 36
logical operator
   matrix 323
Look At 283
LTLE 129
LTLT 129
glossary 710

M
Macintosh 54
macro 333, 388, 407, 414
Mail 55, 57
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make SAS Data Step</td>
<td>355</td>
</tr>
<tr>
<td>Make SAS Data Step Window</td>
<td>355</td>
</tr>
<tr>
<td>manipulate expressions</td>
<td>414–419</td>
</tr>
<tr>
<td>manipulate lists</td>
<td>414–416</td>
</tr>
<tr>
<td>map role</td>
<td></td>
</tr>
<tr>
<td>property parameters</td>
<td>133</td>
</tr>
<tr>
<td>Map1</td>
<td>307</td>
</tr>
<tr>
<td>Map2</td>
<td>309</td>
</tr>
<tr>
<td>Mapgrid1</td>
<td>307</td>
</tr>
<tr>
<td>Mapgrid2</td>
<td>309</td>
</tr>
<tr>
<td>Marker</td>
<td>148, 252–253</td>
</tr>
<tr>
<td>Marker by Column</td>
<td>113</td>
</tr>
<tr>
<td>Marker Of</td>
<td>139–140, 144, 553</td>
</tr>
<tr>
<td>Marker Size</td>
<td>252</td>
</tr>
<tr>
<td>preferred size</td>
<td>252</td>
</tr>
<tr>
<td>Marker State</td>
<td>139–140, 144, 553</td>
</tr>
<tr>
<td>marker, custom</td>
<td>265</td>
</tr>
<tr>
<td>Markers</td>
<td>113</td>
</tr>
<tr>
<td>Match</td>
<td>65</td>
</tr>
<tr>
<td>matching parentheses</td>
<td>27</td>
</tr>
<tr>
<td>MatchMZ</td>
<td>66, 82</td>
</tr>
<tr>
<td>Material</td>
<td>294</td>
</tr>
<tr>
<td>Matrix</td>
<td>250, 315–317</td>
</tr>
<tr>
<td>matrix</td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td>317</td>
</tr>
<tr>
<td>arithmetic operators</td>
<td>434</td>
</tr>
<tr>
<td>comparisons</td>
<td>323</td>
</tr>
<tr>
<td>concatenation</td>
<td>319</td>
</tr>
<tr>
<td>constructing</td>
<td>315</td>
</tr>
<tr>
<td>constructing with expressions</td>
<td>316</td>
</tr>
<tr>
<td>data tables</td>
<td>334–335</td>
</tr>
<tr>
<td>decomposition</td>
<td>329</td>
</tr>
<tr>
<td>definition</td>
<td>315</td>
</tr>
<tr>
<td>deleting rows and columns</td>
<td>317</td>
</tr>
<tr>
<td>diagonal</td>
<td>319</td>
</tr>
<tr>
<td>glossary</td>
<td>710</td>
</tr>
<tr>
<td>introduction</td>
<td>29</td>
</tr>
<tr>
<td>logical operators</td>
<td>323</td>
</tr>
<tr>
<td>normalization</td>
<td>329</td>
</tr>
<tr>
<td>numeric functions</td>
<td>318</td>
</tr>
<tr>
<td>range checks</td>
<td>323</td>
</tr>
<tr>
<td>ranges of rows or columns</td>
<td>322</td>
</tr>
<tr>
<td>ranking</td>
<td>324</td>
</tr>
<tr>
<td>solving linear systems</td>
<td>325</td>
</tr>
<tr>
<td>sorting</td>
<td>324</td>
</tr>
<tr>
<td>special constructors</td>
<td>316</td>
</tr>
<tr>
<td>subcripting</td>
<td>321</td>
</tr>
<tr>
<td>summarizing columns</td>
<td>324</td>
</tr>
<tr>
<td>transposing</td>
<td>320</td>
</tr>
<tr>
<td>Matrix Box</td>
<td>191–192, 231</td>
</tr>
<tr>
<td>matrix boxes</td>
<td></td>
</tr>
<tr>
<td>messages</td>
<td>91</td>
</tr>
<tr>
<td>column</td>
<td>123</td>
</tr>
<tr>
<td>data table</td>
<td>87</td>
</tr>
<tr>
<td>display</td>
<td>182, 209</td>
</tr>
<tr>
<td>glossary</td>
<td>710</td>
</tr>
<tr>
<td>live platform</td>
<td>160</td>
</tr>
<tr>
<td>objects</td>
<td>91</td>
</tr>
<tr>
<td>platform</td>
<td>155</td>
</tr>
<tr>
<td>Report</td>
<td>181</td>
</tr>
<tr>
<td>show properties</td>
<td>94</td>
</tr>
<tr>
<td>messages</td>
<td></td>
</tr>
<tr>
<td>associative arrays</td>
<td>641</td>
</tr>
<tr>
<td>axis boxes</td>
<td>670</td>
</tr>
<tr>
<td>border boxes</td>
<td>674</td>
</tr>
<tr>
<td>columns</td>
<td>653</td>
</tr>
<tr>
<td>data tables</td>
<td>642</td>
</tr>
<tr>
<td>databases</td>
<td>660</td>
</tr>
<tr>
<td>datafeed</td>
<td>660</td>
</tr>
<tr>
<td>display boxes</td>
<td>661</td>
</tr>
<tr>
<td>frame boxes</td>
<td>667</td>
</tr>
<tr>
<td>matrix boxes</td>
<td>674</td>
</tr>
<tr>
<td>nom axis boxes</td>
<td>674</td>
</tr>
<tr>
<td>number col boxes</td>
<td>672</td>
</tr>
<tr>
<td>outline boxes</td>
<td>667</td>
</tr>
<tr>
<td>panel boxes</td>
<td>674</td>
</tr>
<tr>
<td>platforms</td>
<td>676</td>
</tr>
<tr>
<td>rows</td>
<td>657</td>
</tr>
<tr>
<td>SAS results</td>
<td>697</td>
</tr>
<tr>
<td>SAS server objects</td>
<td>684</td>
</tr>
<tr>
<td>scheduling</td>
<td>699</td>
</tr>
<tr>
<td>sockets</td>
<td>700</td>
</tr>
<tr>
<td>stored processes (SAS)</td>
<td>692</td>
</tr>
<tr>
<td>string col boxes</td>
<td>673</td>
</tr>
<tr>
<td>table boxes</td>
<td>668</td>
</tr>
<tr>
<td>text boxes</td>
<td>669</td>
</tr>
<tr>
<td>metadata</td>
<td>105</td>
</tr>
<tr>
<td>glossary</td>
<td>710</td>
</tr>
<tr>
<td>metadata server objects (SAS)</td>
<td></td>
</tr>
<tr>
<td>connections</td>
<td>683</td>
</tr>
<tr>
<td>Min</td>
<td>324</td>
</tr>
<tr>
<td>Minimize</td>
<td>437</td>
</tr>
</tbody>
</table>
missing values 82
OrMZ 66, 82
Ortho 280, 332
Ortho2D 281
orthogonal polynomial 333
Orthographic projections 278
orthonormalize 332
OrthoPoly 333
Outline Box 178, 182, 191–192, 232
outline boxes
messages 667
output
suppressing 166
Oval 257

Panel Box 204
panel boxes
messages 674
Parameters (Enable command) 311
parentheses 5, 27
script editor, adding spaces inside 16
script editor, auto-complete 16
parenthesis matching 27
Parse 411–412
Parse XML 422
Partial Disk 294
password style 201
Patch Editor.jsl 309
path format 30
path variables 30
patterns
case insensitive 426
pausing script execution 68
Pen Color 261
Pen Size 265
Perspective 276, 279
Perspective projections 278
pick 310
pick dialogs 189
Pick Directory 189
Pick File 189
Picture Box 178, 232
Picture data type 217
picture of formula 218
Pie 254
Pixel Line To 265
Pixel Move To 265
Pixel Origin 265
platform
by group 158
scripting 155
platform scripting 153
syntax 157
writing interactively 155
platforms
messages 676
Plot Col Box 232
Point Size 289
POINTS 286
POLYGON 286
Polygon 258
Polygon Mode 290
Polygon Offset 293
Pop Matrix 284
Popup Box 199
easy example 199
POSIX 30
glossary 711
postfix operator 30
glossary 711
precedence 43
pre-evaluated statistics 37, 150
glossary 711
preferences 107
preferences for the script editor 15
pre-fill columns 223
prefix colon 40
prefix operator 30, 99, 317
glossary 711
Prepend 193
Preselect Role 128
Previous Selected 114
Print 4, 5, 406
Print Window 95, 168
procrastination operator 406
Product 63
programming function 59
projections
orthographic 278
perspective 278
property 106, 129
punctuation 26, 61
Push Matrix 284, 297

QC
QC Alarm Script 106
QC Test Alert 106
QR 333
QUAD_STRIP 287
Quadric Draw Style 295
Quadric Normals 295
Quadric Orientation 295
QUADS 287
quotation marks 5, 88
quoting operator 406

R
Radio Box 198
Radio Buttons 227
Random Reset 109, 128
Range Check 128
range check
  property parameters 130
  syntax 129
range checking 128
Rank 324
rearrange columns 98
Rect 256, 260
Recursion 389
Redo Analysis 156, 166
reference 91, 123, 181–182, 209
glossary 711
Reformat Script, in script editor 15
reformatting scripts 15
Regression 335
regression 335–336
regular expressions 427
Remove 414
Remove From 414–415, 417–418
Remove Watch (debugger) 18
Repeat 79
Report 165, 168, 181, 211
Reshow 182
response limits (DOE, desirability profiling)
  property parameters 132
Reverse 415, 419
Reverse Into 415, 419
reverse rotation 330
Revert 92
Rotate 277, 281, 291
rotation 330
Row 41, 99, 101, 138
Row Legend 240, 249
row order levels
  property parameters 131
Row State 137, 140, 144, 553
row state 133
glossary 711
row state combination 139
row state scripting
tutorial 149
row vector 315
Rows menu scripting 110
rowwise functions 151
Run (debugger) 18
Run Formulas 109, 128
Run Script 92
Run Script 25, 68
Run Without Debugger 18
running a script at startup 107

S
SAMPLE_DATA 30
SAMPLE_IMAGES 30
SAMPLE_IMPORT_DATA 30
SAMPLE_SCRIPTS 30
SAS metadata server 356
SAS metadata server objects
  connections 683
SAS Name 355
SAS Open For Var Names 355
SAS results
  messages 697
SAS server objects
  messages 684
SAS stored processes
  messages 692
Save 95
Save Database 355
save formula as picture 218
Save Picture 218
Save Script 165
Save Script for All Objects 156, 167
Save Script to Datatable 107, 156, 166
Save Script to Report 156, 167
Save Script to Script Window 156, 167
Save Script to Window 156
Save Text File 390
scalar 315
glossary 711
scaling 330
Scene Box 275
Schedule 68
schedule
  messages 699
scoping operator 38–39, 317
glossary 711
Script Box 233
Script Editor 12
script editor
  auto-complete parentheses and braces 16
  extra space at bottom of document 16
  indent guides, showing 16
  line numbers, showing 16
  operator tips, showing 16
  preferences 15
  set fonts 15
  spaces in operator names 16
  spaces inside parentheses 16
  tab width 16
  use tabs 16
  variable value tips, showing 16
script style 15
scriptable objects 90
scripting vs. programming 36
Scroll Lock 130
scroll lock 125
Scroll Window 167
Select 182
Select All Matching Cells 112
Select All Rows 111
Select Matching Cells 112
Select Rows 99, 112
Select Where 112
Selected 139–140, 143, 553
Selected State 139–140, 143, 554
semicolon 5, 42, 61
Send 91, 124, 160, 162
Send To Report 183
Sequence 79
set and associative arrays 76
Set Each Value 109
Set Formula 125, 128
Set Label Columns 130
Set Lock 133
Set Matrix 110
Set Name 95, 125
Set Property 106, 129–130
Set Row States 149
Set Script 496
Set Scroll Lock Columns 130
Set Selected 122
Set Table Variable 106
Set Values 125–126, 195
Set Wrap 195
Shade Model 303
Shade State 139–140, 147–148, 554
Shape 317
Shift 415, 419
Shift Into 415, 419
short-circuiting operators 83
Show 3, 55, 406
Show Arcball 283
Show Commands 20
Show Globals 44
Show indent guides, in script editor 16
Show line numbers, in script editor 16
Show operator tips, in script editor 16
Show Properties 20, 94, 123, 162–163, 166, 183, 185
Show Tree Structure 179
Show variable value tips, in script editor 16
Show Window 167
Sib Append 194
sigma
  property parameters 132
Simplify Expr 437
singular value decomposition 332
Size Window 167
slice matrix 321
Slider Box 178, 195–196, 266
socket
  commands 352
sockets
  datagram 351
  example 351
  messages 352, 700
  stream 351
  using 351
Solve 325
solve linear system 324–329
Sort 107, 117
Sort List 415, 419
Sort List Into 415, 419
sort matrix 324
Spaces in operator names, in script editor 16
Spaces inside parentheses, in script editor 16
Speak 55–56
spec limits
  property parameters 132
Sphere 295
Spline Coef, example 169
Spline Eval, example 169
Spline Smooth, example 169
Split 118
Stack 118
startup script 107
StatusMsg 55, 57
Std Dev 150
Step 67
Step (debugger) 18
Stop Script 68
stored expression 419
stored processes (SAS)
messages 692
String Col Box 233
string col boxes
messages 673
strings 28
Subscript 181, 209, 321
subscripting column names 100
subscripts 70
  with column references 100
subscripts with columns 123
Subset 116–117
Substitute 416–417, 419
Substitute Into 212, 416–417, 419
Sum 150
Summarize 61, 100, 102
Summary 102, 116
Summation 61, 63
Suppress Formula Eval 108
Suppressing output 166
SVD 315, 332
Sweep 326–327, 337

T
tab 29
Tab Box 200
  Example 200
Tab width, in script editor 16
Table Box 178, 234
table variable 106
tables boxes
  messages 668
Tables menu scripting 116
tabs
  script editor 16
TEMP 30
Text 260, 277, 296
Text Box 234
Text Box 201
text boxes
  messages 669
text command 249
Text Edit Box 201, 234
text edit box
  password style 201
Text file type 95
Text Wrap, controlling 195
TextEditBox 179
Throw 387–388
time frequency
  property parameters 133
Time Of Day 46
Title 169
title 165–166
toggle 114, 161
glossary 711	okens 26
Trace 320, 331
Translate 277, 281, 291
Transpose 118, 320
transpose 330
TRIANGLE_FAN 287
TRIANGLE_STRIP 287
TRIANGLES 286
trigonometric functions 575
troubleshoot 44, 272
  infinite loop 111
truth tables 82
Try 387–388
tutorial
  custom report 209
data table 100, 102
display 196
display tree 192
matrix 335–336
platform 185
programming 61
QC chart 148
row state 149
Type 53

U
underscores and variables names 421
ungrouping columns 124
uninitialized variable 53
units of measure
  property parameters 132
unload list 225
Unlock Globals 45
unquoting operator 406
Update 120, 276
Use tabs, in script editor 16
Use Value Labels 126

V
value colors
  property parameters 131
Value Labels 126
value labels
  property parameters 131
value ordering
  property parameters 131
Values 97
variable value tooltips, showing in script editor 16
variables
  global, hiding 421
VConcat 319, 337
VecDiag 320
VecQuadratic 320
vector 315
glossary 711
vectors
  normal 302
Vertex 294
VLine 251
VList 223, 226
VList Box 177, 272

W-Z
Wait 56, 68
Watch 43
Web 432
Week Of Year 46
While 63, 243
whitespace characters 27
 wildcard 182
window
  custom 190
window scripting 175
Windows 54
Wrap point, controlling 195
Write 55, 406
X Function 244
XML
  Parse XML 422
  parsing example 422
  xmlAttr 422
  xmlText 422
  xmlAttr 422
  xmlText 422
Y Function 243
Y2K date interpretation 47
Year 46
year 2000 date interpretation 47
Zero Or Missing 82
Zoom Window 167, 183