SAS® 9.4 Component Objects: Reference, Third Edition

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® 9.4 Component Objects: Reference, Third Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.4 Component Objects: Reference, Third Edition

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

November 2016

9.4-P1:lecompobjref

Contents

About This Book

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

- syntax components
- style conventions
- special characters
- references to SAS libraries and external files

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=). The syntax for arguments has multiple forms in order to demonstrate the syntax of multiple arguments, with and without punctuation.

keyword
specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In these examples of SAS syntax, the keywords are bold:

CHAR (*string, position*)

CALL RANBIN (*seed, n, p, x*);

ALTER (*alter-password*)

BEST *w*.

REMOVE <*data-set-name*>

In this example, the first two words of the CALL routine are the keywords:

CALL RANBIN(*seed*, *n*, *p*, *x*)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;

... SAS code ...

END;

Some system options require that one of two keyword values be specified:

DUPLEX | **NODUPLEX**

Some procedure statements have multiple keywords throughout the statement syntax:

CREATE <UNIQUE> **INDEX** *index-name* **ON** *table-name* (*column-1* <, *column-2*, …>)

argument
specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed in angle brackets (< >).

In this example, *string* and *position* follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (*string, position*)

Each argument has a value. In this example of SAS code, the argument *string* has a value of 'summer', and the argument *position* has a value of 4:

```
x=char('summer', 4);
```

In this example, *string* and *substring* are required arguments, whereas *modifiers* and *startpos* are optional.

FIND(*string, substring* <, *modifiers*> <, *startpos*>

argument(s)
specifies that one argument is required and that multiple arguments are allowed. Separate arguments with a space. Punctuation, such as a comma (,) is not required between arguments.

The MISSING statement is an example of this form of multiple arguments:

MISSING *character(s)*;

<LITERAL_ARGUMENT> *argument-1* <<LITERAL_ARGUMENT> *argument-2 ...* >
specifies that one argument is required and that a literal argument can be associated with the argument. You can specify multiple literals and argument pairs. No punctuation is required between the literal and argument pairs. The ellipsis (...) indicates that additional literals and arguments are allowed.

The BY statement is an example of this argument:

BY <DESCENDING> *variable-1* <<DESCENDING> *variable-2 …*>;

argument-1 <option(s)> <argument-2 <option(s)> ...>
specifies that one argument is required and that one or more options can be associated with the argument. You can specify multiple arguments and associated options. No punctuation is required between the argument and the option. The ellipsis (...) indicates that additional arguments with an associated option are allowed.

The FORMAT procedure PICTURE statement is an example of this form of multiple arguments:

PICTURE name <(*format-option(s)*)>
<*value-range-set-1* <(*picture-1-option(s)*)>
<*value-range-set-2* <(*picture-2-option(s)*)> *…*>>;

argument-1=value-1 <argument-2=value-2 ...>
specifies that the argument must be assigned a value and that you can specify multiple arguments. The ellipsis (...) indicates that additional arguments are allowed. No punctuation is required between arguments.

The LABEL statement is an example of this form of multiple arguments:

LABEL *variable-1=label-1 <variable-2=label-2 …>*;

argument-1 <, argument-2, ...>
specifies that one argument is required and that you can specify multiple arguments that are separated by a comma or other punctuation. The ellipsis (...) indicates a continuation of the arguments, separated by a comma. Both forms are used in the SAS documentation.

Here are examples of this form of multiple arguments:

AUTHPROVIDERDOMAIN (*provider-1:domain-1 <, provider-2:domain-2, …>*

INTO :*macro-variable-specification-1 <*, :*macro-variable-specification-2*, *…>*

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

Style Conventions

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

UPPERCASE BOLD
identifies SAS keywords such as the names of functions or statements. In this example, the keyword ERROR is written in uppercase bold:

ERROR <*message*>;

UPPERCASE
identifies arguments that are literals.

In this example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

CMPMODEL=BOTH | CATALOG | XML |

italic
identifies arguments or values that you supply. Items in italic represent user-supplied values that are either one of the following:

- nonliteral arguments. In this example of the LINK statement, the argument *label* is a user-supplied value and therefore appears in italic:

  LINK *label*;

- nonliteral values that are assigned to an argument.

  In this example of the FORMAT statement, the argument DEFAULT is assigned the variable *default-format*:

  FORMAT *variable(s) <format* > <DEFAULT = *default-format*>;

Special Characters

The syntax of SAS language elements can contain the following special characters:

=
an equal sign identifies a value for a literal in some language elements such as system options.

In this example of the MAPS system option, the equal sign sets the value of MAPS:

MAPS=*location-of-maps*

< >
angle brackets identify optional arguments. A required argument is not enclosed in angle brackets.

In this example of the CAT function, at least one item is required:

CAT (*item-1* <, *item-2*, …>)

|
a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.

In this example of the CMPMODEL= system option, you can choose only one of the arguments:

CMPMODEL=BOTH | CATALOG | XML

...
an ellipsis indicates that the argument can be repeated. If an argument and the ellipsis are enclosed in angle brackets, then the argument is optional. The repeated argument must contain punctuation if it appears before or after the argument.

In this example of the CAT function, multiple *item* arguments are allowed, and they must be separated by a comma:

CAT (*item-1 <, item-2*, …>)

'*value*' or "*value*"
indicates that an argument that is enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.

In this example of the FOOTNOTE statement, the argument *text* is enclosed in quotation marks:

FOOTNOTE <*n*> <*ods-format-options* '*text*' | "*text*">;

;
a semicolon indicates the end of a statement or CALL routine.

In this example, each statement ends with a semicolon:

```
data namegame;
   length color name $8;
   color = 'black';
   name = 'jack';
   game = trim(color) || name;
run;
```

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks. If you use a logical name, you typically have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the reference.

Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase *file-specification*. In the examples that use SAS libraries, SAS documentation uses the italicized phrase *SAS-library* enclosed in quotation marks:

```
infile file-specification obs = 100;
libname libref 'SAS-library';
```

What's New in SAS 9.4 Component Objects

Tracking Key Summaries for Hash Objects

Use the *keysum* argument tag in the DECLARE statement or _NEW_ operator to specify the name of a variable that tracks the key summary for all keys.

Iterating over Multiple Keys for Hash Objects

Use the DO_OVER method in an iterative DO loop to traverse through the duplicate keys. The DO_OVER method reads the key on the first method call and continues to iterate over the duplicate key list until it reaches the end. If you need to switch the key in the middle of an iteration, you can use the new RESET_DUP method to reset the pointer to the beginning of the list.

Lock-Down State Restrictions

The LOCKDOWN statement and LOCKDOWN system option are new in the first maintenance release for SAS 9.4. With LOCKDOWN, if you are running in a client/server environment (for example, you use SAS Enterprise Guide), the SAS server administrator can create an environment where your SAS client has access to a set of directories and files. All other directories and files would be inaccessible. In addition to there being restrictions on directories and files, several language elements are not available when SAS is in a locked-down state and the DATA step Java object is not available.For more information, see "SAS Processing Restrictions for Servers in a Locked-Down State" in *SAS Language Reference: Concepts*.

Chapter 1

About SAS Component Objects

DATA Step Component Objects

SAS provides these five predefined component objects for use in a DATA step:

hash and hash iterator objects
: enable you to quickly and efficiently store, search, and retrieve data based on lookup keys. For more information, see "Using the Hash Object " in *SAS Language Reference: Concepts* and "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*.

Java object
: provides a mechanism that is similar to the Java Native Interface (JNI) for instantiating Java classes and accessing fields and methods on the resultant objects. For more information about the Java object, see "Using the Java Object" in *SAS Language Reference: Concepts* .

logger and appender objects
: enable you to record logging events and write these events to the appropriate destination. For more information, see "Component Object Reference" in *SAS Logging: Configuration and Programming Reference*.

The DATA Step Component Interface

The DATA step component object interface enables you to create and manipulate predefined component objects in a DATA step.

To declare and create a component object, you use either the DECLARE statement by itself or the DECLARE statement and _NEW_ operator together.

Component objects are data elements that consist of attributes, methods, and operators. **Attributes** are the properties that specify the information that is associated with an object. **Methods** define the operations that an object can perform. For component objects, **operators** provide special functionality.

You use the DATA step object dot notation to access the component object's attributes and methods.

Note: The DATA step component object's statements, attributes, methods, and operators are limited to those that are defined for these objects. You cannot use the SAS Component Language functionality with these predefined DATA step objects.

Dot Notation and DATA Step Component Objects

Definition

Dot notation provides a shortcut for invoking methods and for setting and querying attribute values. Using dot notation makes your SAS programs easier to read.

To use dot notation with a DATA step component object, you must declare and instantiate the component object by using either the DECLARE statement by itself or the DECLARE statement and the _NEW_ operator together. For more information, see "Using DATA Step Component Objects" in *SAS Language Reference: Concepts* and "Component Object Reference" in *SAS Logging: Configuration and Programming Reference*.

Syntax

The syntax for dot notation is as follows:

object.attribute

or

object.method (<*argument_tag-1*: *value-1* <,… *argument_tag-n*: *value-n*>>;

The arguments are defined as follows:

object
specifies the variable name for the DATA step component object.

attribute
specifies an object attribute to assign or query.

When you set an attribute for an object, the code takes this form:

```
object.attribute = value;
```

When you query an object attribute, the code takes this form:

```
value = object.attribute;
```

method
specifies the name of the method to invoke.

argument_tag
identifies the arguments that are passed to the method. Enclose the argument tag in parentheses. The parentheses are required whether the method contains argument tags.

All DATA step component object methods take this form:

```
return_code=object.method(<argument_tag-1:value-1
   <, ...argument_tag-n:value-n>>);
```

The return code indicates method success or failure. A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message is printed to the log.

value
specifies the argument value.

Tips When Using Component Objects

- You can assign objects in the same manner as you assign DATA step variables. However, the object types must match. The first set of code is valid, but the second generates an error.

```
declare hash h();
declare hash t();
t=h;
```

```
declare hash t();
declare javaobj j();
j=t;
```

- You cannot declare arrays of objects. The following code would generate an error:

```
declare hash h1();
declare hash h2();
array h h1-h2;
```

- You can store a component object in a hash object as data but not as keys.

```
data _null_;
   declare hash h1();
   declare hash h2();

   length key1 key2 $20;

   h1.defineKey('key1');
   h1.defineData('key1', 'h2');
   h1.defineDone();

   key1 = 'abc';
   h2 = _new_ hash();
   h2.defineKey('key2');
   h2.defineDone();

   key2 = 'xyz';
   h2.add();
   h1.add();

   key1 = 'def';
   h2 = _new_ hash();
   h2.defineKey('key2');
```

```
   h2.defineDone();

   key1 = 'abc';
   rc = h1.find();
   h2.output(dataset: 'work.h2');
run;

proc print data=work.h2;
run;
```

The data set WORK.H2 is displayed.

Figure 1.1 *Data Set WORK.H2*

The SAS System

Obs	key2
1	xyz

- You cannot use component objects with comparison operators other than the equal sign (=). If H1 and H2 are hash objects, the following code will generate an error:

```
if h1>h2 then
```

- After you declare and instantiate a component object, you cannot assign a scalar value to it. If J is a Java object, the following code will generate an error:

```
j=5;
```

- You have to be careful to not delete object references that might still be in use or that have already been deleted by reference. In the following code, the second DELETE statement will generate an error because the original H1 object has already been deleted through the reference to H2. The original H2 can no longer be referenced directly.

```
declare hash h1();
declare hash h2();
declare hash t();
t=h2;
h2=h1;
h2.delete();
t.delete();
```

- You cannot use component objects in argument tag syntax. In the following example, using the H2 hash object in the ADD methods will generate an error.

```
declare hash h2();
declare hash h();
h.add(key: h2);
h.add(key: 99, data: h2);
```

- The use of a percent character (%) in the first byte of text output by Java to the SAS log is reserved by SAS. If you need to output a % in the first byte of a Java text line, it must be escaped with another percent immediately next to it (%%).
- You can have a hash table of hash tables.
- A Java object represents an instantiation of a single Java class. A Java object cannot hold anything else. But the Java instance can be arbitrarily complicated just like any

Java instance. A Java object can contain references to other Java entities, but they are not considered Java objects.

- When SAS is in a locked-down state, the Java object is not available. For more information, see "SAS Processing Restrictions for Servers in a Locked-Down State" in *SAS Language Reference: Concepts*.

Chapter 2

Dictionary of Hash and Hash Iterator Object Language Elements

Dictionary

ADD Method

Adds the specified data that is associated with the given key to the hash object.

Applies to: Hash object

Syntax

rc=*object*.**ADD** (<<KEY: *keyvalue-1*>, …<KEY: *keyvalue-n*>, <DATA: *datavalue-1* >, … <DATA: *datavalue-n*>>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: *keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

DATA: *datavalue*
specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call.

The number of "DATA: *datavalue*" pairs depends on the number of data variables that you define by using the DEFINEDATA method.

Details

You can use the ADD method in one of two ways to store data in a hash object.

You can define the key and data item, and then use the ADD method as shown in this code:

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
```

```
   end;
   /* Define constant key and data values */
   k = 'Joyce';
   d = 'Ulysses';
   /* Add key and data values to hash object */
   rc = h.add();
run;
```

Alternatively, you can use a shortcut and specify the key and data item directly in the ADD method call as shown in this code:

```
data _null_;
   length k $8;
   length d $12;
   /* Define hash object and key and data variable names */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Define constant key and data values and add to hash object */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
run;
```

If you add a key that is already in the hash object, the ADD method returns a nonzero value. Use the REPLACE method to replace the data item that is associated with the specified key with new data.

If you do not specify the data variables with the DEFINEDATA method, the data variables are automatically assumed to be the same as the keys.

If you use the KEY and DATA argument tags to specify the key and data directly, you must use both argument tags.

The ADD method does not set the value of the data variable to the value of the data item. The ADD method sets only the value in the hash object.

See Also

- "Storing and Retrieving Data" in *SAS Language Reference: Concepts*

Methods:

- "DEFINEDATA Method" on page 21
- "DEFINEKEY Method" on page 24
- "REF Method" on page 54

CHECK Method

Checks whether the specified key is stored in the hash object.

Applies to: Hash object

Syntax

rc=*object*.**CHECK** (<KEY: *keyvalue-1*, … KEY: *keyvalue-n*>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: ***keyvalue***
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

Details

You can use the CHECK method in one of two ways to find data in a hash object.

You can specify the key, and then use the CHECK method as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();

  /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Define constant key and data values and add to hash object */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   /* Verify that JOYCE key is in hash object */
   k = 'Joyce';
   rc = h.check();
   if (rc = 0) then
      put 'Key is in the hash object.';
run;
```

Alternatively, you can use a shortcut and specify the key directly in the CHECK method call as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
   if _N_ = 1 then do;
```

```
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();

  /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Define constant key and data values and add to hash object */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   /* Verify that JOYCE key is in hash object */
   rc = h.check(key: 'Joyce');
   if (rc  =0) then
      put 'Key is in the hash object.';
run;
```

Comparisons

The CHECK method returns only a value that indicates whether the key is in the hash object. The data variable that is associated with the key is not updated. The FIND method also returns a value that indicates whether the key is in the hash object. However, if the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

See Also

Methods:

- "DEFINEKEY Method" on page 24
- "FIND Method" on page 30

CLEAR Method

Removes all items from the hash object without deleting the hash object instance.

Applies to: Hash object

Syntax

rc=*object*.**CLEAR** ();

Arguments

rc

specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object

specifies the name of the hash object.

Details

The CLEAR method enables you to remove items from and reuse an existing hash object without having to delete the object and create a new one. If you want to remove the hash object instance completely, use the DELETE method.

Note: The CLEAR method does not change the value of the DATA step variables. It clears only the values in the hash object.

Example: Clearing a Hash Object

The following example declares a hash object, gets the number of items in the hash object, and then clears the hash object without deleting it.

```
data mydata;
  do i = 1 to 10000;
    output;
  end;
run;
data _null_;
  length i 8;

/* Declares the hash object named MYHASH using the data set MyData. */
dcl hash myhash(dataset: 'mydata');
 myhash.definekey('i');
 myhash.definedone();
 call missing (i);
/* Uses the NUM_ITEMS attribute, which returns the */
/* number of items in the hash object.             */
 n = myhash.num_items;
 put n=;
/* Uses the CLEAR method to delete all items within MYHASH. */
 rc = myhash.clear();
/* Writes the number of items in the log. */
 n = myhash.num_items;
 put n=;
 run;
```

The first PUT statement writes the number of items in the hash table MYHASH before it is cleared.

```
n=10000
```

The second PUT statement writes the number of items in the hash table MYHASH after it is cleared.

```
n=0
```

See Also

Methods:

- “DELETE Method, Hash and Hash Iterator Objects” on page 26

DECLARE Statement, Hash and Hash Iterator Objects

Declares a hash or hash iterator object; creates an instance of and initializes data for a hash or hash iterator object.

Valid in: DATA step

Category: Action

Type: Executable

Alias: DCL

Applies to: Hash object, Hash iterator object

Syntax

Form 1: **DECLARE** *object object-reference*;

Form 2: **DECLARE** *object object-reference* <(*argument_tag-1*: *value-1*, …*argument_tag-n*: *value-n*)>;

Arguments

object

specifies the component object. It can be one of the following values:

hash

specifies a hash object. The hash object provides a mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys.

See "Using the Hash Object " in *SAS Language Reference: Concepts*

hiter

specifies a hash iterator object. The hash iterator object enables you to retrieve the hash object's data in forward or reverse key order.

See "Using the Hash Object " in *SAS Language Reference: Concepts*

object-reference

specifies the object reference name for the hash or hash iterator object.

argument_tag*:*value

specifies the information that is used to create an instance of the hash object.

There are five valid hash object argument and value tags:

dataset: '*dataset_name* <(*datasetoption*)>'

Specifies the name of a SAS data set to load into the hash object.

The name of the SAS data set can be a literal or character variable. The data set name must be enclosed in single or double quotation marks. Macro variables must be enclosed in double quotation marks.

You can use SAS data set options when declaring a hash object in the DATASET argument tag. Data set options specify actions that apply only to the SAS data set with which they appear. They enable you to perform the following operations:

- renaming variables

- selecting a subset of observations based on observation number for processing
- selecting observations using the WHERE option
- dropping or keeping variables from a data set loaded into a hash object, or for an output data set that is specified in an OUTPUT method call
- specifying a password for a data set.

The following syntax is used:

```
dcl hash h (dataset: 'x (where = (i > 10))');
```

For a list of SAS data set options, see the *SAS Data Set Options: Reference*

Note If the data set contains duplicate keys, the default is to keep the first instance in the hash object; subsequent instances are ignored. To store the last instance in the hash object or an error message written to the SAS log if there is a duplicate key, use the DUPLICATE argument tag.

duplicate: '*option*'
determines whether to ignore duplicate keys when loading a data set into the hash object. The default is to store the first key and ignore all subsequent duplicates. Option can be one of the following values:

'replace' | 'r'
stores the last duplicate key record.

'error' | 'e'
reports an error to the log if a duplicate key is found.

The following example that uses the REPLACE option stores **`brown`** for the key 620 and **`blue`** for the key 531. If you use the default, **`green`** would be stored for 620 and **`yellow`** would be stored for 531.

```
data table;
  input key data $;
  datalines;
  531 yellow
  620 green
  531 blue
  908 orange
  620 brown
  143 purple
 run;

data _null_;
length key 8 data $ 8;
if (_n_ = 1) then do;
    declare hash myhash(dataset: "table", duplicate: "r");
    rc = myhash.definekey('key');
    rc = myhash.definedata('data');
    myhash.definedone();
 end;
rc = myhash.output(dataset:"otable");
run;
```

hashexp: *n*
The hash object's internal table size, where the size of the hash table is 2^n.

The value of HASHEXP is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for HASHEXP equates to a hash table size of 2^4, or 16. The maximum value for HASHEXP is 20.

The hash table size is not equal to the number of items that can be stored. Imagine the hash table as an array of 'buckets.' A hash table size of 16 would have 16 'buckets.' Each bucket can hold an infinite number of items. The efficiency of the hash table lies in the ability of the hashing function to map items to and retrieve items from the buckets.

You should specify the hash table size relative to the amount of data in the hash object in order to maximize the efficiency of the hash object lookup routines. Try different HASHEXP values until you get the best result. For example, if the hash object contains one million items, a hash table size of 16 (HASHEXP = 4) would work, but not very efficiently. A hash table size of 512 or 1024 (HASHEXP = 9 or 10) would result in the best performance.

Default 8, which equates to a hash table size of 2^8 or 256

keysum:'*variable-name***'**
specifies the name of a variable that tracks the key summary for all keys. A key summary is a count of how many times a key has been referenced on a FIND method call.

Note The key summary is in the output data set.

Example "Example 5: Adding the Key Summary to the Output Data Set" on page 20

ordered: '*option***'**
Specifies whether or how the data is returned in key-value order if you use the hash object with a hash iterator object or if you use the hash object OUTPUT method.

option can be one of the following values:

'ascending' | 'a'
Data is returned in ascending key-value order. Specifying '**`ascending`**' is the same as specifying '**`yes`**'.

'descending' | 'd'
Data is returned in descending key-value order.

'YES' | 'Y'
Data is returned in ascending key-value order. Specifying '**`yes`**' is the same as specifying '**`ascending`**'.

'NO' | 'N'
Data is returned in some undefined order.

Default NO

Tip The argument can also be enclosed in double quotation marks.

multidata: '*option***'**
specifies whether multiple data items are allowed for each key.

option can be one of the following values:

'YES' | 'Y'
Multiple data items are allowed for each key.

'NO' | 'N'
Only one data item is allowed for each key.

Default NO

Tip The argument value can also be enclosed in double quotation marks.

See "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

suminc: '*variable-name*'
maintains a summary count of hash object keys. The SUMINC argument tag is given a DATA step variable, which holds the sum increment. The sum increment is how much to add to the key summary for each reference to the key.

See "Maintaining Key Summaries" in *SAS Language Reference: Concepts*

Example A key summary changes using the current value of the DATA step variable.

```
dcl hash myhash(suminc: 'count');
```

See "Initializing Hash Object Data Using a Constructor" in *SAS Language Reference: Concepts* and "Declaring and Instantiating a Hash Iterator Object" in *SAS Language Reference: Concepts*

Details

The Basics

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing predefined component objects from within the DATA step.

For more information about the predefined DATA step component objects, see "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*.

Declaring a Hash or Hash Iterator Object (Form 1)

You use the DECLARE statement to declare a hash or hash iterator object.

```
declare hash h;
```

The DECLARE statement tells SAS that the object reference H is a hash object.

After you declare the new hash or hash iterator object, use the _NEW_ operator to instantiate the object. For example, in the following line of code, the _NEW_ operator creates the hash object and assigns it to the object reference H:

```
h = _new_ hash( );
```

Using the DECLARE Statement to Instantiate a Hash or Hash Iterator Object (Form 2)

As an alternative to the two-step process of using the DECLARE statement and the _NEW_ operator to declare and instantiate a hash or hash iterator object, you can use the DECLARE statement to declare and instantiate the hash or hash iterator object in one step. For example, in the following line of code, the DECLARE statement declares and instantiates a hash object and assigns it to the object reference H:

```
declare hash h( );
```

The previous line of code is equivalent to using the following code:

```
declare hash h;
h = _new_ hash( );
```

A *constructor* is a method that you can use to instantiate a hash object and initialize the hash object data. For example, in the following line of code, the DECLARE statement declares and instantiates a hash object and assigns it to the object reference H. In addition, the hash table size is initialized to a value of 16 (2^4) using the argument tag, HASHEXP.

```
declare hash h(hashexp: 4);
```

Using SAS Data Set Options When Loading a Hash Object

SAS data set options can be used when declaring a hash object that uses the DATASET argument tag. Data set options specify actions that apply only to the SAS data set with which they appear. They enable you to perform the following operations:

- renaming variables
- selecting a subset of observations based on observation number for processing
- selecting observations using the WHERE option
- dropping or keeping variables from a data set loaded into a hash object, or for an output data set that is specified in an OUTPUT method call
- specifying a password for a data set.

The following syntax is used:

```
dcl hash h(dataset: 'x (where = (i > 10))');
```

For more examples of using data set options, see "Example 4: Using SAS Data Set Options When Loading a Hash Object" on page 19. For a list of data set options, see *SAS Data Set Options: Reference*.

Comparisons

You can use the DECLARE statement and the _NEW_ operator, or the DECLARE statement alone to declare and instantiate an instance of a hash or hash iterator object.

Examples

Example 1: Declaring and Instantiating a Hash Object By Using the DECLARE Statement and _NEW_ Operator

This example uses the DECLARE statement to declare a hash object. The _NEW_ operator is used to instantiate the hash object.

```
data _null_;
   length k $15;
   length d $15;
   if _N_ = 1 then do;
      /* Declare and instantiate hash object "myhash" */
      declare hash myhash;
      myhash = _new_ hash( );
      /* Define key and data variables */
      rc = myhash.defineKey('k');
      rc = myhash.defineData('d');
      rc = myhash.defineDone( );
```

```
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Create constant key and data values */
   rc = myhash.add(key: 'Labrador', data: 'Retriever');
   rc = myhash.add(key: 'Airedale', data: 'Terrier');
   rc = myhash.add(key: 'Standard', data: 'Poodle');
   /* Find data associated with key and write data to log */
   rc = myhash.find(key: 'Airedale');
   if (rc = 0) then
      put d=;
   else
      put 'Key Airedale not found';
run;
```

Example 2: Declaring and Instantiating a Hash Object By Using the DECLARE Statement

This example uses the DECLARE statement to declare and instantiate a hash object in one step.

```
data _null_;
   length k $15;
   length d $15;
   if _N_ = 1 then do;
      /* Declare and instantiate hash object "myhash" */
      declare hash myhash( );
      rc = myhash.defineKey('k');
      rc = myhash.defineData('d');
      rc = myhash.defineDone( );
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Create constant key and data values */
   rc = myhash.add(key: 'Labrador', data: 'Retriever');
   rc = myhash.add(key: 'Airedale', data: 'Terrier');
   rc = myhash.add(key: 'Standard', data: 'Poodle');
   /* Find data associated with key and write data to log*/
   rc = myhash.find(key: 'Airedale');
   if (rc = 0) then
      put d=;
   else
      put 'Key Airedale not found';
run;
```

Example 3: Instantiating and Sizing a Hash Object

This example uses the DECLARE statement to declare and instantiate a hash object. The hash table size is set to 16 (2^4).

```
data _null_;
   length k $15;
   length d $15;
   if _N_ = 1 then do;
      /* Declare and instantiate hash object "myhash". */
      /* Set hash table size to 16. */
      declare hash myhash(hashexp: 4);
      rc = myhash.defineKey('k');
```

```
      rc = myhash.defineData('d');
      rc = myhash.defineDone( );
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Create constant key and data values */
   rc = myhash.add(key: 'Labrador', data: 'Retriever');
   rc = myhash.add(key: 'Airedale', data: 'Terrier');
   rc = myhash.add(key: 'Standard', data: 'Poodle');
   rc = myhash.find(key: 'Airedale');
   /* Find data associated with key and write data to log*/
   if (rc = 0) then
      put d=;
   else
      put 'Key Airedale not found';
run;
```

Example 4: Using SAS Data Set Options When Loading a Hash Object

The following examples use various SAS data set options when declaring a hash object:

```
data x;
retain j 999;
do i = 1 to 20;
   output;
 end;
run;
/* Using the WHERE option. */
data _null_;
  length i 8;
  dcl hash h(dataset: 'x (where =(i > 10))', ordered: 'a');
  h.definekey('i');
  h.definedone();
  h.output(dataset: 'out');
  run;
/* Using the DROP option. */
data _null_;
  length i 8;
  dcl hash h(dataset: 'x (drop = j)', ordered: 'a');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out (where =( i < 8))');
  run;
/* Using the FIRSTOBS option. */
data _null_;
  length i j 8;
  dcl hash h(dataset: 'x (firstobs=5)', ordered: 'a');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out');
  run;
/* Using the OBS option. */
data _null_;
  length i j 8;
  dcl hash h(dataset: 'x (obs=5)', ordered: 'd');
  h.definekey(all: 'y');
```

```
      h.definedone();
      h.output(dataset: 'out (rename =(j=k))');
      run;
```

For a list of SAS data set options, see *SAS Data Set Options: Reference*.

Example 5: Adding the Key Summary to the Output Data Set

The following example declares the variable, *ks*, to hold the key summary and adds the variable to the output data set.

```
data key;
   length key data 8;
   input key data;
   datalines;
      1 10
      2 11
      3 20
      5 5
      4 6
run;

data _null_;
   length key data r i sum 8;
   length ks 8;
   i = 0;
   dcl hash h(dataset:'key', suminc: 'i', keysum: 'ks');
   h.definekey('key');
   h.definedata('key', 'data');
   h.definedone();

   i = 1;
   do key = 1 to 5;
      rc = h.find();
   end;

   do key = 1 to 3;
      rc = h.find();
   end;

   rc = h.output(dataset:'out');
 run;

proc print data=out;
run;
```

Output 2.1 *Output of Key Summary Data*

The SAS System

Obs	key	data	ks
1	2	11	2
2	5	5	1
3	1	10	2
4	3	20	2
5	4	6	1

See Also

- "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

DEFINEDATA Method

Defines data, associated with the specified data variables, to be stored in the hash object.

Applies to: Hash object

Syntax

rc=*object*.**DEFINEDATA** ('*datavarname-1*' <, …'*datavarname-n*'>);

rc=*object*.**DEFINEDATA** (ALL: 'YES' | "YES");

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

'*datavarname*'
specifies the name of the data variable.

The data variable name can also be enclosed in double quotation marks.

ALL: 'YES' | “YES”
specifies all the data variables as data when the data set is loaded in the object constructor.

If the *dataset* argument tag is used in the DECLARE statement or _NEW_ operator to automatically load a data set, then you can define all the data set variables as data by using the ALL: 'YES' option.

Details

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

Note: If you use the shortcut notation for the ADD or REPLACE method (for example, **`h.add(key:99, data:'apple', data:'orange')`**) and use the ALL:'YES' option on the DEFINEDATA method, then you must specify the data in the same order as it exists in the data set.

Note: The hash object does not assign values to key variables (for example, **`h.find(key:'abc')`**), and the SAS compiler cannot detect the key and data variable assignments that are performed by the hash object and the hash iterator. Therefore, if no assignment to a key or data variable appears in the program, then SAS will issue a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of the following actions:

- Set the NONOTES system option.
- Provide an initial assignment statement (typically to a missing value) for each key and data variable.
- Use the CALL MISSING routine with all the key and data variables as parameters. Here is an example:

```
length d $20;
length k $20;
if _N_ = 1 then do;
   declare hash h();
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   call missing(k,d);
end;
```

For detailed information about how to use the DEFINEDATA method, see “Defining Keys and Data” in *SAS Language Reference: Concepts*.

Example

The following example creates a hash object and defines the key and data variables:

```
data _null_;
   length d $20;
   length k $20;
   /* Declare the hash object and key and data variables */
```

```
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
run;
```

See Also

- "Defining Keys and Data" in *SAS Language Reference: Concepts*

Methods:

- "DEFINEDONE Method" on page 23
- "DEFINEKEY Method" on page 24

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

DEFINEDONE Method

Indicates that all key and data definitions are complete.

Applies to: Hash object

Syntax

rc = *object*.**DEFINEDONE**( );

rc = *object*.**DEFINEDONE** (MEMRC: 'y');

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

memrc:'y'
enables recovery from memory failure when loading a data set into a hash object.

If a call fails because of insufficient memory to load a data set, a nonzero return code is returned. The hash object frees the principal memory in the underlying array. The

only allowable operation after this type of failure is deletion via the DELETE method.

Details

When the DEFINEDONE method is called and the *dataset* argument tag is used with the constructor, the data set is loaded into the hash object.

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

For detailed information about how to use the DEFINEDONE method, see "Defining Keys and Data" in *SAS Language Reference: Concepts*.

See Also

- "Defining Keys and Data" in *SAS Language Reference: Concepts*

Methods:

- "DEFINEDATA Method" on page 21
- "DEFINEKEY Method" on page 24

DEFINEKEY Method

Defines key variables for the hash object.

Applies to: Hash object

Syntax

rc=*object*.**DEFINEKEY**('*keyvarname-1* '<, … '*keyvarname-n*'>);
rc=*object*.**DEFINEKEY**(ALL: 'YES' | "YES");

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

'keyvarname'
specifies the name of the key variable.

The key variable name can also be enclosed in double quotation marks.

ALL: 'YES' | ”YES”
specifies all the data variables as keys when the data set is loaded in the object constructor.

If you use the *dataset* argument tag in the DECLARE statement or _NEW_ operator to automatically load a data set, then you can define all the key variables by using the ALL: 'YES' option.

Details

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

For more information about how to use the DEFINEKEY method, see “Defining Keys and Data” in *SAS Language Reference: Concepts*.

Note: If you use the shortcut notation for the ADD, CHECK, FIND, REMOVE, or REPLACE methods (for example, **`h.add(key:99, data:'apple', data:'orange')`**) and the ALL:'YES' option on the DEFINEKEY method, then you must specify the keys and data in the same order as they exist in the data set.

Note: The hash object does not assign values to key variables (for example, **`h.find(key:'abc')`**), and the SAS compiler cannot detect the key and data variable assignments done by the hash object and the hash iterator. Therefore, if no assignment to a key or data variable appears in the program, SAS will issue a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of the following actions:

- Set the NONOTES system option.
- Provide an initial assignment statement (typically to a missing value) for each key and data variable.
- Use the CALL MISSING routine with all the key and data variables as parameters. Here is an example:

```
length d $20;
length k $20;
if _N_ = 1 then do;
   declare hash h();
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   call missing(k, d);
end;
```

See Also

- “Defining Keys and Data” in *SAS Language Reference: Concepts*

Methods:

- “DEFINEDATA Method” on page 21
- “DEFINEDONE Method” on page 23

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

DELETE Method, Hash and Hash Iterator Objects

Deletes the hash or hash iterator object.

Applies to: Hash object, Hash interator object

Syntax

rc=*object*.**DELETE**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is printed to the log.

object
specifies the name of the hash or hash iterator object.

Details

DATA step component objects are deleted automatically at the end of the DATA step. If you want to reuse the object reference variable in another hash or hash iterator object constructor, you should delete the hash or hash iterator object by using the DELETE method.

If you attempt to use a hash or hash iterator object after you delete it, you will receive an error in the log.

If you want to delete all the items from within a hash object and save the hash object to use again, use the "CLEAR Method" on page 11.

DO_OVER Method

Traverses a list of duplicate keys in the hash object.

Applies to: Hash object

Syntax

object.**DO_OVER** (KEY:*keyvalue*);

Arguments

object
specifies the name of the hash object.

KEY:*keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

Details

When a hash object has multiple values for a single key, you can use the DO_OVER method in an iterative DO loop to traverse the duplicate keys. The DO_OVER method reads the key on the first method call and continues to traverse the duplicate key list until the key reaches the end.

Note: If you switch the key in the middle of an iteration, you must use the RESET_DUP method to reset the pointer to the beginning of the list. Otherwise, SAS continues to use the first key.

Example

The following example creates a data set, **dup**, that contains duplicate keys. The DO_OVER and RESET_DUP methods are used to iterate through the duplicate keys.

```
data dup;
 length key data 8;
 input key data;
 datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
run;

data _null_;
 length r 8;
 dcl hash h(dataset:'dup', multidata: 'y', ordered: 'y');
 h.definekey('key');
 h.definedata('key', 'data');
 h.definedone();

h.reset_dup();
key = 2;
do while(h.do_over(key:key) eq 0);
  put key= data=;
end;

key = 3;
```

```
do while(h.do_over(key:key) eq 0);
  put key= data=;
end;

key = 2;
do while(h.do_over(key:key) eq 0);
  put key= data=;
end;

run;
```

The following lines are written to the SAS log.

```
key=2 data=11
key=2 data=16
key=2 data=9
key=3 data=20
key=3 data=100
key=2 data=11
key=2 data=16
key=2 data=9
```

See Also

Methods:

- "RESET_DUP Method" on page 66

EQUALS Method

Determines whether two hash objects are equal.

Applies to: Hash object

Syntax

rc=*object*.**EQUALS** (HASH: '*object*', RESULT: *variable name*);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of a hash object.

HASH:'*object*'
specifies the name of the second hash object that is compared to the first hash object.

RESULT: *variable name*
specifies the name of a numeric variable name to hold the result. If the hash objects are equal, the result variable is 1. Otherwise, the result variable is zero.

Details

The following example compares H1 to H2 hash objects:

```
length eq k 8;
declare hash h1();
h1.defineKey('k');
h1.defineDone();

declare hash h2();
h2.defineKey('k');
h2.defineDone();

rc = h1.equals(hash: 'h2', result: eq);
if eq then
   put 'hash objects equal';
else
   put 'hash objects not equal';
```

The two hash objects are defined as equal when all of the following conditions occur:

- Both hash objects are the same size—that is, the HASHEXP sizes are equal.
- Both hash objects have the same number of items—that is, H1.NUM_ITEMS = H2.NUM_ITEMS.
- Both hash objects have the same key and data structure.
- In an unordered iteration over H1 and H2 hash objects, each successive record from H1 has the same key and data fields as the corresponding record in H2—that is, each record is in the same position in each hash object and each such record is identical to the corresponding record in the other hash object.

Example: Comparing Two Hash Objects

In the following example, the first return call to EQUALS returns a nonzero value and the second return call returns a zero value.

```
data x;
   length k eq 8;
   declare hash h1();
   h1.defineKey('k');
   h1.defineDone();

   declare hash h2();
   h2.defineKey('k');
   h2.defineDone();

   k = 99;
   h1.add();
   h2.add();
   rc = h1.equals(hash: 'h2', result: eq);
   put eq=;

   k = 100;
   h2.replace();

   rc = h1.equals(hash: 'h2', result: eq);
   put eq=;
```

```
run;
```

FIND Method

Determines whether the specified key is stored in the hash object.

Applies to: Hash object

Syntax

rc=*object*.**FIND** (<KEY: *keyvalue-1*, …KEY: *keyvalue-n*>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: *keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

Details

You can use the FIND method in one of two ways to find data in a hash object.

You can specify the key, and then use the FIND method as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variables */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   /* Define constant key and data values */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   /* Find the key JOYCE */
   k = 'Joyce';
   rc = h.find();
   if (rc = 0) then
      put 'Key is in the hash object.';
```

```
run;
```

Alternatively, you can use a shortcut and specify the key directly in the FIND method call as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variables */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
    /* Define constant key and data values */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Find the key JOYCE */
   rc = h.find(key: 'Joyce');
   if (rc = 0) then
      put 'Key is in the hash object.';
run;
```

If the hash object has multiple data items for each key, use “FIND_NEXT Method” on page 32 and “FIND_PREV Method” on page 34 in conjunction with the FIND method to traverse a multiple data item list.

Comparisons

The FIND method returns a value that indicates whether the key is in the hash object. If the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call. The CHECK method returns only a value that indicates whether the key is in the hash object. The data variable is not updated.

Example: Using the FIND Method to Find the Key in a Hash Object

The following example creates a hash object. Two data values are added. The FIND method is used to find a key in the hash object. The data value is returned to the data set variable that is associated with the key.

```
data _null_;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      /* avoid uninitialized variable notes */
      call missing(k, d);
      rc = h.defineDone();
   end;
   /* Define constant key and data values and add to hash object */
```

```
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   rc = h.add(key: 'Homer', data: 'Odyssey');
   /* Verify that key JOYCE is in hash object and */
   /* return its data value to the data set variable D */
   rc = h.find(key: 'Joyce');
   put d=;
run;
```

d=Ulysses is written to the SAS log.

See Also

- "Storing and Retrieving Data" in *SAS Language Reference: Concepts*

Methods:

- "CHECK Method" on page 9
- "DEFINEKEY Method" on page 24
- "FIND_NEXT Method" on page 32
- "FIND_PREV Method" on page 34
- "REF Method" on page 54

FIND_NEXT Method

Sets the current list item to the next item in the current key's multiple item list and sets the data for the corresponding data variables.

Applies to: Hash object

Syntax

rc=*object*.**FIND_NEXT**( );

Arguments

rc

specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message is printed to the log.

object

specifies the name of the hash object.

Details

The FIND method determines whether the key exists in the hash object. The HAS_NEXT method determines whether the key has multiple data items associated with it. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example

This example uses the FIND_NEXT method to iterate through a data set where several keys have multiple data items. If a key has more than one data item, subsequent items are marked **dup**.

```
data dup;
   length key data 8;
   input key data;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
   ;
data _null_;
   dcl hash h(dataset:'dup', multidata: 'y');
   h.definekey('key');
   h.definedata('key', 'data');
   h.definedone();
   /* avoid uninitialized variable notes */
   call missing (key, data);
   do key = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put key= data=;
         rc = h.find_next();
         do while(rc = 0);
            put 'dup ' key= data;
            rc = h.find_next();
         end;
      end;
   end;
 run;
```

The following lines are written to the SAS log.

```
key=1 data=10
dup key=1 5
dup key=1 15
key=2 data=11
dup key=2 9
dup key=2 16
key=3 data=20
dup key=3 100
key=4 data=6
key=5 data=5
dup key=5 99
```

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "FIND Method" on page 30
- "FIND_PREV Method" on page 34
- "HAS_NEXT Method" on page 36

FIND_PREV Method

Sets the current list item to the previous item in the current key's multiple item list and sets the data for the corresponding data variables.

Applies to: Hash object

Syntax

rc=*object*.**FIND_PREV**( );

Arguments

rc

specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message is printed to the log.

object

specifies the name of the hash object.

Details

The FIND method determines whether the key exists in the hash object. The HAS_PREV method determines whether the key has multiple data items associated with it. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list. See "HAS_NEXT Method" on page 36 for an example.

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "FIND Method" on page 30
- "FIND_NEXT Method" on page 32
- "HAS_PREV Method" on page 38

FIRST Method

Returns the first value in the underlying hash object.

Applies to: Hash iterator object

Syntax

rc=*object*.**FIRST**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message will be printed to the log.

object
specifies the name of the hash iterator object.

Details

The FIRST method returns the first data item in the hash object. If you use the **`ordered: 'yes'`** or **`ordered: 'ascending'`** argument tag in the DECLARE statement or _NEW_ operator when you instantiate the hash object, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in ascending key-value order in the hash object. Repeated calls to the NEXT method will iteratively traverse the hash object and return the data items in ascending key order. Conversely, if you use the **`ordered: 'descending'`** argument tag in the DECLARE statement or _NEW_ operator when you instantiate the hash object, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in descending key-value order in the hash object. Repeated calls to the NEXT method will iteratively traverse the hash object and return the data items in descending key order.

Use the LAST method to return the last data item in the hash object.

Note: The FIRST method sets the data variable to the value of the data item so that it is available for use after the method call.

Example: Retrieving Hash Object Data

The following example creates a data set that contains sales data. You want to list products in order of sales. The data is loaded into a hash object and the FIRST and NEXT methods are used to retrieve the data.

```
data work.sales;
   input prod $1-6 qty $9-14;
   datalines;
banana  398487
apple   384223
orange  329559
```

```
;
data _null_;
   /* Declare hash object and read SALES data set as ordered */
   if _N_ = 1 then do;
      length prod $10;
      length qty $6;
      declare hash h(dataset: 'work.sales', ordered: 'yes');
      declare hiter iter('h');
      /* Define key and data variables */
      h.defineKey('qty');
      h.defineData('prod');
      h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(qty, prod);
   end;
   /* Iterate through the hash object and output data values */
   rc = iter.first();
   do while (rc = 0);
      put prod=;
      rc = iter.next();
   end;
run;
```

The following lines are written to the SAS log:

```
prod=orange
prod=apple
prod=banana
```

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "LAST Method" on page 40

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

HAS_NEXT Method

Determines whether there is a next item in the current key's multiple data item list.

Applies to: Hash object

Syntax

rc=*object*.**HAS_NEXT** (RESULT: R);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

RESULT:R
specifies the numeric variable **R**, which receives a zero value if there is not another data item in the data item list or a nonzero value if there is another data item in the data item list.

Details

If a key has multiple data items, you can use the HAS_NEXT method to determine whether there is a next item in the current key's multiple data item list. If there is another item, the method will return a nonzero value in the numeric variable **R**. Otherwise, it will return a zero.

The FIND method determines whether the key exists in the hash object. The HAS_NEXT method determines whether the key has multiple data items associated with it. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example: Finding Data Items

This example creates a hash object where several keys have multiple data items. It uses the HAS_NEXT method to find all the data items.

```
data testdup;
   length key data 8;
   input key data;
   datalines;
   1 100
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
data _null_;
   length r 8;
   dcl hash h(dataset:'testdup', multidata: 'y');
   h.definekey('key');
   h.definedata('key', 'data');
```

```
   h.definedone();
   call missing (key, data);
   do key = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put key= data=;
         h.has_next(result: r);
         do while(r ne 0);
             rc = h.find_next();
             put 'dup ' key= data;
             h.has_next(result: r);
         end;
      end;
   end;
run;
```

The following lines are written to the SAS log.

```
key=1 data=100
dup key=1 5
dup key=1 15
key=2 data=11
dup key=2 9
dup key=2 16
key=3 data=20
dup key=3 100
key=4 data=6
key=5 data=5
dup key=5 99
```

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "FIND Method" on page 30
- "FIND_NEXT Method" on page 32
- "FIND_PREV Method" on page 34
- "HAS_PREV Method" on page 38

HAS_PREV Method

Determines whether there is a previous item in the current key's multiple data item list.

Applies to: Hash object

Syntax

rc=*object*.**HAS_PREV** (RESULT: R);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

RESULT:R
specifies the numeric variable **R**, which receives a zero value if there is not another data item in the data item list or a nonzero value if there is another data item in the data item list.

Details

If a key has multiple data items, you can use the HAS_PREV method to determine whether there is a previous item in the current key's multiple data item list. If there is a previous item, the method will return a nonzero value in the numeric variable **R**. Otherwise, it will return a zero.

The FIND method determines whether the key exists in the hash object. The HAS_NEXT method determines whether the key has multiple data items associated with it. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list. See "HAS_NEXT Method" on page 36 for an example.

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "FIND Method" on page 30
- "FIND_NEXT Method" on page 32
- "FIND_PREV Method" on page 34
- "HAS_NEXT Method" on page 36

ITEM_SIZE Attribute

Returns the size (in bytes) of an item in a hash object.

Applies to: Hash object

Syntax

variable_name=*object*.**ITEM_SIZE**;

Arguments

variable_name
specifies the name of the variable that contains the size of the item in the hash object.

object
specifies the name of the hash object.

Details

The ITEM_SIZE attribute returns the size (in bytes) of an item, which includes the key and data variables and some additional internal information. You can get an estimate of how much memory the hash object is using with the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash object requires, nor does it take into account any necessary internal alignments. Therefore, ITEM_SIZE does not provide exact memory usage, but it does return a good approximation.

Example: Returning the Size of a Hash Item

The following example uses ITEM_SIZE to return the size of the item in MYHASH:

```
data work.stock;
   input prod $1-10 qty 12-14;
   datalines;
broccoli 345
corn 389
potato 993
onion 730
;
data _null_;
   if _N_ = 1 then do;
      length prod $10;
   /* Declare hash object and read STOCK data set as ordered */
      declare hash myhash(dataset: "work.stock");
      /* Define key and data variables */
      myhash.defineKey('prod');
      myhash.defineData('qty');
      myhash.defineDone();
   end;
   /* Add a key and data value to the hash object */
   prod = 'celery';
   qty = 183;
   rc = myhash.add();

   /* Use ITEM_SIZE to return the size of the item in hash object */
   itemsize = myhash.item_size;
   put itemsize=;
run;
```

itemsize=40 is written to the SAS log.

LAST Method

Returns the last value in the underlying hash object.

Applies to: Hash iterator object

Syntax

rc=*object*.**LAST**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash iterator object.

Details

The LAST method returns the last data item in the hash object. If you use the **`ordered: 'yes'`** or **`ordered: 'ascending'`** argument tag in the DECLARE statement or _NEW_ operator when you instantiate the hash object, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in ascending key-value order in the hash object. Conversely, if you use the **`ordered: 'descending'`** argument tag in the DECLARE statement or _NEW_ operator when you instantiate the hash object, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in descending key-value order in the hash object.

Use the FIRST method to return the first data item in the hash object.

Note: The LAST method sets the data variable to the value of the data item so that it is available for use after the method call.

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "FIRST Method" on page 35

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

NEW Operator, Hash and Hash Iterator Objects

Creates an instance of a hash or hash iterator object.

Applies to: Hash object, Hash iterator object

Syntax

object-reference = **_NEW_***object* (<*argument_tag-1*: *value-1* <, …*argument_tag-n*: *value-n*> >);

Arguments

object-reference
specifies the object reference name for the hash or hash iterator object.

object
specifies the component object. It can be one of the following:

hash — indicates a hash object. The hash object provides a mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys.

hiter — indicates a hash iterator object. The hash iterator object enables you to retrieve the hash object's data in forward or reverse key order.

See "Using DATA Step Component Objects" in *SAS Language Reference: Concepts* and "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

argument_tag*:*value
specifies the information that is used to create an instance of the hash object.

Valid hash object argument tags and values are

dataset: '*dataset_name* <(*datasetoption*)>'
names a SAS data set to load into the hash object.

The name of the SAS data set can be a literal or character variable. The data set name must be enclosed in single or double quotation marks. Macro variables must be enclosed in double quotation marks.

You can use SAS data set options when declaring a hash object in the DATASET argument tag. Data set options specify actions that apply only to the SAS data set with which they appear. They enable you to perform the following operations:

- renaming variables
- selecting a subset of observations based on observation number for processing
- selecting observations using the WHERE option
- dropping or keeping variables from a data set loaded into a hash object, or for an output data set specified in an OUTPUT method call
- specifying a password for a data set.

The following syntax is used:

```
dcl hash h;
h = _new_ hash (dataset: 'x (where = (i > 10))');
```

For a list of SAS data set options, see the *SAS Data Set Options: Reference*.

Note If the data set contains duplicate keys, the default is to keep the first instance in the hash object; subsequent instances are ignored. To store the

last instance in the hash object or to write an error message in the SAS log if there is a duplicate key, use the DUPLICATE argument tag.

duplicate: '*option*'

determines whether to ignore duplicate keys when loading a data set into the hash object. The default is to store the first key and ignore all subsequent duplicates. Option can be one of the following values:

'replace' | 'r'
stores the last duplicate key record.

'error' | 'e'
reports an error to the log if a duplicate key is found.

The following example using the REPLACE option stores **brown** for the key 620 and **blue** for the key 531. If you use the default, **green** would be stored for 620 and **yellow** would be stored for 531.

```
data table;
  input key data $;
  datalines;
  531 yellow
  620 green
  531 blue
  908 orange
  620 brown
  143 purple
 run;
data _null_;
length key 8 data $ 8;
if (_n_ = 1) then do;
    declare hash myhash;
    myhash = _new_ hash (dataset: "table", duplicate: "r");
    rc = myhash.definekey('key');
    rc = myhash.definedata('data');
    myhash.definedone();
 end;
rc = myhash.output(dataset:"otable");
run;
```

hashexp: *n*

is the hash object's internal table size, where the size of the hash table is 2^n.

The value of HASHEXP is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for HASHEXP equates to a hash table size of 2^4, or 16. The maximum value for HASHEXP is 20.

The hash table size is not equal to the number of items that can be stored. Imagine the hash table as an array of 'buckets.' A hash table size of 16 would have 16 'buckets.' Each bucket can hold an infinite number of items. The efficiency of the hash table lies in the ability of the hashing function to map items to and retrieve items from the buckets.

You should set the hash table size relative to the amount of data in the hash object in order to maximize the efficiency of the hash object lookup routines. Try different HASHEXP values until you get the best result. For example, if the hash object contains one million items, a hash table size of 16 (HASHEXP = 4) would work, but not very efficiently. A hash table size of 512 or 1024 (HASHEXP = 9 or 10) would result in the best performance.

Default 8, which equates to a hash table size of 2^8 or 256

keysum:'*variable-name*'
specifies the name of a variable that tracks the key summary for all keys. A key summary is a count of how many times a key has been referenced on a FIND method call.

Note The key summary is in the output data set.

ordered: '*option*'
specifies whether or how the data is returned in key-value order if you use the hash object with a hash iterator object or if you use the hash object OUTPUT method.

The argument value can also be enclosed in double quotation marks.

option can be one of the following values:

'ascending' | 'a'	Data is returned in ascending key-value order. Specifying '**ascending**' is the same as specifying '**yes**'.
'descending' | 'd'	Data is returned in descending key-value order.
'YES' | 'Y'	Data is returned in ascending key-value order. Specifying '**yes**' is the same as specifying '**ascending**'.
'NO' | 'N'	Data is returned in some undefined order.

Default NO

multidata: '*option*'
specifies whether multiple data items are allowed for each key.

The argument value can also be enclosed in double quotation marks.

option can be one of the following values:

'YES' | 'Y'	Multiple data items are allowed for each key.
'NO' | 'N'	Only one data item is allowed for each key.

Default NO

See "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

suminc: '*variable-name*'
maintains a summary count of hash object keys. The SUMINC argument tag is given a DATA step variable, which holds the sum increment. The sum increment is how much to add to the key summary for each reference to the key. For example, a key summary changes using the current value of the DATA step variable.

```
dcl hash myhash(suminc: 'count');
```

For more information, see "Maintaining Key Summaries" in *SAS Language Reference: Concepts*.

See "Initializing Hash Object Data Using a Constructor" in *SAS Language Reference: Concepts* and "Declaring and Instantiating a Hash Object" in *SAS Language Reference: Concepts*

Details

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing the predefined component objects from within the DATA step.

If you use the _NEW_ operator to instantiate the component object, you must first use the DECLARE statement to declare the component object. For example, in the following lines of code, the DECLARE statement tells SAS that the object reference H is a hash object. The _NEW_ operator creates the hash object and assigns it to the object reference H.

```
declare hash h();
h = _new_ hash( );
```

Note: You can use the DECLARE statement to declare and instantiate a hash or hash iterator object in one step.

A constructor is a method that is used to instantiate a component object and to initialize the component object data. For example, in the following lines of code, the _NEW_ operator instantiates a hash object and assigns it to the object reference H. In addition, the data set WORK.KENNEL is loaded into the hash object.

```
declare hash h();
h = _new_ hash(datset: "work.kennel");
```

For more information about the predefined DATA step component objects and constructors, see "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*.

Comparisons

You can use the DECLARE statement and the _NEW_ operator, or the DECLARE statement alone to declare and instantiate an instance of a hash or hash iterator object.

Example: Using the _NEW_ Operator to Instantiate and Initialize Hash Object Data

This example uses the _NEW_ operator to instantiate and initialize data for a hash object and instantiate a hash iterator object.

The hash object is filled with data, and the iterator is used to retrieve the data in key order.

```
data kennel;
   input name $1-10 kenno $14-15;
   datalines;
Charlie      15
Tanner       07
Jake         04
Murphy       01
Pepe         09
Jacques      11
Princess Z   12
;
run;
data _null_;
   if _N_ = 1 then do;
      length kenno $2;
```

```
      length name $10;
      /* Declare the hash object */
      declare hash h();
      /* Instantiate and initialize the hash object */
      h = _new_ hash(dataset:"work.kennel", ordered: 'yes');
      /* Declare the hash iterator object */
      declare hiter iter;
      /* Instantiate the hash iterator object */
      iter = _new_ hiter('h');
      /* Define key and data variables */
      h.defineKey('kenno');
      h.defineData('name', 'kenno');
      h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(kenno, name);
   end;
   /* Find the first key in the ordered hash object and output to the log */
   rc = iter.first();
   do while (rc = 0);
      put kenno '   ' name;
      rc = iter.next();
   end;
run;
```

The following lines are written to the SAS log:

```
NOTE: There were 7 observations read from the data set WORK.KENNEL.
01    Murphy
04    Jake
07    Tanner
09    Pepe
11    Jacques
12    Princess Z
15    Charlie
```

See Also

- "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

NEXT Method

Returns the next value in the underlying hash object.

Applies to: Hash iterator object

Syntax

rc=*object*.**NEXT**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash iterator object.

Details

Use the NEXT method iteratively to traverse the hash object and return the data items in key order.

The FIRST method returns the first data item in the hash object.

You can use the PREV method to return the previous data item in the hash object.

Note: The NEXT method sets the data variable to the value of the data item so that it is available for use after the method call.

Note: If you call the NEXT method without calling the FIRST method, then the NEXT method will still start at the first item in the hash object.

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "FIRST Method" on page 35
- "PREV Method" on page 53

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

NUM_ITEMS Attribute

Returns the number of items in the hash object.

Applies to: Hash object

Syntax

variable_name=*object*.**NUM_ITEMS**;

Arguments

variable_name
specifies the name of the variable that contains the number of items in the hash object.

object
specifies the name of the hash object.

Example: Returning the Number of Items in a Hash Object

This example creates a data set and loads the data set into a hash object. An item is added to the hash object and the total number of items in the resulting hash object is returned by the NUM_ITEMS attribute.

```
data work.stock;
   input item $ qty;
   datalines;
broccoli 345
corn 389
potato 993
onion 730
;
data _null_;
   if _N_ = 1 then do;
      length item $10;
      length qty 8;
      length totalitems 8;
      /* Declare hash object and read STOCK data set as ordered */
      declare hash myhash(dataset: "work.stock");
      /* Define key and data variables */
      myhash.defineKey('item');
      myhash.defineData('qty');
      myhash.defineDone();
   end;
   /* Add a key and data value to the hash object */
   item = 'celery';
   qty = 183;
   rc = myhash.add();
   if (rc ne 0) then
      put 'Add failed';
   /* Use NUM_ITEMS to return updated number of items in hash object */
   totalitems = myhash.num_items;
   put totalitems=;
run;
```

totalitems=5 is written to the SAS log.

OUTPUT Method

Creates one or more data sets, each of which contains the data in the hash object.

Applies to: Hash object

Syntax

rc=*object*.**OUTPUT** (DATASET: '*dataset-1* <(*datasetoption*)> '
<, …<DATASET: '*dataset-n*'> ('*datasetoption* <(*datasetoption*)> ');

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

DATASET: '*dataset*'
specifies the name of the output data set.

The name of the SAS data set can be a character literal or character variable. The data set name can also be enclosed in double quotation marks. When specifying the name of the output data set, you can use SAS data set options in the DATASET argument tag. Macro variables must be enclosed in double quotation marks.

datasetoption
specifies a data set option.

For more information about how to specify data set options, see "Syntax" in *SAS Data Set Options: Reference*.

Details

Hash object keys are not automatically stored as part of the output data set. The keys can be defined as data items to be included in the output data set by using the DEFINEDATA method. In addition, if no data items are defined using the DEFINEDATA method, the keys are written to the data set specified in the OUTPUT method.

If you use the **`ordered: 'yes'`** or **`ordered: 'ascending'`** argument tag in the DECLARE statement or the _NEW_ operator when you instantiate the hash object, then the data items are written to the data set in ascending key-value order. If you use the **`ordered: 'descending'`** argument tag in the DECLARE statement or the _NEW_ operator when you instantiate the hash object, then the data items are written to the data set in descending key-value order. If you do not use the **`ordered`** argument tag, the order is undefined.

When specifying the name of the output data set, you can use SAS data set options in the DATASET argument tag. Data set options specify actions that apply only to the SAS data set with which they appear and enable you to complete these actions:

- rename variables
- select a subset of observations based on the observation number for processing
- select observations using the WHERE option
- drop or keep variables from a data set loaded into a hash object, or for an output data set that is specified in an OUTPUT method call

  Note: The variables that are dropped or kept must have been included in the hash table by using the DEFINEDATA or DEFINEKEY method. Otherwise, an error occurs.

- specify a password for a data set

This example uses the WHERE data set option to select specific data for the output data set named OUT.

```
data x;
 do i = 1 to 20;
   output;
 end;
run;

/* Using the WHERE option. */
data _null_;
  length i 8;
  dcl hash h(dataset:'x');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out (where =( i < 8))');
run;
```

This example uses the RENAME data set option to rename the variable J to K for the output data set named OUT.

```
data x;
 do i = 1 to 20;
   output;
 end;
run;
/* Using the RENAME option. */
data _null_;
  length i j 8;
  dcl hash h(dataset:'x');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out (rename =(i=k))');
run;
```

For a list of data set options, see *SAS Data Set Options: Reference*.

Note: When you use the OUTPUT method to create a data set, the hash object is not part of the output data set. In this example, the H2 hash object is omitted from the output data set and a warning is written to the SAS log.

```
data _null_;
   length k 8;
   length d $10;
   declare hash h2();
   declare hash h(ordered: 'y');
   h.defineKey('k');
   h.defineData('k', 'd', 'h2');
   h.defineDone();
   k = 99;
   d = 'abc';
   h.add();
   k = 199;
   d = 'def';
   h.add();
   h.output(dataset:'work.x');
run;
```

Example

Using the data set ASTRO that contains astronomical data, the following code creates a hash object with the Messier (OBJ) objects sorted in ascending order by their right-ascension (RA) values and uses the OUTPUT method to save the data to a data set.

```
data astro;
input obj $1-4 ra $6-12 dec $14-19;
datalines;
 M31 00 42.7 +41 16
 M71 19 53.8 +18 47
 M51 13 29.9 +47 12
 M98 12 13.8 +14 54
 M13 16 41.7 +36 28
 M39 21 32.2 +48 26
 M81 09 55.6 +69 04
M100 12 22.9 +15 49
 M41 06 46.0 -20 44
 M44 08 40.1 +19 59
 M10 16 57.1 -04 06
 M57 18 53.6 +33 02
  M3 13 42.2 +28 23
 M22 18 36.4 -23 54
 M23 17 56.8 -19 01
 M49 12 29.8 +08 00
 M68 12 39.5 -26 45
 M17 18 20.8 -16 11
 M14 17 37.6 -03 15
 M29 20 23.9 +38 32
 M34 02 42.0 +42 47
 M82 09 55.8 +69 41
 M59 12 42.0 +11 39
 M74 01 36.7 +15 47
 M25 18 31.6 -19 15
;
run;
data _null_;
   if _N_ = 1 then do;
      length obj $10;
      length ra $10;
      length dec $10;
      /* Read ASTRO data set as ordered */
   declare hash h(hashexp: 4, dataset:"work.astro", ordered: 'yes');
      /* Define variables RA and OBJ as key and data for hash object */
      h.defineKey('ra');
      h.defineData('ra', 'obj');
      h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(ra, obj);
end;
/* Create output data set from hash object */
rc = h.output(dataset: 'work.out');
run;

proc print data=work.out;
   var ra obj;
   title 'Messier Objects Sorted by Right-Ascension Values';
```

```
run;
```

Output 2.2 *Messier Objects Sorted by Right-Ascension Values*

Messier Objects Sorted by Right-Ascension Values

Obs	ra	obj
1	00 42.7	M31
2	01 36.7	M74
3	02 42.0	M34
4	06 46.0	M41
5	08 40.1	M44
6	09 55.6	M81
7	09 55.8	M82
8	12 13.8	M98
9	12 22.9	M100
10	12 29.8	M49
11	12 39.5	M68
12	12 42.0	M59
13	13 29.9	M51
14	13 42.2	M3
15	16 41.7	M13
16	16 57.1	M10
17	17 37.6	M14
18	17 56.8	M23
19	18 20.8	M17
20	18 31.6	M25
21	18 36.4	M22
22	18 53.6	M57
23	19 53.8	M71
24	20 23.9	M29
25	21 32.2	M39

See Also

- "Saving Hash Object Data in a Data Set" in *SAS Language Reference: Concepts*

Methods:

- "DEFINEDATA Method" on page 21

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

PREV Method

Returns the previous value in the underlying hash object.

Applies to: Hash iterator object

Syntax

rc=*object*.**PREV**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash iterator object.

Details

Use the PREV method iteratively to traverse the hash object and return the data items in reverse key order.

The FIRST method returns the first data item in the hash object. The LAST method returns the last data item in the hash object.

You can use the NEXT method to return the next data item in the hash object.

Note: The PREV method sets the data variable to the value of the data item so that it is available for use after the method call.

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "FIRST Method" on page 35
- "LAST Method" on page 40
- "NEXT Method" on page 46

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

REF Method

Consolidates the CHECK and ADD methods into a single method call.

Applies to: Hash object

Syntax

rc=*object*.**REF** (<<KEY: *keyvalue-1*>, … <KEY: *keyvalue-n*>, <DATA: *datavalue-1*> , …<DATA: *datavalue-n*>>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: *keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

DATA: *datavalue*
specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call.

The number of "DATA: *datavalue*" pairs depends on the number of data variables that you define by using the DEFINEDATA method.

Details

You can consolidate CHECK and ADD methods into a single REF method. You can change the following code:

```
rc = h.check();
  if (rc ne 0) then
    rc = h.add();
```

to

```
rc = h.ref();
```

The REF method is useful for counting the number of occurrences of each key in a hash object. The REF method initializes the key summary for each key on the first ADD, and then changes the ADD for each subsequent CHECK.

For more information about key summaries, see "Maintaining Key Summaries" in *SAS Language Reference: Concepts*.

Example: Using the REF Method for Key Summaries

The following example uses the REF method for key summaries:

```
data keys;
 input key;
datalines;
1
2
1
3
5
2
3
2
4
1
5
1
;
data count;
 length count key 8;
 keep key count;
 if _n_ = 1 then do;
   declare hash myhash(suminc: "count", ordered: "y");
   declare hiter iter("myhash");
   myhash.defineKey('key');
   myhash.defineDone();
   count = 1;
  end;
 do while (not done);
   set keys end=done;
   rc = myhash.ref();
 end;
 rc = iter.first();
 do while(rc = 0);
   rc = myhash.sum(sum: count);
   output;
   rc = iter.next();
 end;
 stop;
run;

proc print data=count;
run;
```

Output 2.3 *Output of DATA Using the REF Method*

The SAS System

Obs	count	key
1	4	1
2	3	2
3	2	3
4	1	4
5	2	5

See Also

Methods:

- "ADD Method" on page 8
- "CHECK Method" on page 9

REMOVE Method

Removes the data that is associated with the specified key from the hash object.

Applies to: Hash object

Syntax

rc=*object*.**REMOVE** (<KEY: *keyvalue-1*, …KEY: *keyvalue-n*>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: *keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

Restriction If an associated hash iterator is pointing to the *keyvalue*, then the REMOVE method will not remove the key or data from the hash object. An error message is issued.

Details

The REMOVE method deletes both the key and the data from the hash object.

You can use the REMOVE method in one of two ways to remove the key and data in a hash object.

You can specify the key, and then use the REMOVE method as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   /* Specify the key */
   k = 'Joyce';
   /* Use the REMOVE method to remove the key and data */
   rc = h.remove();
   if (rc = 0) then
      put 'Key and data removed from the hash object.';
run;
```

Alternatively, you can use a shortcut and specify the key directly in the REMOVE method call as shown in the following code:

```
data _null_;
   length k $8;
   length d $12;
   if _N_ = 1 then do;
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
   end;
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   rc = h.add(key: 'Homer', data: 'Iliad');
   /* Specify the key in the REMOVE method parameter */
   rc = h.remove(key: 'Homer');
   if (rc  =0) then
      put 'Key and data removed from the hash object.';
run;
```

Note: The REMOVE method does not modify the value of data variables. It removes only the value in the hash object.

Note: If you specify **`multidata:'y'`** in the hash object constructor, the REMOVE method will remove all data items for the specified key.

Example: Removing a Key in the Hash Table

This example illustrates how to remove a key in the hash table.

```
/* Generate test data */
data x;
   do k = 65 to 70;
      d = byte (k);
      output;
   end;
run;
data _null_;
   length k 8 d $1;
   /* define the hash table and iterator */
   declare hash H (dataset:'x', ordered:'a');
   H.defineKey  ('k');
   H.defineData ('k', 'd');
   H.defineDone ();
   call missing (k,d);
   declare hiter HI ('H');
/*Use this logic to remove a key in the hash object when an*/
/*iterator is pointing to that key. The NEXT method will*/
/*start at the first item in the hash object if it is called*/
/*without calling the FIRST method. */
    do while (hi.next() = 0);
      if flag then rc=h.remove(key:key);
         if d = 'C' then do;
            key=k;
            flag=1;
         end;
         else flag=0;
      end;
  if flag then rc=h.remove(key:key);
  rc = h.output(dataset: 'work.out');
  stop;
run;
proc print;
run;
```

The following output shows that the key and data for the third object (key=67, data=C) is deleted.

Output 2.4 *Key and Data Removed from Output*

The SAS System

Obs	k	d
1	65	A
2	66	B
3	68	D
4	69	E
5	70	F

See Also

- "Replacing and Removing Data in the Hash Object" in *SAS Language Reference: Concepts*

Methods:

- "ADD Method" on page 8
- "DEFINEKEY Method" on page 24
- "REMOVEDUP Method" on page 59

REMOVEDUP Method

Removes the data that is associated with the specified key's current data item from the hash object.

Applies to: Hash object

Syntax

rc=*object*.**REMOVEDUP** (<KEY: *keyvalue-1*, …KEY: *keyvalue-n*>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: ***keyvalue***
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

Restriction If an associated hash iterator is pointing to the *keyvalue*, then the REMOVEDUP method does not remove the key or data from the hash object. An error message is issued.

Details

The REMOVEDUP method deletes both the key and the data from the hash object.

You can use the REMOVEDUP method in one of two ways to remove the key and data in a hash object. You can specify the key, and then use the REMOVEDUP method. Alternatively, you can use a shortcut and specify the key directly in the REMOVEDUP method call.

Note: The REMOVEDUP method does not modify the value of data variables. It removes only the value in the hash object.

Note: If only one data item is in the key's data item list, the key and data are removed from the hash object.

Comparisons

The REMOVEDUP method removes the data that is associated with the specified key's current data item from the hash object. The REMOVE method removes the data that is associated with the specified key from the hash object.

Example: Removing Duplicate Items in Keys

This example creates a hash object where several keys have multiple data items. The second data item in the key is removed.

```
data testdup;
   length key data 8;
   input key data;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
data _null_;
   length r 8;
   dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
   h.definekey('key');
   h.definedata('key', 'data');
   h.definedone();
   call missing (key, data);
   do key = 1 to 5;
      rc = h.find();
```

```
      if (rc = 0) then do;
         h.has_next(result: r);
         if (r ne 0) then do;
            h.find_next();
            h.removedup();
         end;
      end;
   end;
   dcl hiter i('h');
   rc = i.first();
   do while (rc = 0);
      put key= data=;
      rc = i.next();
   end;
run;
```

The following lines are written to the SAS log:

```
key=1 data=10
key=1 data=5
key=2 data=11
key=2 data=9
key=3 data=20
key=4 data=6
key=5 data=5
```

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "REMOVE Method" on page 56

REPLACE Method

Replaces the data that is associated with the specified key with new data.

Applies to: Hash object

Syntax

rc=*object*.**REPLACE** (<<KEY: *keyvalue-1*>, …<KEY: *keyvalue-n*>, <DATA: *datavalue-1*>,
…<DATA: *datavalue-n*>>);

Arguments

rc

specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: ***keyvalue***
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: **`keyvalue`**" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

Requirement The KEY:*keyvalue* arguments must be in the same order as they were defined in the hash object because the hash object variable names are not specified.

DATA: ***datavalue***
specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call.

The number of "DATA: **`datavalue`**" pairs depends on the number of data variables that you define by using the DEFINEDATA method.

Requirement The DATA:*datavalue* arguments must be in the same order as they were defined in the hash object because the hash object variable names are not specified.

Details

You can use the REPLACE method in one of two ways to replace data in a hash object.

You can define the key and data item, and then use the REPLACE method as shown in the following code. In this example, the data for the key 'Rottwlr' is changed from '1st' to '2nd'.

```
data work.show;
   length brd $10 plc $8;
   input brd plc;
datalines;
Terrier    2nd
LabRetr    3rd
Rottwlr    1st
Collie     bis
ChinsCrstd 2nd
Newfnlnd   3rd
;

proc print data=work.show;
   title 'SHOW Data Set Before Replace';
run;

data _null_;
   length brd $12;
   length plc $8;
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.show');
      rc = h.defineKey('brd');
      rc = h.defineData('brd', 'plc');
      rc = h.defineDone();
   end;
   /* Specify the key and new data value */
   brd = 'Rottwlr';
```

```
   plc = '2nd';
   /* Call the REPLACE method to replace the data value */
   rc = h.replace();
   /* Write the hash table to the data set. */
   rc = h.output(dataset: 'work.show');
run;

proc print data=work.show;
   title 'SHOW Data Set After Replace';
run;
```

Alternatively, you can use a shortcut and specify the key and data directly in the REPLACE method call as shown in the following code:

```
data work.show;
   length brd $10 plc $8;
   input brd plc;
datalines;
Terrier    2nd
LabRetr    3rd
Rottwlr    1st
Collie     bis
ChinsCrstd 2nd
Newfnlnd   3rd
;
data _null_;
   length brd $12;
   length plc $8;
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.show');
      rc = h.defineKey('brd');
      rc = h.defineData('brd', 'plc');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(brd, plc);
   end;
   /* Specify the key and new data value in the REPLACE method */
   rc = h.replace(key: 'Rottwlr', data: '2nd');
   /* Write the hash table to the data set. */
   rc = h.output(dataset: 'work.show');
run;
```

Note: The hash object's REPLACE method is intended for use with hash tables that have a single item for each key (**`MULTIDATA: 'NO'`**), whereas the REPLACEDUP method is intended for use with hash tables that have multiple data items for each key (**`MULTIDATA: 'YES'`**). In the SAS 9.4 release, if you call the REPLACE method and the hash object was declared using the **`multidata:'y'`** option, then all data items for the current key are replaced with the new data. In previous releases, no items are replaced and the new data is added to the current key.For more information about the MULTIDATA option, see "DECLARE Statement, Hash and Hash Iterator Objects" on page 13.

Note: If you call the REPLACE method and the key is not found, then the key and data are added to the hash object.

Note: The REPLACE method does not replace the value of the data variable with the value of the data item. It replaces only the value in the hash object.

Comparisons

The REPLACE method replaces the data that is associated with the specified key with new data. The REPLACEDUP method replaces the data that is associated with the current key's current data item with new data.

See Also

- "Replacing and Removing Data in the Hash Object" in *SAS Language Reference: Concepts*

Methods:

- "DEFINEDATA Method" on page 21
- "DEFINEKEY Method" on page 24
- "REPLACEDUP Method" on page 64

REPLACEDUP Method

Replaces the data that is associated with the current key's current data item with new data.

Applies to: Hash object

Syntax

rc=*object*.**REPLACEDUP** (<DATA: *datavalue-1*, …DATA: *datavalue-n*>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

DATA: ***datavalue***
specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call.

The number of "DATA: *datavalue*" pairs depends on the number of data variables that you define by using the DEFINEDATA method for the current key.

Details

You can use the REPLACEDUP method in one of two ways to replace data in a hash object.

You can define the data item, and then use the REPLACEDUP method. Alternatively, you can use a shortcut and specify the data directly in the REPLACEDUP method call.

Note: If you call the REPLACEDUP method and the key is not found, then the key and data are added to the hash object.

Note: The REPLACEDUP method does not replace the value of the data variable with the value of the data item. It replaces only the value in the hash object.

Comparisons

The REPLACEDUP method replaces the data that is associated with the current key's current data item with new data. The REPLACE method replaces the data that is associated with the specified key with new data.

Example: Replacing Data in the Current Key

This example creates a hash object where several keys have multiple data items. When a duplicate data item is found, 300 is added to the value of the data item.

```
data testdup;
   length key data 8;
   input key data;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
data _null_;
   length r 8;
   dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
   h.definekey('key');
   h.definedata('key', 'data');
   h.definedone();
   call missing (key, data);
   do key = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put key= data=;
         h.has_next(result: r);
         do while(r ne 0);
            rc = h.find_next();
            put 'dup ' key= data;
            data = data + 300;
            rc = h.replacedup();
            h.has_next(result: r);
         end;
      end;
   end;
   put 'iterating...';
   dcl hiter i('h');
   rc = i.first();
   do while (rc = 0);
      put key= data=;
```

```
      rc = i.next();
    end;
run;
```

The following lines are written to the SAS log.

```
key=1 data=10
dup key=1 15
dup key=1 5
key=2 data=11
dup key=2 16
dup key=2 9
key=3 data=20
dup key=3 100
key=4 data=6
key=5 data=5
dup key=5 99
iterating...
key=1 data=10
key=1 data=315
key=1 data=305
key=2 data=11
key=2 data=316
key=2 data=309
key=3 data=20
key=3 data=400
key=4 data=6
key=5 data=5
key=5 data=399
```

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "REPLACE Method" on page 61

RESET_DUP Method

Resets the pointer to the beginning of a duplicate list of keys when you use the DO_OVER method.

Applies to: Hash object

Syntax

rc=object.**RESET_DUP**( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

Details

When a hash object has multiple values for a single key, you can use the DO_OVER method in an iterative DO loop to traverse the duplicate keys. The DO_OVER method reads the key on the first method call and continues to traverse the duplicate key list until the key reaches the end.

If you switch the key in the middle of an iteration, you must use the RESET_DUP method to reset the pointer to the beginning of the list. Otherwise, SAS continues to use the first key.

For an example, see the DO_OVER method example on page 27.

See Also

Methods:

- "DO_OVER Method" on page 26

SETCUR Method

Specifies a starting key item for iteration.

Applies to: Hash iterator object

Syntax

rc=*object*.**SETCUR** (KEY: '*keyvalue-1*' <, …KEY: '*keyvalue-n*'>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash iterator object.

KEY: '*keyvalue*'
specifies a key value as the starting key for the iteration.

Details

The hash iterator enables you to start iteration on any item in the hash object. The SETCUR method sets the starting key for iteration. You use the KEY option to specify the starting item.

Example: Specifying the Starting Key Item

The following example creates a data set that contains astronomical data. You want to start iteration at RA= 18 31.6 instead of the first or last items. The data is loaded into a hash object and the SETCUR method is used to start the iteration. Because the *ordered* argument tag was set to YES, note that the output is sorted in ascending order.

```
data work.astro;
input obj $1-4 ra $6-12 dec $14-19;
datalines;
 M31 00 42.7 +41 16
 M71 19 53.8 +18 47
 M51 13 29.9 +47 12
 M98 12 13.8 +14 54
 M13 16 41.7 +36 28
 M39 21 32.2 +48 26
 M81 09 55.6 +69 04
M100 12 22.9 +15 49
 M41 06 46.0 -20 44
 M44 08 40.1 +19 59
 M10 16 57.1 -04 06
 M57 18 53.6 +33 02
  M3 13 42.2 +28 23
 M22 18 36.4 -23 54
 M23 17 56.8 -19 01
 M49 12 29.8 +08 00
 M68 12 39.5 -26 45
 M17 18 20.8 -16 11
 M14 17 37.6 -03 15
 M29 20 23.9 +38 32
 M34 02 42.0 +42 47
 M82 09 55.8 +69 41
 M59 12 42.0 +11 39
 M74 01 36.7 +15 47
 M25 18 31.6 -19 15
;
```

The following code sets the starting key for iteration to **'18 31.6'**:

```
data _null_;
length obj $10;
length ra $10;
length dec $10;
declare hash myhash(hashexp: 4, dataset:"work.astro", ordered:"yes");

declare hiter iter('myhash');
myhash.defineKey('ra');
myhash.defineData('obj', 'ra');
myhash.defineDone();
call missing (ra, obj, dec);
rc = iter.setcur(key: '18 31.6');
  do while (rc = 0);
    put obj= ra=;
    rc = iter.next();
  end;
run;
```

The following lines are written to the SAS log.

```
obj=M25 ra=18 31.6
  obj=M22 ra=18 36.4
  obj=M57 ra=18 53.6
  obj=M71 ra=19 53.8
  obj=M29 ra=20 23.9
  obj=M39 ra=21 32.2
```

You can use the FIRST method or the LAST method to start iteration on the first item or the last item, respectively.

See Also

- "Using the Hash Iterator Object " in *SAS Language Reference: Concepts*

Methods:

- "FIRST Method" on page 35
- "LAST Method" on page 40

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

SUM Method

Retrieves the summary value for a given key from the hash table and stores the value in a DATA step variable.

Applies to: Hash object

Syntax

rc=*object*.**SUM** (<KEY: *keyvalue-1*, …KEY: *keyvalue-n*,> SUM: *variable-name*);

Required Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: *keyvalue*
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of "KEY: *keyvalue*" pairs depends on the number of key variables that you define by using the DEFINEKEY method.

SUM: *variable-name*
specifies a DATA step variable that stores the current summary value of a given key.

Details

You use the SUM method to retrieve key summaries from the hash object. For more information, see "Maintaining Key Summaries" in *SAS Language Reference: Concepts*.

Comparisons

The SUM method retrieves the summary value for a given key when only one data item exists per key. The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key.

Example: Retrieving the Key Summary for a Given Key

The following example uses the SUM method to retrieve the key summary for each given key, K=99 and K=100.

```
k = 99;
count = 1;
h.add();
/* key=99 summary is now 1 */
k = 100;
h.add();
/* key=100 summary is now 1 */
k = 99;
h.find();
/* key=99 summary is now 2 */
count = 2;
h.find();
/* key=99 summary is now 4 */
k = 100;
h.find();
/* key=100 summary is now 3 */
h.sum(sum: total);
put 'total for key 100 = ' total;
k = 99;
h.sum(sum:total);
put 'total for key 99 = ' total;
run;
```

The first PUT statement prints the summary for k=100:

```
total for key 100 = 3
```

The second PUT statement prints the summary for k=99:

```
total for key 99 = 4
```

See Also

Methods:

- "ADD Method" on page 8
- "FIND Method" on page 30
- "CHECK Method" on page 9
- "DEFINEKEY Method" on page 24
- "REF Method" on page 54

- "SUMDUP Method" on page 71

Operators:

- "_NEW_ Operator, Hash and Hash Iterator Objects" on page 41

Statements:

- "DECLARE Statement, Hash and Hash Iterator Objects" on page 13

SUMDUP Method

Retrieves the summary value for the current data item of the current key and stores the value in a DATA step variable.

Applies to: Hash object

Syntax

rc=*object*.**SUMDUP** (SUM: *variable-name*);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a nonzero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message is printed to the log.

object
specifies the name of the hash object.

SUM: ***variable-name***
specifies a DATA step variable that stores the summary value for the current data item of the current key.

Details

You use the SUMDUP method to retrieve key summaries from the hash object when a key has multiple data items. For more information, see "Maintaining Key Summaries" in *SAS Language Reference: Concepts*.

Comparisons

The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key. The SUM method retrieves the summary value for a given key when only one data item exists per key.

Example: Retrieving a Summary Value

The following example uses the SUMDUP method to retrieve the summary value for the current data item. It also illustrates that it is possible to loop backward through the list by using the HAS_PREV and FIND_PREV methods. The FIND_PREV method works

similarly to the FIND_NEXT method with respect to the current list item except that it moves backward through the multiple item list.

```
data dup;
   length key data 8;
   input key data;
   cards;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
data _null_;
   length r i sum 8;
   i = 0;
   dcl hash h(dataset:'dup', multidata: 'y', suminc: 'i');
   h.definekey('key');
   h.definedata('key', 'data');
   h.definedone();
   call missing (key, data);
   i = 1;
  do key = 1 to 5;
     rc = h.find();
     if (rc = 0) then do;
        h.has_next(result: r);
        do while(r ne 0);
           rc = h.find_next();
           rc = h.find_prev();
           rc = h.find_next();
           h.has_next(result: r);
        end;
     end;
  end;
 i = 0;
 do key = 1 to 5;
   rc = h.find();
   if (rc = 0) then do;
       h.sum(sum: sum);
       put key= data= sum=;
       h.has_next(result: r);
       do while(r ne 0);
          rc = h.find_next();
          h.sumdup(sum: sum);
          put 'dup ' key= data= sum=;
          h.has_next(result: r);
       end;
     end;
 end;
 run;
```

The following lines are written to the SAS log.

```
key=1 data=10 sum=2
dup key=1 data=15 sum=3
dup key=1 data=5 sum=2
key=2 data=11 sum=2
dup key=2 data=16 sum=3
dup key=2 data=9 sum=2
key=3 data=20 sum=2
dup key=3 data=100 sum=2
key=4 data=6 sum=1
key=5 data=5 sum=2
dup key=5 data=99 sum=2
```

To see how this works, consider the key 1, which has three data values: 10, 15, and 5 (which are stored in that order).

```
key=1 data=10 sum=2
dup key=1 data=15 sum=3
dup key=1 data=5 sum=2
```

When traversing the data list in the first **`do key = 1 to 5;`** loop, the key summary for data item 10 is set to 1 on the initial FIND method call. The first FIND_NEXT method call sets the key summary for data item 15 to 1. The next FIND_PREV method call moves back to data item 10 and increments its key summary to 2. Finally, the last call to the FIND_NEXT method increments the key summary for data item 15 to 2. The next iteration through the loop sets the key summary for data item 5 to 1 and the key summary for data item 15 to 3. Finally, the key summary for data item 5 is incremented to 2.

You do not call the HAS_PREV method before calling the FIND_PREV method in this example because you already know that there is a previous entry in the list. Otherwise, you would not be in the loop.

Also shown here is the necessity of using special methods for some duplicate operations. (In this case, the SUMDUP method works similarly to the SUM method by retrieving the key summary for the current data item.)

See Also

- "Non-Unique Key and Data Pairs" in *SAS Language Reference: Concepts*

Methods:

- "SUM Method" on page 69

Chapter 3

Dictionary of Java Object Language Elements

Java Object Methods by Category

There are five categories of Java object methods.

Table 3.1 *Java Object Methods by Category*

Category	Description
Deletion	enables you to delete a Java object.
Exception	enables you to gather information about and clear an exception.
Field reference	enables you to return or set the value of static and non-static instance fields of the Java object.
Method reference	enables you to access static and non-static Java methods.
Output	enables you to send the Java output to its destination immediately.

The following table provides brief descriptions of the Java object methods. For more detailed descriptions, see the dictionary entry for each method.

Category	Language Elements	Description
Deletion	DELETE Method, Java Object (p. 83)	Deletes the Java object.
Exception	EXCEPTIONCHECK Method (p. 84)	Determines whether an exception occurred during a method call.
	EXCEPTIONCLEAR Method (p. 85)	Clears any exception that is currently being thrown.
	EXCEPTIONDESCRIBE Method (p. 87)	Turns the exception debug logging on or off and prints exception information.
Field Reference	GET*type*FIELD Method (p. 90)	Returns the value of a non-static field for a Java object.
	GETSTATIC*type*FIELD Method (p. 92)	Returns the value of a static field for a Java object.
	SET*type*FIELD Method (p. 95)	Modifies the value of a non-static field for a Java object.
	SETSTATIC*type*FIELD Method (p. 97)	Modifies the value of a static field for a Java object.
Method Reference	CALL*type*METHOD Method (p. 76)	Invokes an instance method on a Java object from a non-static Java method.
	CALLSTATIC*type*METHOD Method (p. 79)	Invokes an instance method on a Java object from a static Java method.
Output	FLUSHJAVAOUTPUT Method (p. 88)	Specifies that the Java output is sent to its destination.

Dictionary

CALL*type*METHOD Method

Invokes an instance method on a Java object from a non-static Java method.

Category: Method Reference

Applies to: Java object

Syntax

object.**CALL***type***METHOD** ("*method-name*", <*method-argument-1*, …*method-argument-n*>,
<*return-value*>);

Arguments

object
specifies the name of the Java object.

type
specifies the result type for the non-static Java method. The type can be one of the following values:

BOOLEAN
specifies that the result type is BOOLEAN.

BYTE
specifies that the result type is BYTE.

CHAR
specifies that the result type is CHAR.

DOUBLE
specifies that the result type is DOUBLE.

FLOAT
specifies that the result type is FLOAT.

INT
specifies that the result type is INT.

LONG
specifies that the result type is LONG.

SHORT
specifies that the result type is SHORT.

STRING
specifies that the result type is STRING.

VOID
specifies that the result type is VOID.

See "Type Issues" in *SAS Language Reference: Concepts*

method-name
specifies the name of the non-static Java method.

Requirement The method name must be enclosed in either single or double quotation marks.

method-argument
specifies the parameters to pass to the method.

return-value
specifies the return value if the method returns one.

Details

Once you instantiate a Java object, you can access any non-static Java method through method calls on the Java object by using the CALL*type*METHOD method.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts* .

Comparisons

Use the CALL*type*METHOD method for non-static Java methods. If the Java method is static, use the CALLSTATIC*type*METHOD method.

Example: Setting and Retrieving Field Values

The following example creates a simple class that contains three non-static fields. The Java object **j** is instantiated, and then the field values are set and retrieved using the CALL*type*FIELD method.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
   public int i;
   public double d;
   public string s;
   public int im()
   {
      return i;
   }
   public String sm()
   {
       return s;
   }
   public double dm()
   {
      return d;
   }
}
```

```
/*  DATA step code */
data _null_;
   dcl javaobj j("ttest");
   length val 8;
   length str $20;
    j.setIntField("i", 100);
   j.setDoubleField("d", 3.14159);
   j.setStringField("s", "abc");
   j.callIntMethod("im", val);
   put val=;
   j.callDoubleMethod("dm", val);
   put val=;
   j.callStringMethod("sm", str);
   put str=;
run;
```

The following lines are written to the SAS log:

```
val=100
val=3.14159
str=abc
```

See Also

Methods:

- "CALLSTATICtypeMETHOD Method" on page 79

CALLSTATIC*type*METHOD Method

Invokes an instance method on a Java object from a static Java method.

Category: Method Reference

Applies to: Java object

Syntax

object.**CALLSTATIC***type***METHOD** ("*method-name*", <*method-argument-1* , …*method-argument-n*>, <*return-value*>);

Arguments

object
specifies the name of the Java object.

type
specifies the result type for the static Java method. The type can be one of the following values:

BOOLEAN
specifies that the result type is BOOLEAN.

BYTE
specifies that the result type is BYTE.

CHAR
specifies that the result type is CHAR.

DOUBLE
specifies that the result type is DOUBLE.

FLOAT
specifies that the result type is FLOAT.

INT
specifies that the result type is INT.

LONG
specifies that the result type is LONG.

SHORT
specifies that the result type is SHORT.

STRING
specifies that the result type is STRING.

VOID
specifies that the result type is VOID.

See "Type Issues" in *SAS Language Reference: Concepts*

method-name
specifies the name of the static Java method.

Requirement The method name must be enclosed in either single or double quotation marks.

method-argument
specifies the parameters to pass to the method.

return-value
specifies the return value if the method returns one.

Details

Once you instantiate a Java object, you can access any static Java method through method calls on the Java object by using the CALLSTATIC*type*METHOD method.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts* .

Comparisons

Use the CALLSTATIC*type*METHOD method for static Java methods. If the Java method is not static, use the CALL*type*METHOD method.

Example: Setting and Retrieving Static Fields

The following example creates a simple class that contains three static fields. The Java object **j** is instantiated, and then the field values are set and retrieved using the CALLSTATIC*type*FIELD method.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttestc
{
   public static double d;
   public static double dm()
   {
      return d;
   }
```

```
/* DATA step code */
data x;
   declare javaobj j("ttestc");
   length d 8;
   j.SetStaticDoubleField("d", 3.14159);
   j.callStaticDoubleMethod("dm", d);
   put d=;
run;
```

The following line is written to the SAS log:

```
d=3.14159
```

See Also

Methods:

- "CALLtypeMETHOD Method" on page 76

DECLARE Statement, Java Object

Declares a Java object; creates an instance of and initializes data for a Java object.

Alias: DCL

Syntax

Form 1: **DECLARE JAVAOBJ** *object-reference*;

Form 2: **DECLARE JAVAOBJ** *object-reference* ("*java-class*", <*argument-1*, … *argument-n*>);

Arguments

object-reference
specifies the object reference name for the Java object.

java-class
specifies the name of the Java class to be instantiated.

Requirements The Java class name must be enclosed in either double or single quotation marks.

If you specify a Java package path, you must use forward slashes (/) and not periods (.) in the path. For example, an incorrect class name is *"java.util.Hashtable"*. The correct class name is *"java/util/Hashtable"*.

argument
specifies the information that is used to create an instance of the Java object. Valid values for *argument* depend on the Java object.

See "Using the DECLARE Statement to Instantiate a Java Object (Form 2)" on page 82

Details

The Basics

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing predefined component objects from within the DATA step.

For more information, see "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*.

Declaring a Java Object (Form 1)

You use the DECLARE statement to declare a Java object.

```
declare javaobj j;
```

The DECLARE statement tells SAS that the object reference J is a Java object.

After you declare the new Java object, use the _NEW_ operator to instantiate the object. For example, in the following line of code, the _NEW_ operator creates the Java object and assigns it to the object reference J:

```
j = _new_ javaobj("somejavaclass");
```

Using the DECLARE Statement to Instantiate a Java Object (Form 2)

Instead of the two-step process of using the DECLARE statement and the _NEW_ operator to declare and instantiate a Java object, you can use the DECLARE statement to declare and instantiate the Java object in one step. For example, in the following line of code, the DECLARE statement declares and instantiates a Java object and assigns the Java object to the object reference J:

```
declare javaobj j("somejavaclass");
```

The preceding line of code is equivalent to using the following code:

```
declare javaobj j;
j = _new_ javaobj("somejavaclass");
```

A *constructor* is a method that you can use to instantiate a component object and initialize the component object data. For example, in the following line of code, the DECLARE statement declares and instantiates a Java object and assigns the Java object to the object reference J. Note that the only required argument for a Java object constructor is the name of the Java class to be instantiated. All other arguments are constructor arguments for the Java class itself. In the following example, the Java class name, **`testjavaclass`**, is the constructor, and the values **`100`** and **`.8`** are constructor arguments.

```
declare javaobj j("testjavaclass", 100, .8);
```

Comparisons

You can use the DECLARE statement and the _NEW_ operator, or the DECLARE statement alone to declare and instantiate an instance of a Java object.

Examples

Example 1: Declaring and Instantiating a Java Object By Using the DECLARE Statement and the _NEW_ Operator

In the following example, a simple Java class is created. The DECLARE statement and the _NEW_ operator are used to create an instance of this class.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class simpleclass
{
   public int i;
   public double d;
}

/* DATA step code
data _null_;
   declare javaobj myjo;
```

```
   myjo = _new_ javaobj("simpleclass");
run;
```

Example 2: Using the DECLARE Statement to Create and Instantiate a Java Object

In the following example, a Java class is created for a hash table. The DECLARE statement is used to create and instantiate an instance of this class by specifying the capacity and load factor. In this example, a wrapper class, **mhash**, is necessary because the DATA step's only numeric type is equivalent to the Java type DOUBLE.

```
/* Java code */
import java.util.*;
public class mhash extends Hashtable;
{
   mhash (double size, double load)
      {
         super ((int)size, (float)load);
      }
}

/* DATA step code */
data _null_;
   declare javaobj h("mhash", 100, .8);
run;
```

See Also

- "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*

Operators:

- "_NEW_ Operator, Java Object" on page 94

DELETE Method, Java Object

Deletes the Java object.

Category: Deletion

Applies to: Java object

Syntax

object.**DELETE**( );

Arguments

object
specifies the name of the Java object.

Details

DATA step component objects are deleted automatically at the end of the DATA step. If you want to reuse the object reference variable in another Java object constructor, you should delete the Java object by using the DELETE method.

If you attempt to use a Java object after you delete it, you will receive an error in the log.

EXCEPTIONCHECK Method

Determines whether an exception occurred during a method call.

Category: Exception

Applies to: Java object

Syntax

object.**EXCEPTIONCHECK** (*status*);

Arguments

object
specifies the name of the Java object.

status
specifies the exception status that is returned.

Tip The status value that is returned by Java is of type DOUBLE, which corresponds to a SAS numeric data value.

Details

Java exceptions are handled through the EXCEPTIONCHECK, EXCEPTIONCLEAR, and EXCEPTIONDESCRIBE methods.

The EXCEPTIONCHECK method is used to determine whether an exception occurred during a method call. Ideally, the EXCEPTIONCHECK method should be called after every call to a Java method that can throw an exception.

Example: Checking an Exception

In the following example, the Java class contains a method that throws an exception. The DATA step calls the method and checks for an exception.

```
/* Java code */
public class a
   {
      public void m() throws NullPointerException
      {
         throw new NullPointerException();
      }
   }

/* DATA step code */
data _null_;
   length e 8;
   dcl javaobj j('a');
   rc = j.callvoidmethod('m');
   /* Check for exception.  Value is returned in variable 'e' */
   rc = j.exceptioncheck(e);
   if (e) then
```

```
    put 'exception';
  else
    put 'no exception';
  run;
```

The following line is written to the SAS log:

```
exception
```

See Also

Methods:

- "EXCEPTIONCLEAR Method" on page 85
- "EXCEPTIONDESCRIBE Method" on page 87

EXCEPTIONCLEAR Method

Clears any exception that is currently being thrown.

Category: Exception

Applies to: Java object

Syntax

object.**EXCEPTIONCLEAR**( );

Arguments

object
specifies the name of the Java object.

Details

Java exceptions are handled through the EXCEPTIONCHECK, EXCEPTIONCLEAR, and EXCEPTIONDESCRIBE methods.

If you call a method that throws an exception, it is strongly recommended that you check for an exception after the call. If an exception was thrown, you should take appropriate action and then clear the exception by using the EXCEPTIONCLEAR method.

If no exception is currently being thrown, this method has no effect.

Examples

Example 1: Checking and Clearing an Exception

In the following example, the Java class contains a method that throws an exception. The method is called in the DATA step, and the exception is cleared.

```
/* Java code */
public class a
   {
      public void m() throws NullPointerException
      {
```

```
            throw new NullPointerException();
         }
      }

/* DATA step code */
data _null_;
   length e 8;
   dcl javaobj j('a');
   rc = j.callvoidmethod('m');
   /* Check for exception.  Value is returned in variable 'e' */
   rc = j.exceptioncheck(e);
   if (e) then
     put 'exception';
   else
     put 'no exception';
   /* Clear the exception and check it again */
   rc = j.exceptionclear( );
   rc = j.exceptioncheck(e);
   if (e) then
      put 'exception';
   else
      put 'no exception';
run;
```

The following lines are written to the SAS log:

```
exception
no exception
```

Example 2: Checking for an Exception When Reading an External File

In this example, the Java IO classes are used to read an external file from the DATA step. The Java code creates a wrapper class for **`DataInputStream`**, which enables you to pass a **`FileInputStream`** to the constructor. The wrapper is necessary because the constructor actually takes an **`InputStream`**, which is the parent of **`FileInputStream`**, and the current method lookup is not robust enough to perform the superclass lookup.

```
/* Java code */
public class myDataInputStream extends java.io.DataInputStream
{
   myDataInputStream(java.io.FileInputStream fi)
   {
      super(fi);
   }
}
```

After you create the wrapper class, you can use it to create a **`DataInputStream`** for an external file and read the file until the end-of-file is reached. The EXCEPTIONCHECK method is used to determine when the **`readInt`** method throws an **`EOFException`**, which enables you to end the input loop.

```
/* DATA step code */
data _null_;
   length d e 8;
   dcl javaobj f("java/io/File", "c:\temp\binint.txt");
   dcl javaobj fi("java/io/FileInputStream", f);
   dcl javaobj di("myDataInputStream", fi);
```

```
   do while(1);
      di.callIntMethod("readInt", d);
      di.ExceptionCheck(e);
      if (e) then
         leave;
      else
         put d=;
   end;
run;
```

See Also

Methods:

- "EXCEPTIONCHECK Method" on page 84
- "EXCEPTIONDESCRIBE Method" on page 87

EXCEPTIONDESCRIBE Method

Turns the exception debug logging on or off and prints exception information.

Category: Exception

Applies to: Java object

Syntax

object.**EXCEPTIONDESCRIBE** (*status*);

Arguments

object
specifies the name of the Java object.

status
specifies whether exception debug logging is on or off. The **`status`** argument can be one of the following values:

0
specifies that debug logging is off.

1
specifies that debug logging is on.

Default 0 (off)

Tip The status value that is returned by Java is of type DOUBLE, which corresponds to a SAS numeric data value.

Details

The EXCEPTIONDESCRIBE method is used to turn exception debug logging on or off. If exception debug logging is on, exception information is printed to the JVM standard output.

Note: By default, JVM standard output is redirected to the SAS log.

Example: Printing Exception Information to Standard Output

In the following example, exception information is printed to the standard output.

```
/* Java code */
public class a
   {
      public void m() throws NullPointerException
      {
         throw new NullPointerException();
      }
   }

/* DATA step code */
data _null_;
   length e 8;
   dcl javaobj j('a');
   j.exceptiondescribe(1);
   rc = j.callvoidmethod('m');
run;
```

The following lines are written to the SAS log:

```
java.lang.NullPointerException
     at a.m(a.java:5)
```

See Also

Methods:

- "EXCEPTIONCHECK Method" on page 84
- "EXCEPTIONCLEAR Method" on page 85

FLUSHJAVAOUTPUT Method

Specifies that the Java output is sent to its destination.

Category: Output

Applies to: Java object

Syntax

object.**FLUSHJAVAOUTPUT**( );

Arguments

object
specifies the name of the Java object.

Details

Java output that is directed to the SAS log is flushed when the DATA step terminates. If you use the FLUSHJAVAOUTPUT method, the Java output will appear after any output that was issued while the DATA step was running.

Example: Displaying Java Output

In the following example, the “In Java class” lines are written after the DATA step is complete.

```
/* Java code */
 public class p
{
  void p()
     {
        System.out.println("In Java class");
     }
}
```

```
/* DATA step code */
data _null_;
   dcl javaobj j('p');
   do i = 1 to 3;
        j.callVoidMethod('p');
        put 'In DATA Step';
   end;
run;
```

The following lines are written to the SAS log:

```
In DATA Step
In DATA Step
In DATA Step
In Java class
In Java class
In Java class
```

If you use the FLUSHJAVAOUTPUT method, the Java output is written to the SAS log in the order of execution.

```
/* DATA step code */
data _null_;
   dcl javaobj j('p');
   do i = 1 to 3;
        j.callVoidMethod('p');
        j.flushJavaOutput();
        put 'In DATA Step';
   end;
run;
```

The following lines are written to the SAS log:

```
In Java class
In DATA Step
In Java class
In DATA Step
In Java class
In DATA Step
```

See Also

"Java Standard Output" in *SAS Language Reference: Concepts*

GET*type*FIELD Method

Returns the value of a non-static field for a Java object.

Category: Field Reference

Applies to: Java object

Syntax

object.**GET***type***FIELD** ("*field-name*", *value*);

Arguments

object
specifies the name of a Java object.

type
specifies the type for the Java field. The type can be one of the following values:

BOOLEAN
specifies that the field type is BOOLEAN.

BYTE
specifies that the field type is BYTE.

CHAR
specifies that the field type is CHAR.

DOUBLE
specifies that the field type is DOUBLE.

FLOAT
specifies that the field type is FLOAT.

INT
specifies that the field type is INT.

LONG
specifies that the field type is LONG.

SHORT
specifies that the field type is SHORT.

STRING
specifies that the field type is STRING.

See "Type Issues" in *SAS Language Reference: Concepts*

field-name
specifies the Java field name.

Requirement The field name must be enclosed in either single or double quotation marks.

value
specifies the name of the variable that receives the returned field value.

Details

Once you instantiate a Java object, you can access and modify its public fields through method calls on the Java object. The GET*type*FIELD method enables you to access non-static fields.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts*.

Comparisons

The GET*type*FIELD method returns the value of a non-static field for a Java object. To return the value of a static field, use the GETSTATIC*type*FIELD method.

Example: Retrieving the Value of a Non-Static Field

The following example creates a simple class that contains three non-static fields. The Java object `j` is instantiated, and then the field values are modified and retrieved using the GET*type*FIELD method.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
   public int i;
   public double d;
   public string s;
}
}
```

```
/*  DATA step code */
data _null_;
   dcl javaobj j("ttest");
   length val 8;
   length str $20;
   j.setIntField("i", 100);
   j.setDoubleField("d", 3.14159);
   j.setStringField("s", "abc");
   j.getIntField("i", val);
   put val=;
   j.getDoubleField("d", val);
   put val=;
   j.getStringField("s", str);
   put str=;
run;
```

The following lines are written to the SAS log:

```
val=100
val=3.14159
str=abc
```

See Also

Methods:

- "GETSTATICtypeFIELD Method" on page 92
- "SETtypeFIELD Method" on page 95

GETSTATIC*type*FIELD Method

Returns the value of a static field for a Java object.

Category: Field Reference

Applies to: Java object

Syntax

object.**GETSTATIC***type***FIELD** ("*field-name*", *value*);

Arguments

object
specifies the name of a Java object.

type
specifies the type for the Java field. The type can be one of the following values:

BOOLEAN
specifies that the field type is BOOLEAN.

BYTE
specifies that the field type is BYTE.

CHAR
specifies that the field type is CHAR.

DOUBLE
specifies that the field type is DOUBLE.

FLOAT
specifies that the field type is FLOAT.

INT
specifies that the field type is INT.

LONG
specifies that the field type is LONG.

SHORT
specifies that the field type is SHORT.

STRING
specifies that the field type is STRING.

See "Type Issues" in *SAS Language Reference: Concepts*

field-name
specifies the Java field name.

Requirement The field name must be enclosed in either single or double quotation marks.

value
specifies the name of the variable that receives the returned field value.

Details

Once you instantiate a Java object, you can access and modify its public fields through method calls on the Java object. The GETSTATIC*type*FIELD method enables you to access static fields.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts*.

Comparisons

The GETSTATIC*type*FIELD method returns the value of a static field for a Java object. To return the value of a non-static field, use the GET*type*FIELD method.

Example: Retrieving the Value of a Static Field

The following example creates a simple class that contains three static fields. The Java object **j** is instantiated, and then the field values are set and retrieved using the GETSTATIC*type*FIELD method.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
   public int i;
   public double d;
   public string s;
}
}
```

```
/*  DATA step code */
data _null_;
   dcl javaobj j("ttest");
   length val 8;
   length str $20;
   j.setStaticIntField("i", 100);
   j.setStaticDoubleField("d", 3.14159);
   j.setStaticStringField("s", "abc");
   j.getStaticIntField("i", val);
   put val=;
   j.getStaticDoubleField("d", val);
   put val=;
   j.getStaticStringField("s", str);
   put str=;
run;
```

The following lines are written to the SAS log:

```
val=100
val=3.14159
```

```
str=abc
```

See Also

Methods:

- "GETtypeFIELD Method" on page 90
- "SETSTATICtypeFIELD Method" on page 97

NEW Operator, Java Object

Creates an instance of a Java object.

Valid in: DATA step

Applies to: Java object

Syntax

object-reference = **_NEW_ JAVAOBJ** ("*java-class*", <*argument-1*, …*argument-n*>);

Arguments

object-reference
specifies the object reference name for the Java object.

java-class
specifies the name of the Java class to be instantiated.

Requirement The Java class name must be enclosed in either single or double quotation marks.

argument
specifies the information that is used to create an instance of the Java object. Valid values for *argument* depend on the Java object.

Details

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing the predefined component objects from within the DATA step.

If you use the _NEW_ operator to instantiate the Java object, you must first use the DECLARE statement to declare the Java object. For example, in the following lines of code, the DECLARE statement tells SAS that the object reference J is a Java object. The _NEW_ operator creates the Java object and assigns it to the object reference J.

```
declare javaobj j;
j = _new_ javaobj("somejavaclass" );
```

Note: You can use the DECLARE statement to declare and instantiate a Java object in one step.

A constructor is a method that is used to instantiate a component object and to initialize the component object data. For example, in the following lines of code, the _NEW_ operator instantiates a Java object and assigns it to the object reference J. Note that the

only required argument for a Java object constructor is the name of the Java class to be instantiated. All other arguments are constructor arguments for the Java class itself. In the following example, the Java class name, **testjavaclass**, is the constructor, and the values **100** and **.8** are constructor arguments.

```
declare javaobj j;
j = _new_ javaobj("testjavaclass", 100, .8);
```

For more information about the predefined DATA step component objects and constructors, see "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*.

Comparisons

You can use the DECLARE statement and the _NEW_ operator, or the DECLARE statement alone to declare and instantiate an instance of a Java object.

Example: Using the _NEW_ Operator to Instantiate and Initialize a Java Class

In the following example, a Java class is created for a hash table. The _NEW_ operator is used to create and instantiate an instance of this class by specifying the capacity and load factor. In this example, a wrapper class, **mhash**, is necessary because the DATA step's only numeric type is equivalent to the Java type DOUBLE.

```
/* Java code */
import java.util.*;
public class mhash extends Hashtable;
{
   mhash (double size, double load)
      {
         super ((int)size, (float)load);
      }
}

/* DATA step code */
data _null_;
   declare javaobj h;
   h = _new_ javaobj("mhash", 100, .8);
run;
```

See Also

- "Using DATA Step Component Objects" in *SAS Language Reference: Concepts*

Statements:

- "DECLARE Statement, Java Object" on page 81

SET*type*FIELD Method

Modifies the value of a non-static field for a Java object.

Category: Field Reference

Applies to: Java object

Syntax

object.**SET***type***FIELD** ("*field-name*", *value*);

Arguments

object
specifies the name of a Java object.

type
specifies the type for the Java field. The type can be one of the following values:

BOOLEAN
specifies that the field type is BOOLEAN.

BYTE
specifies that the field type is BYTE.

CHAR
specifies that the field type is CHAR.

DOUBLE
specifies that the field type is DOUBLE.

FLOAT
specifies that the field type is FLOAT.

INT
specifies that the field type is INT.

LONG
specifies that the field type is LONG.

SHORT
specifies that the field type is SHORT.

STRING
specifies that the field type is STRING.

See "Type Issues" in *SAS Language Reference: Concepts*

field-name
specifies the Java field name.

Requirement The field name must be enclosed in either single or double quotation marks.

value
specifies the value for the field.

Details

Once you instantiate a Java object, you can access and modify its public fields through method calls on the Java object. The SET*type*FIELD method enables you to modify non-static fields.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts*.

Comparisons

The SET*type*FIELD method modifies the value of a non-static field for a Java object. To modify the value of a static field, use the SETSTATIC*type*FIELD method.

Example: Creating a Java Class with Non-Static Fields

The following example creates a simple class that contains three non-static fields. The Java object **j** is instantiated, the field values are set using the SET*type*FIELD method, and then the field values are retrieved.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
   public int i;
   public double d;
   public string s;
}
}
```

```
/*  DATA step code */
data _null_;
   dcl javaobj j("ttest");
   length val 8;
   length str $20;
   j.setIntField("i", 100);
   j.setDoubleField("d", 3.14159);
   j.setStringField("s", "abc");
   j.getIntField("i", val);
   put val=;
   j.getDoubleField("d", val);
   put val=;
   j.getStringField("s", str);
   put str=;
run;
```

The following lines are written to the SAS log:

```
val=100
val=3.14159
str=abc
```

See Also

Methods:

- "GETtypeFIELD Method" on page 90
- "SETSTATICtypeFIELD Method" on page 97

SETSTATIC*type*FIELD Method

Modifies the value of a static field for a Java object.

Category: Field Reference

Applies to: Java object

Syntax

object.**SETSTATIC***type***FIELD** ("*field-name*", *value*);

Arguments

object
specifies the name of a Java object.

type
specifies the type for the Java field. The type can be one of the following values:

BOOLEAN
specifies that the field type is BOOLEAN.

BYTE
specifies that the field type is BYTE.

CHAR
specifies that the field type is CHAR.

DOUBLE
specifies that the field type is DOUBLE.

FLOAT
specifies that the field type is FLOAT.

INT
specifies that the field type is INT.

LONG
specifies that the field type is LONG.

SHORT
specifies that the field type is SHORT.

STRING
specifies that the field type is STRING.

See "Type Issues" in *SAS Language Reference: Concepts*

field-name
specifies the Java field name.

Requirement The field name must be enclosed in either single or double quotation marks.

value
specifies the value for the field.

Details

Once you instantiate a Java object, you can access and modify its public fields through method calls on the Java object. The SETSTATIC*type*FIELD method enables you to modify static fields.

Note: The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see "Type Issues" in *SAS Language Reference: Concepts*.

Comparisons

The SETSTATIC*type*FIELD method modifies the value of a static field for a Java object. To modify the value of a non-static field, use the SET*type*FIELD method.

Example: Creating a Java Class with Static Fields

The following example creates a simple class that contains three static fields. The Java object **j** is instantiated, the field values are set using the SETSTATIC*type*FIELD method, and then the field values are retrieved.

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttestc
{
   public static double d;
   public static double dm()
   {
      return d;
   }
}
```

```
/*  DATA step code */
data _null_;
   dcl javaobj j("ttest");
   length val 8;
   length str $20;
   j.setStaticIntField("i", 100);
   j.setStaticDoubleField("d", 3.14159);
   j.setStaticStringField("s", "abc");
   j.getStaticIntField("i", val);
   put val=;
   j.getStaticDoubleField("d", val);
   put val=;
   j.getStaticStringField("s", str);
   put str=;
run;
```

The following lines are written to the SAS log:

```
val=100
val=3.14159
str=abc
```

See Also

Methods:

- "GETSTATICtypeFIELD Method" on page 92
- "SETtypeFIELD Method" on page 95

Recommended Reading

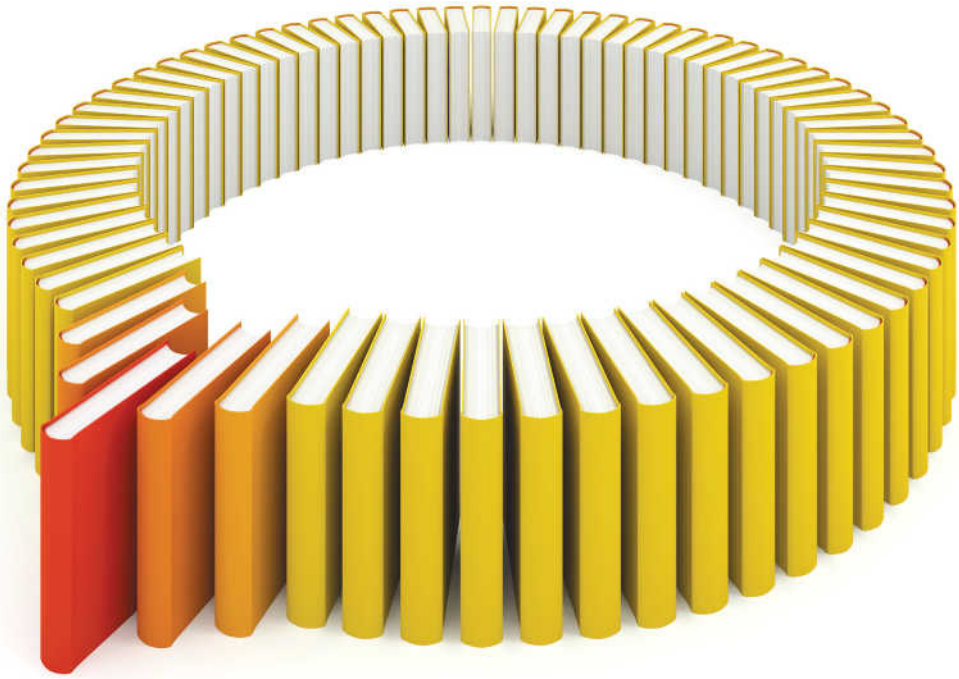
Here is the recommended reading list for this title:

- *SAS Data Set Options: Reference*
- *SAS Hash Object Programming Made Easy*
- *SAS Language Reference: Concepts*
- *SAS Logging: Configuration and Programming Reference*
- *SAS Statements: Reference*

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Index



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.