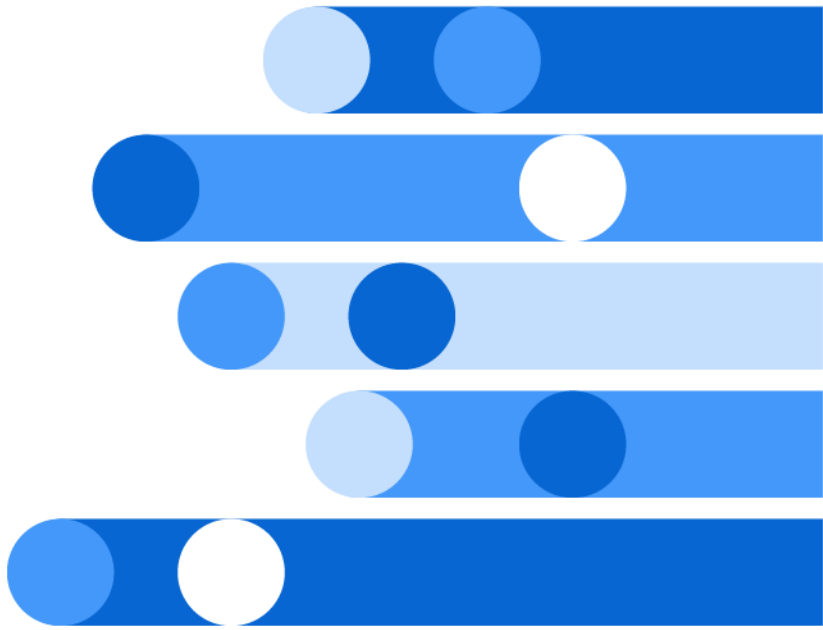
SAS® Component Objects: Reference

2020.1 - 2025.03*



* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

SAS® Documentation
August 19, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2020. *SAS® Component Objects: Reference*. Cary, NC: SAS Institute Inc.

SAS® Component Objects: Reference

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

March 2025

v_001-P1:lecompobjref

Contents

Syntax Conventions for the SAS Language

Syntax Conventions for the SAS Language

Overview of 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

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 <*options*> <*argument-2* <*options*> ...>
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-options*)>
<*value-range-set-1* <(*picture-1-options*)>
<*value-range-set-2* <(*picture-2-options*)> ...>>;

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

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

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

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';
```

PART 1

About Component Objects

1

About SAS Component Objects

Component Objects

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

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

hash and hash iterator objects
: enables you to quickly and efficiently store, search, and retrieve data based on lookup keys. For more information, see Chapter 2, "Using the Hash and Hash Iterator Objects"

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" .

logger and appender objects
: enables you to record logging events and write these events to the appropriate destination. For more information, see "Component Objects " in *SAS Viya Logging Facility for SAS Programs and Jobs*.

SAS provides these predefined component objects for use in PROC FCMP:

dictionary object
: Dictionary objects provide an alternative and broader method of storing data inside a PROC FCMP program by reference or value. For more information see "Using FCMP Dictionary Objects"

hash and hash iterator objects
: enables you to quickly and efficiently store, search, and retrieve data based on lookup keys. For more information see "Using the PROC FCMP Hash Object and PROC FCMP Hash Iterator Object"

Python objects
: enables you to submit and execute Python functions to a Python interpreter from SAS. For more information see "Using PROC FCMP Python Objects" on page 249

The Component Interface

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 object dot notation to access the component object's attributes and methods.

Note: The 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 and PROC FCMP objects.

Dot Notation and 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 PROC FCMP component objects, use only the DECLARE statement by itself. and SAS Viya Logging Facility for SAS Programs and Jobs.

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 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.

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 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.

```
data _null_;
   declare hash h1();
   declare hash h2;      /* h2 is declared, but not instantiated here */

   length key1 key2 $20;

   /* This sets up h1 to have a character key, with key (key1)
      and object data (h2) */
   h1.defineKey('key1');
   h1.defineData('key1', 'h2');
   h1.defineDone();

   /* Add a record to an instance of h2 with key = 'xyz' */
   key2 = 'xyz';
   h2 = _new_ hash();
   h2.defineKey('key2');
   h2.defineDone();
   h2.add();

   /* Add a record to h1 with key = 'abc', and data equal to 'abc'
      and the current h2 */
   key1 = 'abc';
   h1.add();

   /* Create an instance of h2. Add another record to h1 with a
      different key and another instance of h2 as data */
   key2 = 'pqr';
   h2 = _new_ hash();
   h2.definekey('key2');
   h2.definedone();
```

```
   h2.add();
   key1 = 'def';
   h1.add();

   /* Retrieve the object in h1 with key1 */
   key1 = 'abc';
   rc = h1.find();
   h2.output(dataset: 'work.h1');

   /* Retrieve another object stored in h1 */
   key1 = 'def';
   rc = h1.find();
   h2.output(dataset: 'work.h2');
run;

proc print data=work.h1; run;
proc print data=work.h2; run;
```

Here is the data set WORK.H1.

Obs	key2
1	xyz

Here is the data set WORK.H2.

Obs	key2
1	pqr

- You cannot store a component object in a hash object as a key. The following code generates an error:

```
declare hash h();
length k 8;
h.definekey('k');
h.definedone();

declare hash h2();
h2.definekey('h');   /* Error */
h2.definedone();
```

- You cannot use component objects with comparison operators other than the equal sign (=). If H1 and H2 are hash objects, the following code generates 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 generates 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 generates 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 generates 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 written by Java to the SAS log is reserved by SAS. If you need to write 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 "LOCKDOWN" in *SAS Programmer's Guide: Essentials*.
- Component objects are not supported for an in-database DATA step. For more information, see "DATA Step Acceleration" in *SAS In-Database Products: User's Guide*.

PART 2

DATA Step Component Objects

2

Using the Hash and Hash Iterator Objects

Using the Hash Object

Why Use the Hash Object?

The hash object provides an efficient and convenient in-memory mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys.

To use the DATA step Component Object Interface, follow these steps:

1 Declare the hash object.

2 Create an instance of (instantiate) the hash object.

3 Initialize lookup keys and data.

After you declare and instantiate a hash object, you can perform many tasks, including these:

- Store and retrieve data.
- Maintain key summaries.
- Replace and remove data.
- Compare hash objects.
- Generate a data set that contains the data in the hash object.

For example, suppose you have a large data set that contains numeric lab results corresponding to a unique patient number and weight. And suppose you have a small data set that contains patient numbers (a subset of those in the large data set). You can load the large data set into a hash object using the unique patient number as the key and the weight values as the data. A single pass is made over the small data set using the patient number to look up the current patient in the hash object whose weight is over a certain value and output that data to a different data set.

Depending on the number of lookup keys and the size of the data set, the hash object lookup can be significantly faster than a standard format lookup. If you are just looking up keys, you have a lot of memory, and you want fast performance, load the large data set first. If you do not want to use a lot of memory, load the small data set first.

Create a Hash Object

Create a hash object using the DECLARE statement. After you declare the new hash object, use the _NEW_ operator to instantiate the object. For example:

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

The DECLARE statement tells the compiler that the object reference `myhash` is of type hash. At this point, you have declared only the object reference `myhash`. It has the potential to hold a component object of type hash. Declare the hash object only once. The _NEW_ operator creates an instance of the hash object and assigns it to the object reference `myhash`.

There is an alternative to the two-step process of using the DECLARE statement and the _NEW_ operator to declare and instantiate a component object. You can use the DECLARE statement to declare and instantiate the hash object in one step.

```
declare hash myhash();
```

The above statement is equivalent to the following code:

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

For more information, see “DECLARE Statement: Hash and Hash Iterator Objects” on page 30 and the “_NEW_ Operator: Hash and Hash Iterator” on page 64.

Initialize Hash Object Data Using a Constructor

When you create a hash object, A constructor is a method that you can use to instantiate a hash object and initialize the hash object data.

The hash object constructor can have either of the following formats:

- declare hash *object_name*(*argument_tag-1*: *value-1*
  <, ...*argument_tag-n*: *value-n*>);
- *object_name* = _new_ hash(*argument_tag-1*: *value-1*
  <, ...*argument_tag-n*: *value-n*>);

For more information, see the "DECLARE Statement: Hash and Hash Iterator Objects" on page 30 and the "_NEW_ Operator: Hash and Hash Iterator" on page 64.

Define Keys and Data

The hash object uses lookup keys to store and retrieve data. The keys and the data are DATA step variables that you use to initialize the hash object by using dot notation method calls. A key is defined by passing the key variable name to the DEFINEKEY method. Data is defined by passing the data variable name to the DEFINEDATA method. After you have defined all key and data variables, the DEFINEDONE method is called. Keys and data can consist of any number of character or numeric DATA step variables.

For example, the following code initializes a character key and a character data variable:

```
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();
end;
```

You can have multiple key and data variables, but the entire key must be unique, unless you create the hash object with the MULTIDATA:"YES" argument tag. For more information, see "Duplicate Key Values and Data Pairs".

You can store more than one data item with a particular key. For example, you could modify the previous example to store auxiliary numeric values with the character key and data. In this example, each key and each data item consists of a character value and a numeric value:

```
length d1 8;
length d2 $20;
```

```
length k1 $20;
length k2 8;

if _N_ = 1 then do;
   declare hash h();
   rc = h.defineKey('k1', 'k2');
   rc = h.defineData('d1', 'd2');
   rc = h.defineDone();
end;
```

For more information, see the "DEFINEDATA Method" on page 40, "DEFINEDONE Method" on page 42, and the "DEFINEKEY Method" on page 43.

Note: The hash object does not assign values to key variables (for example, **`h.find(key:'abc')`**), and the SAS compiler cannot detect the 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, SAS issues 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;
```

Duplicate Key Values and Data Pairs

By default, one set of data variables exists for each key. However, some situations might warrant duplicate keys in the hash object (that is, associate more than one set of data variables with a key).

For example, assume that the key is a patient ID and the data is a visit date. If the patient were to visit multiple times, multiple visit dates would be associated with the patient ID. When you create a hash object with the MULTIDATA:"YES" argument tag, multiple sets of the data variables are associated with the key.

If the data set contains duplicate keys, by default, the first instance is stored in the hash object and subsequent instances are ignored. To store the last instance in the

hash object, use the DUPLICATE argument tag. The DUPLICATE argument tag also writes an error to the SAS log if there is a duplicate key.

However, the hash object allows storage of multiple values for each key if you use the MULTIDATA argument tag in the DECLARE statement or _NEW_ operator. The hash object keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

To traverse a multiple data item list, you must know the current list item. Start by calling the FIND method for a given key. The FIND method sets the current list item. Then to determine whether the key has multiple data values, call the HAS_NEXT method. After you have determined that the key has another data value, you can retrieve that value with the FIND_NEXT method. The FIND_NEXT method sets the current list item to the next item in the list and sets the corresponding data variable or variables for that item.

In addition to moving forward through the list for a given key, you can loop backward through the list by using the HAS_PREV and FIND_PREV methods in a similar manner.

When you have a hash object that has multiple values for a single key, you can 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.

Note: The items in a multiple data item list are maintained in the order in which you insert them.

For more information about these and other methods associated with non-unique key and data pairs, see "Dictionary of Hash and Hash Iterator Object Language Elements" on page 23.

Store and Retrieve Data

How to Store and Retrieve Data

After you initialize the hash object's key and data variables, you can store data in the hash object using the ADD method, or you can use the *dataset* argument tag to load a data set into the hash object. If you use the *dataset* argument tag, and if the data set contains more than one observation with the same value of the key, by default, SAS keeps the first observation in the hash table and ignores subsequent observations. To store the last instance in the hash object or to send an error to the log if there is a duplicate key, use the DUPLICATE argument tag. To allow duplicate values for each key, use the MULTIDATA argument tag.

You can then use the FIND method to search and retrieve data from the hash object if one data value exists for each key. Use the FIND_NEXT and FIND_PREV methods to search and retrieve data if multiple data items exist for each key.

For more information, see “ADD Method” on page 24, “FIND Method” on page 50, “FIND_NEXT Method” on page 53, and the “FIND_PREV Method” on page 55.

You can consolidate a FIND method and ADD method using the REF method. In the following example, you can reduce the amount of code from this:

```
rc = h.find();
  if (rc != 0) then
    rc = h.add();
```

to a single method call:

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

For more information, see the “Using the Hash Iterator Object ” on page 19.

Note: You can also use the hash iterator object to retrieve the hash object data, one data item at a time, in forward and reverse order. For more information, see “How to Store and Retrieve Data”.

Maintain Key Summaries

You can maintain a summary count for a hash object key by using the SUMINC argument tag when you declare the hash object. The tag value is a string expression that resolves to the name of a numeric DATA step variable: the SUMINC variable.

This SUMINC tag instructs the hash object to allocate internal storage for maintaining a summary value for each key.

The summary value of a hash key is initialized to the value of the SUMINC variable whenever the ADD or REPLACE method is used.

The summary value of a hash key is incremented by the value of the SUMINC variable whenever the FIND, CHECK, or REF method is used.

Note that the SUMINC variable can be negative, positive, or zero valued. The variable does not need to be an integer. The SUMINC value for a key is zero by default.

In the following example, the initial ADD method sets the summary count for K=99 to 1 before the ADD. Then each time a new COUNT value is given, the following FIND method adds the value to the key summary. In this example, one data value exists for each key. The SUM method retrieves the current value of the key summary and the value is stored in the DATA step variable TOTAL. If multiple items exist for each key, the SUMDUP method retrieves the current value of the key summary.

```
data _null_;
 length k count 8;
 length total 8;
 dcl hash myhash(suminc: 'count');
 myhash.defineKey('k');
 myhash.defineDone();
```

```
 k = 99;
 count = 1;
 myhash.add();


/* COUNT is given the value 2.5 and the */
/* FIND sets the summary to 3.5*/
 count = 2.5;
 myhash.find();

/* The COUNT of 3 is added to the FIND and */
/* sets the summary to 6.5. */
 count = 3;
 myhash.find();


/* The COUNT of -1 sets the summary to 5.5. */
 count = -1;
 myhash.find();

/* The SUM method gives the current value of */
/* the key summary to the variable TOTAL. */
 myhash.sum(sum: total);

/* The PUT statement prints total=5.5 in the log. */
 put total=;
 run;
```

In this example, a summary is maintained for each key value K=99 and K=100:

```
 k = 99;
 count = 1;
 myhash.add();
 /* key=99 summary is now 1 */

 k = 100;
 myhash.add();
 /* key=100 summary is now 1 */

 k = 99;
 myhash.find();
 /* key=99 summary is now 2 */

 count = 2;
 myhash.find();
 /* key=99 summary is now 4 */

 k = 100;
 myhash.find();
 /* key=100 summary is now 3 */

 myhash.sum(sum: total);
 put 'total for key 100 = 'total;

 k = 99;

myhash.sum(sum:total);
```

```
 put 'total for key 99 = ' total;
```

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

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

And the second PUT statement prints the summary for K=99:

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

You can use key summaries in conjunction with the *dataset* argument tag. As the data set is read into the hash object using the DEFINEDONE method, all key summaries are set to the SUMINC value. And, all subsequent FIND, CHECK, or ADD methods change the corresponding key summaries.

```
declare hash myhash(suminc: "keycount", dataset: "work.mydata");
```

Use Hash Object Attributes

You can use the DATA Step Component Interface to retrieve information from a hash object using an attribute. Use the following syntax for an attribute:

```
attribute_value=obj.attribute_name;
```

There are two attributes available to use with hash objects. NUM_ITEMS returns the number of items in a hash object and ITEM_SIZE returns the size (in bytes) of an item. The following example retrieves the number of items in a hash object:

```
n = myhash.num_items;
```

The following example retrieves the size of an item in a hash object:

```
s = myhash.item_size;
```

You can obtain an idea 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, the use of ITEM_SIZE does not provide exact memory usage, but it gives a good approximation.

For more information, see the "NUM_ITEMS Attribute" on page 71 and the "ITEM_SIZE Attribute" on page 61.

Using the Hash Iterator Object

About the Hash Iterator Object

Use the hash iterator object to store and search data based on lookup keys. The hash iterator object enables you to retrieve the hash object data in either forward or reverse key order.

Declaring and Instantiating a Hash Iterator Object

You declare a hash iterator object by using the DECLARE statement. After you declare the new hash iterator object, use the _NEW_ operator to instantiate the object. Use the hash object name as an argument tag. For example:

```
declare hiter myiter;
myiter = _new_ hiter('h');
```

The DECLARE statement tells the compiler that the object reference MyIter is of type hash iterator. At this point, you have declared only the object reference MyIter. It has the potential to hold a component object of type hash iterator. You should declare the hash iterator object only once. The _NEW_ operator creates an instance of the hash iterator object and assigns it to the object reference MyIter. The hash object, H, is passed as a constructor argument. The hash object, not the hash object variable, is specifically assigned to the hash iterator.

As an alternative to the two-step process of using the DECLARE statement and the _NEW_ operator to declare and instantiate a component object, you can declare and instantiate a hash iterator object in one step by using the DECLARE statement as a constructor method. The syntax is as follows:

```
declare hiter object_name(hash_object_name);
```

In the above example, the hash object name must be enclosed in single or double quotation marks.

For example:

```
declare hiter myiter('h');
```

The previous statement is equivalent to these:

```
declare hiter myiter;
myiter = _new_ hiter('h');
```

Note: You must declare and instantiate a hash object before you create a hash iterator object. For more information, see "Create a Hash Object".

For example:

```
if _N_ = 1 then do;
   length key $10;
   declare hash myhash(dataset:"work.x", ordered: 'yes');
   declare hiter myiter('myhash');
   myhash.defineKey('key');
   myhash.defineDone();
end;
```

This code creates an instance of a hash iterator object with the variable name MyIter. The hash object, MyHash, is passed as the constructor argument. Because the hash object was created with the ORDERED argument tag set to **'yes'**, the data is returned in ascending key-value order.

For more information about the DECLARE statement and the _NEW_ operator, see the *SAS DATA Step Statements: Reference*.

Example: Retrieving Hash Object Data By Using the Hash Iterator

Using the data set ASTRO that contains astronomical data, the following code creates the data set that contains Messier objects (OBJ) whose right-ascension (RA) values are greater than 12. The FIRST and NEXT methods are used to retrieve the data in ascending order. For more information about the FIRST and NEXT methods, see *SAS Component Objects: Reference*.

```
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 out;
   if _N_ = 1 then do;
      length obj $10;
      length ra $10;
      length dec $10;
      /* Read ASTRO data set and store in asc order in hash obj */
      declare hash h(dataset:"work.astro", ordered: 'yes');
      /* Define variables RA and OBJ as key and data for hash object */
      declare hiter iter('h');
      h.defineKey('ra');
      h.defineData('ra', 'obj');
      h.defineDone();
      /* Avoid uninitialized variable notes */
      call missing(obj, ra, dec);
   end;
/* Retrieve RA values in ascending order */
rc = iter.first();
do while (rc = 0);
/* Find hash object keys greater than 12 and output data */
   if ra GE '12' then
      output;
   rc = iter.next();
end;
run;

proc print data=work.out;
   var ra obj;
   title 'Messier Objects Greater than 12 Sorted by Right Ascension
Values';
run;
```

The following output shows the report that PROC PRINT generates.

Messier Objects Greater than 12 Sorted by Right Ascension Values

Obs	ra	obj
1	12 13.8	M98
2	12 22.9	M100
3	12 29.8	M49
4	12 39.5	M68
5	12 42.0	M59
6	13 29.9	M51
7	13 42.2	M3
8	16 41.7	M13
9	16 57.1	M10
10	17 37.6	M14
11	17 56.8	M23
12	18 20.8	M17
13	18 31.6	M25
14	18 36.4	M22
15	18 53.6	M57
16	19 53.8	M71
17	20 23.9	M29
18	21 32.2	M39

3

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.

Restriction The *keyvalue* cannot contain a component 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.

Restriction The *datavalue* cannot contain a component object.

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;
   rc = h.output(dataset:'beforeadd');
   /* Define constant key and data values */
   k = 'Joyce';
   d = 'Ulysses';
   /* Add key and data values to hash object */
   rc = h.add();
   rc = h.output(dataset:'afteradd');
run;

proc print data = beforedadd; run;
proc print data = afteradd; run;
```

This error is written to the log when an attempt is made to display the BEFOREADD data set before we add data to the hash:

```
ERROR: File WORK.BEFOREDADD.DATA does not exist.
```

After the key and data values are defined, the data within the hash can be displayed.

Obs	d
1	Ulysses

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;
   rc = h.output(dataset:'beforeadd');
   /* Define constant key and data values and add to hash object */
   rc = h.add(key: 'Joyce', data: 'Ulysses');
   rc = h.output(dataset:'afteradd');
run;

proc print data = beforedadd; run;
proc print data = afteradd ; 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. Because no assignment to a key or data variable appears in this program, SAS issues a note stating that the variables are uninitialized. In order to avoid receiving these notes, use the CALL MISSING routine to specify the key and data variables as parameters.

See Also

- "Store and Retrieve Data"

Methods:

- "DEFINEDATA Method" on page 40
- "DEFINEKEY Method" on page 43
- "OUTPUT Method" on page 72
- "REF Method" on page 78

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 43
- “FIND Method” on page 50

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 45

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 "Store and Retrieve Data"

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 "Store and Retrieve Data"

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 seven valid hash object argument and value tags:

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

Note: Hash objects do not support the VALIDMEMNAME=EXTEND system option for data set names. Data set names cannot contain special characters or national characters.

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. A literal data set name must be enclosed in single or double quotation marks. A character variable that contains the name of the data set should not be enclosed in quotation marks. Macro variables must be enclosed in double quotation marks.

IMPORTANT SAS data set options are not supported when the hash object is processed in CAS.

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 and enable you to perform these operations:

- 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
- specify 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 *SAS Data Set Options: Reference*

Note If the data set contains duplicate keys, the default is to load the first instance in the hash object; subsequent instances are ignored. Use the DUPLICATE argument tag to store the last instance in the hash object or write an error message to the SAS log. Use the MULTIDATA argument tag to allow multiple data items for a key.

duplicate: '*option*'

determines how to handle duplicate keys if they are not allowed. The default is to store the first key and ignore subsequent duplicates. Option can be one of these 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 uses the REPLACE option and stores **`blue`** for the *color_id* key 531 and **`brown`** for the *color_id* key 620. If you use the default, the first values (**`yellow`** for 531 and **`green`** for 620) are loaded and the duplicate values are ignored.

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

data _null_;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
    declare hash myhash(dataset: "table", duplicate: "r");
    rc = myhash.definekey('color_id');
    rc = myhash.definedata('color_id','color_name');
    myhash.definedone();
    /* avoid uninitialized variable notes */
    call missing(color_id, color_name);
 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 39

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 "Duplicate Key Values and Data Pairs"

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.

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 "Maintain Key Summaries"

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

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

See "Initialize Hash Object Data Using a Constructor" "Declaring and Instantiating a Hash Iterator Object"

Details

The Basics

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. You can access the predefined hash and hash iterator component objects through the DATA step component interface.

For more information about the predefined DATA step component objects, see Chapter 2, "Using the Hash and Hash Iterator Objects"

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 this 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

IMPORTANT SAS data set options are not supported when the hash object is processed in CAS.

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 and enable you to perform these operations:

- 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
- specify 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 38. 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:

IMPORTANT SAS data set options are not supported when the hash object is processed in CAS.

```
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: 'example1');
run;
proc print data=example1; 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: 'example2 (where =( i < 8))');
run;
proc print data=example2; 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: 'example3');
run;
proc print data=example3; 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: 'example4 (rename =(j=k))');
```

```
run;
proc print data=example4; 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, *key_sum*, to hold the key summary for each key and adds the variable to the output data set.

```
data numbers;
   input key_id amount;
   datalines;
      1 10
      2 11
      3 20
      5 5
      4 6
run;

data _null_;
   dcl hash h(dataset:'numbers', suminc: 'i', keysum: 'key_sum');
   h.definekey('key_id');
   h.definedata('key_id', 'amount');
   h.definedone();
   /* avoid uninitialized variable notes */
   call missing(key_id, amount, key_sum);
   i = 1;
   do key_id = 1 to 5;
      rc = h.find();
   end;
   do key_id = 1 to 3;
      rc = h.find();
   end;
   rc = h.output(dataset:'out');
run;
proc print data=out; run;
```

Output 3.1 *Output of Key Summary Data*

Obs	key_id	amount	key_sum
1	2	11	2
2	5	5	1
3	1	10	2
4	3	20	2
5	4	6	1

See Also

- "Component Objects"

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

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 "Define Keys and Data".

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

- "Define Keys and Data"

Methods:

- "DEFINEDONE Method" on page 42
- "DEFINEKEY Method" on page 43

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

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 "Define Keys and Data".

See Also

- "Define Keys and Data"

Methods:

- "DEFINEDATA Method" on page 40
- "DEFINEKEY Method" on page 43

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 "Define Keys and Data".

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

- "Define Keys and Data"

Methods:

- "DEFINEDATA Method" on page 40
- "DEFINEDONE Method" on page 42

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

DELETE Method: Hash and Hash Iterator Objects

Deletes the hash or hash iterator object.

Applies to: Hash object, Hash iterator 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 name in another hash or hash iterator object or if you want to release the memory used by that object, 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 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 28.

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;
   input key_id value;
   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_;
   dcl hash h(dataset:'dup', multidata: 'y', ordered: 'y');
   h.definekey('key_id');
   h.definedata('key_id', 'value');
   h.definedone();
   call missing(key_id, value);

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

   key_id = 3;
   do while(h.do_over(key:key_id) eq 0);
      put key_id= value=;
   end;
```

```
   key_id = 2;
   do while(h.do_over(key:key_id) eq 0);
     put key_id= value=;
   end;
run;
```

The following lines are written to the SAS log:

```
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
key_id=3 value=20
key_id=3 value=100
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
```

See Also

Methods:

- "RESET_DUP Method" on page 92

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 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:

```
data compare;
    declare hash h1();
    h1.defineKey('k');
    h1.defineDone();

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

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

The following line is written to the SAS log:

```
hash objects are equal
```

The two hash objects are defined as equal when all of these 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 H1 and H2 hash objects are declared and instantiated. A record with a key value of 99 is added to both objects. The first call to EQUALS returns a nonzero result value indicating that the hash objects are equivalent.

The key in H2 is then replaced with a value of 100. A second call to EQUALS compares the H1 and H2 hash objects and returns a zero result value indicating that the hash objects are no longer equivalent.

```
data _null_;
   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=;

put rc=;
   k = 100;
   h2.replace();

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

put rc=;
 run;
```

The following lines are written to the SAS log:

```
eq=1
eq=0
```

FIND Method

Determines whether the specified key is stored in the hash object and if found, updates the data variables in the data set.

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 this code:

```
data _null_;
   length k $8  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 this code:

```
data _null_;
   length k $8  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 53 and "FIND_PREV Method" on page 55 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 stored 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 stored 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');
      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');
   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

- "Using the Hash Object"

Methods:

- "CHECK Method" on page 26
- "DEFINEKEY Method" on page 43
- "FIND_NEXT Method" on page 53
- "FIND_PREV Method" on page 55
- "REF Method" on page 78

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_id value 8;
   input key_id value;
   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_id');
   h.definedata('key_id', 'value');
   h.definedone();
   /* avoid uninitialized variable notes */
   call missing (key_id, value);
   do key_id = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put @5 key_id= @15 value=;
         rc = h.find_next();
         do while(rc = 0);
            put 'dup ' @5 key_id= @15 value=;
            rc = h.find_next();
         end;
      end;
   end;
 run;
```

The following lines are written to the SAS log:

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

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method" on page 50
- "FIND_PREV Method" on page 55
- "HAS_NEXT Method" on page 58

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 58 for an example.

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method" on page 50
- "FIND_NEXT Method" on page 53
- "HAS_PREV Method" on page 60

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, then an error message indicating that the key is not found is written to the log. The hash iterator traverses the hash table using the keys.

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 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 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_;
      length prod $10 qty $6;
   /* Declare hash object and read SALES data set as ordered */
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.sales', ordered: 'yes');
      declare hiter iter('h');
      /* Define key and data variables */
      h.defineKey('qty');
      h.defineData('prod', 'qty');
      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=  qty=;
      rc = iter.next();
   end;
run;
```

The following lines are written to the SAS log:

```
prod=orange qty=329559
prod=apple qty=384223
prod=banana qty=398487
```

See Also

- Chapter 2, “Using the Hash and Hash Iterator Objects”

Methods:

- “LAST Method” on page 63

Operators:

- “_NEW_ Operator: Hash and Hash Iterator” on page 64

Statements:

- “DECLARE Statement: Hash and Hash Iterator Objects” on page 30

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 returns a nonzero value in the numeric variable **R**. Otherwise, it returns 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 billing;
   length customer_num invoice_amount  8;
   input customer_num invoice_amount;
   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:'billing', multidata: 'y');
   h.definekey('customer_num');
   h.definedata('customer_num', 'invoice_amount');
   h.definedone();
   call missing (customer_num, invoice_amount);
   do customer_num = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put @5 customer_num= @20 invoice_amount=;
         h.has_next(result: r);
         do while(r ne 0);
             rc = h.find_next();
             put 'dup ' @5 customer_num= @20 invoice_amount=;
             h.has_next(result: r);
```

```
        end;
      end;
   end;
run;
```

The following lines are written to the SAS log:

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

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method" on page 50
- "FIND_NEXT Method" on page 53
- "FIND_PREV Method" on page 55
- "HAS_PREV Method" on page 60

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 58 for an example.

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method" on page 50
- "FIND_NEXT Method" on page 53
- "FIND_PREV Method" on page 55
- "HAS_NEXT Method" on page 58

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 error message indicating that the key is not found is written to the log. The hash iterator traverses the hash table using the keys.

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

- Chapter 2, "Using the Hash and Hash Iterator Objects"

Methods:

- "FIRST Method" on page 56

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

NEW Operator: Hash and Hash Iterator

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 these objects:

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 Chapter 2, “Using the Hash and Hash Iterator Objects”

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 and enable you to perform these operations:

- 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 specified in an OUTPUT method call
- specify 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 *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 **`blue`** for the key 531 and **`brown`** for the key 620. If you use the default, **`yellow`** would be stored for 531 and **`green`** would be stored for 620.

```
data table;
  input color_id color_name $;
  datalines;
  531 yellow
  620 green
  531 blue
  908 orange
  620 brown
  143 purple
 run;
data _null_;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
    declare hash myhash;
    myhash = _new_ hash (dataset: "table", duplicate: "r");
    rc = myhash.definekey('color_id');
    rc = myhash.definedata('color_id', 'color_name');
    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 "Duplicate Key Values and Data Pairs"

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 "Maintain Key Summaries".

See "Initialize Hash Object Data Using a Constructor" and "Declaring and Instantiating a Hash Iterator 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 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 "Initialize Hash Object Data Using a Constructor".

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

- "Component Objects"

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

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 error message indicating that the key is not found is written to the log. The hash iterator traverses the hash table using the keys.

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 Object"

Methods:

- "FIRST Method" on page 56
- "PREV Method" on page 77

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

■ “DECLARE Statement: Hash and Hash Iterator Objects” on page 30

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-1 datasetoption-2*...)> ' , ...DATASET: '*dataset-n* <(*datasetoption-1 datasetoption-2*...)>' >);

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 perform these operations:

- 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. The code 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 3.2 *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

- "Store and Retrieve Data"

Methods:

- "DEFINEDATA Method" on page 40

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

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 error message indicating that the key is not found is written to the log. The hash iterator traverses the hash table using the keys.

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

- Chapter 2, "Using the Hash and Hash Iterator Objects"

Methods:

- "FIRST Method" on page 56
- "LAST Method" on page 63
- "NEXT Method" on page 69

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

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.

Restriction The *keyvalue* cannot contain a component 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.

Restriction The *datavalue* cannot contain a component object.

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 "Maintain Key Summaries".

Example: Using the REF Method for Key Summaries

The following example uses the REF method for key summaries:

```
data keys;
 input key_id;
datalines;
1
2
1
3
5
2
3
2
4
```

```
1
5
1
;
data key_count;
 length count key_id 8;
 keep count key_id;
 if _n_ = 1 then do;
   declare hash myhash(suminc: "count", ordered: "y");
   declare hiter iter("myhash");
   myhash.defineKey('key_id');
   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=key_count;
run;
```

Output 3.3 *Output of COUNT Using the REF Method*

Obs	count	key_id
1	4	1
2	3	2
3	2	3
4	1	4
5	2	5

See Also

Methods:

- “ADD Method” on page 24
- “CHECK Method” on page 26

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 does 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 this 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 this 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 removes 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 3.4 *Key and Data Removed from Output*

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

See Also

- "Using the Hash Object"

Methods:

- "ADD Method" on page 24
- "DEFINEKEY Method" on page 43
- "REMOVEDUP Method" on page 84

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_id value 8;
   input key_id value;
   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_id');
   h.definedata('key_id', 'value');
   h.definedone();
   call missing (key_id, value);
   do key_id = 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_id= value=;
      rc = i.next();
   end;
run;
```

The following lines are written to the SAS log:

```
key_id=1 value=10
key_id=1 value=5
key_id=2 value=11
key_id=2 value=9
key_id=3 value=20
key_id=4 value=6
key_id=5 value=5
```

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "REMOVE Method" on page 81

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.

Restriction The *keyvalue* cannot contain a component object.

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.

Restriction The *datavalue* cannot contain a component object.

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 to replace the data as shown in the following code. In this example, the **`Place`** data for the key 'Collie' is changed from 'BestInShow' to 'bis'.

TIP The key is defined as a data item in the defineData method so that it is written to the data set.

```
data work.dogshow;
   length breed $10 place $10;
   input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;
```

```
proc print data=work.dogshow;
   title 'Dog Show Data Set Before Replace';
run;

data _null_;
   length breed $10 place $10;
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.dogshow');
      rc = h.defineKey('breed');
      rc = h.defineData('breed', 'place');
      rc = h.defineDone();
   end;
   /* Specify the key and new data value */
   breed = 'Collie';
   place = 'bis';
   /* 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.dogshow');
run;

proc print data=work.dogshow;
   title 'Dog 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.

Note: Because the key is defined as a data item in the DEFINEDATA method, it must be included in the REPLACE method as both a key and a data item.

```
data work.dogshow2;
   length breed $10 place $10;
   input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;
proc print data=work.dogshow2;
   title 'Second Dog Show Data Set Before Replace';
run;

data _null_;
   length breed $10 place $10;
   if _N_ = 1 then do;
      declare hash h(dataset:'work.dogshow2');
      rc = h.defineKey('breed');
      rc = h.defineData('breed', 'place');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(breed, place);
```

```
   end;
   /* Specify the key and new data value in the REPLACE method */
   rc = h.replace(key: 'Collie', data: 'Collie', data: 'bis');
   /* Write the hash table to the data set. */
   rc = h.output(dataset: 'work.dogshow2');
run;

proc print data=work.dogshow2;
   title 'Second Dog Show Data Set After Replace';
run;
```

Note: The hash object's REPLACE method is intended for use with hash tables that have a single data 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'`). 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.For more information about the MULTIDATA option, see "DECLARE Statement: Hash and Hash Iterator Objects" on page 30.

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

- "Using the Hash Object"

Methods:

- "DEFINEDATA Method" on page 40
- "DEFINEKEY Method" on page 43
- "REPLACEDUP Method" on page 90

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_id value 8;
   input key_id value;
   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_id');
   h.definedata('key_id', 'value');
   h.definedone();
   call missing (key_id, value);
   do key_id = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put @5 key_id= @15 value=;
         h.has_next(result: r);
         do while(r ne 0);
            rc = h.find_next();
            put 'dup ' @5 key_id= @15 value=;
            value = value + 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 @5 key_id= @15 value=;
```

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

The following lines are written to the SAS log:

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

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "REPLACE Method" on page 86

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 47.

See Also

Methods:

- "DO_OVER Method" on page 46

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 the data set ASTRO that contains astronomical data. The data includes Messier objects (OBJ) and right-ascension (RA) values. The method starts the 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, 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

- Chapter 2, “Using the Hash and Hash Iterator Objects”

Methods:

- “FIRST Method” on page 56
- “LAST Method” on page 63

Operators:

- “_NEW_ Operator: Hash and Hash Iterator” on page 64

Statements:

- “DECLARE Statement: Hash and Hash Iterator Objects” on page 30

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 "Maintain Key Summaries".

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.

```
data _NULL_;
   retain total 0;
   if _N_=1 then
      do;
         declare hash h(suminc:'count');
         rc=h.defineKey('k');
         rc=h.definedone();
      end;
   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 24
- “FIND Method” on page 50
- “CHECK Method” on page 26
- “DEFINEKEY Method” on page 43

- "REF Method" on page 78
- "SUMDUP Method" on page 98

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash and Hash Iterator Objects" on page 30

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 "Maintain Key Summaries".

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. The method 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_id value 8;
   input key_id value;
   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_id');
   h.definedata('key_id', 'value');
   h.definedone();
   call missing (key_id, value);
   i = 1;
  do key_id = 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_id = 1 to 5;
```

```
   rc = h.find();
   if (rc = 0) then do;
       h.sum(sum: sum);
       put @5 key_id= @15 value= @26 sum=;
       h.has_next(result: r);
       do while(r ne 0);
          rc = h.find_next();
          h.sumdup(sum: sum);
          put 'dup ' @5 key_id= @15 value= @26 sum=;
          h.has_next(result: r);
       end;
     end;
end;
run;
```

The following lines are written to the SAS log:

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

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

```
    key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
```

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

See Also

- “Duplicate Key Values and Data Pairs”

Methods:

- “SUM Method” on page 96

4

Using the Java Object

Using the Java Object

About the Java Object

The 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. You can create hybrid applications that contain both Java and DATA step code.

CLASSPATH and Java Options

In previous versions of SAS, Java classes were found using the JREOPTIONS system option.

You must set the CLASSPATH environment variable so that the Java object can find your Java classes. Any class that you use must appear in the CLASSPATH. If the class is in a .jar file, then the .jar filename must appear in the CLASSPATH.

Here is an example of the syntax needed to set the CLASSPATH environment variable in Linux:

- Global SAS configuration file `set CLASSPATH ~/HelloWorld.jar`
- Local EXPORT command `export CLASSPATH=~/HelloWorld.jar`

Restrictions and Requirements for Using the Java Object

The following restrictions and requirements apply when using the Java object:

- The Java object is designed to call Java methods from SAS. The Java object is not intended to extend the SAS library of functions. Calling PROC FCMP functions is much more efficient for fast in-process extensions to the DATA step, especially when large data sets are involved. Using the Java object to perform this type of processing with large data sets takes significantly more time.
- The only Java Runtime Environments (JREs) that are supported by SAS are those that are explicitly required during the installation of the SAS software.
- The only Java options that are supported by SAS are those that are set when SAS is installed.
- Ensure that your Java application runs correctly before using it with the Java object.
- 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 write a % in the first byte of a Java text line, it must be escaped with another percent immediately next to it (%%).
- When SAS is in a locked-down state, the Java object is not available. For more information, see "LOCKDOWN" in *SAS Programmer's Guide: Essentials*.

Declaring and Instantiating a Java Object

You declare a Java object by using the DECLARE statement. After you declare the new Java object, use the _NEW_ operator to instantiate the object, using the Java object name as an argument tag.

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

In this example, the DECLARE statement tells the compiler that the object reference J is of type Java. That is, the instance of the Java object is stored in the

variable J. At this point, you have declared only the object reference J. It has the potential to hold a component object of type Java. You should declare the Java object only once. The _NEW_ operator creates an instance of the Java object and assigns it to the object reference J. The Java class name, SOMEJAVACLASS, is passed as a constructor argument, which is the first-and-only argument that is required for the Java object constructor. All other arguments are constructor arguments to the Java class itself.

As an alternative to the two-step process of using the DECLARE statement and the _NEW_ operator to declare and instantiate a Java object, you can declare and instantiate a Java object in one step by using the DECLARE statement as a constructor method. The syntax is as follows:

DECLARE JAVAOBJ*object-name*(“*java-class*”, <*argument-1*, ... *argument–n*>);

For more information, see the “DECLARE Statement: Java Object” on page 130 and the “_NEW_ Operator: Java Object” on page 145.

Accessing Object Fields

Once you instantiate a Java object, you can access and modify its public and class fields in a DATA step through method calls on the Java object. Public fields are non-static and declared as public in the Java class. Class fields are static and accessed from Java classes.

Method calls to access object fields have one of these forms, depending on whether you are accessing non-static or static fields:

GET*type***FIELD**(“*field-name*”, *value*);

GETSTATIC*type***FIELD**(“*field-name*”, *value*);

Method calls to modify object fields have one of these forms, depending on whether you access static or non-static fields:

SET*type***FIELD**(“*field-name*”, *value*);

SETSTATIC*type***FIELD**(“*field-name*”, *value*);

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”. The *field-name* argument specifies the type for the Java field, and *value* specifies the value that is returned or set by the method.

For more information and examples, see “Dictionary of Java Object Language Elements”.

Accessing Object Methods

Once you instantiate a Java object, you can access its public and class methods in a DATA step through method calls on the Java object. Public methods are non-static and declared as public in the Java class. Class methods are static and accessed from Java classes.

Method calls to access Java methods have one of these forms, depending on whether you are accessing non-static or static methods:

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

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

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".

For more information and examples, see "Dictionary of Java Object Language Elements".

Type Issues

The Java type set is a superset of the SAS data types. Java has data types such as BYTE, SHORT, and CHAR in addition to the standard numeric and character values. SAS has only two data types: numeric and character.

The following table describes how Java data types are mapped to SAS data types when using the Java object method calls.

Table 4.1 *How Java Data Types Map to SAS Data Types*

Java Data Type	SAS Data Type
BOOLEAN	numeric
BYTE	numeric
CHAR	numeric
DOUBLE	numeric
FLOAT	numeric

Java Data Type	SAS Data Type
INT	numeric
LONG	numeric
SHORT	numeric
STRING	character[1]

1 Java string data types are mapped to SAS character data types as UTF-8 strings.

Other than STRING, it is not possible to return objects from Java classes to the DATA step. However, it is possible to pass objects to Java methods. For more information, see "Passing Java Object Arguments".

Some Java methods that return objects can be handled by creating wrapper classes to convert the object values. In the following example, the Java hash table returns object values. However, you can still use the hash table from the DATA step by creating simple Java wrapper classes to handle the type conversions. Then you can access the **dhash** and **shash** classes from the DATA step.

```
/* Java code */
import java.util.*;

public class dhash
{
   private Hashtable  table;

   public dhash()
   {
      table = new Hashtable  ();
   }

   public void put(double key, double value)
   {
      table.put(new Double(key), new Double(value));
   }

   public double get(double key)
   {
      Double ret = table.get(new Double(key));
      return ret.doubleValue();
   }
}

import java.util.*;

public class shash
{
   private Hashtable  table;

   public shash()
  {
      table = new Hashtable  ();
```

```
   }

   public void put(double key, String value)
  {
      table.put(new Double(key), value);
  }

   public String get(double key)
   {
      return table.get(new Double(key));
   }
}
```

```
/* DATA step code */
data _null_;
   dcl javaobj sh('shash');
   dcl javaobj dh('dhash');
   length d 8;
   length s $20;

   do i = 1 to 10;
      dh.callvoidmethod('vput', i, i * 2);
   end;

   do i = 1 to 10;
      sh.callvoidmethod('put', i, 'abc' || left(trim(i)));  end;

   do i = 1 to 10;
      dh.calldoublemethod('get', i, d);
      sh.callstringmethod('get', i, s);
      put d= s=;
   end;
run;
```

The following lines are written to the SAS log:

```
d=2 s=abc1
d=4 s=abc2
d=6 s=abc3
d=8 s=abc4
d=10 s=abc5
d=12 s=abc6
d=14 s=abc7
d=16 s=abc8
d=18 s=abc9
d=20 s=abc10
```

Java Objects and Arrays

You can pass DATA step arrays to Java objects.

In the following example, the arrays **d** and **s** are passed to the Java object **j**.

```
/* Java code
*/
```

```
import java.util.*;
import java.lang.*;
class jtest
{
   public void dbl(double args[])
  {
      for(int i = 0; i < args.length; i++)
         System.out.println(args[i]);
  }

  public void str(String args[])
  {
     for(int i = 0; i < args.length; i++)
        System.out.println(args[i]);
  }
}

/* DATA Step code */
data _null_;
   dcl javaobj j("jtest");
   array s{3} $20 ("abc", "def", "ghi");
   array d{10} (1:10);
   j.callVoidMethod("dbl", d);
   j.callVoidMethod("str", s);
run;
```

The following lines are written to the SAS log:

```
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
abc
def
ghi
```

Only one-dimensional array parameters are supported. However, it is possible to pass multidimensional array arguments by taking advantage of the fact that the arrays are passed in row-major order. You must handle the dimensional indexing manually in the Java code. That is, you must declare a one-dimensional array parameter and index to the subarrays accordingly.

Passing Java Object Arguments

Although it is not possible to return objects from Java classes to the DATA step, it is possible to pass objects, as well as strings, to Java class methods.

For example, suppose you have the following wrapper classes for **`java/util/Vector`** and its iterator:

```
/* Java code */
import java.util.*;

class mVector extends Vector
{
   public mVector()
  {
     super();
  }

  public mVector(double d)
  {
     super((int)d);
  }

  public void addElement(String s)
  {
     addElement((Object)s);
  }
}

import java.util.*;
public class mIterator
{
   protected mVector m_v;
   protected Iterator iter;

   public mIterator(mVector v)
   {
      m_v = v;
      iter = v.iterator();
   }

  public boolean hasNext()
  {
      return iter.hasNext();
  }

  public String next()
  {
     String ret = null;
     ret = (String)iter.next();
     return ret;
  }
}
```

These wrapper classes are useful for performing type conversions (for example, the **mVector** constructor takes a DOUBLE argument). Overloading the constructor is necessary because **java/util/Vector**'s constructor takes an integer value, but the DATA step has no integer type.

The following DATA step program uses these classes. The program creates and fills a vector, passes the vector to the iterator's constructor, and then lists all the values in the vector. Note that you must create the iterator after the vector is filled. The iterator keeps a copy of the vector's modification count at the time of creation, and this count must stay in synchronization with the vector's current modification

count. The code would throw an exception if the iterator were created before the vector was filled.

```
/* DATA step code */
data _null_;
   length b 8;
   length val $200;
   dcl javaobj v("mVector");

   v.callVoidMethod("addElement", "abc");
   v.callVoidMethod("addElement", "def");
   v.callVoidMethod("addElement", "ghi");
   dcl javaobj iter("mIterator", v);

   iter.callBooleanMethod("hasNext", b);
   do while(b);
      iter.callStringMethod("next", val);
      put val=;
      iter.callBooleanMethod("hasNext", b);
   end;

   m.delete();
   v.delete();
   iter.delete();
run;
```

The following lines are written to the SAS log:

```
val=abc
val=def
val=ghi
```

One current limitation to passing objects is that the JNI method lookup routine does not perform a full class lookup based on a given signature. This means that you could not change the **mIterator** constructor to take a **Vector** as shown in the following code:

```
/* Java code */
public mIterator(Vector v)
{
    m_v = v;
    iter = v.iterator();
}
```

Even though **mVector** is a subclass of **Vector**, the method lookup routine cannot find the constructor. Currently, the only solution is to manage the types in Java by adding new methods or by creating wrapper classes.

Java Exceptions

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. If you call a method that can throw an exception, it

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

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. By default, JVM standard output is redirected to the SAS log. Exception debugging is off by default.

For more information, see the "EXCEPTIONCHECK Method" on page 133, "EXCEPTIONCLEAR Method" on page 135, and the "EXCEPTIONDESCRIBE Method" on page 137.

Java Standard Output

Output from statements in Java that are directed to standard output such as the following are sent to the SAS log by default.

```
System.out.println("hello");
```

The Java output that is directed to the SAS log is flushed when the DATA step ends. This flushing causes the Java output to appear after any output that was generated while the DATA step was running. Use the FLUSHJAVAOUTPUT method to synchronize the output so that it appears in the order of execution.

Java Object Examples

Example 1: Calling a Simple Java Method

This Java class creates a simple method that sums three numbers.

```
/* Java code */
class MyClass
{
   double compute(double x, double y, double z)
      {
         return (x + y + z);
      }
}
```

```
/* DATA step code */
data _null_;
   dcl javaobj j("MyClass");

   rc = j.callDoubleMethod("compute", 1, 2, 3, r);

   put rc= r=;
run;
```

The following line is written to the SAS log:

```
rc=0 rc=6
```

Example 2: Creating a User Interface

In addition to providing a Java component access mechanism, you can use the Java object to create a simple Java user interface.

This Java class creates a simple user interface with several buttons. The user interface also maintains a queue of values that represent the sequence of button choices that are entered by a user.

```
/* Java code */
import java.awt.*;
import java.util.*;
import java.awt.event.*;

class colorsUI extends Frame
{
  private Button red;
  private Button blue;
  private Button green;
  private Button quit;
  private Vector list;
  private boolean d;
  private colorsButtonListener cbl;

  public colorsUI()
  {
    d = false;
    list = new Vector();
    cbl = new colorsButtonListener();

    setBackground(Color.lightGray);
    setSize(320,100);
    setTitle("New Frame");
    setVisible(true);
    setLayout(new FlowLayout(FlowLayout.CENTER, 10, 15));
    addWindowListener(new colorsUIListener());

    red = new Button("Red");
    red.setBackground(Color.red);
    red.addActionListener(cbl);

    blue = new Button("Blue");
    blue.setBackground(Color.blue);
    blue.addActionListener(cbl);

    green = new Button("Green");
    green.setBackground(Color.green);
    green.addActionListener(cbl);

    quit = new Button("Quit");
```

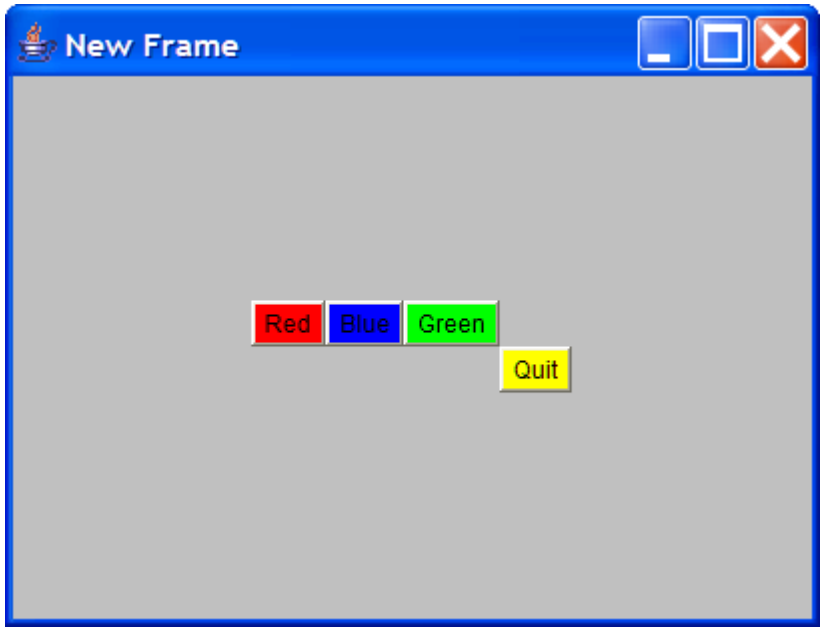
```
    quit.setBackground(Color.yellow);
    quit.addActionListener(cbl);

    this.add(red);
    this.add(blue);
    this.add(green);
    this.add(quit);

    show();
  }

  public synchronized void enqueue(Object o)
  {
    synchronized(list)
      {
        list.addElement(o);
        notify();
      }
  }

  public synchronized Object dequeue()
  {
    try
      {
        while(list.isEmpty())
          wait();

        if (d)
          return null;

        synchronized(list)
          {
            Object ret = list.elementAt(0);
            list.removeElementAt(0);
            return ret;
          }
      }
    catch(Exception e)
      {
        return null;
      }
  }

  public String getNext()
  {
    return (String)dequeue();
  }

  public boolean done()
  {
    return d;
  }

  class colorsButtonListener implements ActionListener
  {
     public void actionPerformed(ActionEvent e)
```

```
      {
         Button b;
         String l;
         b = (Button)e.getSource();
         l = b.getLabel();
         if ( l.equals("Quit") )
         {
            d = true;
            hide();
            l = "";
         }
         enqueue(l);
      }
   }

   class colorsUIListener extends WindowAdapter
   {
      public void windowClosing(WindowEvent e)
      {
         Window w;
         w = e.getWindow();
         d = true;
         enqueue("");
         w.hide();
      }
   }

   public static void main(String s[])
   {
       colorsUI cui;
       cui = new colorsUI();
   }
}
/* DATA step code */
data colors;
   length s $10;
   length done 8;
   drop done;

   if (_n_ = 1) then do;
    /* Declare and instantiate colors object (from colorsUI.class) */
      dcl javaobj j("colorsUI");
   end;

   /*
    * colorsUI.class will display a simple UI and maintain a
    * queue to hold color choices.
    */

   /* Loop until user hits quit button */
   do while (1);
      j.callBooleanMethod("done", done);
      if (done) then
         leave;
      else do;
       /* Get next color back from queue */
```

```
            j.callStringMethod("getNext", s);
            if s ne "" then
               output;
            end;
      end;
run;
proc print data=colors;
run;
quit;
```

In the DATA step code, the **colorsUI** class is instantiated and the user interface is displayed. You enter a loop that is terminated when you click **Quit**. This action is communicated to the DATA step through the Done variable. While looping, the DATA step retrieves the values from the Java class's queue and writes the values successively to the output data set.

Figure 4.1 *User Interface Created by the Java Object*



Example 3: Creating a Custom Class Loader

You might not want to put all your Java classes in the classpath. You can write your own class loader to find the classes and load them. The following example illustrates how you can create a custom class loader.

In this example, you create a class, **x**, which resides in a folder or directory, **y**. You call the methods in this class by using the Java object with the classpath that includes the **y** folder.

```
/* Java code */
package com.sas;

public class x
```

```
{
   public void m()
   {
      System.out.println("method m in y folder");
   }

   public void m2()
   {
      System.out.println("method m2 in y folder");
   }
}
```

```
/* DATA step code */
data _null_;
   dcl javaobj j('com/sas/x');
   j.callvoidmethod('m');
   j.callvoidmethod('m2');
run;
```

The following lines are written to the SAS log.

```
method m in y folder
method m2 in y folder
```

Suppose you have another class, **x**, that is stored in a different folder, **z**.

```
/* Java code
*/
package com.sas;

public class z
{
   public void m()
   {
      System.out.println("method m in y folder");
   }

   public void m2()
   {
      System.out.println("method m2 in y folder");
   }
}
```

You can call methods in this class instead of the class in folder **y** by changing the classpath, but this requires restarting SAS. The following method allows for more dynamic control of how classes are loaded.

To create a custom class loader, first you create an interface that contains all the methods that you will call through the Java object—in this program, **m** and **m2**.

```
/* Java
code */
public interface apiInterface
{
   public void m();
   public void m2();
}
```

Then you create a class for the actual implementation.

```
/* Java code */
import com.sas.x;

public class apiImpl implements apiInterface
{
   private x x;

   public apiImpl()
   {
      x = new x();
   }

   public void m()
   {
      x.m();
   }

   public void m2()
   {
      x.m2();
   }
}
```

These methods are called by delegating to the Java object instance class. Note that the code to create the **apiClassLoader** custom class loader is provided later in this section.

```
/* Java code */
public class api
{
  /* Load classes from the z folder */
  static ClassLoader customLoader =  new apiClassLoader("C:\\z");
  static String API_IMPL = "apiImpl";
  apiInterface cp = null;

  public api()
    {
      cp = load();
    }

  public void m()
    {
      cp.m();
    }

  public void m2()
    {
      cp.m2();
    }

  private static apiInterface load()
    {
      try
        {
          Class aClass = customLoader.loadClass(API_IMPL);
          return (apiInterface) aClass.newInstance();
        }
```

```
      catch (Exception e)
       {
          e.printStackTrace();
          return null;
        }
    }
}
```

The following DATA step program calls these methods by delegating through the **api** Java object instance class. The Java object instantiates the **api** class, which creates a custom class loader to load classes from the **z** folder. The **api** class calls the custom loader and returns an instance of the **apiImpl** interface implementation class to the Java object. When methods are called through the Java object, the **api** class delegates them to the implementation class.

```
/* DATA step code */
data _null_;
   dcl javaobj j('api');
   j.callvoidmethod('m');
   j.callvoidmethod('m2');
run;
```

The following lines are written to the SAS log:

```
method m is z folder
method m2 in z folder
```

In the previous Java code, you could also use .jar files to augment the classpath in the **ClassLoader** constructor.

```
static ClassLoader customLoader = new apiClassLoader("C:\\z;C:\\temp\some.jar");
```

In this case, the Java code for the custom class loader is as follows. This code for this class loader can be added to or modified as needed.

```
import java.io.*;
import java.util.*;
import java.util.jar.*;
import java.util.zip.*;

public class apiClassLoader extends ClassLoader
{
  //class repository where findClass performs its search
  private List classRepository;

  public apiClassLoader(String loadPath)
    {
      super(apiClassLoader.class.getClassLoader());
      initLoader(loadPath);
    }

  public apiClassLoader(ClassLoader parent,String loadPath)
    {
      super(parent);
      initLoader(loadPath);
    }

  /**
   * This method will look for the class in the class repository. If
```

```
 * the method cannot find the class, the method will delegate to its parent
 * class loader.
 *
 * @param className A String specifying the class to be loaded
 * @return  A Class object loaded by the apiClassLoader
 * @throws ClassNotFoundException if the method is unable to load the class
 */
public Class loadClass(String name) throws ClassNotFoundException
  {
    // Check if the class is already loaded
    Class loadedClass = findLoadedClass(name);

    // Search for class in local repository before delegating
   if (loadedClass == null)
     {
       loadedClass = myFindClass(name);
     }

    // If class not found, delegate to parent
    if (loadedClass == null)
      {
        loadedClass = this.getClass().getClassLoader().loadClass(name);
      }
    return loadedClass;
  }

private Class myFindClass(String className) throws ClassNotFoundException
  {
    byte[] classBytes = loadFromCustomRepository(className);
    if(classBytes != null)
      {
        return defineClass(className,classBytes,0,classBytes.length);
      }
    return null;
  }

/**
 *  This method loads binary class file data from the classRepository.
 */
private byte[] loadFromCustomRepository(String classFileName)
  throws ClassNotFoundException
  {
    Iterator dirs = classRepository.iterator();
    byte[] classBytes = null;
    while (dirs.hasNext())
      {
        String dir = (String) dirs.next();

        if (dir.endsWith(".jar"))
          {
            // Look for class in jar

            String jclassFileName = classFileName;

            jclassFileName = jclassFileName.replace('.', '/');
            jclassFileName += ".class";
```

```
                try
                  {
                    JarFile j = new JarFile(dir);
                    for (Enumeration e = j.entries(); e.hasMoreElements() ;)
                      {
                        Object n = e.nextElement();

                        if (jclassFileName.equals(n.toString()))
                          {
                            ZipEntry zipEntry = j.getEntry(jclassFileName);
                            if (zipEntry == null)
                              {
                                return null;
                              }
                            else
                              {
                                // read file
                                InputStream is = j.getInputStream(zipEntry);
                                classBytes = new byte[is.available()];
                                is.read(classBytes);
                                break;
                              }
                          }
                      }
                  }
                catch (Exception e)
                  {
                    System.out.println("jar file exception");
                    return null;
                  }
              }
            else
              {
                // Look for class in directory
                String fclassFileName = classFileName;

                fclassFileName = fclassFileName.replace('.', File.separatorChar);
                fclassFileName += ".class";

                try
                  {
                    File file = new File(dir,fclassFileName);
                    if(file.exists()) {
                      //read file
                      InputStream is = new FileInputStream(file);
                      classBytes = new byte[is.available()];
                      is.read(classBytes);
                      break;
                    }
                  }
                catch(IOException ex)
                  {
                    System.out.println("IOException raised while reading class
file data");
                    ex.printStackTrace();
```

```
                    return null;
                  }
              }
          }
        return classBytes;
      }

    private void initLoader(String loadPath)
      {
        /*
         * loadPath is passed in as a string of directories/jar files
         * separated by the File.pathSeparator
         */
        classRepository = new ArrayList();
        if((loadPath != null) && !(loadPath.equals("")))
          {
            StringTokenizer tokenizer =
              new StringTokenizer(loadPath,File.pathSeparator);
            while(tokenizer.hasMoreTokens())
              {
                classRepository.add(tokenizer.nextToken());
              }
          }
      }
}
```

5

Dictionary of Java Object Language Elements

Java Object Methods by Category

There are five categories of Java object methods.

Table 5.1 *Java Object Methods by Category*

Category	Description
Deletion	enables you to delete a Java object.

Category	Description
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. 133)	Deletes the Java object.
Exception	EXCEPTIONCHECK Method (p. 133)	Determines whether an exception occurred during a method call.
	EXCEPTIONCLEAR Method (p. 135)	Clears any exception that is currently being thrown.
	EXCEPTIONDESCRIBE Method (p. 137)	Turns the exception debug logging on or off and prints exception information.
Field Reference	GETtypeFIELD Method (p. 140)	Returns the value of a non-static field for a Java object.
	GETSTATICtypeFIELD Method (p. 143)	Returns the value of a static field for a Java object.
	SETtypeFIELD Method (p. 147)	Modifies the value of a non-static field for a Java object.
	SETSTATICtypeFIELD Method (p. 150)	Modifies the value of a static field for a Java object.
Method Reference	CALLtypeMETHOD Method (p. 125)	Invokes an instance method on a Java object from a non-static Java method.
	CALLSTATICtypeMETHOD Method (p. 127)	Invokes an instance method on a Java object from a static Java method.
Output	FLUSHJAVAOUTPUT Method (p. 139)	Specifies that the Java output is sent to its destination.

Dictionary

CALLtypeMETHOD 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"

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" .

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 127

CALLSTATICtypeMETHOD 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"

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" .

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 125

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 131

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 "Component Objects".

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

- "Component Objects"

Operators:

- "_NEW_ Operator: Java Object" on page 145

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 135
- “EXCEPTIONDESCRIBE Method” on page 137

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 133
- “EXCEPTIONDESCRIBE Method” on page 137

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 133
- “EXCEPTIONCLEAR Method” on page 135

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"

GETtypeFIELD 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"

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".

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 143
- “SETtypeFIELD Method” on page 147

GETSTATICtypeFIELD 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"

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".

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 140
- “SETSTATICtypeFIELD Method” on page 150

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 “Component Objects”.

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

Statements:

- "DECLARE Statement: Java Object" on page 130

SETtypeFIELD 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"

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".

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 140
- “SETSTATICtypeFIELD Method” on page 150

SETSTATICtypeFIELD 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"

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".

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 143
- “SETtypeFIELD Method” on page 147

PART 3

FCMP Component Objects

6

Using PROC FCMP Hash Objects and Hash Iterator Objects

Using the PROC FCMP Hash Object and PROC FCMP Hash Iterator Object

The PROC FCMP hash object and PROC FCMP hash iterator object enable you to quickly and efficiently store, search, filter, and retrieve data based on lookup keys. These 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. Operators provide special functionality. You use the DATA step object dot notation to access the component object's attributes and methods.

Through the use of a hash function, an input string or number (key) is converted to an integer (hash). This hash value is then used as an index into a hash table. For each key, the hash table can then store and retrieve associated data. Hashing is considered the fastest way to search a large amount of information that is referenced through keys.

For a list of supported statements, methods, and syntax for the PROC FCMP hash object and hash iterator object, see PROC FCMP Hash and Hash Iterator Language Elements.

For information about how to use hashing in PROC FCMP and to review examples, see Hashing in PROC FCMP to Enhance Your Productivity.

For another example of using PROC FCMP and hash objects, see Load a SAS Data Set into a Hash Object Using PROC FCMP.

7

PROC FCMP Hash and Hash Iterator Language Elements

Dictionary

ADD Method

Adds a key-value pair to the PROC FCMP hash object.

Applies to: PROC FCMP 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 0 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 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 *keyvalue* must be a literal.

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 *datavalue* must be a literal.

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:

```
proc fcmp;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
   /* Define constant key and data values */
   k = 'Joyce';
   d = 'Ulysses';
   /* Add key and data values to hash object */
   rc = h.add();
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.

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

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

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. Because no assignment to a key or data variable appears in this program, SAS issues a note stating that the variables are uninitialized. In order to avoid receiving these notes, use the CALL MISSING routine to specify the key and data variables as parameters.

See Also

- “Using the Hash Object”

Methods:

- "DEFINEDATA Method" on page 163
- "DEFINEKEY Method" on page 167

CHECK Method

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

Applies to: PROC FCMP 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 0 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 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 *keyvalue* must be a literal.

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 this code:

```
proc fcmp;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
```

```
  /* avoid uninitialized variable notes */
      call missing(k, d);
   /* 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 this code:

```
proc fcmp;
   length k $8;
   length d $12;
   /* Declare hash object and key and data variable names */
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();

  /* avoid uninitialized variable notes */
      call missing(k, d);
   /* 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, 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 167
- “FIND Method” on page 173

CLEAR Method

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

Applies to: PROC FCMP hash object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 written to the log.

object
specifies the name of the hash object.

Details

The CLEAR method enables you to remove items from an existing hash object and reuse it without deleting the object and creating 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 variables. It clears only the values in the hash object.

Example: Clearing a Hash Object

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

```
proc fcmp ;

/* Declares the hash object named h */
   declare hash h1();
   rc= h1.defineKey('key1');
   rc=h1.defineData('key1');
   rc=h1.defineDone();
```

```
   key1 = 'abc';
   rc= h1.add();
/* Uses the NUM_ITEMS attribute, which returns the */
/* number of items in the hash object.             */
   n = h1.num_items();
   put 'Number of items before clear = ' n;
/* Uses the CLEAR method to delete all items within h. */
   rc = h1.clear();
/* Writes the number of items in the log. */
   n = h1.num_items();
   put 'Number of items after clear = ' n;
 quit;
 run;
```

The preceding statements produce these results:

```
Number of items before clear = 1
Number of items after clear = 0
```

See Also

Methods:

- "DELETE Method: Hash and Hash Iterator Objects" on page 169

DEFINEDATA Method

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

Applies to: PROC FCMP hash object

Syntax

rc=*object*.**DEFINEDATA** ('*datavarname-1*' <'*datavarname-2*', ...>);

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 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 into the object constructor.

If the *dataset* argument tag is used in the DECLARE statement to automatically load a data set, 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: PROC FCMP does not execute the DEFINEDATA method multiple times. Unlike the DATA step DEFINEDATA method, there is no need to create a conditional do block to avoid repeated execution of the DEFINEDATA method in PROC FCMP.

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, 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, SAS issues a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of these 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.

For information about how to use the DEFINEDATA method, see "Define Keys and Data".

Example

This example creates a hash object and defines the key and data variables.

```
proc fcmp;
   length d $20;
   length k $20;
   /* Declare the hash object and key and data variables */
      declare hash h();
      rc = h.defineKey('k');
      rc = h.defineData('d');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(k, d);
run;
```

See Also

- "Define Keys and Data"

Methods:

- "DEFINEDONE Method" on page 165
- "DEFINEKEY Method" on page 167

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

DEFINEDONE Method

Indicates that all key and data definitions are complete for the PROC FCMP hash object.

Applies to: PROC FCMP hash object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 written to the log.

object
specifies the name of the hash object.

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.

Note: Unlike the DATA step DEFINEDATA method, there is no need to create a conditional do block to avoid repeated execution of the DEFINEDATA method in PROC FCMP.

For more information about how to use the DEFINEDONE method, see "Define Keys and Data".

See Also

- "Define Keys and Data"

Methods:

- "DEFINEDATA Method" on page 163
- "DEFINEKEY Method" on page 167

DEFINEKEY Method

Defines key variables for the PROC FCMP hash object.

Applies to: PROC FCMP hash object

Syntax

rc=*object*.**DEFINEKEY**('*keyvarname-1* '<, ...'*keyvarname-2*', ...>);

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 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 into the object constructor.

If you use the *dataset* argument tag in the DECLARE statement or _NEW_ operator to automatically load a data set, 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 "Define Keys and Data".

Note: PROC FCMP does not execute the DEFINEKEY method multiple times. Unlike the DATA step DEFINEKEY method, there is no need to create a conditional do block to avoid repeated execution of the DEFINEKEY method in PROC FCMP.

Note: If you use the shortcut notation for the ADD, CHECK, FIND, REMOVE, or REPLACE method (for example, **`h.add(key:99, data:'apple', data:'orange')`**) and the ALL: 'YES' option on the DEFINEKEY method, 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 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, SAS issues a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of these 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;
   declare hash h();
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   call missing(k, d);
```

See Also

- "Define Keys and Data"

Methods:

- "DEFINEDATA Method" on page 163
- "DEFINEDONE Method" on page 165

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

DELETE Method: Hash and Hash Iterator Objects

Deletes the PROC FCMP hash object or PROC FCMP hash iterator object.

Applies to: PROC FCMP hash object, PROC FCMP hash iterator object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 or hash iterator object.

Details

PROC FCMP component objects are deleted automatically at the end of the PROC. If you want to reuse the object reference name in another hash or hash iterator object or if you want to release the memory used by that object, use the DELETE method to delete the hash or hash iterator object.

If you attempt to use a hash or hash iterator object after you delete it, you 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..

DO_OVER Method

Traverses a list of duplicate keys in the hash object.

Applies to: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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;
   input key_id value;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
run;

proc fcmp;
file log;
```

```
   dcl hash h(dataset:'dup', multidata: 'y', ordered: 'y');
   rc= h.definekey('key_id');
   rc= h.definedata('key_id', 'value');
   rc= h.definedone();
   call missing(key_id, value);

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

   key_id = 3;
   do while(h.do_over(key:3) eq 0);
      put key_id= value=;
   end;

   key_id = 2;
   do while(h.do_over(key:2) eq 0);
     put key_id= value=;
   end;
run;
quit;
```

The following lines are written to the SAS log:

```
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
key_id=3 value=20
key_id=3 value=100
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
```

See Also

Methods:

- "RESET_DUP Method"

EQUALS Method

Determines whether two hash objects are equal.

Applies to: PROC FCMP hash object

Note: The PROC FCMP EQUALS method is available as of 2024.07.

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.

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 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:

```
proc fcmp;
file log
    declare hash h1();
    rc=h1.defineKey('k');
    rc=h1.defineDone();

    declare hash h2();
    rc=h2.defineKey('k');
    rc=h2.defineDone();
    call missing(k);

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

The following line is written to the SAS log:

```
hash objects are equal
```

The two hash objects are defined as equal when all of these 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 H1 and H2 hash objects are declared and instantiated. A record with a key value of 99 is added to both objects. The first call to EQUALS returns a nonzero result value indicating that the hash objects are equivalent.

The key in H2 is then replaced with a value of 100. A second call to EQUALS compares the H1 and H2 hash objects and returns a zero result value indicating that the hash objects are no longer equivalent.

```
proc fcmp;
file log;
 declare hash h1();
   rc=h1.defineKey('k');
   rc=h1.defineDone();

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

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

   k = 100;
   rc=h2.replace();

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

The following lines are written to the SAS log:

```
eq=1
eq=0
```

FIND Method

Determines whether the specified key is stored in the PROC FCMP hash object and if found, updates the data variables in the data set.

Applies to: PROC FCMP hash object

Syntax

rc=*object*.**FIND** (<KEY: *keyvalue-1*, KEY: *keyvalue-2*, ...>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 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 *keyvalue* must be a literal.

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 this code:

```
proc fcmp;
   length k $8  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 this code:

```
proc fcmp;
   length k $8  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;
```

Comparisons

The FIND method returns a value that indicates whether the key is stored in the hash object. If the key is in the hash object, 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 stored in the hash object. The data variable is not updated.

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

This 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.

```
   proc fcmp;
      length k $8;
      length d $12;
      /* Declare hash object and key and data variable names */
         declare hash h();
         rc = h.defineKey('k');
         rc = h.defineData('d');
         rc = h.defineDone();
         /* avoid uninitialized variable notes */
         call missing(k, d);
      /* 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;
```

The preceding statements produce this result:

```
d=Ulysses
```

See Also

- "Store and Retrieve Data"

Methods:

- "CHECK Method" on page 160
- "DEFINEKEY Method" on page 167

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: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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

The following lines are written to the SAS log:

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

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method"
- "FIND_PREV Method"
- "HAS_NEXT Method"

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: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 58 for an example.

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method" on page 50
- "FIND_NEXT Method" on page 53
- "HAS_PREV Method" on page 60

HAS_NEXT Method

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

Applies to: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 returns a nonzero value in the numeric variable **R**. Otherwise, it returns 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 billing;
   length customer_num invoice_amount  8;
   input customer_num invoice_amount;
   datalines;
   1 100
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
proc fcmp;
file log;
   length r 8;
   dcl hash h(dataset:'billing', multidata: 'y');
   rc= h.definekey('customer_num');
   rc= h.definedata('customer_num', 'invoice_amount');
   rc= h.definedone();
   call missing (customer_num, invoice_amount);
   do customer_num = 1 to 5;
```

```
      rc = h.find();
      if (rc = 0) then do;
         put @5 customer_num= @20 invoice_amount=;
         rc= h.has_next(result: r);
         do while(r ne 0);
             rc = h.find_next();
             put 'dup ' @5 customer_num= @20 invoice_amount=;
             rc= h.has_next(result: r);
         end;
      end;
   end;
run;
```

The following lines are written to the SAS log:

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

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method"
- "FIND_NEXT Method"
- "FIND_PREV Method"
- "HAS_NEXT Method"

HAS_PREV Method

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

Applies to: Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 58 for an example.

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "FIND Method"
- "FIND_NEXT Method"
- "FIND_PREV Method"
- "HAS_NEXT Method"

DECLARE Statement: Hash Object and Hash Iterator Object

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

Valid in: PROC FCMP

Category: Action

Type: Declarative

Alias: DCL

Applies to: PROC FCMP hash object, PROC FCMP hash iterator object

Syntax

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

Form 2: **DECLARE HITER** *object-reference* ("*hash-reference*");

Form 3: **DECLARE** *object object-reference* (<*argument_tag-1*: *value-1*, *argument_tag-2*: *value-2*, ...>);

Arguments

object
specifies the component object. *object* can be one of these 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 Chapter 2, "Using the Hash and Hash Iterator Objects"

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 Chapter 2, "Using the Hash and Hash Iterator Objects"

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

hash-reference
specifies the reference name of a hash object to attach the hash iterator to.

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

There are four 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 must be a literal. A literal data set name must be enclosed in single or double quotation marks.

This syntax is used:

```
dcl hash h (dataset: 'x');
```

Note If the data set contains duplicate keys, the default is to load the first instance in the hash object; subsequent instances are ignored. Use the DUPLICATE argument tag to store the last instance in the hash object or write an error message to the SAS log.

duplicate: *'option'*
determines how to handle duplicate keys if they are not allowed. The default is to store the first key and ignore subsequent duplicates. *option* can be one of these values:

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

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

This example uses the REPLACE option and stores **`blue`** for the *color_id* key 531 and **`brown`** for the *color_id* key 620. If you use the default, the first values (**`yellow`** for 531 and **`green`** for 620) are loaded and the duplicate values are ignored.

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

proc fcmp;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
    declare hash myhash(dataset: "table", duplicate: "r");
    rc = myhash.definekey('color_id');
    rc = myhash.definedata('color_id','color_name');
    rc=myhash.definedone();
    /* avoid uninitialized variable notes */
    call missing(color_id, color_name);
 end;
 n = myhash.num_items();
 put n=;
```

```
    run;

n=4
```

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 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

multidata: '***option***'

Default NO

Restriction The multidata argument is available starting with the 2024.11 release.

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

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.

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.

option can be one of these 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.

suminc: '*variable-name*'

Restriction The suminic argument is available starting with the 2024.11 release.

See "Maintain Key Summaries"

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

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

maintains a summary count of hash object keys. The SUMINC argument tag is given a PROC FCMP 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 "Initialize Hash Object Data Using a Constructor"

Details

The Basics

To use a PROC FCMP component object in your SAS program, you must declare and create (instantiate) the object. You can access the predefined hash and hash iterator component objects through the PROC FCMP component interface.

Declaring a Hash Object (Form 1)

You use the DECLARE statement to declare a hash object.

```
declare hash h;
```

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

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

Use the DECLARE statement to declare and instantiate the hash or hash iterator object in one step. For example, in this line of code, the DECLARE statement declares and instantiates a hash object and assigns it to the object reference H:

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

A *constructor* is a method that you can use to instantiate a hash object and initialize the hash object data. For example, in this 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);
```

Examples

Example 1: 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.

```
proc fcmp;
   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;
```

The preceding statements produce this result:

```
d=Terrier
```

Example 2: 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).

```
proc fcmp;
   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;
```

The preceding statements produce this result:

```
d=Terrier
```

NUM_ITEMS Method Attribute

Returns the number of items in the PROC FCMP hash object.

Applies to: PROC FCMP 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 METHOD.

```
data work.stock;
   input item $ qty;
   datalines;
broccoli 345
corn 389
potato 993
onion 730
;
proc fcmp;
   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 */
      rc=myhash.defineKey('item');
      rc=myhash.defineData('qty');
      rc=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;
```

The preceding statements produce this result:

```
totalitems=5
```

REF Method

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

Applies to: Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

Restriction The *keyvalue* cannot contain a component 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.

Restriction The *datavalue* cannot contain a component object.

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 "Maintain Key Summaries".

Example: Using the REF Method for Key Summaries

The following example uses the REF method for key summaries:

```
data keys;
 input key_id;
datalines;
1
2
1
3
5
2
3
2
4
1
5
1
;

proc fcmp data=keys;
file log;
 length count key_id 8;
 if _n_ = 1 then do;
   declare hash myhash(suminc: "count", ordered: "y");
   declare hiter iter("myhash");
   rc = myhash.defineKey('key_id');
   rc = myhash.defineDone();
   count = 1;
  end;

 rc = myhash.ref();
 file log;
 if (_n_ = 12) then do;
     rc = iter.first();
     do while(rc = 0);
     rc = myhash.sum(sum: count);
         put key_id= count=;
         rc = iter.next();
     end;
 end;
```

```
run;
```

The following lines are written to the SAS log:

```
key_id=1 count=4
key_id=2 count=3
key_id=3 count=2
key_id=4 count=1
key_id=5 count=2
```

See Also

Methods:

- "ADD Method"
- "CHECK Method"

REMOVEDUP Method

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

Applies to: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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_id value 8;
   input key_id value;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
```

```
;
proc fcmp;
file log;
   length r 8;
   dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
   rc= h.definekey('key_id');
   rc= h.definedata('key_id', 'value');
   rc= h.definedone();
   call missing (key_id, value);
   do key_id = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         rc= h.has_next(result: r);
         if (r ne 0) then do;
            rc= h.find_next();
            rc= h.removedup();
         end;
      end;
   end;
   dcl hiter i('h');
   rc = i.first();
   do while (rc = 0);
      put key_id= value=;
      rc = i.next();
   end;
run;
```

The following lines are written to the SAS log:

```
key_id=1 value=10
key_id=1 value=5
key_id=2 value=11
key_id=2 value=9
key_id=3 value=20
key_id=4 value=6
key_id=5 value=5
```

See Also

- "Duplicate Key Values and Data Pairs"

Methods:

- "REMOVE Method"

REMOVE Method

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

Applies to: PROC FCMP hash object

Syntax

rc=*object*.**REMOVE** (<KEY: *keyvalue-1*, KEY: *keyvalue-2*, ...>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 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 *keyvalue* must be a literal.

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*, the REMOVE method does not remove the key or data from the hash object.

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 this code:

```
proc fcmp;
   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 this code:

```
proc fcmp;
   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. The method removes only the value in the hash object.

See Also

Methods:

- “ADD Method” on page 158
- “DEFINEKEY Method” on page 167

REPLACE Method

Replaces the data that is associated with the specified key with new data in the PROC FCMP hash object.

Applies to: PROC FCMP hash object

Syntax

rc=*object*.**REPLACE** (<<KEY: *keyvalue-1*>, <KEY: *keyvalue-2*, ...>, <DATA: *datavalue-1*>,
<DATA: *datavalue-2*>, ...>);

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 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 *keyvalue* must be a literal.

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 *datavalue* must be a literal.

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 to replace the data as shown in the following code. In this example, the **`Place`** data for the key 'Collie' is changed from 'BestInShow' to 'bis'.

TIP The key is defined as a data item in the defineData method so that it is written to the data set.

```
data work.dogshow;
   length breed $10 place $10;
   input breed place;
datalines;
Terrier    2nd
```

```
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;

proc fcmp;
   length breed $10 place $10;
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.dogshow');
      rc = h.defineKey('breed');
      rc = h.defineData('breed', 'place');
      rc = h.defineDone();
   end;
   /* Specify the key and new data value */
   breed = 'Collie';
   place = 'bis';
   /* Call the REPLACE method to replace the data value */
   rc = h.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.

Note: Because the key is defined as a data item in the DEFINEDATA method, it must be included in the REPLACE method as both a key and a data item.

```
data work.dogshow2;
   length breed $10 place $10;
   input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;

proc fcmp;
   length breed $10 place $10;
   if _N_ = 1 then do;
      declare hash h(dataset:'work.dogshow2');
      rc = h.defineKey('breed');
      rc = h.defineData('breed', 'place');
      rc = h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(breed, place);
   end;
   /* Specify the key and new data value in the REPLACE method */
   rc = h.replace(key: 'Collie', data: 'Collie', data: 'bis');
run;
```

Note: If you call the REPLACE method and the key is not found, 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. The method replaces only the value in the hash object.

See Also

Methods:

- "DEFINEDATA Method" on page 163
- "DEFINEKEY Method" on page 167

REPLACEDUP Method

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

Applies to: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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_id value 8;
   input key_id value;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
proc fcmp;
file log;
   length r 8;
   dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
   rc= h.definekey('key_id');
   rc= h.definedata('key_id', 'value');
```

```
   rc= h.definedone();
   call missing (key_id, value);
   do key_id = 1 to 5;
      rc = h.find();
      if (rc = 0) then do;
         put @5 key_id= @15 value=;
         rc= h.has_next(result: r);
         do while(r ne 0);
            rc = h.find_next();
            put 'dup ' @5 key_id= @15 value=;
            value = value + 300;
            rc = h.replacedup();
            rc= h.has_next(result: r);
         end;
      end;
   end;
   put 'iterating...';
   dcl hiter i('h');
   rc = i.first();
   do while (rc = 0);
      put @5 key_id= @15 value=;
      rc = i.next();
    end;
run;
```

The following lines are written to the SAS log:

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

See Also

- “Duplicate Key Values and Data Pairs”

Methods:

- "REPLACE Method"

RESET_DUP Method

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

Applies to: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 47.

See Also

Methods:

- "DO_OVER Method"

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: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 “Maintain Key Summaries”.

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. The method 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_id value 8;
   input key_id value;
   cards;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
proc fcmp;
file log;
   length r i sum 8;
   i = 0;
   dcl hash h(dataset:'dup', multidata: 'y', suminc: 'i');
   rc= h.definekey('key_id');
   rc= h.definedata('key_id', 'value');
   rc= h.definedone();
   call missing (key_id, value);
   i = 1;
  do key_id = 1 to 5;
     rc = h.find();
     if (rc = 0) then do;
        rc= h.has_next(result: r);
        do while(r ne 0);
           rc = h.find_next();
           rc = h.find_prev();
           rc = h.find_next();
           rc= h.has_next(result: r);
        end;
     end;
  end;
 i = 0;
 do key_id = 1 to 5;
   rc = h.find();
   if (rc = 0) then do;
       rc= h.sum(sum: sum);
       put @5 key_id= @15 value= @26 sum=;
       rc= h.has_next(result: r);
       do while(r ne 0);
          rc = h.find_next();
          rc= h.sumdup(sum: sum);
          put 'dup ' @5 key_id= @15 value= @26 sum=;
          rc= h.has_next(result: r);
       end;
     end;
 end;
```

```
run;
```

The following lines are written to the SAS log:

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

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

```
    key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
```

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

See Also

- “Duplicate Key Values and Data Pairs”

Methods:

- “SUM Method”

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: PROC FCMP Hash object

Restriction: This method is available starting with the 2024.11 release.

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.

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 "Maintain Key Summaries".

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.

```
proc fcmp;
file log;
   retain total 0;
   if _N_=1 then
      do;
         declare hash h(suminc:'count');
         rc=h.defineKey('k');
         rc=h.definedone();
      end;
   k = 99;
   count = 1;
   rc= h.add();
   /* key=99 summary is now 1 */
   k = 100;
   rc= h.add();
   /* key=100 summary is now 1 */
   k = 99;
   rc= h.find();
   /* key=99 summary is now 2 */
   count = 2;
   rc= h.find();
   /* key=99 summary is now 4 */
   k = 100;
   rc= h.find();
   /* key=100 summary is now 3 */
   rc= h.sum(sum: total);
   put 'total for key 100 = ' total;
   k = 99;
   rc= 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”
- “FIND Method” on page 173
- “CHECK Method” on page 160
- “DEFINEKEY Method” on page 167
- “REF Method”
- “SUMDUP Method” on page 203

Operators:

- "_NEW_ Operator: Hash and Hash Iterator" on page 64

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object"

FIRST Method

Returns the first value in the underlying PROC FCMP hash object.

Applies to: PROC FCMP hash iterator object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 error message indicating that the key is not found is written 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 when you instantiate the hash object, 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 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 when you instantiate the hash object, 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 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

This 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
;
proc fcmp;
      length prod $10 qty $6;
   /* Declare hash object and read SALES data set as ordered */
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.sales', ordered: 'yes');
      declare hiter iter('h');
      /* Define key and data variables */
      rc=h.defineKey('qty');
      rc=h.defineData('prod', 'qty');
      rc=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=  qty=;
      rc = iter.next();
   end;
run;
```

The following lines are written to the SAS log:

```
prod=orange qty=329559
prod=apple qty=384223
prod=banana qty=398487
```

See Also

- "Using the Hash Iterator Object "

Methods:

- "LAST Method" on page 210

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

LAST Method

Returns the last value in the underlying PROC FCMP hash object.

Applies to: PROC FCMP hash iterator object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 error message indicating that the key is not found 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 when you instantiate the hash object, 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 when you instantiate the hash object, 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 "

Methods:

- "FIRST Method" on page 208

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

NEXT Method

Returns the next value in the underlying PROC FCMP hash object.

Applies to: PROC FCMP hash iterator object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 error message indicating that the key is not found 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, the NEXT method starts at the first item in the hash object.

See Also

- "Using the Hash Iterator Object "

Methods:

- "FIRST Method" on page 208
- "PREV Method" on page 212

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

PREV Method

Returns the previous value in the underlying PROC FCMP hash object.

Applies to: PROC FCMP hash iterator object

Syntax

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

Arguments

rc
specifies whether the method succeeded or failed.

A return code of 0 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 error message indicating that the key is not found is written to the log. The hash iterator traverses the hash table using the keys.

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: If you call the PREV method without calling the LAST method, the PREV method still starts at the last 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 "

Methods:

- "FIRST Method" on page 208
- "LAST Method" on page 210
- "NEXT Method" on page 211

Statements:

- "DECLARE Statement: Hash Object and Hash Iterator Object" on page 183

SETCUR Method

Specifies a starting key item for iteration.

Applies to: PROC FCMP hash iterator object

Note: The PROC FCMP SETCUR method is available as of 2024.07.

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

This example creates a data set that contains sales data. You want to list products that sold greater than or equal to 329559 units. The data is loaded into a hash object and the SETCUR and NEXT methods are used to retrieve the data.

```
data work.sales;
   input prod $1-6 qty $9-14;
   datalines;
banana  398487
apple   384223
kiwi    364858
orange  329559
mango   295874
;
proc fcmp;
file log;
      length prod $10 qty $6;
   /* Declare hash object and read SALES data set as ordered */
   if _N_ = 1 then do;
      declare hash h(dataset: 'work.sales', ordered: 'yes');
      declare hiter iter('h');
      /* Define key and data variables */
      rc=h.defineKey('qty');
      rc=h.defineData('prod', 'qty');
      rc=h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(qty, prod);
   end;
   /* Iterate through the hash object and output data values */
   rc = iter.setcur(key: '329559');
   do while (rc = 0);
      put prod=  qty=;
      rc = iter.next();
   end;
run;
```

The following lines are written to the SAS log:

```
prod=orange qty=329559
prod=kiwi qty=364858
prod=apple qty=384223
prod=banana qty=398487
```

8

Using Dictionary Objects

Using FCMP Dictionary Objects

Why Use Dictionary Objects

Dictionary objects provide an alternative method of storing data inside a PROC FCMP program. Hash objects are limited to storing only numeric and character data, but dictionaries can store numeric and character data in addition to arrays, hash objects, and even other dictionaries. Dictionaries can store data by reference or by value. To store your data in a dictionary, the data does not need to be predefined.

Getting Started

For individual method syntaxes, see “Dictionary Object Language Elements”.

Create a Dictionary

Create a dictionary by using the DECLARE DICTIONARY statement. The abbreviation DNARY is also a valid keyword for creating dictionaries.

```
proc fcmp;
   declare dictionary d1;
   declare dnary d2;
```

```
run;
```

Store Your Data

By default, numeric data and character data is stored in dictionaries by value. Arrays, hash objects, hash iterators, ASTORE objects, Python objects, and other dictionaries are stored in dictionaries by reference, which is the default. Data is stored in a dictionary using any of these methods:

Table 8.1 *Data Storage Methods*

Task and Method	Description
Direct Assignment	Data items can be directly stored in a dictionary by assigning a dictionary key to a data item. When using direct assignment, data is automatically stored by value or reference depending on the type of data. In some cases, the default behavior can be overridden using the other storage methods. `d1[key] = your-data;`
Store data by value ("CLONE Method")	The CLONE method forces an array to be stored by value in a dictionary. The CLONE method can also be used to store character data and numeric data in a dictionary. Hash, hash iterators, and other dictionaries cannot be stored by value using the CLONE method. `rc = d1.clone(your-data, key);`
Store data by reference ("REF Method")	The REF method forces numeric data and character data to be stored by reference in a dictionary. By default, numeric data and character data is stored by value. Arrays, hash objects, hash iterators, and other dictionaries can also be stored in a dictionary using the REF method. `rc = d1.ref(your-data, key);`

Example 1: Storing Your Data in a Dictionary

This example shows you all of the methods for storing data items in a dictionary.

```
proc fcmp;
   declare dictionary d1;
   array arr[3] 5 10 15; x = 25; y = 'string';
   d1[key-1] = x;
   rc = d1.clone(arr, key-2);
   rc = d1.ref(x, key-3);
run;
```

Retrieve Your Data

After your data is in a dictionary, there are multiple ways to return that data to a variable outside the dictionary.

- If you know the key that is assigned to a data item, directly assign a variable to the dictionary data item.
- If you want to return dictionary data items without specifying a key, use the other return methods described in Table 7.2 Data Retrieving Methods. These return methods enable you to programmatically return data items based on position in the dictionary. The variable that you specify to return the dictionary data item must match the data type you are copying, or the variable is set to missing.

Data items that are stored in a dictionary are returned using any of these methods:

Table 8.2 *Data Retrieving Methods*

Task and Method	Description
Direct Assignment	Data items in a dictionary can be directly assigned to a new variable by setting the new variable equal to a dictionary key. `a = d1[key];`
Return the first item ("FIRST Method")	The FIRST method copies the first data item in a dictionary to a specified variable and sets the dictionary iterator to the first position. Using the FIRST method initializes the dictionary iterator. `rc = d1.first(data-variable);`
Return the next item ("NEXT Method")	The NEXT method advances the dictionary iterator one position, and then returns a copy of the data item at that position to a specified variable. If the dictionary iterator is not initialized, the NEXT method copies the first item in the dictionary and sets the iterator to the first position. `data-variable = d1.next();`
Return the previous item ("PREV Method")	The PREV method moves the dictionary iterator to the previous position, and then returns a copy of the data item at that position to a specified variable. If the dictionary iterator is not initialized, the PREV method copies the last item in the dictionary and sets the iterator to the last position. `data-variable = d1.prev();`
Return the last item ("LAST Method")	The LAST method copies the last data item in a dictionary to a specified variable and sets the dictionary iterator to the first position. Using the LAST method initializes the dictionary iterator.

Task and Method	Description
	`rc = d1.last(data-variable);`

Example 2: Retrieving Data Items from a Dictionary

This example shows you how to use all of the different data retrieving methods to return data items from a dictionary.

```
proc fcmp;
   declare dictionary d1;
   array arr[3] 5 10 15; array myvar1[3]; array myvar3[3];
   x = 25; y = 'string';
   length myvar2 $6;
   d1[1] = x;
   rc = d1.clone(arr,2);
   rc = d1.ref(y, 3);
   a = d1[1];
   rc = d1.first(myvar1);
   myvar2 = d1.next();
   myvar3 = d1.prev();
   rc = d1.last(myvar4);
   put a= myvar1= myvar2= myvar3= myvar4=;
run;
```

The preceding statements produce these results:

```
a=25 myvar1[1]=5 myvar1[2]=10 myvar1[3]=15 myvar2=string myvar3[1]=5 myvar3[2]=10
myvar3[3]=15
myvar4=25
```

Navigate Your Dictionary

The preceding methods enable you to cycle through dictionary items while also returning copies of the items to variables that you specify. If you want to cycle through dictionary items without returning copies of the items or check whether an item exists, use these methods:

Table 8.3 *Methods to Navigate Your Dictionary*

Task and Method	Description
Check for a next data item	The HASNEXT method looks ahead in a dictionary without advancing the dictionary iterator. The return code indicates whether a next data item exists. If an item exists, the data type can be returned using the optional arguments. When starting from the first

Task and Method	Description
("HASNEXT Method")	data item in a dictionary, the HASNEXT method can be used to indicate when the end of the dictionary has been reached. If the dictionary iterator is not initialized, the HASNEXT method checks for the first item in the dictionary. `item-found = d1.hasnext(data-type-rc,array-rc);`
Check for a previous data item ("HASPREV Method")	The HASPREV method looks behind in a dictionary without moving the dictionary iterator. The return code indicates whether a previous data item exists. If an item exists, attributes can be returned using the optional arguments. When starting from the last data item in a dictionary, the HASPREV method can be used to indicate when the beginning of the dictionary has been reached. If the dictionary iterator is not initialized, the HASPREV method checks for the last item in the dictionary. `item-found = d1.hasprev(data-type-rc, array-rc);`
Move the iterator forward one position ("SKIPNEXT Method")	The SKIPNEXT method advances the iterator one position. The return code indicates whether an item exists at the new position, but no attributes can be returned. `item-found = d1.skipnext();`
Move the iterator backward one position ("SKIPPREV Method")	The SKIPPREV method moves the iterator backward one position. The return code indicates whether an item exists at the new position, but no attributes can be returned. `item-found = d1.skipnext();`

Example 3: Navigate through Your Dictionary and Cycle the Dictionary Iterator

This example shows you how to manipulate the dictionary iterator and check for data items as you approach either end of your dictionary.

```
proc fcmp;
   declare dictionary d1;
   array arr[3] 5 10 15; x = 25; y = 'string';
   HasNext = .; HasPrev = .;
   ItemFound = .; DType = .; IsArr = .;
   d1[1] = x;
   rc = d1.clone(arr,2);
   rc = d1.ref(y, 3);
   rc = d1.first(myvar1);
```

```
   HasNext = d1.hasnext(DType,IsArr);
   put HasNext= Dtype= IsArr=;
   ItemFound = d1.skipnext();
   put ItemFound=;
   ItemFound = d1.skipnext();
   put ItemFound=;
   ItemFound = d1.skipprev();
   put ItemFound=;
   HasPrev = d1.hasprev(DType,IsArr);
   put HasPrev= Dtype= IsArr=;
run;
```

The preceding statements produce these results:

```
HasNext=1 DType=2 IsArr=0
ItemFound=1
ItemFound=1
ItemFound=1
HasPrev=1 DType=1 IsArr=1
```

Replace and Remove Items from Your Dictionary

Assigning a data item to an existing dictionary key replaces the data item at that location. The type of data previously stored at that key does not matter. For example, assigning a character string to a key and later assigning numeric data at the same key requires no additional code or type conversions. Replacing data items can be completed using any of the data storage methods previously discussed (direct assignment, CLONE, and REF). Here is an example.

Replace a Data Item in a Dictionary

```
proc fcmp;
   declare dictionary d1;
   length a $6;
   d1[1] = 'string';
   a = d1[1];
   put a=;
   d1[1] = 25;
   b = d1[1];
   put b=;
run;
```

The preceding statements produce these results:

```
a=string
b=25
```

Table 8.4 Removing Data Items from a Dictionary

Task and Method	Description
Remove a specified data item or multiple data items ("REMOVE Method")	The REMOVE method deletes one or more specified data items from a dictionary. The keys specified and all data stored at the specified keys are deleted from the dictionary. `rc = d1.remove(key-1, key-2);`
Clear all data items ("CLEAR Method")	The CLEAR method deletes all keys and data stored in a dictionary. The dictionary object itself is not deleted and can be used to store new data. `rc = d1.clear();`

Count and Describe the Items in Your Dictionary

The attributes of your dictionary and the attributes of the data items are returned using these methods:

Table 8.5 Identify Dictionary Data Item Attributes

Task and Method	Description
Count the number of items ("NUM_ITEMS Method")	The NUM_ITEMS method returns the number of items stored in a dictionary. If data is being assigned to a dictionary through a function or loop, the NUM_ITEMS method can quickly validate that the number of expected items in the dictionary is correct. `n-items = d1.num_items();`
Describe the data at a specified key or keys ("DESCRIBE Method")	The DESCRIBE method returns what type of data is stored at the specified key or keys and whether that data is an array. Those return values can then be used to programmatically determine what variable data type is needed to return a data item from a dictionary. `data-type-rc = d1.describe(array-rc, key);`

Example 4: Identifying Dictionary Attributes

This example shows you how to return different dictionary attributes.

```
proc fcmp;
   declare dictionary d1;
   array arr[3] 5 10 15;
```

```
      DType = .; IsArr = .;
      x = 25; y = 'string';
      d1[1] = x;
      rc = d1.clone(arr,2);
      rc = d1.ref(y, 3);
      n = d1.num_items();
      DType = d1.describe(IsArr, 2);
      put n= Dtype= IsArr=;
   run;
```

The preceding statements produce these results:

```
n=3 DType=1 IsArr=1
```

9

Dictionary Object Language Elements

Dictionary

CLEAR Method

Removes all key-value pairs from the dictionary.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

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

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

Details

The CLEAR method deletes all keys and data that are stored inside a dictionary object. The dictionary object itself is not deleted and can be used to store new keys and data after clearing.

Example: Clearing a Dictionary Object

```
proc fcmp;
   declare dictionary d;
   Num = .;
   d[1] = 25;
   Num = d.num_items();
   put Num=;
   rc = d.clear();
   Num = d.num_items();
   put Num=;
run;
```

The preceding statements produce these results:

```
Num=1
Num=0
```

CLONE Method

Stores an array by value.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

rc=*object-reference*.**CLONE**(*data*, *key-1* <, ...*key-n*>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

data
specifies the data to pair with the specified key.

key-1
specifies the key value that indicates where data is to be placed. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

Optional Argument

key-n
specifies additional key values to indicate where data is placed. A maximum of six keys can be stored in a dictionary.

Details

For performance reasons, complex data types such as arrays, hash objects, and dictionaries are stored within a dictionary by reference by default. The CLONE method stores an array by value. Dictionaries do not support storing hash objects, hash iterators, and dictionaries by value.

Example

```
proc fcmp;
   declare dnary d;
   array a[10]; array b[10];
   do i= 1 to 10;
    a[i] = i**2;
   end;
   rc = d.clone(a, 5);
   a[1] = 17;
   b = d[5];
   put b=;
run;
```

The preceding statements produce these results:

```
b[1]=1 b[2]=4 b[3]=9 b[4]=16 b[5]=25 b[6]=36 b[7]=49 b[8]=64 b[9]=81 b[10]=100
```

DECLARE Statement: Dictionary Object

Declares a dictionary object.

Alias: DCL

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

DECLARE DICTIONARY *object-reference*;

Required Argument

object-reference
specifies a reference name for the dictionary object.

Details

To create a dictionary object in a PROC FCMP program, use the DECLARE statement with the keyword DICTIONARY. The abbreviation DNARY is also a valid keyword for creating a dictionary object.

Example

```
proc fcmp;
   declare dictionary d1;
   declare dnary d2;
   n1 = d1.num_items();
   n2 = d2.num_items();
   put n1=; put n2=;
run;
```

The preceding statements produce these results:

```
n1=0
n2=0
```

DESCRIBE Method

Returns information about data stored in a dictionary at a location.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

data-type=*object-reference*.**DESCRIBE**(*array-indicator*, *key* <, ...,*key-n*>);

Required Arguments

data-type
specifies a variable that stores the returned data type found as a numeric value or MISSING if there is no data.

object-reference
specifies the name of the object.

array-indicator
specifies a variable that stores the array indicator value. If there is no data, the value is set to MISSING.

key
specifies the key value that indicates which data item to describe.

Details

The DESCRIBE method returns information about data that is stored at a specified location in a dictionary. The return code argument, *data-type*, is set to a numeric value that indicates the data type. The *array-indicator* argument is set to a numeric value to indicate whether the data is an array. The following tables contain information about the return code values for data types and array indicators.

Table 9.1 *Data Type Return Codes*

Return Code	Data Type
1	double
2	char
0	MISSING

Return Code	Data Type
–1	dictionary
–2	hash object
–3	hash iterator
–4	Other object
–5	ASTORE object
–6	Python object

Table 9.2 *Array Indicator Return Codes*

Return Code	Description
1	The data at the specified location is an array.
0	The data at the specified location is not an array.
MISSING	There is no data at the specified location.

Example: Using the DESCRIBE Method to Return Attributes from a Dictionary

```
proc fcmp;
   declare dictionary d;
   delcare dictionary d2;
   DType = .; IsArr = .;
   array Arr{3} 5 10 15;
   d[1] = 25; d[2] = 'char-string'; d[3] = Arr; d[4] = d2;
   do i = 1 to 4;
    DType = d.describe(IsArr, i);
    put DType=;
    put IsArr=;
   end;
run;
```

The preceding statements produce these results:

```
dtype=1
isarr=0
dtype=2
isarr=0
dtype=1
isarr=1
dtype=-1
isarr=0
```

FIRST Method

Copies the first item in a dictionary to a specified variable and sets the iterator to the first position.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

rc=*object-reference*.**FIRST**(*data-variable*<, *data-type*, *array-indicator*>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

data-variable
specifies a variable to store a copy of the first item in the dictionary. If the *data-variable* specified is not the same data type as the first item in the dictionary, the *data-variable* is set to MISSING.

Optional Arguments

data-type
specifies a variable that stores the returned data type found as a numeric value or MISSING if there is no data.

array-indicator
specifies a variable that stores the array indicator value. If there is no data, the value is set to MISSING.

Details

The FIRST method stores a copy of the first data item in a dictionary in a specified variable and sets the dictionary iterator to the first position in the dictionary. The copy is stored in the first argument, *data-variable*. The argument *data-variable* must be the same data type as the dictionary data item or it is set to missing. Additional information about the first data item can be returned by using the optional arguments *data-type* and *array-indicator*.

Note: The data item that is returned by the FIRST method does not correspond to the first item sequentially stored in the dictionary in your program. The order of data items in a dictionary is not sequential and can change as the data items in the dictionary change.

The optional arguments use these return codes:

Table 9.3 *Data Type Return Codes*

Return Code	Data Type
1	double
2	char
0	MISSING
–1	dictionary
–2	hash object
–3	hash iterator
–4	Other object
–5	ASTORE object
–6	Python object

Table 9.4 *Array Indicator Return Codes*

Return Code	Description
1	The data at the specified location is an array.
0	The data at the specified location is not an array.
MISSING	There is no data at the specified location.

Example: Returning the First Item in a Dictionary Using the FIRST Method

```
proc fcmp;
   declare dictionary d;
   DFirst = .; DType = .;IsArr = .;
   d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
   rc = d.first(DFirst, DType, IsArr);
   put DFirst= DType= IsArr=;
run;
```

The preceding statements produce these results:

```
DFirst=534 DType=1 IsArr=0
```

HASNEXT Method

Indicates whether there is a next data item in a dictionary.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

item-found=*object-reference*.**HASNEXT**(<*data-type*, *array-indicator*>);

Required Arguments

item-found
specifies an assignment variable that returns whether a next data item exists in the dictionary. The variable is set to 1 if a next data item exists or 0 if a next data item does not exist in the dictionary.

object-reference
specifies the name of the object.

Optional Arguments

data-type
specifies a variable that stores the returned data type found as a numeric value or MISSING if there is no data.

array-indicator
specifies a variable that stores the array indicator value. If there is no data, the value is set to MISSING.

Details

Note: The HASNEXT method does not change the position of the dictionary iterator.

The HASNEXT method indicates whether a specified dictionary has a next data item beyond the current iterator value. If the iterator is not initialized, the HASNEXT method checks for the first data item in the dictionary. Additional information about the data item can be returned using the optional arguments *data-type* and *array-indicator*. The following tables contain information about return code values for data types and array indicators.

Table 9.5 *Data Type Return Codes*

Return Code	Data Type
1	double
2	char
0	MISSING
–1	dictionary
–2	hash object
–3	hash iterator
–4	Other object
–5	ASTORE object
–6	Python object

Table 9.6 *Array Indicator Return Codes*

Return Code	Description
1	The data at the specified location is an array.
0	The data at the specified location is not an array.
MISSING	There is no data at the specified location.

Example: Using the HASNEXT Method on a Dictionary

```
proc fcmp;
   declare dictionary d;
   FirstItem = .; ItemFound = .; DType = .; IsArr = .;
   d[1] = 25; d[2] = 'string'; d["key"] = 534;
   rc = d.first(FirstItem);
   ItemFound = d.hasnext(DType, IsArr);
   put ItemFound= DType= IsArr=;
run;
```

The preceding statements produce these results:

```
ItemFound=1 DType=2 IsArr=0
```

HASPREV Method

Indicates whether there is a previous data item in a dictionary.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects. on page 217

Syntax

item-found=*object-reference*.**HASPREV**(<*data-type*, *array-indicator*>);

Required Arguments

item-found
specifies an assignment variable that returns whether a previous data item exists in the dictionary. The variable is set to 1 if a previous data item exists or 0 if a previous data item does not exist in the dictionary.

object-reference
specifies the name of the object.

Optional Arguments

data-type
specifies a variable that stores the returned data type found as a numeric value or MISSING if there is no data.

array-indicator
specifies a variable that stores the array indicator value. If there is no data, the value is set to MISSING.

Details

Note: The HASPREV method does not change the position of the dictionary iterator.

The HASPREV method indicates whether a specified dictionary has a previous data item before the current iterator value. If the iterator is not initialized, the HASPREV method checks for the last data item in the dictionary. Additional information about the data item can be returned using the optional arguments *data-type* and *array-indicator*. The following tables contain information about the return code values for data types and array indicators.

Table 9.7 *Data Type Return Codes*

Return Code	Data Type
1	double
2	char
0	MISSING
–1	dictionary
–2	hash object
–3	hash iterator
–4	Other object
–5	ASTORE object
–6	Python object

Table 9.8 *Array Indicator Return Codes*

Return Code	Description
1	The data at the specified location is an array.
0	The data at the specified location is not an array.
MISSING	There is no data at the specified location.

Example: Using the HASPREV Method to Check for a Previous Item in a Dictionary

```
proc fcmp;
   declare dictionary d;
   LastItem = .; ItemFound = .; DType = .; IsArr = .;
   d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
   rc = d.last(LastItem);
   ItemFound = d.hasprev(DType, IsArr);
   put ItemFound= DType= IsArr=;
run;
```

The preceding statements produce these results:

```
ItemFound=1 DType=2 IsArr=0
```

LAST Method

Copies the last data item in a dictionary in a specified variable and sets the iterator to the last position.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects. on page 217

Syntax

rc=*object-reference*.**LAST**(*data-variable* <, *data-type*, *array-indicator*>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

data-variable
specifies a variable to store a copy of the last item in the dictionary. If the *data-variable* specified is not the same data type as the last item, the *data-variable* is set to missing.

Optional Arguments

data-type
specifies a variable that stores the returned data type found as a numeric value or MISSING if there is no data.

array-indicator
specifies a variable that stores the array indicator value. If there is no data, the value is set to MISSING.

Details

The LAST method stores a copy of the last data item in a dictionary in a specified variable and sets the dictionary iterator to the last position in the dictionary. The copy is stored in the argument, *data-variable*. If the specified variable is not the same data type as the last data item in the dictionary, the variable is set to missing. Additional information about the last data item can be returned by using the optional arguments *data-type* and *array indicator*.

Note: The data item that is returned by the LAST method does not correspond to the last item sequentially stored in the dictionary in your program. The order of data items in a dictionary is not sequential and can change as the data items in the dictionary change.

The following tables contain information about the return code values for data types and array-indicators.

Table 9.9 *Data Type Return Codes*

Return Code	Data Type
1	double
2	char
0	MISSING
–1	dictionary
–2	hash object
–3	hash iterator
–4	Other object
–5	ASTORE object
–6	Python object

Table 9.10 *Array Indicator Return Codes*

Return Code	Description
1	The data at the specified location is an array.
0	The data at the specified location is not an array.
MISSING	There is no data at the specified location.

Example: Using the LAST Method to Return a Copy of the Last Item in a Dictionary

```
proc fcmp;
   declare dictionary d;
   DLast = .; DType = .; IsArr = .;
   d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
   rc = d.last(DLast, DType, IsArr);
   put DLast= DType= IsArr=;
run;
```

The preceding statements produce these results:

```
DLast=25 DType=1 IsArr=0
```

NEXT Method

Advances the iterator one position and copies the data item at that position.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

data-variable=*object-reference*.**NEXT**();

Required Arguments

data-variable
specifies a variable to store a copy of the next item in a dictionary. The variable must be the same data type as the dictionary data item or the variable is set to missing.

object-reference
specifies the name of the object.

Details

The NEXT method returns a copy of the next data item in the dictionary to the assignment variable *data-variable*, and advances the iterator to the next position in the dictionary. The *data-variable* specified must be the same data type as the dictionary data item, or the *data-variable* is set to missing. If the iterator is not initialized, the first data item in the dictionary is returned. If no next data item exists in the dictionary, the return variable is set to missing.

Example

```
proc fcmp;
   declare dictionary d;
   length NextVar $8;
   array Arr[5];
   array FirstVar[5];
   do i =1 to 5;
    Arr[i] = i**2;
   end;
   d[1] = 25; d[2] = 'string'; d["key"] = arr;
   rc = d.first(FirstVar);
   put FirstVar=;
   NextVar = d.next();
   put NextVar=;
run;
```

The preceding statements produce these results:

```
FirstVar[1]=1 FirstVar[2]=4 FirstVar[3]=9 FirstVar[4]=16 FirstVar[5]=25
NextVar=string
```

NUM_ITEMS Method

Returns the number of data items in a dictionary.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

number-items=*object-reference*.**NUM_ITEMS**();

Required Arguments

number-items
specifies a variable that stores the number of items returned by the method.

object-reference
specifies the name of the object.

Details

The NUM_ITEMS method returns the number of data items that are stored in a dictionary. The number of items is returned in the assignment variable *number-items*.

Example

```
proc fcmp;
   declare dictionary d;
   n = .;
   d[1] = 25; d[2] = 'string';
   n = d.num_items();
   put n=;
run;
```

The preceding statements produce this result:

```
n=2
```

PREV Method

Returns a copy of the previous data item in a dictionary and moves the iterator to the previous position.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

data-variable=*object-reference*.**PREV**();

Required Arguments

data-variable
specifies a variable to store a copy of the previous data item in a dictionary. The variable must be the same data type as the dictionary data item or the variable is set to missing.

object-reference
specifies the name of the object.

Details

The PREV method stores a copy of the previous data item in the dictionary to the assignment variable *data-variable*, and moves the iterator backward one position if possible. If the iterator is not initialized, the PREV method returns the last data item in the dictionary.

Example

```
proc fcmp;
   declare dictionary d;
   array Arr[5];
   do i =1 to 5;
    Arr[i] = i**2;
   end;
   PrevVar = '      '; LastVar = .;
   d[1] = 25; d[2] = 'string'; d["key"] = Arr;
   rc = d.last();
   put LastVar=;
   PrevVar = d.prev();
   put PrevVar=;
run;
```

The preceding statements produce these results:

```
LastVar=25
PrevVar=string
```

REF Method

Stores numeric and character variables by reference.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

rc=*object-reference*.**REF**(*data*, *key-1* ...< *key-n*>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

data
specifies the data to pair with the specified key.

key-1
specifies the key value that indicates where data is to be placed. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

Optional Argument

key-n
specifies additional key values to indicate where data is placed. A maximum of six keys can be stored in a dictionary.

Details

The REF method stores numeric data and character data by reference. By default, dictionaries store numeric data and character data by value.

Example: Using the REF Method to Store Data by Reference

```
proc fcmp;
   declare dictionary d;
   RefVar =.; NumVar = 25;
   rc = d.ref(NumVar, 1);
   RefVar = d[1];
   put RefVar=;
   NumVar = 30; RefVar = d[1];
   put RefVar=;
```

```
run;
```

The preceding statements produce these results:

```
RefVar= 25
RefVar= 30
```

REMOVE Method

Removes a specified key-value pair from the dictionary.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

rc=*object-reference*.**REMOVE**(*key-1* <, ...,*key-n*>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the name of the object.

key-1
specifies a key to be removed from the dictionary.

Optional Argument

key-n
specifies an additional key or keys to remove from the dictionary.

Details

The REMOVE method deletes a specified key and data pair from a dictionary.

Example: Remove an Item from a Dictionary Using the REMOVE Method

```
proc fcmp;
   declare dictionary d;
   array Arr[5];
   do x = 1 to 5;
    Arr[x] = x**2;
   end;
   n = .;
   d[1] = 25; d[2] = 'string'; d[3] = Arr;
   n = d.num_items();
   put n=;
   rc = d.remove(1);
   put rc=;
   n = d.num_items();
   put n=;
run;
```

The preceding statements produce these results:

```
N=3
rc=0
N=2
```

SKIPNEXT Method

Advances the iterator one position if possible.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

item-found=*object-reference*.**SKIPNEXT**();

Required Arguments

item-found
returns a value that indicates whether a next data item exists in the dictionary. Returns 1 if a next data item exists, or returns 0 if the end of the dictionary is reached.

object-reference
specifies the name of the object.

Details

The SKIPNEXT method advances the iterator one position. The assignment variable *item-found* indicates whether the iterator was advanced one position and whether a next item exists in the dictionary.

Example: Using the SKIPNEXT Method to Advance the Dictionary Iterator and Check for Items in the Dictionary

```
proc fcmp;
   declare dictionary d;
   DFirst = .; ItemFound = .; array Arr[5];
   do x = 1 to 5;
     arr[x] = x**2;
   end;
   d[4] = 25; d[5] = 'string'; d[6] = Arr;
   rc = d.first(DFirst);
   ItemFound = d.skipnext();
   put ItemFound=;
run;
```

The preceding statements produce this result:

```
ItemFound = 1
```

SKIPPREV Method

Moves the iterator to the previous position if possible.

Applies to: Dictionary Object

See: For more information, see Using Dictionary Objects.

Syntax

item-found=*object-reference*.**SKIPPREV**();

Required Arguments

item-found
returns a value that indicates whether a next data item exists in the dictionary. Returns 1 if a next data item exists, or returns 0 if the end of the dictionary is reached.

object-reference
specifies the name of the object.

Details

The SKIPPREV method moves the iterator backward one previous position in the dictionary if possible. The assignment variable *item-found* indicates whether the iterator was moved backward one position and whether a data item exists in the dictionary at that position.

Example: Using the SKIPPREV Method to Move the Iterator Backward and Check for Items in the Dictionary

```
proc fcmp;
   declare dictionary d;
   Dlast = .; ItemFound = .; array Arr[5];
   do x = 1 to 5;
      Arr[x] = x**2;
   end;
   d[4] = 25; d[5] = 'string'; d[6] = Arr;
   rc = d.last(Dlast);
   ItemFound = d.skipprev();
   put ItemFound=;
run;
```

The preceding statements produce this result:

```
ItemFound = 1
```

10

Using Python Objects

Using PROC FCMP Python Objects

Working with Python Objects That Use Stateful Data in a Multi-threaded Environment

Python code that uses stateful data can return unexpected results when executed in a multi-threaded SAS environment. When Python objects are used in a multi-threaded SAS environment, multiple Python interpreter sessions are launched, one per thread, to execute the Python code. The results that are returned in scenarios where multiple Python interpreter sessions are used can be unpredictable if global data or other stateful constructs are used. Caution must be used to avoid stateful Python programs when the Python code is executed in a multi-threaded SAS environment. See “Stateful Python Program” for an example.

Why Use PROC FCMP Python Objects

PROC FCMP Python objects enable you to embed and import Python functions into SAS programs. The Python code is not converted to SAS code. Instead, the Python code runs in the Python interpreter of your choice and returns the results to SAS. With a small Python code modification, you can run your Python functions from SAS and easily program in both languages at the same time.

Requirements

Python objects require environment variables to be set before you can use Python objects in your SAS environment. If the environment variables have not been set, or if they have been set incorrectly, SAS returns an error when you publish your Python code. Environment variable related errors can look like these examples:

```
ERROR: MAS_PYPATH environment variable is undefined.
```

```
ERROR: The executable C:\file-path\python.exe cannot be located
       or is not a valid executable.
```

If you encounter errors when publishing your Python code, contact your site administrator. Site administrators can follow the setup steps listed in the Deployment Variables for Python to enable the use of Python objects in your SAS environment.

Restrictions

The FCMP Python object does not support Python decorators.

Python Function Workflow in PROC FCMP

For individual method syntax descriptions, see Python Object Language Elements.

Here is a typical workflow for using Python objects in PROC FCMP.

1 Declare a Python object and a dictionary object:

   - Use the DECLARE statement to declare an object that is type python. Python objects are used to store Python source code. Here is an example:

     ```
     declare object py(python);
     ```

2 Insert Python source code into SAS:

- Choose a submission method to read Python source code directly into the Python object. Here is an example using the SUBMIT INTO statement:

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(var1, var2):
    "Output: MyOutputKey"
    MyPyResult = var1 * var2
    return MyPyResult
endsubmit;
run;
```

3 Publish Python source code:

- Use the PUBLISH method to submit Python code stored in the Python object to the Python interpreter for compilation. Here is an example:

```
rc = py.publish();
```

4 Call the Python source code:

- Use the CALL method to run a published Python function and store the results in a dictionary. Here is an example:

```
rc = py.call("MyPyFunc", var1, var2);
```

5 Return results from the dictionary:

- Results that are returned by Python functions are stored in dictionaries. By default, the output is stored in the Python object RESULTS member dictionary. To return results from the dictionary, specify the output key from the output string in the Python function as the dictionary key. Here is an example:

```
MyResult = py.results["MyOutputKey"];
```

Using the previous steps, a complete PROC FCMP program using a Python object can look like this example:

```
proc fcmp;
declare object py(python);
submit into py;
def PyProduct(var1, var2):
    "Output: MyKey"
    newvar = var1 * var2
    return newvar,
endsubmit;
rc = py.publish();
rc = py.call("PyProduct", 5, 10);
MyResult = py.results["MyKey"];
put MyResult=;
run;
```

The preceding statements produce this result:

```
MyResult=50
```

Choosing a Submission Method

> **IMPORTANT** All Python code submitted using the Python object submission methods must follow the indention and whitespace character rules required by your Python interpreter.

Python source code can be submitted in PROC FCMP in multiple ways. Using multiple code submissions in one program is permitted. Each additional code submission is appended after the previous submission. To determine which method to use, consider these differences:

- The SUBMIT INTO statement enables users to submit an embedded Python code block or submit the file path to a Python source code file. Using the file path option is different from using the INFILE method. The Python code is stored in the SAS function data set and submitted at FCMP parse time.
- The APPEND method is similar to the SUBMIT INTO embedded method, but the APPEND method submits only a single line of code. The Python code is stored in the SAS function data set and submitted at run time. Because the code is submitted at run time, it is possible to use a variable as the line of code.
- The INFILE method enables users to specify a file path to a Python source code file. Only the file path reference is stored in the SAS function data set, not the Python source code. The reference is stored as a string literal. The code is submitted at FCMP parse time.
- The RTINFILE method is similar to the INFILE method because it stores only the file path reference in the SAS function data set, not the Python source code. However, the code is submitted at run time not FCMP parse time. Because the code is submitted at run time, it is possible to use a variable as the file path reference. The reference is stored as a string literal.

Clear Source Code from a Python Object

Use the CALL CLEAR method to delete source code and file references stored in a Python object and clear the Python results dictionary. The Python object is not deleted, and you can submit new source code to the object.

Using Python Functions in the DATA Step

The Python object and all of the Python object methods are valid only inside a PROC FCMP statement. For example, attempting to declare a Python object in a DATA step program results in an error. However, it is possible to call Python

functions from the DATA step by creating PROC FCMP functions or subroutines that contain Python functions. PROC FCMP functions that contain Python functions are valid in the DATA step and can be called like other functions that are created using PROC FCMP. Here is an example.

```
proc fcmp outlib=work.fcmp.pyfuncs;
function MyPyFunc(FCMParg);
declare object py(python);
submit into py;
def TimesFive(PythonArg):
    "Output: MyKey"
    newvar = PythonArg * 5
    return newvar
endsubmit;
rc = py.publish();
rc = py.call("TimesFive",FCMParg);
MyFCMPResult = py.results["MyKey"];
return(MyFCMPResult);
endsub;
run;

options cmplib=work.fcmp;
data _null_;
   x = MyPyFunc(5);
   put x=;
run;
```

The preceding statements produce this result:

```
x= 25
```

Differences between Python and PROC FCMP Python Objects

Tuples and Output Keys

Python functions that are called by PROC FCMP return values in a Python tuple. The items in the Python tuples are then stored as separate elements in the Python object member dictionary using the RESULTS dictionary or a specified PROC FCMP dictionary. Python functions that return a value or values and are called using the Python object CALL method, must have an output string line as the first line of the Python function body. The output string must begin and end with double quotation marks and must contain only "Output: " followed by a comma-separated list of output keys. The keys correspond to the elements of the tuple that is being returned by the function. The output keys become the dictionary keys that are used to access the output from the Python tuple. There must be an output key for every tuple element item in the return statement. The order of the output keys matches

the order of elements in the Python tuple. This example shows the output string details.

The value of *Tuple_Element1* is stored in the dictionary at *Python_Return_Key1*, and *Tuple_Element2* is stored at *Python_Return_Key2*.

```
def MyFunction(foo):
    "Output: Python_Return_Key1, Python_Return_Key2"
    Tuple_Element1 = foo * 2
    Tuple_Element2 = foo + 2
    return Tuple_Element1, Tuple_Element2
```

To access the output from the function, you can use either the Python object member dictionary or specify a dictionary.

Using the member variable dictionary to access output:

```
My_Output1 = py.results["Python_Return_Key1"];
My_Output2 = py.results["Python_Return_Key2"];
```

Using a specified dictionary to access output:

```
My_Output1 = My_Dictionary["Python_Return_Key1"];
My_Output2 = My_Dictionary["Python_Return_Key2"];
```

Line Size Limit

Python code lines that are submitted with PROC FCMP must be shorter than 255 bytes. If you have lines longer than 255 bytes in your function, they can be continued using the Python line continuation character, "\" (backslash). Here is an example:

```
def MyPythonFunc (arg1, arg2, arg3):
    "Output: MyOutputKey"
    Result = arg1 + arg2 - arg3 + \
    arg2 * arg1
    return Result,
```

Type Conversions

SAS automatically converts data values created by Python code to SAS data types when returning values. The following conversion table lists what Python data types are supported and what the Python data types are converted to in SAS. Python data types not listed in the conversion table are not supported by PROC FCMP.

Table 10.1 *Python to SAS Conversion Table*

Python Data Type	Converted SAS Data Type
NoneType*	Double*

Python Data Type	Converted SAS Data Type
String	Fixed Character
List of Strings**	Fixed Character Array**
Integer	Double
List of Integers**	Double Array**
Long	Double
List of Longs**	Double Array**
Double	Double
List of Doubles**	Double Array**
Date	Double
Date Array**	Double Array**
Time	Double
List of Times**	Double Array**
DateTime	Double
List of DateTimes**	Double Array**
Other	Not Supported

* A Python variable of any type can be changed to the NoneType type. The NoneType has the value None to indicate special conditions such as a variable that has no value. This special condition is similar to how the SAS "missing" values are used to imply that a variable's value has not been set. When a Python function returns None, PROC FCMP does not know whether the None represents "no value" for a double SAS data type or for a character variable. So, PROC FCMP always transforms a returned Python None value to a missing value for the double SAS data type. Therefore, code should be written in a way that avoids trying to store a returned Python None in a SAS character variable.

** Python lists such as "[1, 2, 3]" or "['a', 'b', 'c']" are converted to SAS arrays of the corresponding type, as described in the conversion table. SAS does not support returning Python lists of mixed types such as "[3, 1.5]" or "[4, 'x']". When a Python list is returned to SAS, the SAS array type is determined by the first element in the list that does not have the value "None". All elements of the list are converted to the same SAS data type.

Note: When returning the list "[1, 2.3, 4.01]", SAS receives a list of integers: [1, 2, 4]. If you are returning a list of numbers, you must ensure that unintended truncation does not occur. For example, returning "[float(1), 2.3, 4.01]" avoids the previous truncation.

Working with Date, Datetime, and Time Data in SAS and PROC FCMP Python

When a Python function returns a Python date, datetime, or time object to PROC FCMP, the object is converted to the numeric value that represents the corresponding date, datetime, or time value in SAS. No additional code is needed to pass Python date, datetime, or time objects from Python to SAS.

When passing a SAS date, datetime, or time value to a Python function, the user should ensure that the SAS numeric values are correctly converted to the appropriate Python object. SAS stores dates, datetimes, and times as numeric values that represent a certain unit of time added (or subtracted) from a reference point. For SAS dates, the numeric value represents the number of days since January 1, 1960. For SAS datetimes, the numeric value represents the number of seconds since midnight, January 1, 1960. For SAS times, the numeric value represents the number of seconds since midnight.

In each of these examples, a SAS date, datetime, or time value is passed into a Python function, and the Python function converts the received numeric value to the appropriate Python object and returns a formatted string representation.

Example of converting SAS date values to Python date values:

```
proc fcmp;
length d_str $ 36;
length d_out $ 36;
declare object py(python);
submit into py;
import datetime
def get_date(indate):
   "Output: outdate"
   d = datetime.date(1960, 1, 1) + \
       datetime.timedelta(days=indate)
   return d.strftime('%m/%d/%Y')
endsubmit;
rc = py.publish();
d_in = "22Dec1983"d;
put d_in=;
d_str=put(d_in,mmddyy10.);
put d_str=;
rc = py.call("get_date", d_in);
d_out = py.results["outdate"];
put d_out=;
quit;
```

The preceding statements produce these results:

```
d_in=8756
d_str=12/22/1983
d_out=12/22/1983
```

Example of converting SAS datetime values to Python datetime values:

```
proc fcmp;
length dt_str $ 36;
length dt_out $ 36;
declare object py(python);
submit into py;
import datetime
def get_datetime(indatetime):
   "Output: outdatetime"
   dt = datetime.datetime(1960, 1, 1, 0, 0, 0) + \
        datetime.timedelta(seconds=indatetime)
   return dt.strftime('%d%b%Y:%H:%M:%S')
endsubmit;
rc = py.publish();
dt_in = "22Dec1983:13:52:00"dt;
put dt_in=;
dt_str=put(dt_in,datetime20.);
put dt_str=;
rc = py.call("get_datetime", dt_in);
dt_out = py.results["outdatetime"];
put dt_out=;
quit;
```

The preceding statements produce these results:

```
dt_in=756568320
dt_str=22DEC1983:13:52:00
dt_out=22Dec1983:13:52:00
```

Example of converting SAS time values to Python time values:

```
proc fcmp;
length t_str $ 36;
length t_out $ 36;
declare object py(python);
submit into py;
from datetime import date, datetime, time, timedelta
def get_time(intime):
   "Output: outtime"
   t = ( datetime.combine(date.today(), time()) + \
         timedelta(seconds=intime) ).time()
   return t.strftime('%H:%M:%S')
endsubmit;
rc = py.publish();
t_in = "17:23:45"t;
put t_in=;
t_str=put(t_in,time.);
put t_str=;
rc = py.call("get_time", t_in);
t_out = py.results["outtime"];
put t_out=;
quit;
```

The preceding statements produce these results:

```
t_in=62625
t_str=17:23:45
t_out=17:23:45
```

Commenting

Python functions that are submitted with PROC FCMP support only the hash character (#) commenting method. For example, docstring comments such as `"""This is my comment."""` cause errors. Here is an example of valid commenting Python code in PROC FCMP:

```
proc fcmp;
declare object py(python);
submit into py;
def MyFunc(arg1):
    "Output: MyKey"
    #This is a comment
    ReturnVar = arg1 * 5 #This is another comment
    return ReturnVar
endsubmit;
rc = py.publish();
rc = py.call("MyFunc", 10);
x = py.results["MyKey"];
put x=;
run;
```

The preceding statements produce this result:

```
x= 50
```

Stateful Python Program

The following example is a Python program that uses stateful data, specifically the code `yield x`. As discussed in "Working with Python Objects That Use Stateful Data in a Multi-threaded Environment", when Python programs that use stateful data are executed in a multi-threaded SAS environment, the results can be nondeterministic.

```
function py_pq3(price);
declare object py_pq3_module(python);
rc=0;
submit into py_pq3_module;
def my_generator():
    for x in range(5):
        yield x

shared_gen = my_generator()
```

```
def get_sum_share():
    "Output: sum_share"
    sum = 0
    for x in shared_gen:
        sum += x
    return sum
```

The function containing the `yield` keyword is a generator function declared at a global level. After the function is called, the local state inside the function is retained. In a multi-threaded SAS environment, multiple threads create multiple corresponding Python interpreter sessions (one per thread) to call the global generator function. This results in multiple threads writing to the same global storage, which can lead to unexpected results. This behavior is not limited to the `yield` statement. In this next example, the class variable `Counter` is instantiated at the global level.

```
class Counter(object):
    def __init__(self):
        self.internal_counter = 0
    def increment(self):
        self.internal_counter += 1
    @property
    def count(self):
        return self.internal_counter

myCounter = Counter()

def click():
    "Output: count"
    myCounter.increment()
    return myCounter.count
```

11

Python Object Language Elements

Dictionary

APPEND Method

Appends a line of Python code to a Python object at run time.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

rc=*object-reference*.**APPEND**("*Python-code*");

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the reference name of a Python object.

"Python-code"
specifies a line of Python code to be appended to the Python object. The line of code must be enclosed in double quotation marks and follow standard Python syntax and indention rules.

Details

The APPEND method appends a line of code to existing Python code in a Python object. If no Python code is stored in the object, the APPEND method creates the first line of Python code. Multiple lines of code can be appended to a Python object by using multiple append method statements. Successful appends do not check for valid Python code syntax. Check for correct Python code syntax by using the PUBLISH method. For differences between the APPEND method and other code submission methods, see “Choosing a Submission Method”.

Example: Appending Python Code to a Python Object

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(arg1):
    "Output: MyKey"
    myvar = arg1 * 10
endsubmit;
rc = py.append("    return myvar");
rc = py.publish();
rc = py.call("MyPyFunc", 5);
a = py.results["MyKey"];
put a=;
run;
```

The preceding statements produce this result:

```
a= 50
```

CALL Method

Executes Python functions in stored in Python objects.

Applies to: Python Object

Note: If you execute the CALL method on the CAS server and there is an error, the error message is written to the app.cas.actions.FCMPACT logger instead of the SAS log.

See: For more information, see Using Python Objects.

For more information about CAS logs, see CAS Server Loggers.

Syntax

rc=*object-reference*.**CALL**(<*dictionary-reference*>, "*python-function-name*" <, *python-argument-1*> <, *python-argument-2*, ...>);

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the reference name of a Python object.

"python-function-name"
specifies a Python function name stored inside a Python object. The function name must be enclosed in double quotation marks (" ").

Optional Arguments

dictionary-reference
specifies the reference name of a dictionary object.

python-argument
specifies the argument or arguments, if any, required by the Python function.

Details

The CALL method is used to execute published Python functions that are stored inside a Python object. The name of the Python function, not the Python object, must be specified and enclosed in double quotation marks. The Python function name is case sensitive. If the Python function requires arguments, they must be specified after the function name and separated by commas (,).

Output from the CALL method is stored inside a dictionary object. An optional dictionary object can be specified to store output that is returned from the function. If no dictionary object is specified, the results are stored in the Python object member dictionary and returned using the RESULTS dictionary after the function is called.

Output values are returned in SAS by indexing the dictionary object with the key-name specified in the Python function's output string. The following examples show details about both forms of the CALL method.

Examples

Example 1: Calling a Python Function from a Python Object and Using the Python Object Member Dictionary

```
proc fcmp;
declare object py(python);
x = 5; FCMP_Result = .; rc = 0;

submit into py;
def MyPyFunction(Arg1):
    "Output: MyOutputKey"
    MyPyResult = Arg1 * 10
    return MyPyResult
endsubmit;

rc = py.publish();
rc = py.call("MyPyFunction", x);
FCMP_Result = py.results["MyOutputKey"];
put FCMP_Result=;
run;
```

The preceding statements produce this result:

```
FCMP_Result=50
```

Example 2: Calling a Python Function from a Python Object and Specifying a Dictionary Object

```
proc fcmp;
declare object py(python);
declare dictionary d;
x = 5; FCMP_Result = .; rc = 0;

submit into py;
def MyPyFunction(Arg1):
    "Output: MyOutputKey"
    MyPyResult = Arg1 * 10
    return MyPyResult
endsubmit;

rc = py.publish();
```

```
rc = py.call(d, "MyPyFunction", x);
FCMP_Result = d["MyOutputKey"];
put FCMP_Result=;
run;
```

The preceding statements produce this result:

```
Result=50
```

CALL CLEAR Method

Clears the submit buffer and results dictionary, which enables new code to be published to an existing object.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

CALL *object-reference*.**CLEAR**();

Required Argument

object-reference
specifies the reference name of a Python object.

Details

The CALL CLEAR method clears Python source code that is stored in the submit buffer by using the PUBLISH method and all results stored in the RESULTS dictionary. The Python object remains and can be used to submit new Python source code. A successful clear of the submit buffer enables users to submit new Python source code into an existing Python object and publish that source code to the submit buffer.

Example: Clear a Python Object

```
proc fcmp;
   declare object py(python);
   rc = py.publish();
   call py.clear();
run;
```

The CALL CLEAR method returns no output.

DECLARE Statement: Python Object

Declares a Python object.

Alias: DCL

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

DECLARE OBJECT *object-reference*(**PYTHON**<("*module-name*")>);

Required Argument

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

Optional Argument

"module-name"
specifies a module name stored inside the Python object.

Note: This is not the name of the function and does not create a function.

Details

The DECLARE statement is used to create a Python object and name the Python module. Python objects can be declared only in a PROC FCMP program.

Example

```
proc fcmp;
   declare object py(python("MyPyModule"));
   declare object py2(python);
run;
```

There is no output returned from declaring Python objects.

INFILE Method

Reads Python source code from a file into a Python object at parse time.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

rc=*object-reference*.**INFILE**("*file-path*");

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the reference name of a Python object.

"file-path"
specifies the file path to a local Python file. The file path value must be enclosed in double quotation marks (" ").

Details

The INFILE method reads Python source code from a local file into a Python object. The source code in the file must follow Python syntax and indention rules. Source code must be modified with an output string on page 253 after the Python function definition statement. For differences between the INFILE method and other code submission methods, see "Choosing a Submission Method".

Example: Read Python Source Code into a Python Object with the INFILE Method

In this example, we read Python source code from a file, "MyPyFunction.py".

```
def MyPyFunc(arg1):
    "Output: MyOutputKey"
    PyResult = arg1 * 10
    return PyResult
```

Using the file path to this file, you can submit and call the Python function from PROC FCMP.

```
proc fcmp;
   declare object py(python);
   rc = py.infile("Your-File-Path/MyPyFunction.py");
   put rc=;
   rc = py.publish();
   rc = py.call("MyPyFunc", 5);
   Result = py.results["MyOutputKey"];
   put Result=;
run;
```

The preceding statements produce these results:

```
rc=0
Result=50
```

PUBLISH Method

Submits Python source code to the Python interpreter in PROC FCMP.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

rc=*object-reference*.**PUBLISH**();

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the reference name of a Python object.

Details

The PUBLISH method submits all the Python source code that is stored in a Python object to the Python interpreter. Python source code can be loaded into a Python object using these methods and statements:

- APPEND Method
- INFILE Method

- RTINFILE Method
- SUBMIT INTO Statement

The PUBLISH method must be run before Python source code can be executed using the CALL method on a Python object. Python code syntax errors cause the PUBLISH method to fail and return an error to the log.

Example: Use the PUBLISH Method to Submit Python Source Code to the Python Interpreter

```
proc fcmp;
declare object py(python);
rc = 0;

submit into py;
def MyPyFunction(Arg1):
    "Output: MyOutputKey"
    MyPyResult = Arg1 * 10
    return MyPyResult
endsubmit;

rc = py.publish();
run;
```

There is no output returned by the PUBLISH method.

RESULTS Dictionary Method

Returns results from a Python function call using the member variable dictionary.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

Form 1: *data-variable*=*object-reference*.**RESULTS**["*output-key*"];

Form 2: *rc*=*object-reference*.**RESULTS**.**CLEAR**();

Required Arguments

data-variable
specifies a variable to store a copy of the output at the specified key from the RESULTS dictionary.

object-reference
specifies the reference name of a Python object.

"output-key"
specifies the output key name for a Python function return value.

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

Details

The RESULTS dictionary is used to access output from a called Python function without declaring a dictionary object. When no dictionary object is specified in a "CALL Method", the CALL method stores output in the RESULTS member dictionary. Results are accessed from the member dictionary by specifying the output key on page 253 for the desired return value.

The output that is stored in the RESULTS member dictionary is cleared by using the dictionary "CLEAR Method" on page 225 shown in the Form 2 of the syntax. Appending .CLEAR to the RESULTS dictionary removes all data items from the dictionary.

Note: The CALL method automatically overwrites same name data items in the RESULTS dictionary. Clearing the RESULTS dictionary is not required to run the same function and return new results.

Example: Use the RESULTS Method to Return Results from a Python Function

```
proc fcmp;
   declare object py(python);
   x = 10; myresult = .; rc = 0;
   submit into py;
   def MyFunc(arg1):
       "Output: MyPyResult"
       result = arg1 * 2
       return result
   endsubmit;
   rc = py.publish();
   rc = py.call("MyFunc", x);
   myresult = py.results["MyPyResult"];
   put myresult=;
quit;
```

The preceding statements produce this result:

```
myresult= 20
```

RTINFILE Method

Reads Python source code from a file into a Python object at run time.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

rc=*object-reference*.**RTINFILE**("*file-path*");

Required Arguments

rc
specifies whether the method succeeded or failed. A return code of 0 indicates success; a nonzero value indicates failure.

object-reference
specifies the reference name of a Python object.

file-path
specifies the file path to a Python file on the server. The file path value must be enclosed in double quotation marks.

Details

The RTINFILE method is used in client/server environments to read Python source code from files on the server. The source code must follow Python syntax and indention rules. Source code must be modified with an output string after the Python function definition in PROC FCMP. For differences between the RTINFILE method and other code submission methods, see "Choosing a Submission Method".

Example: Reading Python Source Code from a File on a Server

In this example, we read Python source code from a file, "MyPyFunction.py".

```
def MyPyFunc(arg1):
    "Output: MyOutputKey"
    PyResult = arg1 * 10
    return PyResult
```

Using the file path to this file, you can submit and call the Python function from PROC FCMP.

```
proc fcmp;
   declare object py(python);
   rc = py.rtinfile("Your-File-Path/MyPyFunction.py");
   put rc=;
   rc = py.publish();
   rc = py.call("MyPyFunc", 5);
   Result = py.results["MyOutputKey"];
   put Result=;
run;
```

The preceding statements produce these results:

```
rc=0
Result=50
```

SUBMIT INTO Statement

Inserts Python source code into a Python object at parse time.

Applies to: Python Object

See: For more information, see Using Python Objects.

Syntax

Form 1: **SUBMIT INTO** *object-reference*; *...python-code...* **ENDSUBMIT;**

Form 2: **SUBMIT INTO** *object-reference*("*file-path*");

Required Arguments

object-reference
specifies the reference name of a Python object.

"file-path"
specifies the file path to a file that contains Python source code.

Details

The SUBMIT INTO statement is used to insert Python source code into a Python object with either embedded Python code or a file that contains Python code. Both methods require the Python code to be modified with an output string after the function definition to run in PROC FCMP. The embedded submission block creates a source code submission block for users to embed Python functions in their SAS

programs. Python functions that are written in the embedded submission block can be called to return results. To end a submission block, use the ENDSUBMIT statement. The SUBMIT INTO statement also supports Python file submissions by using the second form and supplying a *file-path* argument that is similar to the INFILE method.

Note: If you are using the "*file-path*" form, do not use the ENDSUBMIT statement. Using the ENDSUBMIT statement after specifying a file path causes an error.

When submitting embedded code, the lines between the SUBMIT INTO and ENDSUBMIT statements are white space sensitive. The lines are read one at a time and sequentially stored into the specified Python object's submit buffer. Do not indent the SUBMIT INTO and ENDSUBMIT statements when using the embedded Python code block. These statements must start at the first (left most) position in your SAS Program Editor. Standard rules for Python syntax and indention apply to source code written in the embedded submission block. The embedded submission block does not detect Python syntax errors. Errors are detected when Python objects are submitted to the Python interpreter using the PUBLISH method. For differences between the SUBMIT INTO statement and other code submissions methods, see "Choosing a Submission Method" on page 252.

Examples

Example 1: Submit Python Source Code Using the Embedded Submission Block Form

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(Arg1):
    "Output: MyOutputKey"
    MyPyResult = Arg1 * 10
    return MyPyResult
endsubmit;
run;
```

There is no output returned by these statements.

Example 2: Submit Python Source Code Using the File Submission Form

In this example, we read the same Python source code from a file, "MyPyFunction.py". The file contains this code:

```
def MyPyFunc(Arg1):
    "Output: MyOutputKey"
    MyPyResult = Arg1 * 10
    return MyPyResult
```

```
proc fcmp;
   declare object py(python);
```

```
   submit into py("Your-File-Path/MyPyFunction.py");
run;
```

There is no output returned by these statements.