SAS® Viya™ コンポーネントオブジェクト: リファレンス

SAS® ドキュメント

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Viya™ コンポーネントオブジェクト: リファレンス*. Cary, NC: SAS Institute Inc.

SAS® Viya™ コンポーネントオブジェクト: リファレンス

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

September 2016

3.1-P1:lecompobjref

目次

Early Adopter Software

---

1 章
SAS コンポーネントオブジェクトについて

DATA ステップコンポーネントオブジェクトについて

SAS では、DATA ステップで使用するために事前定義された、次の 5 つのコンポーネントオブジェクトが用意されています。

ハッシュオブジェクトとハッシュ反復子オブジェクト
: ルックアップキーに基づいて、データをすばやく効率的に保存、検索および取得できます。ハッシュオブジェクトキーとデータは DATA ステップ変数です。SAS データセットからの定数値をキーとデータ値に直接割り当てられます。ハッシュオブジェクトとハッシュ反復子オブジェクト言語要素についての詳細は、2 章, “ハッシュオブジェクトとハッシュ反復子オブジェクトの言語要素のディクショナリ” (41 ページ)を参照してください。

Java オブジェクト
: Java クラスをインスタンス化して結果オブジェクトのフィールドとメソッドにアクセスする、Java ネイティブインターフェイス (JNI)に似たメカニズムを提供します。詳細については、3 章, “Java オブジェクトの言語要素のディクショナリ” (111 ページ)を参照してください。

DATA ステップコンポーネントインタフェイスでは、ステートメント、属性、演算子、メソッドを使用するコンポーネントオブジェクトを作成して操作できます。コンポーネントオブジェクトの属性とメソッドにアクセスするには、DATA ステップオブジェクトのドット表記を使用します。ドット表記と DATA ステップオブジェクトのステートメント、属性、メソッド、演算子の詳細については、*SAS Viya コンポーネントオブジェクト: リファレンス*のコンポーネント言語要素のディクショナリを参照してください。

注: DATA ステップコンポーネントオブジェクトのステートメント、属性、メソッド、演算子は、これらのオブジェクト用に定義されている場合のみ使用できます。これらの事前定義された DATA ステップオブジェクトで SAS Component Language 機能を使用することはできません。

ドット表記と DATA ステップコンポーネントオブジェクト

定義

ドット表記は、メソッド呼び出しおよび属性値の設定とクエリに対するショートカットとして使用できます。ドット表記を使用することで、SAS プログラムが読みやすくなります。

DATA ステップコンポーネントオブジェクトでドット表記を使用するには、DECLARE ステートメントのみを使用するか、DECLARE ステートメントと_NEW_演算子を組み合わせて使用し、コンポーネントオブジェクトを宣言してインスタンス化する必要があります。

構文

ドット表記の構文を次に示します。

object.attribute

または

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

引数は次のように定義されます。

object
DATA ステップコンポーネントオブジェクトの変数名を指定します。

attribute
割り当てまたはクエリ用のオブジェクト属性を指定します。

オブジェクトの属性を設定する場合、コードの形式は次のようになります。

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

オブジェクトの属性をクエリする場合、コードの形式は次のようになります。

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

method
呼び出すメソッド名を指定します。

argument_tag
メソッドに渡される引数を識別します。引数タグはかっこで囲みます。メソッドに引数タグが含まれるかどうかに関わらず、かっこは必須です。

すべての DATA ステップコンポーネントオブジェクトのメソッドの形式は次のようになります。

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

リターンコードは、メソッドの成功または失敗を示します。ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

value
引数の値を指定します。

ハッシュオブジェクトの使用

ハッシュオブジェクトを使用する理由

ハッシュオブジェクトは、迅速なデータの保存および取得のための効率的かつ便利なメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

DATA ステップコンポーネントオブジェクトインターフェイスを使用するには、次の操作を行います。

1. ハッシュオブジェクトを宣言します。
2. ハッシュオブジェクトのインスタンスを作成(インスタンス化）します。
3. ルックアップキーとデータを初期化します。

ハッシュオブジェクトを宣言してインスタンス化した後に、次を始めとする、多くのタスクを実行できます。

- データの保存と取得
- キー集計の維持
- データの置換と削除
- ハッシュオブジェクトの比較
- ハッシュオブジェクトのデータを含むデータセットの生成

たとえば、重複しない患者の番号と体重に対応する、数値の検査結果を含む大規模なデータセットがあるとします。また、患者番号を含む小規模なデータセット(大規模なデータセットの検査結果のサブセト)があるとします。重複しない患者番号をキーに使用し、体重の値をデータに使用して、ハッシュオブジェクトに大規模なデータセットをロードできます。小規模なデータセットを介するシングルパスが作成され、患者番号を使用して、体重がある値を上回る現在の患者がハッシュオブジェクトでルックアップされます。そして、患者データは別のデータセットに書き込まれます。

ルックアップキーの数とデータセットのサイズによりますが、ハッシュオブジェクトのルックアップは、標準形式のルックアップよりも大幅に高速化されます。キーのみ検索し、メモリが大量にあり、パフォーマンスの高速化が必要な場合は、最初に大規模なデータセットをロードします。大量のメモリを使用しないようにする場合は、小規模なデータセットを最初にロードします。

ハッシュオブジェクトの宣言とインスタンス化

DECLARE ステートメントを使用して、ハッシュオブジェクトを宣言します。新しいハッシュオブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。次はその一例です。

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

DECLARE ステートメントは、コンパイラにオブジェクト MyHash の種類はハッシュということを示します。ここでは、種類がハッシュのコンポーネントオブジェクトを保持する可能性があるオブジェクト参照 MyHash のみ宣言しています。ハッシュオブジェクトの宣言は一度のみ行います。_NEW_演算子でハッシュオブジェクトのインスタンスが作成され、このハッシュオブジェクトがオブジェクト参照 MyHash に割り当てられます。

DECLARE ステートメントと_NEW_演算子を使用してコンポーネントオブジェクトを宣言してインスタンス化する 2 ステップのプロセス以外の方法もあります。DECLARE ステートメントを使用して、1 ステップでコンポーネントオブジェクトを宣言してインスタンス化できます。

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

The preceding statement is equivalent to this code:

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

詳細については、“DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)および“_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)を参照してください。

コンストラクタを使用したハッシュオブジェクトデータの初期化

ハッシュオブジェクトの作成時、初期化データの提供が必要な場合があります。コンストラクタは、ハッシュオブジェクトをインスタンス化し、ハッシュオブジェクトデータを初期化するメソッドです。

ハッシュオブジェクトのコンストラクタは、次のいずれかの形式になります。

- 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*>);

詳細については、“DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)および“_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)を参照してください。

キーとデータの定義

ハッシュオブジェクトはルックアップキーを使用して、データの保存と取得を行います。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。すべてのキー変数とデータ変数を定義した後、DEFINEDONE メソッドが呼び出されます。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

たとえば、次のコードは、文字キー変数や文字データ変数を初期化します。

```
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;
```

複数のキー変数とデータ変数を保有できますが、MULTIDATA:“YES”引数タグ付きのハッシュオブジェクトを作成しない場合はキーは重複しない必要があります。詳細については、“重複するキーとデータペア” (6 ページ)を参照してください。

特定の 1 つのキーを使用して、複数のデータ項目を保存できます。たとえば、前の例を変更して、補助数値を文字キーと文字データと一緒に保存できるようにできます。この例では、キーとデータ項目は、文字値と数値から構成されます。

```
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;
```

詳細については、“DEFINEDATA メソッド” (56 ページ), “DEFINEDONE メソッド” (58 ページ)および“DEFINEKEY メソッド” (59 ページ)を参照してください。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')**)に割り当てないため、SAS コンパイラはハッシュオブジェクトとハッシュ反復子が実行するデータ変数割り当てを検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- すべてのキー変数とデータ変数をパラメータにして、CALL MISSING ルーチンを使用します。次に例を示します。

```
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;
```

重複するキーとデータペア

デフォルトでは、ハッシュオブジェクトのキーのすべては 重複しません。つまり、キーごとに 1 つのデータ変数のセットが存在します。ハッシュオブジェクト内で重複したキーが必要な場合もあります。つまり、データ変数の複数のセットを 1 つのキーに関連付けることが必要になることがあります。

たとえば、キーが患者 ID であり、データは来院日であるとします。患者が複数回来院した場合、複数の日付が患者 ID に関連付けられます。MULTIDATA:“YES” 引数タグ付きのハッシュオブジェクトを作成すると、データ変数の複数のセットが 1 つのキーに関連付けられます。

データセットに重複するキーが含まれている場合、デフォルトでは、最初のインスタンスがハッシュオブジェクトに保存され、次のインスタンスは無視されます。ハッシュオブジェクトに最後のインスタンスを保存するには、DUPLICATE 引数タグを使用します。また、DUPLICATE 引数タグは、重複したキーが存在する場合に SAS ログにエラーを書き込みます。

しかし、DECLARE ステートメントまたは_NEW_演算子に MULTIDATA 引数タグを使用する場合、ハッシュオブジェクトでは各キーに複数の値を保存できます。ハッシュオブジェクトは、キーに関連付けられているリストに複数の値を保持します。このリストは、HAS_NEXT や FIND_NEXT などのメソッドを使用して移動や操作を行えます。

複数のデータ項目を含むリスト内を移動するには、現在のリスト項目を確認する必要があります。開始するには、キーを指定して FIND メソッドを呼び出します。FIND メソッドによって、現在のリスト項目が設定されます。そして、キー

に複数のデータ値があるかを確認するには、HAS_NEXT メソッドを呼び出します。キーが別のデータ値を持つことを確認した後、FIND_NEXT メソッドでその値を取得できます。FIND_NEXT メソッドは、現在のリスト項目をリスト内の次の項目に設定し、その項目のデータ変数を設定します。

指定されたキーのリストを前方に移動する以外に、同じように HAS_PREV メソッドと FIND_PREV メソッドを使用して、後方にリストをループできます。

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

注: 複数のデータ項目を含むリストの項目は、挿入順序で保持されます。

重複キーとデータペアに関連付けられたこれらのメソッドや他のメソッドの詳細については、2 章, “ハッシュオブジェクトとハッシュ反復子オブジェクトの言語要素のディクショナリ” (41 ページ)を参照してください。

データの保存と取得

データの保存法と取得法

ハッシュオブジェクトのキー変数とデータ変数を初期化した後に、ADD メソッドを使用してハッシュオブジェクトにデータを保存したり、*dataset* 引数タグを使用してハッシュオブジェクトにデータセットをロードしたりできます。*dataset* 引数タグを使用する際にデータセットに同じキー値を持つ複数のオブザベーションが含まれている場合、デフォルトでは、SAS はハッシュテーブル内の最初のオブザベーションを保持し、次のオブザベーションは無視します。重複するキーがある場合に、ハッシュオブジェクトの最後のインスタンスを保存するか、またはログにエラーを書き込むには、DUPLICATE 引数タグを使用します。キーの重複値を許可するには、MULTIDATA 引数タグを使用します。

次に、キーに 1 つのデータ値が存在する場合、FIND メソッドを使用して、ハッシュオブジェクトからデータを検索して取得できます。キーに複数のデータ項目が存在する場合、FIND_NEXT メソッドと FIND_PREV メソッドを使用して、データを検索して取得します。

詳細については、“ADD メソッド” (42 ページ)、“FIND メソッド” (64 ページ)、“FIND_NEXT メソッド” (67 ページ)、“FIND_PREV メソッド” (68 ページ)を参照してください。

REF メソッドを使用して、FIND メソッドと ADD メソッドを連結できます。この例のコードの量は縮約できます。

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

ここでは、このコードは 1 つのメソッド呼び出しに縮約されます。

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

詳細については、“REF メソッド” (90 ページ)を参照してください。

注: また、ハッシュ反復子オブジェクトを使用し、ハッシュオブジェクトのデータを一度に 1 つのデータ項目づつ、前方または後方の順で取得できます。詳細については、“ハッシュ反復子オブジェクトの使用 ” (16 ページ)を参照してください。

例 1: ADD メソッドと FIND メソッドを使用したデータの保存と取得

次の例では、ADD メソッドを使用して、ハッシュオブジェクトにデータを保存してキーとデータを関連付けます。次に FIND メソッドを使用して、キー値 **Homer** に関連付けられているデータを取得します。

```
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();
end;

/* Define constant value for key and data */
k = 'Homer';
d = 'Odyssey';
/* Use the ADD method to add the key and data to the hash object */
rc =h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Define constant value for key and data */
k = 'Joyce';
d = 'Ulysses';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

k = 'Homer';
/* Use the FIND method to retrieve the data associated with 'Homer' key */
rc = h.find();
if (rc = 0) then
  put d=;
else
  put 'Key Homer not found.';
run;
```

FIND メソッドは、データ値 **Odyssey** を変数 D に割り当てます。このデータ値は、キー値 **Homer** に関連付けられています。

例 2: データセットのロードと FIND メソッドを使用したデータの取得

データセット Small には 2 つの数値変数 K (キー)と S (データ)が含まれ、もう 1 つのデータセット Large にはキー変数 K が含まれているとします。次のコードは Small データセットをハッシュオブジェクトにロードし、Large データセットの変数 K に一致するキーをハッシュオブジェクトから検索します。

```
data match;
  length k 8;
  length s 8;
  if _N_ = 1 then do;
    /* load SMALL data set into the hash object */
    declare hash h(dataset: "work.small");
```

```
    /* define SMALL data set variable K as key and S as value */
    h.defineKey('k');
    h.defineData('s');
    h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, s);
  end;

/* use the SET statement to iterate over the LARGE data set using */
/* keys in the LARGE data set to match keys in the hash object */
set large;
rc = h.find();
if (rc = 0) then output;
run;
```

DEFINEDONE メソッの処理中に、*dataset* 引数タグによって、キーとデータがハッシュオブジェクトによって読み込まれてロードされた Small データセットが指定されます。次に、FIND メソッドを使用して、データが取得されます。

キー集計の維持

ハッシュオブジェクトの宣言時に SUMINC 引数タグを使用すると、ハッシュオブジェクトキーの集計数を維持できます。タグ値は、数値 DATA ステップ変数 (SUMINC 変数)の名前に解決される文字列式です。

この SUMINC 引数タグによって、ハッシュオブジェクトにキーの集計値を維持する内部ストレージを割り当てるように示されます。

ADD メソッドまたは REPLACE メソッドが使用されると、常にハッシュキーの集計値は初期化され、SUMINC 変数の値にされます。

FIND メソッド、CHECK メソッド、または REF メソッドが使用されると、常にハッシュキーの集計値に SUMINC 変数の値が加算されます。

SUMINC 変数は、負、正、またはゼロに評価されることがあります。変数は整数である必要はありません。キーの SUMINC 値は、デフォルトではゼロです。

次の例では、ADD の前に、最初の ADD メソッドによって K=99 の集計数が 1 に設定されます。.新しい COUNT 値が指定されると、常に FIND メソッドはキー集計にこの新しい値を加算します。この例では、キーごとに 1 つのデータ値が存在します。SUM メソッドは、キー集計の現在値を取得します。値は、DATA ステップ変数 TOTAL に保存されます。キーに複数の値が存在する場合、SUMDUP メソッドはキー集計の現在値を取得します。

```
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;
```

この例では、キー値 K=99 と K=100 の集計が維持されます。

```
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;
```

最初の PUT ステートメントは K=100 の集計を出力します。

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

2 番目の PUT ステートメントは K=99 の集計を出力します。

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

キー集計は、*dataset* 引数タグと組み合わせて使用できます。DEFINEDONE メソッドを使用してデータセットをハッシュオブジェクトに読み込み、すべてのキー集計を SUMINC 値に設定します。すべての後続の FIND、CHECK、ADD メソッドは、キー集計を変更します。

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

キー集計を使用して、指定のキーの出現数をカウントできます。次の例では、データセット MyData はハッシュオブジェクトにロードされ、キー集計を使用してデータセット Keys のキーの出現数がカウントされます。(SUMINC 変数に値が設定されていないため、デフォルト初期値の 0 が使用されます。.)

```
data mydata;
 input key;
datalines;
1
2
3
4
5
;
run;


data keys;
 input key;
datalines;
1
2
1
3
5
2
3
2
4
1
5
1
;
run;

data count;
 length total key 8;
 keep key total;

 declare hash myhash(suminc: "count", dataset:"mydata");
 myhash.defineKey('key');
 myhash.defineDone();
 count = 1;

 do while (not done);
  set keys end=done;
  rc = myhash.find();
 end;

 done = 0;
```

```
do while (not done);
  set mydata end=done;
  rc = myhash.sum(sum: total);
  output;
end;
stop;
run;
```

次にその結果のデータセットの出力を示します。

アウトプット 1.1 キー集計の出力

The SAS System

Obs	total	key
1	4	1
2	3	2
3	2	3
4	1	4
5	2	5

注: DECLARE ステートメントまたは_NEW_演算子の KEYSUM コンストラクタは、すべてのキーのキー集計をトラックする変数を宣言します。KEYSUM 変数は出力データセットの一部であり、キーに 1 つ以上のデータ項目が存在する場合に機能します。

詳細については、“SUM メソッド” (104 ページ)および “SUMDUP メソッド” (106 ページ)を参照してください。

ハッシュオブジェクトのデータの置換と削除

次のいずれかのメソッドを使用して、ハッシュオブジェクトに保存されたデータを置換または削除できます。

- REMOVE メソッドを使用して、すべてのデータ項目を削除します。
- REPLACE メソッドを使用して、すべてのデータ項目を置換します。
- REMOVEDUP メソッドを使用して、現在のデータ項目のみ削除します。
- REPLACEDUP メソッドを使用して、現在のデータ項目のみ置換します。

次の例では、REPLACE メソッドはデータ **Odyssey** を **Iliad** で置換し、REMOVE メソッドはハッシュオブジェクトから **Joyce** キーに関連付けられたデータエントリ全体を削除します。

```
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();
end;

/* Define constant value for key and data */
k = 'Joyce';
d = 'Ulysses';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Define constant value for key and data */
k = 'Homer';
d = 'Odyssey';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Use the REPLACE method to replace 'Odyssey' with 'Iliad' */
k = 'Homer';
d = 'Iliad';
rc = h.replace();
if (rc = 0) then
  put d=;
else
  put 'Replace not successful.';

/* Use the REMOVE method to remove the 'Joyce' key and data */
k = 'Joyce';
rc = h.remove();
if (rc = 0) then
  put k 'removed from hash object';
else
  put 'Deletion not successful.';

run;
```

次の行が SAS ログに書き込まれます。

```
d=Iliad
Joyce removed from hash object
```

注: 関連付けられているハッシュ反復子がキーを参照している場合、REMOVE メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージがログに発行されます。

詳細については、“REMOVE メソッド” (92 ページ), “REMOVEDUP メソッド” (95 ページ)、“REPLACE メソッド” (97 ページ)、および“REPLACEDUP メソッド” (99 ページ)を参照してください。

データセットにハッシュオブジェクトデータを保存

OUTPUT メソッドを使用して、指定したハッシュオブジェクトのデータを含む出力データセットを作成できます。次の例では、2 つのキーとデータがハッシュオブジェクトに追加され、Work.Out データセットに書き込まれます。

```
options pageno=1 nodate;

data test;
length d1 8;
length d2 $20;
length k1 $20;
length k2 8;

/* Declare the hash object and two key and data variables */
if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k1', 'k2');
  rc = h.defineData('d1', 'd2');
  rc = h.defineDone();
end;

/* Define constant value for key and data */
k1 = 'Joyce';
k2 = 1001;
d1 = 3;
d2 = 'Ulysses';
rc = h.add();

/* Define constant value for key and data */
k1 = 'Homer';
k2 = 1002;
d1 = 5;
d2 = 'Odyssey';
rc = h.add();

/* Use the OUTPUT method to save the hash object data to the OUT data set */
rc = h.output(dataset: "work.out");
run;

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

出力 1.2 には、PROC PRINT が生成したレポートが示されています。

アウトプット 1.2 ハッシュオブジェクトから作成されたデータセット

The SAS System

Obs	d1	d2
1	5	Odyssey
2	3	Ulysses

ハッシュオブジェクトキーは、出力データセットの一部として自動的には保存されません。キーは、DEFINEDATA メソッドを使用してデータ項目として定義し、出力データセットに含めることができます。また、データ項目が DEFINEDATA メソッドを使用して定義されていない場合、キーが OUTPUT メソッドで指定さ

れたデータセットに書き込まれます。前述の例では、DEFINEDATA メソッドは次のように記述されます。

```
rc = h.defineData('k1', 'k2', 'd1', 'd2');
```

詳細については、“OUTPUT メソッド” (83 ページ)を参照してください。

ハッシュオブジェクトの比較

EQUALS メソッドを使用して、1 つのハッシュオブジェクトを別のハッシュオブジェクトと比較できます。次の例では、2 つのハッシュオブジェクトが比較されます。EQUALS メソッドには、2 つの引数タグがあります。HASH 引数タグは、2 番目のハッシュオブジェクトの名前です。RESULTS 引数は、比較結果を保有する変数名です(等しい場合は 1、等しくない場合はゼロ)。

```
length eq k 8;

declare hash myhash1();
myhash1.defineKey('k');
myhash1.defineDone();

declare hash myhash2();
myhash2.defineKey('k');
myhash2.defineDone();

rc = myhash1.equals(hash: 'myhash2', result: eq);
```

詳細については、“EQUALS メソッド” (63 ページ)を参照してください。

ハッシュオブジェクト属性の使用

DATA ステップコンポーネントインターフェイスでは、属性を使用してハッシュオブジェクトから情報を取得できます。属性には、次の構文を使用します。

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

ハッシュオブジェクトには 2 つの属性を使用できます。NUM_ITEMS はハッシュオブジェクトの項目数を返し、ITEM_SIZE は項目のサイズ(バイト)を返します。この例では、ハッシュオブジェクトの項目数を取得します。

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

この例では、ハッシュオブジェクトの項目のサイズを取得します。

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

ITEM_SIZE 属性と NUM_ITEMS 属性から、ハッシュオブジェクトが使用しているメモリ量を概算できます。ITEM_SIZE 属性は、ハッシュオブジェクトが必要とする初期オーバーヘッドを反映せず、必要な内部配置も考慮しません。ITEM_SIZE は正確なメモリ使用を提供しませんが、優れた概算を提供します。

詳細については、“NUM_ITEMS 属性 ” (82 ページ)および“ITEM_SIZE 属性” (74 ページ)を参照してください。

ハッシュ反復子オブジェクトの使用

ハッシュ反復子オブジェクトについて

ハッシュ反復子オブジェクトを使用して、ルックアップキーに基づきデータを保存して検索します。ハッシュ反復子オブジェクトにより、ハッシュオブジェクトのデータを前向きまたは逆向きのキー順序で取得できます。

ハッシュ反復子オブジェクトの宣言とインスタンス化

DECLARE ステートメントを使用して、ハッシュ反復子オブジェクトを宣言します。新しいハッシュ反復子オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。引数タグとしてハッシュオブジェクト名を使用します。次に、例を示します。

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

DECLARE ステートメントは、オブジェクト参照 MyIter の種類がハッシュ反復子であることをコンパイラに示します。ここでは、種類がハッシュ反復子のコンポーネントオブジェクトを保持する可能性があるオブジェクト参照 MyIter のみ宣言しています。ハッシュ反復子オブジェクトの宣言は一度のみ行います。_NEW_演算子でハッシュ反復子オブジェクトのインスタンスが作成され、このハッシュ反復子がオブジェクト参照 MyIter に割り当てられます。ハッシュオブジェクト H がコンストラクタ引数として渡されます。つまり、ハッシュオブジェクト変数ではなく、ハッシュオブジェクトがハッシュ反復子に割り当てられます。

DECLARE ステートメントと_NEW_演算子を使用してコンポーネントオブジェクトの宣言とインスタンス化を行う 2 ステップのプロセス以外に、コンストラクタメソッドとして DECLARE ステートメントを使用して、1 ステップでハッシュ反復子オブジェクトの宣言とインスタンス化を行えます。構文は次のとおりです。

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

この例では、ハッシュオブジェクト名は、一重引用符または二重引用符で囲む必要があります。

次に例を示します。

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

この例は、次のステートメントと同等です。

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

注: ハッシュオブジェクトを宣言してインスタンス化してから、ハッシュ反復子オブジェクトを作成する必要があります。詳細については、“ハッシュオブジェクトの宣言とインスタンス化” (4 ページ)を参照してください。

次に例を示します。

```
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;
```

このコードは、変数名 MyIter を使用して、ハッシュ反復子オブジェクトのインスタンスを作成します。ハッシュオブジェクト MyHash がコンストラクタ引数として渡されます。ORDERED 引数タグを**'yes'**に設定してハッシュオブジェクトを作成したため、データは昇順の key-value で返されます。

DECLARE ステートメントと_NEW_演算子の詳細については、*SAS Viya ステートメント: リファレンス*を参照してください。

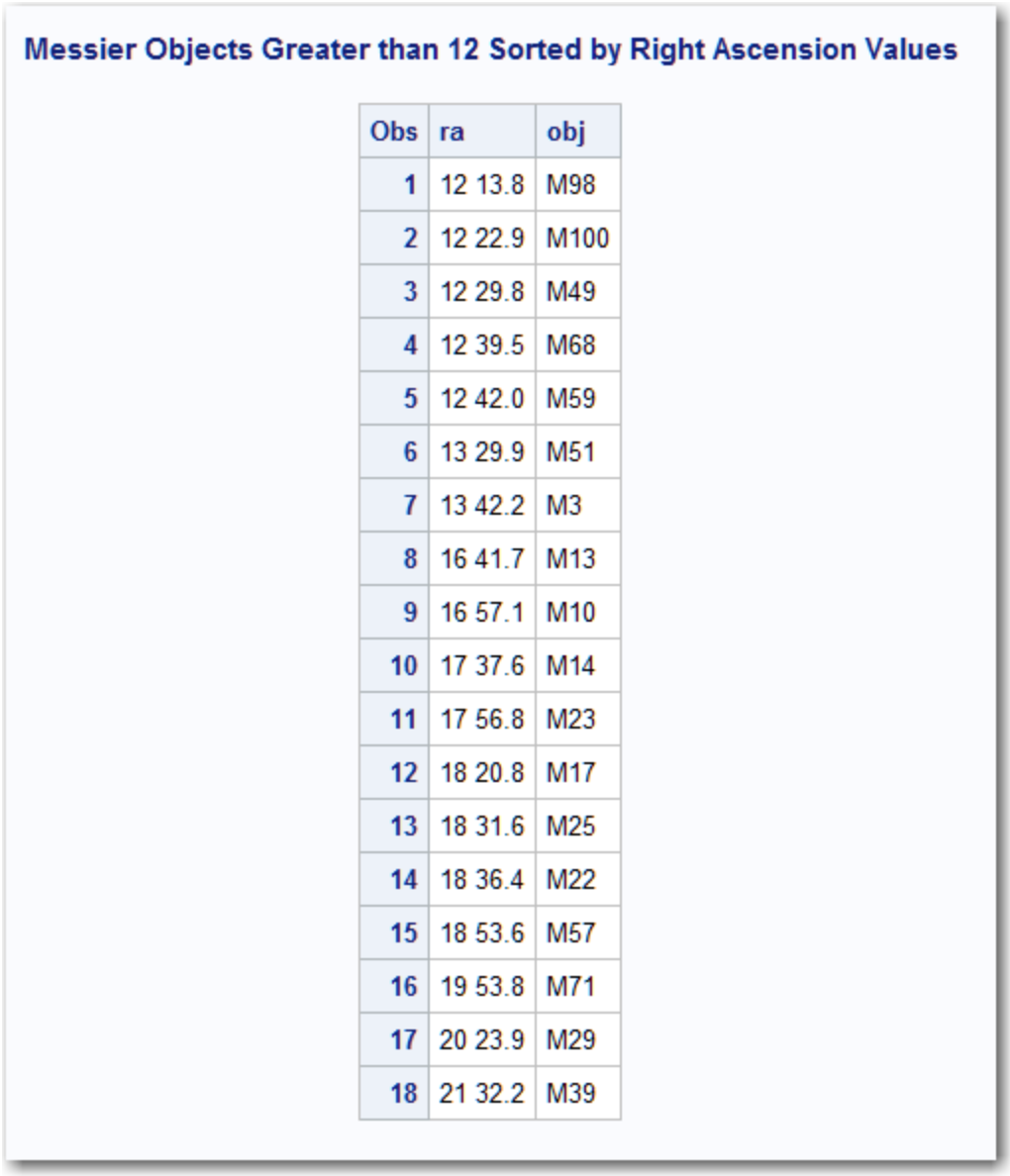
例: ハッシュ反復子を使用したハッシュオブジェクトデータの取得

次のコードは天文データを含むデータセット ASTRO を使用して、赤経(RA)の値が 12 より大きいメシエ天体を含むデータセット（OBJ）を作成します。FIRST メソッドと NEXT メソッドを使用して、昇順にデータを取得します。FIRST メソッドと NEXT メソッドの詳細については、*SAS Viya コンポーネントオブジェクト: リファレンス*を参照してください。.

```
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;
```

出力 1.3 には、PROC PRINT が生成したレポートが示されています。



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

Java オブジェクトの使用

Java オブジェクトについて

Java クラスをインスタンス化して結果オブジェクトのフィールドとメソッドにアクセスする、Java ネイティブインターフェイス (JNI)に似たメカニズムを提供します。Java と DATA ステップコードの両方を含むハイブリッドアプリケーションを作成できます。

CLASSPATH と Java オプション

SAS のこれまでのバージョンでは、Java クラスの検索に JREOPTIONS システムオプションを使用していました。

SAS では、Java オブジェクトが Java クラスを検索できるように CLASSPATH 環境変数を設定する必要があります。Java オブジェクトは、現在の Java クラスパスで検索された Java クラスのインスタンスを表します。使用するクラスは、クラスパスに含まれる必要があります。クラスが JAR ファイルにある場合、その JAR ファイル名がクラスパスに含まれる必要があります。

CLASSPATH 環境変数の設定法は、使用する動作環境によって異なります。多くの OS では、CLASSPATH 環境変数を、グローバルまたはローカル(ユーザーの SAS セッション内でのみの使用)に設定できます。 表 1.1 (20 ページ) には、Linux 動作環境に対応した方法と例が示されています。

表 1.1 動作環境に対応した CLASSPATH 環境変数の設定

Linux 動作環境	方法	例
グローバル	SAS 構成ファイル	set classpath ~/HelloWorld.jar
ローカル	EXPORT コマンド*	export classpath=~/HelloWorld.jar;

* 構文は、シェルによって異なります。

Java オブジェクトの使用の制限と要件

Java オブジェクトの使用時に次の制約と要件が適用されます。

- Java オブジェクトは、SAS から Java メソッドを呼び出すために設計されています。Java オブジェクトは、関数の SAS ライブラリを拡張するためのものではありません。特に大きなデータセットを含む場合は、DATA ステップへの高速なインプロセス拡張としては、PROC FCMP 関数を呼び出すほうが効率的です。Java オブジェクトを使用して大きなデータセットに対してこの種類の処理を実行すると、かなりの時間がかかります。
- SAS は、SAS ソフトウェアのインストール中に明示的に要求した Java ランタイム環境(JRE)のみサポートします。
- SAS は、SAS のインストール中に設定された Java オプションのみサポートします。
- Java オブジェクトで使用する前に、Java アプリケーションが正しく実行されることを確認します。
- Java によって SAS ログに書き込まれるテキストの 1 バイト目のパーセント文字(%)は、SAS によって予約されています。Java テキスト行の 1 バイト目に%を出力する必要がある場合は、そのすぐ後に別のパーセントを追加してエスケープします(%%)。
- SAS がロックダウン状態の時、Java オブジェクトは使用できません。LOCKDOWN ステートメントの詳細については、*SAS Viya ステートメント: リファレンス*を参照してください。

Java オブジェクトの宣言とインスタンス化

DECLARE ステートメントを使用して、Java オブジェクトを宣言します。新しい Java オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化して、Java オブジェクト名を引数タグとして使用します。

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

この例では、DECLARE ステートメントは、コンパイラにオブジェクト参照 J の種類は Java ということを示します。つまり、Java オブジェクトのインスタンスは変数 J に保存されます。ここでは、種類が Java のコンポーネントオブジェクトを保持する可能性があるオブジェクト参照 J のみを宣言しています。Java オブジェクトの宣言は一度のみ行います。_NEW_演算子は Java オブジェクトのインスタンスを作成し、オブジェクト参照 J に割り当てます。Java クラス名の SOMEJAVACLASS は、コンストラクタ引数として渡されます。これは、Java オブジェクトコンストラクタに必要な最初で唯一の引数です。その他の引数はすべて Java クラス自体のコンストラクタ引数です。

DECLARE ステートメントと_NEW_演算子を使用して Java オブジェクトの宣言とインスタンス化を行う 2 ステップのプロセス以外に、コンストラクタメソッドとして DECLARE ステートメントを使用して、1 ステップで Java オブジェクトの宣言とインスタンス化を行えます。

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

詳細については、“DECLARE ステートメント、Java オブジェクト” (117 ページ)および“_NEW_演算子の Java オブジェクト” (130 ページ)を参照してください。

オブジェクトフィールドへのアクセス

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、DATA ステップのパブリックフィールドとクラスフィールドにアクセスして変更できます。パブリックフィールドは静的でなく、Java クラスで public として宣言されています。クラスフィールドは静的であり、Java クラスからアクセスされます。

静的でないフィールドまたは静的なフィールドにアクセスするのかによって、オブジェクトフィールドにアクセスするメソッド呼び出しは、次のいずれかの形式になります。

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

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

静的でないフィールドまたは静的なフィールドにアクセスするのかによって、オブジェクトフィールドを変更するメソッド呼び出しは、次のいずれかの形式になります。

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

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

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。*field-name* 引数は、Java フィールドの種類を指定します。*value* は、メソッドによって返されるか、または設定される値を指定します。

詳細と例については、3 章, “Java オブジェクトの言語要素のディクショナリ” (111 ページ)を参照してください。

オブジェクトメソッドへのアクセス

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、DATA ステップのパブリックメソッドとクラスメソッドにアクセスできます。パブリックメソッドは静的でなく、Java クラスで public とし

て宣言されています。クラスメソッドは静的であり、Java クラスからアクセスされます。

静的でないメソッドまたは静的なメソッドにアクセスするのかによって、Java メソッドにアクセスするメソッド呼び出しは、次のいずれかの形式になります。

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*>);

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

詳細と例については、3 章, “Java オブジェクトの言語要素のディクショナリ” (111 ページ)を参照してください。

型の問題

Java 型のセットは、SAS データの型のスーパーセットです。Java のデータ型には、標準数値と標準文字値以外に、BYTE、SHORT、CHAR があります。SAS のデータ型は、数値と文字の 2 種類のみです。

表 1.3 には、Java オブジェクトメソッドの呼び出しの使用時における、Java データ型と SAS データ型のマッピングが示されています。

表 1.2　Java データ型と SAS データ型のマッピング

Java のデータ型	SAS のデータ型
BOOLEAN	数値
BYTE	数値
CHAR	数値
DOUBLE	数値
FLOAT	数値
INT	数値
LONG	数値
SHORT	数値
STRING	文字*

* Java STRING データ型は、UTF-8 文字列として SAS 文字データの種類にマッピングされています。

STRING 以外は、Java クラスから DATA ステップにオブジェクトを返すことができません。しかし、オブジェクトを Java メソッドに渡すことはできます。詳細については、“Java オブジェクト引数を渡す” (25 ページ)を参照してください。

オブジェクトを返す Java メソッドの中には、オブジェクト値を変換するラッパークラスを作成して処理されるメソッドもあります。次の例では、Java ハッシュテーブルはオブジェクト値を返します。しかし、型の変換を処理する単純な Java ラッパークラスを作成することで、DATA ステップからハッシュテーブルを使用できます。つまり、DATA ステップから **dhash** クラスと **shash** クラスにアクセスできます。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

```
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 オブジェクトと配列

DATA ステップ配列を Java オブジェクトに渡せます。

この例では、配列 **d** と配列 **s** が Java オブジェクト **j** に渡されます。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

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

1 次元配列パラメータのみサポートされます。しかし、配列が行順に渡されることを利用して、多次元配列引数を渡すことが可能です。この Java コードでは、次元のインデックスを手動で作成する必要があります。つまり、一次元配列パラメータを宣言して、部分配列のインデックスを作成します。

Java オブジェクト引数を渡す

Java クラスから DATA ステップにオブジェクトを返すことはできませんが、Java クラスにオブジェクトと文字列を渡すことはできます。

たとえば、**java/util/Vector** とその反復子に次のラッパークラスがあるとします。

```
/* 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;
 }
}
```

これらのラッパークラスは、型の変換の実行に役立ちます(たとえば、**mVector** コンストラクタは DOUBLE 引数を使用します)。**java/util/Vector** コンストラクタは整数値を使用しますが、DATA ステップには整数型がないため、コンストラクタのオーバーロードが必要です。

次の DATA ステッププログラムはこれらのクラスを使用します。プログラムはベクトルを作成してフィルし、反復子のコンストラクタにベクトルを渡し、ベクトルのすべての値をリストします。ベクトルのフィル後に、反復子を作成する必要があります。反復子は、作成時点のベクトルの変更数のコピーを保持します。この値は、ベクトルの最新の変更数と同期されている必要があります。ベクトルのフィル前に反復子を作成すると、コードに例外が発生します。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

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

オブジェクトを渡す際の現在の制限の 1 つとして、指定された署名に基づいて、JNI メソッドルックアップルーチンがフルクラスルックアップを実行しないことがあります。つまり、次に示すように、**mIterator** コンストラクタを変更して **Vector** を使用するようにできません。

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

mVector が **Vector** のサブクラスであっても、メソッドルックアップルーチンはコンストラクタを検索できません。現時点では、新しいメソッドを追加するか、またはラッパークラスを作成して Java で型を管理することでのみ解決できます。

Java 例外

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

メソッド呼び出し時に例外が発生したかどうかを確認するには、EXCEPTIONCHECK メソッドを使います。例外が発生するメソッドを呼び出す場合は、呼び出し後に例外の有無を確認することをお勧めします。例外が発生した場合は、適切な操作を実行してから、EXCEPTIONCLEAR メソッドを使って例外をクリアします。

EXCEPTIONDESCRIBE メソッドは、例外のデバッグログのオンとオフを切り替えます。例外のデバッグログがオンの場合、例外情報が JVM の標準出力に出力されます。デフォルトでは、JVM の標準出力は SAS ログにリダイレクトされます。例外のデバッグはデフォルトでオフとなっています。

詳細については、“EXCEPTIONCHECK メソッド” (120 ページ)、“EXCEPTIONCLEAR メソッド” (121 ページ)、および“EXCEPTIONDESCRIBE メソッド” (123 ページ)を参照してください。

Java 標準出力

デフォルトでは、次のように標準出力に送られる Java のステートメントからの出力は SAS ログに送られます。

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

SAS ログに送られる Java 出力は、DATA ステップの終了時にフラッシュされます。このフラッシュにより、Java 出力は DATA ステップの実行中に生成された出力の後に表示されます。FLUSHJAVAOUTPUT メソッドを使用して出力の同期をとると、実行順に表示されます。

Java オブジェクトの例

例 1: 単純な Java メソッドの呼び出し

この Java クラスは、3 つの数値を合計する簡単なメソッドを作成します。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

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

例 2: ユーザーインターフェイスの作成

Java コンポーネントアクセスメカニズムを提供することに加えて、Java オブジェクトを使用して単純な Java ユーザーインターフェイスを作成できます。

この Java クラスは、複数のボタンがある単純なユーザーインターフェイスを作成します。また、このユーザーインターフェイスは、ユーザーによって入力されたボタンの選択の順序を表す値のキューも管理します。

```
/* 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();
```

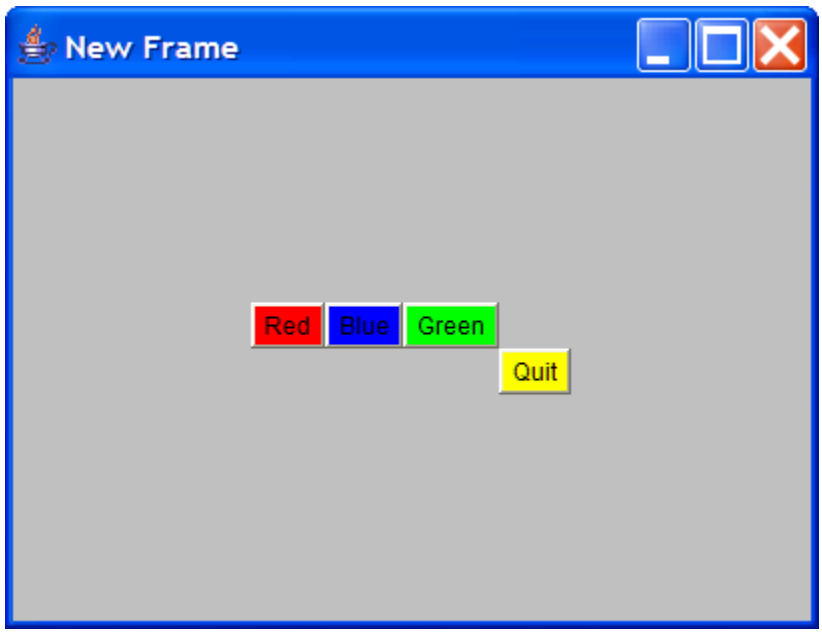
```
  setBackground(Color.lightGray);
  setSize(320,100);
  setTitle("New Frame");
  setVisible(true);
  setLayout(new FlowLayout(FlowLayout.CENTER, 10, 15));
  addWindowListener(new colorsUIListener());

  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;
```

DATA ステップコードで、**colorsUI** クラスはインスタンス化され、ユーザーインターフェイスが表示されます。ループを入力します。このループは、**Quit** をクリックすると終了します。このアクションは、Done 変数を介して DATA ステップに伝えられます。ループ中、DATA ステップは Java クラスのキューから値を取得し、出力データセットに値を逐次書き込みます。

図 1.1 Java オブジェクトによって作成されたユーザーインターフェイス



例 3: カスタムクラスローダーの作成

クラスパスに Java クラスをすべて 含める必要がない場合があります。独自のクラスローダーを書き、クラスを検索してロードできます。次の例では、カスタムクラスローダーの作成法を示しています。

この例では、クラス **x** を作成し、フォルダまたはディレクトリ **y** に常駐させています。このクラスのメソッドを、Java オブジェクトと **y** フォルダを含むクラスパスを使用して呼び出します。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

```
method m in y folder
```

```
method m2 in y folder
```

別のクラス **x** が別のフォルダ **z** にあるとします。

```
/* 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");
  }
}
```

クラスパスを変更することでフォルダ **y** のクラスではなくこのクラスのメソッドを呼び出せますが、SAS の再起動が必要です。次のメソッドは、クラスのロードにより動的なコントロールを可能にします。

カスタムクラスロードを作成するには、まず、Java オブジェクトから呼び出すメソッドをすべて含むインターフェイスを作成します。このプログラムの場合、**m** と **m2** になります。

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

次に、実際の実装に使用するクラスを作成します。

```
/* 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();
```

```
  }
}
```

Java オブジェクトインスタンスクラスに委任して、これらのメソッドが呼び出されます。**apiClassLoader** カスタムクラスローダーを作成するコードがこのセクションの後で提供されます。

```
/* 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;
     }
  }
}
```

api Java オブジェクトインスタンスクラスを介して委任して、次の DATA ステッププログラムはこれらのメソッドを呼び出します。この Java オブジェクトは、**z** フォルダからクラスをロードするカスタムクラスロードを作成する **api** クラスをインスタンス化します。**api** クラスはカスタムローダーを呼び出し、Java オブジェクトに **apiImpl** インターフェイス実装クラスのインスタンスを返します。メソッドが Java オブジェクトから呼び出されると、**api** クラスによってメソッドが実装クラスに委任されます。

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

```
run;
```

次の行が SAS ログに書き込まれます。

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

これまでの Java コードでは、JAR ファイルを使用して、**ClassLoader** コンストラクタのクラスパスを補うこともできました。

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

この場合、カスタムクラスローダーの Java コードは次のようになります。このクラスローダーのコードは、必要に応じて追加または変更できます。

```
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());
          }
        }
      }
    }
```

コンポーネントオブジェクト使用時のヒント

- DATA ステップ変数の割り当てと同じ方法でオブジェクトを割り当てることができます。ただし、オブジェクトの種類は一致する必要があります。最初のコードセットは有効ですが、2 番目のコードセットではエラーが発生します。

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

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

- オブジェクトの配列は宣言できません。次のコードではエラーが生成されます。

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

- コンポーネントオブジェクトはハッシュオブジェクト内にデータとして保存できますが、キーとしては保存できません。

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

  length key1 key2 $20;

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

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

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

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

  key1 = 'abc';
```

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

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

データセット WORK.H2 が表示されます。

図 1.2 データセット WORK.H2

Obs	key2
1	xyz

- 等号記号(=)以外の比較演算子を含むコンポーネントオブジェクトは使用できません。H1 と H2 がハッシュオブジェクトの場合、次のコードではエラーが生成されます。

  ```
  if h1>h2 then
  ```

- コンポーネントオブジェクトを宣言してインスタンス化した後に、スカラ値を割り当てることはできません。J が Java オブジェクトの場合、次のコードではエラーが生成されます。

  ```
  j=5;
  ```

- まだ使用されている可能性のあるオブジェクト参照、または参照によってすでに削除されているオブジェクト参照を削除しないように注意する必要があります。次のコードでは、元の H1 オブジェクトは H2 への参照によってすでに削除されているため、2 番目の DELETE ステートメントでエラーが生成されます。元の H2 は直接参照できなくなります。

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

- 引数タグの構文ではコンポーネントオブジェクトを使用できません。この例では、ADD メソッドでの H2 ハッシュオブジェクトの使用でエラーが生成されます。

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

- Java による SAS ログへのテキスト出力の 1 バイト目でのパーセント文字(%)の使用は、SAS によって予約されています。Java テキスト行の 1 バイト目に%を出力する必要がある場合は、そのすぐ後に別のパーセントを追加してエスケープします(%%)。
- ハッシュテーブルのハッシュテーブルを 持つこともできます。
- Java オブジェクトは単一 Java クラスのインスタンス化を表します。Java オブジェクトは他のものを持つことができません。しかし、Java インスタンスは他の通常の Java インスタンスと同様に任意に複雑化できます。Java オブ

ジェクトは他の Java エンティティの参照を持つことができますが、参照は Java オブジェクトとは考慮されません。

- SAS がロックダウン状態の時、Java オブジェクトは使用できません。LOCKDOWN ステートメントの詳細については、*SAS Viya ステートメント: リファレンス* を参照してください。

2 章

ハッシュオブジェクトとハッシュ反復子オブジェクトの言語要素のディクショナリ

ディクショナリ

ADD メソッド

特定のキーに関連付けられた指定データをハッシュオブジェクトに追加します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

"DATA: *datavalue*"ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで ADD メソッドを使用して、ハッシュオブジェクト内にデータを保存できます。

1 つ目は、次のコードに示すように、キーおよびデータ項目を定義してから ADD メソッドを使用する方法です。

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

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

2 つ目は、次のコードに示すように、ショートカットを使用して、ADD メソッド呼び出しでキーおよびデータを直接指定する方法です。

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

すでにハッシュオブジェクト内にあるキーを追加した場合、ADD メソッドはゼロ以外の値を返し、キーがすでにハッシュオブジェクト内にあることを示します。指定したキーに関連付けられたデータを新しいデータに置き換えるには、REPLACE メソッドを使用します。

DEFINEDATA メソッドでデータ変数を指定しない場合、データ変数は自動的にキーと同じであるとみなされます。

KEY:および DATA:引数タグを使用して直接キーとデータを指定する場合、両方の引数タグを使用する必要があります。

ADD メソッドでは、データ変数の値はデータ項目の値に設定されません。ハッシュオブジェクトの値のみが設定されます。

関連項目:

- “データの保存と取得” (7 ページ)

メソッド:

- “DEFINEDATA メソッド” (56 ページ)
- “DEFINEKEY メソッド” (59 ページ)
- “REF メソッド” (90 ページ)

CHECK メソッド

指定したキーがハッシュオブジェクト内に保存されているかどうかを確認します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで CHECK メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、次のコードに示すように、キーを指定してから CHECK メソッドを使用する方法です。

```
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;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、CHECK メソッド呼び出しでキーを直接指定する方法です。

```
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;
```

比較

CHECK メソッドは、キーがハッシュオブジェクト内にあるかどうかを示す値のみを返します。キーに関連付けられたデータ変数は更新されません。FIND メソッドも、キーがハッシュオブジェクト内にあるかどうかを示す値を返します。ただし、キーがハッシュオブジェクト内にある場合、FIND メソッドはさらにデータ変数にデータ項目の値を設定し、メソッド呼び出し後にそのデータ項目を使用できるようにします。

関連項目:

メソッド:

- “DEFINEKEY メソッド” (59 ページ)
- “FIND メソッド” (64 ページ)

CLEAR メソッド

ハッシュオブジェクトインスタンスを削除せずに、ハッシュオブジェクトからすべての項目を削除します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc

メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

CLEAR メソッドでは、既存のハッシュオブジェクトを削除して新しいオブジェクトを作成することなく、そのオブジェクトから項目を削除して再利用できます。ハッシュオブジェクトインスタンスを完全に削除する場合は、DELETE メソッドを使用します。

注: CLEAR メソッドは、DATA ステップ変数の値を変更しません。ハッシュオブジェクトの値のみをクリアします。

例: ハッシュオブジェクトのクリア

次の例では、ハッシュオブジェクトを宣言し、ハッシュオブジェクト内の項目数を取得してから、ハッシュオブジェクトを削除せずにクリアします。

```
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;
```

最初の PUT ステートメントは、ハッシュテーブル MYHASH をクリアする前にその項目数を書き込みます。

```
n=10000
```

2 番目の PUT ステートメントは、ハッシュテーブル MYHASH をクリアした後にその項目数を書き込みます。

```
n=0
```

関連項目:

メソッド:

- “DELETE メソッド、ハッシュオブジェクトとハッシュ反復子オブジェクト” (60 ページ)

DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言します。ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを作成し、データを初期化します。

別名: DCL

適用対象: ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

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

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

引数

object
コンポーネントオブジェクトを指定します。次のいずれかの値を使用できます。

hash
ハッシュオブジェクトを指定します。ハッシュオブジェクトは、迅速なデータの保存および取得のメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

参照項目: “ハッシュオブジェクトの使用 ” (3 ページ)

hiter
ハッシュ反復子オブジェクトを指定します。ハッシュ反復子オブジェクトを使用すると、ハッシュオブジェクトのデータを正または逆のキー順序で取得できます。

参照項目: “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

object-reference
ハッシュオブジェクトまたはハッシュ反復子オブジェクトのオブジェクト参照名を指定します。

argument_tag*:*value
ハッシュオブジェクトのインスタンスの作成に使用する情報を指定します。

有効なハッシュオブジェクト引数および値タグには、次の 5 つがあります。

dataset: '*dataset_name* <(*datasetoption*)>'
ハッシュオブジェクトに読み込む SAS データセットの名前を指定します。

SAS データセットの名前には、リテラルまたは文字変数を使用できます。データセット名は一重引用符または二重引用符で囲む必要があります。マクロ変数は二重引用符で囲む必要があります。

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、データ

セットオプションが表示される SAS データセットにのみ適用されるアクションを指定します。次の操作を実行できます。

- 変数名の変更
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードの指定

次の構文が使用されます。

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

SAS データセットオプションのリストについては、*SAS Viya Data Set Options: Reference* を参照してください。

制限事項 データセットオプションは、CAS Server では無効です。

注 データセットに重複するキーが含まれている場合、デフォルトでは、ハッシュオブジェクトの最初のインスタンスが保持され、後続のインスタンスは無視されます。重複するキーがある場合に、ハッシュオブジェクトの最後のインスタンスまたは SAS ログに書き込まれたエラーメッセージを保存するには、DUPLICATE 引数タグを使用します。

duplicate: '*option*'
ハッシュオブジェクトにデータセットを読み込むときに重複するキーを無視するかどうかを判断します。デフォルトでは、最初のキーが保存され、後続の重複はすべて無視されます。オプションには、次のいずれかの値を使用できます。

'replace' | 'r'
最後の重複するキーレコードを保存します。

'error' | 'e'
重複するキーが検出された場合に、ログにエラーを報告します。

REPLACE オプションを使用する次の例では、キー 620 には **brown**、キー 531 には **blue** が保存されます。デフォルトを使用する場合は、620 には **green**、 531 には **yellow** が保存されます。

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

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

hashexp: *n*
ハッシュオブジェクトの内部テーブルサイズです。ハッシュテーブルのサイズは 2^n です。

HASHEXP の値は、ハッシュテーブルサイズを作成する 2 の累乗指数として使用されます。たとえば、HASHEXP の値 4 は、ハッシュテーブルサイズ 2^4(つまり 16)と同じです。HASHEXP の最大値は 20 です。

ハッシュテーブルサイズは、保存可能な項目数とは等しくありません。ハッシュテーブルを、一連の'バケット'と考えます。ハッシュテーブルサイズ 16 には、16 個の'バケット'があります。各バケットに保持できる項目の数は無限です。ハッシュテーブルの効率は、項目をバケットにマップし、バケットから項目を取得するハッシュ関数の機能にあります。

ハッシュオブジェクトのルックアップルーチンの効率を最大にするには、ハッシュオブジェクトのデータ量に応じてハッシュテーブルサイズを指定する必要があります。最適な結果が得られるまで、さまざまな HASHEXP 値を試してみてください。たとえば、ハッシュオブジェクトに 100 万個の項目が含まれている場合、ハッシュテーブルサイズ 16(HASHEXP = 4)でも機能しますが、効率は良くありません。ハッシュテーブルサイズ 512 または 1024(HASHEXP = 9 または 10)を使用すると、最高の処理速度が得られます。

デフォルト 8。これは、ハッシュテーブルサイズ 2^8(つまり 256)と同じです。

keysum:'*variable-name*'
すべてのキーのキー集計をトラッキングする変数の名前を指定します。キー集計はキーが何回 FIND メソッド呼び出しに参照されたかのカウントです。

注 キー集計は出力データセットにあります。

例 “例 5: 出力データセットにキー集計を追加する” (54 ページ)

ordered: '*option*'
ハッシュオブジェクトとハッシュ反復子オブジェクトを併用する場合、またはハッシュオブジェクト OUTPUT メソッドを使用する場合、データがキー値の順序で返されるかどうか、またはどのように返されるかを指定します。

option には、次のいずれかの値を使用できます。

'ascending' | 'a'
データはキー値の昇順で返されます。'**ascending**'の指定は、'**yes**'を指定することと同じです。

'descending' | 'd'
データはキー値の降順で返されます。

'YES' | 'Y'
データはキー値の昇順で返されます。'**yes**'の指定は、'**ascending**'を指定することと同じです。

'NO' | 'N'
データは未定義の順序で返されます。

デフォルト NO

制限事項 ハッシュ反復子を持つハッシュ項目内を並べ替え順に移動できます。しかし、CAS Server から並べ替え順にハッシュテーブルを生成できません。.

注 ORDERED 引数タグは'ascending'、'descending'または'yes'に設定されていますが、.VARCHAR はサポートされていません。

ヒント 引数は二重引用符で囲むこともできます。

multidata: '*option*'
各キーに複数のデータ項目を許可するかどうかを指定します。

option には、次のいずれかの値を使用できます。

'YES' | 'Y'
各キーに複数のデータ項目が許可されます。

'NO' | 'N'
各キーにデータ項目が 1 つのみ許可されます。

デフォルト NO

ヒント 引数値は二重引用符で囲むこともできます。

参照項目: “重複するキーとデータペア” (6 ページ)

suminc: '*variable-name*'
ハッシュオブジェクトキーの集計数を維持します。SUMINC 引数タグは、DATA ステップ変数を指定します。この変数には、合計増分が保持されます。合計増分はキーが参照されるたびにキー集計に追加される数です。

参照項目: “キー集計の維持” (9 ページ)

例 キー集計は、DATA ステップ変数の現在の値を使用して変更されます。

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

参照項目: “コンストラクタを使用したハッシュオブジェクトデータの初期化” (5 ページ) and “ハッシュ反復子オブジェクトの宣言とインスタンス化” (16 ページ)

詳細

基本

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステ

ップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

定義済みの DATA ステップコンポーネントオブジェクトの詳細については、“DATA ステップコンポーネントオブジェクトについて” (2 ページ)を参照してください。

ハッシュオブジェクトまたはハッシュ反復子オブジェクトの宣言(フォーム 1)

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言するには、DECLARE ステートメントを使用します。

```
declare hash h;
```

DECLARE ステートメントは、オブジェクト参照 H がハッシュオブジェクトであることを SAS に示します。

新しいハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。たとえば、次のコード行では、_NEW_演算子でハッシュオブジェクトが作成され、オブジェクト参照 H に割り当てられます。

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

DECLARE ステートメントを使用したハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンス生成(フォーム 2)

DECLARE ステートメントと_NEW_演算子を使用してハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言し、インスタンス化する 2 ステップのプロセスの代わりに、DECLARE ステートメントを使用して、ハッシュオブジェクトまたはハッシュ反復子オブジェクトを 1 つのステップで宣言し、インスタンス化することができます。たとえば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。

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

前述のコード行は、次のコードを使用するのと同等です。

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

*コンストラクタ*は、ハッシュオブジェクトをインスタンス化し、ハッシュオブジェクトデータを初期化するために使用できるメソッドです。たとえば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。さらに、引数タグ HASHEXP を使用してハッシュテーブルサイズが値 16(2^4)に初期化されます。

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

ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、データセットオプションが表示される SAS データセットにのみ適用されるアクションを指定します。次の操作を実行できます。

- 変数名の変更
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択

- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードの指定

次の構文が使用されます。

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

データセットオプションのその他の使用例については、“例 4: ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する” (54 ページ)を参照してください。データセットオプションのリストについては、*SAS Viya Data Set Options: Reference* を参照してください。

比較

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを宣言し、インスタンス化するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例

例 1: DECLARE ステートメントと_NEW_演算子を使用したハッシュオブジェクトの宣言とインスタンス生成

この例では、DECLARE ステートメントを使用してハッシュオブジェクトを宣言します。ハッシュオブジェクトのインスタンス化には_NEW_演算子が使用されます。

```
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;
```

例 2: DECLARE ステートメントを使用したハッシュオブジェクトの宣言とインスタンス生成

この例では、DECLARE ステートメントを使用してハッシュオブジェクトの宣言とインスタンス化を 1 つのステップで実行します。

```
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;
```

例 3: ハッシュオブジェクトのインスタンスを生成してサイズ順に並べる

この例では、DECLARE ステートメントを使用してハッシュオブジェクトを宣言し、インスタンス化します。ハッシュテーブルサイズは 16(2^4)に設定されています。

```
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;
```

例 4: ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する

次の例では、ハッシュオブジェクトを宣言するときにさまざまな SAS データセットオプションを使用します。

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

SAS データセットオプションのリストについては、*SAS Viya Data Set Options: Reference* を参照してください。

例 5: 出力データセットにキー集計を追加する

次の例では、変数 *ks* はキー集計を持っていて、その変数を出力データセットに追加することを宣言します。

```
data key;
 length key data 8;
```

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

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

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

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

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

proc print data=out;
run;
```

アウトプット 2.1 キー集計データの出力

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

関連項目:

演算子:

• “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

DEFINEDATA メソッド

ハッシュオブジェクトに保存するデータを、指定のデータ変数に関連付けて定義します。

適用対象: ハッシュオブジェクト

構文

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

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'*datavarname*'
データ変数の名前を指定します。

データ変数名は二重引用符で囲むこともできます。

ALL: 'YES' | “YES”
すべてのデータ変数をデータに指定して、データセットをオブジェクトコンストラクタに読み込みます。

DECLARE ステートメントまたは_NEW_演算子で *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'を使ってすべてのキー変数を定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

注: ADD メソッドまたは REPLACE メソッドにショートカット表記(例: **h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEDATA メソッドに ALL:'YES'オプションを使う場合は、データの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの

変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメータとして使用します。例:

```
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;
```

DEFINEDATA メソッドの使用方法の詳細については、“キーとデータの定義” (5 ページ)を参照してください。

例

次の例では、ハッシュオブジェクトを作成し、キー変数とデータ変数を定義します。

```
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;
```

関連項目:

- “キーとデータの定義” (5 ページ)

メソッド:

- “DEFINEDONE メソッド” (58 ページ)
- “DEFINEKEY メソッド” (59 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

DEFINEDONE メソッド

キーとデータの定義がすべて完了したことを指定します。

適用対象: ハッシュオブジェクト

構文

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

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

memrc:'y'
ハッシュオブジェクトへのデータセットの読み込みでメモリエラーが発生した場合に、エラーからの回復を可能にします。

データセット読み込み時のメモリ不足が原因で呼び出しが失敗すると、ゼロ以外のリターンコードが返されます。ハッシュオブジェクトは基礎配列の主メモリを解放します。この種のエラーが発生した後に実行可能な操作は、DELETE メソッドを使った削除のみです。

詳細

DEFINEDONE メソッド呼び出しでコンストラクタに *dataset* 引数タグを使用すると、データセットはハッシュオブジェクトに読み込まれます。

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

DEFINEDONE メソッドの使用方法の詳細については、“キーとデータの定義” (5 ページ)を参照してください。

関連項目:

- “キーとデータの定義” (5 ページ)

メソッド:

- “DEFINEDATA メソッド” (56 ページ)
- “DEFINEKEY メソッド” (59 ページ)

DEFINEKEY メソッド

ハッシュオブジェクトのキー変数を定義します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'keyvarname'
キー変数の名前を指定します。

キー変数名は二重引用符で囲むこともできます。

ALL: 'YES' | ”YES”
すべてのデータ変数をキーに指定して、データセットをオブジェクトコンストラクタに読み込みます。

DECLARE ステートメントまたは_NEW_演算子で *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'オプションを使ってすべてのキー変数を定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

DEFINEKEY メソッドの使用方法の詳細については、“キーとデータの定義” (5 ページ)を参照してください。

注: ADD、CHECK、FIND、REMOVE、REPLACE などのメソッドにショートカット表記(例: **h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEKEY

メソッドに ALL:'YES'オプションを使う場合は、キーとデータの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメータとして使用します。例:

```
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;
```

関連項目:

- “キーとデータの定義” (5 ページ)

メソッド:

- “DEFINEDATA メソッド” (56 ページ)
- “DEFINEDONE メソッド” (58 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

DELETE メソッド、ハッシュオブジェクトとハッシュ反復子オブジェクト

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを削除します。

適用対象: ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクトまたはハッシュ反復子オブジェクトの名前を指定します。

詳細

DATA ステップのコンポーネントオブジェクトは、DATA ステップの終了時に自動的に削除されます。他のハッシュオブジェクトコンストラクタまたはハッシュ反復子オブジェクトコンストラクタでオブジェクト参照変数を再利用する場合は、DELETE メソッドでハッシュオブジェクトまたはハッシュ反復子オブジェクトを削除する必要があります。

削除したハッシュオブジェクトまたはハッシュ反復子オブジェクトを使用しようとすると、エラーがログに書き込まれます。

ハッシュオブジェクト内からすべての項目を削除し、ハッシュオブジェクトを再利用できるように保存するには、“CLEAR メソッド” (45 ページ)を使用します。

DO_OVER メソッド

ハッシュオブジェクトの重複キーのリスト内を移動します。

適用対象: ハッシュオブジェクト

構文

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

引数

object
ハッシュオブジェクト名を指定します。

KEY:*keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

注: 反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

例

次の例では、重複キーを含むデータセット **dup** を作成します。DO_OVER メソッドと RESET_DUP メソッドを使って、重複キー間を移動します。

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

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

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

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

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

run;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “RESET_DUP メソッド” (102 ページ)

EQUALS メソッド

2 つのハッシュオブジェクトが等しいかどうかを確認します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

HASH:'*object*'
1 つ目のハッシュオブジェクトに対して比較する 2 つ目のハッシュオブジェクトの名前を指定します。

RESULT: *variable name*
結果を保持する数値変数の名前を指定します。ハッシュオブジェクトが等しい場合、結果変数は 1 です。それ以外の場合の結果変数はゼロです。

詳細

次の例では、H1 を H2 ハッシュオブジェクトに対して比較します。

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

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

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

次のすべての条件に該当する場合、2 つのハッシュオブジェクトは等しいと定義されます。

- 2 つのハッシュオブジェクトが同サイズである(HASHEXP サイズが等しい)。
- 2 つのハッシュオブジェクトに同数の項目がある(H1.NUM_ITEMS = H2.NUM_ITEMS)。

- 2 つのハッシュオブジェクトのキー構造とデータ構造が同一である。
- H1 と H2 ハッシュオブジェクトの比較を順不同で反復した場合に、H1 からの一連のレコードが対応する H2 のレコードと同じキーおよびデータのフィールドを持っている。すなわち、各ハッシュオブジェクトの同じ位置に各レコードがあり、それらのレコードが対応するもう一方のハッシュオブジェクトのレコードと同一である。

例: 2 つのハッシュオブジェクトの比較

次の例では、EQUALS の 1 つ目のリターンコードはゼロ以外の値を返し、2 つ目はゼロ値を返します。

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

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

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

  k = 100;
  h2.replace();

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

 run;
```

FIND メソッド

指定したキーがハッシュオブジェクトに保存されているかどうかを確認します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc

メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで FIND メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、次のコードに示すように、キーを指定してから FIND メソッドを使用する方法です。

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

2 つ目は、次のコードに示すように、ショートカットを使用して、FIND メソッド呼び出しでキーを直接指定する方法です。

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

```
    put 'Key is in the hash object.';
run;
```

ハッシュオブジェクトの各キーに複数のデータ項目がある場合は、“FIND_NEXT メソッド” (67 ページ)と “FIND_PREV メソッド” (68 ページ) を FIND メソッドと組み合わせ、複数データ項目のリスト内を移動します。

比較

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを示す値を返します。FIND メソッドでは、キーがハッシュオブジェクトに存在する場合、同時にデータ項目の値がデータ変数に設定されます。これによって、メソッド呼び出し後に値が使えるようになります。CHECK メソッドは、キーがハッシュオブジェクト内にあるかどうかを示す値のみを返します。データ変数は更新されません。

例: FIND メソッドを使用したハッシュオブジェクト内のキー検索

次の例では、ハッシュオブジェクトを作成します。2 つのデータ値を追加します。FIND メソッドを使用してハッシュオブジェクト内のキーを検索します。データ値は、キーに関連付けられたデータセット変数に返されます。

```
data _null_;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    /* avoid uninitialized variable notes */
    call missing(k, d);
    rc = h.defineDone();
  end;
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Odyssey');
  /* Verify that key JOYCE is in hash object and */
  /* return its data value to the data set variable D */
  rc = h.find(key: 'Joyce');
  put d=;
run;
```

d=Ulysses が SAS ログに書き込まれます。

関連項目:

- “データの保存と取得” (7 ページ)

メソッド:

- “CHECK メソッド” (43 ページ)
- “DEFINEKEY メソッド” (59 ページ)
- “FIND_NEXT メソッド” (67 ページ)
- “FIND_PREV メソッド” (68 ページ)

- “REF メソッド” (90 ページ)

FIND_NEXT メソッド

現在のキーの複数項目リストで現在のリスト項目を次の項目に設定し、対応するデータ変数にデータを設定します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータ項目が関連付けられているかどうかを判断します。キーに別のデータ項目があることが判明した場合、FIND_NEXT メソッドを使用してそのデータ項目を取得できます。これによりデータ変数にデータ項目の値が設定され、メソッド呼び出し後に使用できるようになります。データ項目リスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドを使用することでリスト内を移動できます。

例

この例では、FIND_NEXT メソッドを使用して、複数のキーに複数のデータ項目があるデータセットを反復処理します。キーに複数のデータ項目がある場合、後続の項目は **dup** とマークされます。

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

```
  5 99
  ;
data _null_;
  dcl hash h(dataset:'dup', multidata: 'y');
  h.definekey('key');
  h.definedata('key', 'data');
  h.definedone();
  /* avoid uninitialized variable notes */
  call missing (key, data);
  do key = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put key= data=;
      rc = h.find_next();
      do while(rc = 0);
        put 'dup ' key= data;
        rc = h.find_next();
      end;
    end;
  end;
 run;
```

次の行が SAS ログに書き出されます。

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

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “FIND メソッド” (64 ページ)
- “FIND_PREV メソッド” (68 ページ)
- “HAS_NEXT メソッド” (71 ページ)

FIND_PREV メソッド

現在のキーの複数項目リストで現在のリスト項目を次の項目に設定し、対応するデータ変数にデータを設定します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_PREV メソッドは、キーに複数のデータ項目が関連付けられているかどうかを判断します。キーに前のデータ項目があることが判明した場合、FIND_PREV メソッドを使用してそのデータ項目を取得できます。これによりデータ変数にデータ項目の値が設定され、メソッド呼び出し後に使用できるようになります。データ項目リスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (71 ページ)を参照してください。

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “FIND メソッド” (64 ページ)
- “FIND_NEXT メソッド” (67 ページ)
- “HAS_PREV メソッド” (73 ページ)

FIRST メソッド

基となるハッシュオブジェクトの開始値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

FIRST メソッドは、ハッシュオブジェクト内の最初のデータ項目を返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントま

たは_NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータ項目はキー値の昇順で並べ替えられるため、'最小'キー(最小の数値またはアルファベット順の最初)のデータ項目が返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの昇順でデータ項目を返します。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは _NEW_演算子で **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータ項目はキー値の降順で並べ替えられるため、'最大'キー(最大の数値またはアルファベット順の最後)のデータ項目が返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの降順でデータ項目を返します。

ハッシュオブジェクト内の最後のデータ項目を返すには、LAST メソッドを使用します。

注: FIRST メソッドは、データ変数にデータ項目の値を設定し、メソッド呼び出し後にそのデータ項目を使用できるようにします。

例: ハッシュオブジェクトデータの取得

次の例では、売上データを含むデータセットを作成します。製品を売上順に表示します。データがハッシュオブジェクト内に読み込まれ、データの取得に FIRST メソッドと NEXT メソッドが使用されます。

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

次の行が SAS ログに書き出されます。

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

関連項目:

- “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

メソッド:

- “LAST メソッド” (75 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

HAS_NEXT メソッド

現在のキーの複数データ項目リスト内に次の項目があるかどうかを判断します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データ項目リスト内に別のデータ項目がない場合はゼロ値、データ項目リスト内に別のデータ項目がある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータ項目がある場合、HAS_NEXT メソッドを使用して、現在のキーの複数データ項目リスト内に次の項目があるかどうかを判断できます。別の項目がある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別の項目がない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータ項目が関連付けられているかどうかを判断します。キーに別のデータ項目があることが判明した場合、FIND_NEXT メソッドを使用してそのデータ項目を取得できます。これによりデ

ータ変数にデータ項目の値が設定され、メソッド呼び出し後に使用できるようになります。データ項目リスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。

例: データ項目の検索

この例では、複数のキーに複数のデータ項目があるハッシュオブジェクトを作成します。さらに、HAS_NEXT メソッドを使用してすべてのデータ 項目を検索します。

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

次の行が SAS ログに書き出されます。

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

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “FIND メソッド” (64 ページ)
- “FIND_NEXT メソッド” (67 ページ)
- “FIND_PREV メソッド” (68 ページ)
- “HAS_PREV メソッド” (73 ページ)

HAS_PREV メソッド

現在のキーの複数データ項目リスト内に前の項目があるかどうかを判断します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データ項目リスト内に別のデータ項目がない場合はゼロ値、データ項目リスト内に別のデータ項目がある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータ項目がある場合、HAS_PREV メソッドを使用して、現在のキーの複数データ項目リスト内に前の項目があるかどうかを判断できます。前の項目がある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別の項目がない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータ項目が関連付けられているかどうかを判断します。キーに前のデータ項目があることが判明した場合、FIND_PREV メソッドを使用してそのデータ項目を取得できます。これによりデータ変数にデータ項目の値が設定され、メソッド呼び出し後に使用できるようになります。データ項目リスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (71 ページ)を参照してください。

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “FIND メソッド” (64 ページ)
- “FIND_NEXT メソッド” (67 ページ)
- “FIND_PREV メソッド” (68 ページ)
- “HAS_NEXT メソッド” (71 ページ)

ITEM_SIZE 属性

ハッシュオブジェクト内の項目のサイズ(バイト)を返します。

適用対象: ハッシュオブジェクト

構文

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

引数

variable_name
ハッシュオブジェクト内の項目のサイズを含む変数名を指定します。

object
ハッシュオブジェクト名を指定します。

詳細

ITEM_SIZE 属性は、項目のサイズ(バイト)を返します。これには、キーおよびデータ変数の他に、いくつかの追加内部情報が含まれます。ハッシュオブジェクトが ITEM_SIZE 属性と NUM_ITEMS 属性で使用するメモリ量の推定値を取得できます。ITEM_SIZE 属性は、ハッシュオブジェクトが必要とする初期オーバーヘッドを反映せず、必要な内部配置も考慮しません。そのため、ITEM_SIZE では正確なメモリ使用量は提供されず、近似値が返されます。

例: ハッシュ項目のサイズを返す

次の例では、ITEM_SIZE を使用して MYHASH 内の項目のサイズを返します。

```
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 が SAS ログに書き込まれます。

LAST メソッド

基となるハッシュオブジェクトの最終値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

LAST メソッドは、ハッシュオブジェクト内の最後のデータ項目を返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータ項目はキー値の昇順で並べ替えられるため、'最大'キー(最大の数値またはアルファベット順の最後)のデータ項目が返されます。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータ項目はキー値の降順で並べ替えられるため、'最小'キー(最小の数値またはアルファベット順の最初)のデータ項目が返されます。

ハッシュオブジェクト内の最初のデータ項目を返すには、FIRST メソッドを使用します。

注: LAST メソッドは、データ変数にデータ項目の値を設定し、メソッド呼び出し後にそのデータ項目を使用できるようにします。

関連項目:

- “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

メソッド:

- “FIRST メソッド” (69 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを作成します。

適用対象: ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

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

引数

object-reference
ハッシュオブジェクトまたはハッシュ反復子オブジェクトのオブジェクト参照名を指定します。

object
コンポーネントオブジェクトを指定します。次のいずれかを指定できます。

hash
ハッシュオブジェクトを示します。ハッシュオブジェクトは、迅速なデータの保存および取得のメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

hiter
ハッシュ反復子オブジェクトを示します。ハッシュ反復子オブジェクトを使用すると、ハッシュオブジェクトのデータを正または逆のキー順序で取得できます。

参照項目: “ハッシュオブジェクトの使用 ” (3 ページ)と“ハッシュ反復子オブジェクトの使用 ” (16 ページ)

argument_tag*:*value
ハッシュオブジェクトのインスタンスの作成に使用する情報を指定します。

有効なハッシュオブジェクトの引数タグおよび値は、次のとおりです。

dataset: '*dataset_name* <(*datasetoption*)>'
ハッシュオブジェクトに読み込む SAS データセットの名前を指定します。

SAS データセットの名前には、リテラルまたは文字変数を使用できます。データセット名は一重引用符または二重引用符で囲む必要があります。マクロ変数は二重引用符で囲む必要があります。

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、データセットオプションが表示される SAS データセットにのみ適用されるアクションを指定します。次の操作を実行できます。

- 変数名の変更
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードの指定

次の構文が使用されます。

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

SAS データセットオプションのリストについては、*SAS Viya Data Set Options: Reference* を参照してください。

制限事項 データセットオプションは、CAS Server では無効です。

注 データセットに重複するキーが含まれている場合、デフォルトでは、ハッシュオブジェクトの最初のインスタンスが保持され、後続のインスタンスは無視されます。重複するキーがある場合に、ハッシュオブジェクトの最後のインスタンスを保存するか、または SAS ログにエラーメッセージを書き込むには、DUPLICATE 引数タグを使用します。

duplicate: '*option*'
ハッシュオブジェクトにデータセットを読み込むときに重複するキーを無視するかどうかを判断します。デフォルトでは、最初のキーが保存され、後続の重複はすべて 無視されます。オプションには、次のいずれかの値を使用できます。

'replace' | 'r'
最後の重複するキーレコードを保存します。

'error' | 'e'
重複するキーが検出された場合に、ログにエラーを報告します。

REPLACE オプションを使用する次の例では、キー 620 には **brown**、キー 531 には **blue** が保存されます。デフォルトを使用する場合は、620 には **green**、531 には **yellow** が保存されます。

```
data table;
```

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

hashexp: *n*

ハッシュオブジェクトの内部テーブルサイズです。ハッシュテーブルのサイズは 2^n です。

HASHEXP の値は、ハッシュテーブルサイズを作成する 2 の累乗指数として使用されます。たとえば、HASHEXP の値 4 は、ハッシュテーブルサイズ 2^4(つまり 16)と同じです。HASHEXP の最大値は 20 です。

ハッシュテーブルサイズは、保存可能な項目数とは等しくありません。ハッシュテーブルを、一連の'バケット'と考えます。ハッシュテーブルサイズ 16 には、16 個の'バケット'があります。各バケットに保持できる項目の数は無限です。ハッシュテーブルの効率は、項目をバケットにマップし、バケットから項目を取得するハッシュ関数の機能にあります。

ハッシュオブジェクトのルックアップルーチンの効率を最大にするには、ハッシュオブジェクトのデータ量に応じてハッシュテーブルサイズを設定する必要があります。最適な結果が得られるまで、さまざまな HASHEXP 値を試してみてください。たとえば、ハッシュオブジェクトに 100 万個の項目が含まれている場合、ハッシュテーブルサイズ 16(HASHEXP = 4)でも機能しますが、効率は良くありません。ハッシュテーブルサイズ 512 または 1024(HASHEXP = 9 または 10)を使用すると、最高の処理速度が得られます。

デフォルト 8。これは、ハッシュテーブルサイズ 2^8(つまり 256)と同じです。

keysum:'*variable-name*'

すべてのキーのキー集計をトラッキングする変数の名前を指定します。キー集計はキーが何回 FIND メソッド呼び出しに参照されたかのカウントです。

注 キー集計は出力データセットにあります。

ordered: '*option*'

ハッシュオブジェクトとハッシュ反復子オブジェクトを併用する場合、またはハッシュオブジェクト OUTPUT メソッドを使用する場合、データが

キー値の順序で返されるかどうか、またはどのように返されるかを指定します。

引数値は二重引用符で囲むこともできます。

option には、次のいずれかの値を使用できます。

'ascending' | 'a'	データはキー値の昇順で返されます。'**ascending**'の指定は、'**yes**'を指定することと同じです。
'descending' | 'd'	データはキー値の降順で返されます。
'YES' | 'Y'	データはキー値の昇順で返されます。'**yes**'の指定は、'**ascending**'を指定することと同じです。
'NO' | 'N'	データは未定義の順序で返されます。

デフォルト NO

制限事項 ハッシュ反復子を持つハッシュ項目内を並べ替え順に移動できます。しかし、CAS Server から並べ替え順にハッシュテーブルを生成できません。.

注 ORDERED 引数タグは'ascending'、'descending'または'yes'に設定されていますが、.VARCHAR はサポートされていません。

ヒント 引数は二重引用符で囲むこともできます。

multidata: '*option*'

各キーに複数のデータ項目を許可するかどうかを指定します。

引数値は二重引用符で囲むこともできます。

option には、次のいずれかの値を使用できます。

'YES' | 'Y'	各キーに複数のデータ項目が許可されます。
'NO' | 'N'	各キーにデータ項目が 1 つのみ許可されます。

デフォルト NO

参照項目: “重複するキーとデータペア” (6 ページ)

suminc: '*variable-name*'

ハッシュオブジェクトキーの集計数を維持します。SUMINC 引数タグは、DATA ステップ変数を指定します。この変数には、合計増分が保持されます。合計増分はキーが参照されるたびにキー集計に追加される数です。

参照項目: “キー集計の維持” (9 ページ)

例 キー集計は、DATA ステップ変数の現在の値を使用して変更されます。

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

参照項目: “コンストラクタを使用したハッシュオブジェクトデータの初期化” (5 ページ) と“ハッシュ反復子オブジェクトの宣言とインスタンス化” (16 ページ)

詳細

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

_NEW_演算子を使用してコンポーネントオブジェクトをインスタンス化する場合は、最初に DECLARE ステートメントを使用してコンポーネントオブジェクトを宣言する必要があります。たとえば、次のコード行では、DECLARE ステートメントは、オブジェクト参照 H がハッシュオブジェクトであることを SAS に示します。_NEW_演算子でハッシュオブジェクトが作成され、オブジェクト参照 H に割り当てられます。

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

注: DECLARE ステートメントを使用してハッシュオブジェクトまたはハッシュ反復子オブジェクトの宣言とインスタンス化を 1 つのステップで実行できます。

コンストラクタは、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用するメソッドです。たとえば、次のコード行では、_NEW_演算子でハッシュオブジェクトがインスタンス化され、オブジェクト参照 H に割り当てられます。さらに、データセット WORK.KENNEL がハッシュオブジェクトに読み込まれます。

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

定義済みの DATA ステップコンポーネントオブジェクトおよびコンストラクタの詳細については、“DATA ステップコンポーネントオブジェクトについて” (2 ページ)を参照してください。

比較

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを宣言し、インスタンス化するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例: _NEW_演算子を使用したハッシュオブジェクトデータのインスタンス化と初期化

この例では、_NEW_演算子を使用して、ハッシュオブジェクトのデータをインスタンス化して初期化し、ハッシュ反復子オブジェクトをインスタンス化します。

ハッシュオブジェクトにはデータが含まれており、データをキー順序で取得するために反復子が使用されます。

```
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;
```

次の行が SAS ログに書き出されます。

```
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
```

関連項目:

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

NEXT メソッド

基となるハッシュオブジェクトの次の値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

NEXT メソッドを繰り返し使用することで、ハッシュオブジェクト内を移動し、データ項目をキーの順序で返すことができます。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータ項目を返します。

ハッシュオブジェクト内の前のデータ項目を返すには、PREV メソッドを使用できます。

注: NEXT メソッドは、データ変数にデータ項目の値を設定し、メソッド呼び出し後にそのデータ項目を使用できるようにします。

注: FIRST メソッドを呼び出さずに NEXT メソッドを呼び出した場合でも、NEXT メソッドはハッシュオブジェクト内の最初の項目から開始します。

関連項目:

- “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

メソッド:

- “FIRST メソッド” (69 ページ)
- “PREV メソッド” (89 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

NUM_ITEMS 属性

ハッシュオブジェクト内の項目数を返します。

適用対象: ハッシュオブジェクト

構文

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

引数

variable_name
ハッシュオブジェクト内の項目数を含む変数名を指定します。

object
ハッシュオブジェクト名を指定します。

例: ハッシュオブジェクト内の項目数を返す

この例では、データセットを作成し、そのデータセットをハッシュオブジェクト内に読み込みます。項目がハッシュオブジェクトに追加され、その結果のハッシュオブジェクト内の項目の合計数が NUM_ITEMS 属性によって返されます。

```
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 が SAS ログに書き込まれます。

OUTPUT メソッド

ハッシュオブジェクトにデータが格納された 1 つ以上のデータセットを作成します。

適用対象: ハッシュオブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

DATASET: '*dataset*'
出力データセット名を指定します。

SAS データセットの名前には、文字リテラルまたは文字変数を使用できます。データセット名は二重引用符で囲むこともできます。出力データセットの名前を指定するときには、DATASET 引数タグで SAS データセットオプションを使用できます。マクロ変数は二重引用符で囲む必要があります。

datasetoption
データセットオプションを指定します。

データセットオプションを指定する方法の詳細については、“Syntax” (*SAS Viya Data Set Options: Reference*)を参照してください。

詳細

ハッシュオブジェクトキーは、出力データセットの一部として自動的には保存されません。DEFINEDATA メソッドを使用して出力データセットにキーを含めると、キーをデータ項目として定義できます。また、データ項目が DEFINEDATA メソッドを使用して定義されていない場合、キーが OUTPUT メソッドで指定されたデータセットに書き込まれます。

ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用する場合、データ項目はキー値の昇順でデータセットに書き込まれます。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'descending'**引数タグを使用する場合、データ項目はキー値の降順でデータセットに書き込まれます。**ordered** 引数タグを使用しない場合、順序は未定義になります。

出力データセットの名前を指定するときには、DATASET 引数タグで SAS データセットオプションを使用できます。データセットオプションは、データセットオプションが表示される SAS データセットにのみ適用されるアクションを指定します。次の操作を実行できます。

- 変数名の変更
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持

注: 削除または保持する変数は、DEFINEDATA メソッドまたは DEFINEKEY メソッドを使用することにより、ハッシュ テーブルに含めておく必要があります。そうでない場合、エラーが発生します。

- データセットのパスワードの指定

次の例では、WHERE データセットオプションを使用して、出力データセット OUT の特定のデータを選択します。

```
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;
```

次の例では、RENAME データセットオプションを使用して、出力データセット OUT の変数名 J を K に変更します。

```
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;
```

データセットオプションのリストについては、*SAS Viya Data Set Options: Reference* を参照してください。

注: OUTPUT メソッドを使用してデータセットを作成すると、ハッシュオブジェクトは出力データセットには含まれません。次の例では、H2 ハッシュオブジェクトは出力データセットから省略され、警告が SAS ログに書き込まれます。

```
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;
```

例

次のコードでは、天文学データを含むデータセット ASTRO を使用して、メシエ(OBJ)天体が赤経(RA)値の昇順で並べ替えられたハッシュオブジェクトを作成し、OUTPUT メソッドを使用してデータセットにデータを保存します。

```
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;
```

アウトプット 2.2 赤経値で並べ替えられたメシエ天体

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

関連項目:

- “データセットにハッシュオブジェクトデータを保存” (13 ページ)

メソッド:

- “DEFINEDATA メソッド” (56 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

PREV メソッド

基となるハッシュオブジェクトの前の値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

ハッシュオブジェクトを移動して逆のキー順序でデータ項目を返すには、PREV メソッドを反復して使用します。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータ項目を返します。LAST メソッドは、ハッシュオブジェクト内の最後のデータ項目を返します。

ハッシュオブジェクトの次のデータ項目を返すには、NEXT メソッドを使用できます。

注: PREV メソッドは、メソッド呼び出し後に使用できるように、データ変数をデータ項目の値に設定します。

関連項目:

- “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

メソッド:

- “FIRST メソッド” (69 ページ)

- “LAST メソッド” (75 ページ)
- “NEXT メソッド” (81 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

REF メソッド

CHECK メソッドと ADD メソッドを 1 つのメソッド呼び出しに統合します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

"DATA: *datavalue*"ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

詳細

CHECK メソッドと ADD メソッドを 1 つの REF メソッド呼び出しに統合できます。次のコードを変更できます。

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

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

変更後のコード

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

REF メソッドは、ハッシュオブジェクト内の各キーの出現数を数えるのに便利です。REF メソッドは、最初の ADD で各キーのキー集計を初期化し、後続の CHECK が出現するたびに ADD を変更します。

キー集計の詳細については、“キー集計の維持” (9 ページ) を参照してください。

例: REF メソッドを使用したキー集計

次の例では、キー集計に REF メソッドを使用します。

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

proc print data=count;
run;
```

アウトプット 2.3 REF メソッドを使用した DATA の出力

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

関連項目:

メソッド:

- “ADD メソッド” (42 ページ)
- “CHECK メソッド” (43 ページ)

REMOVE メソッド

指定したキーに関連付けられているデータをハッシュオブジェクトから削除します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定されている対応するキー変数と型が一致するキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を示している場合、REMOVE メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージが発行されます。

詳細

REMOVE メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVE メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。

1 つ目は、次のコードに示すように、キーを指定してから REMOVE メソッドを使用する方法です。

```
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;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、REMOVE メソッド呼び出しでキーを直接指定する方法です。

```
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;
```

注: REMOVE メソッドではデータ変数の値は変更されません。ハッシュオブジェクト内の値のみが削除されます。

注: ハッシュオブジェクトコンストラクタで **multidata:'y'**を指定すると、REMOVE メソッドは指定したキーのすべてのデータ項目を削除します。

例: ハッシュテーブルのキーの削除

この例では、ハッシュテーブルのキーを削除する方法を示します。

```
/* 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;
```

次の出力は、3 番目のオブジェクトのキーとデータ(キー=67、データ=C)が削除されることを示しています。

アウトプット 2.4 出力から削除されるキーとデータ

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

関連項目:

- “ハッシュオブジェクトのデータの置換と削除” (12 ページ)

メソッド:

- “ADD メソッド” (42 ページ)
- “DEFINEKEY メソッド” (59 ページ)
- “REMOVEDUP メソッド” (95 ページ)

REMOVEDUP メソッド

指定したキーの現在のデータ項目に関連付けられているデータをハッシュオブジェクトから削除します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を示している場合、REMOVEDUP メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージが発行されます。

詳細

REMOVEDUP メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVEDUP メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。1 つ目は、キーを指定してから、REMOVEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REMOVEDUP メソッド呼び出しでキーを直接指定する方法です。

注: REMOVEDUP メソッドでは、データ変数の値は変更されません。ハッシュオブジェクトの値のみが削除されます。

注: キーのデータ項目リストにあるデータ項目が 1 つのみの場合は、キーとデータがハッシュオブジェクトから削除されます。

比較

REMOVEDUP メソッドでは、指定したキーの現在のデータ項目に関連付けられているデータがハッシュオブジェクトから削除されます。REMOVE メソッドでは、指定したキーに関連付けられているデータがハッシュオブジェクトから削除されます。

例: キーの重複する項目の削除

この例では、複数のキーに複数のデータ項目があるハッシュオブジェクトを作成します。キーの 2 番目のデータ項目が削除されます。

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

次の行が SAS ログに書き出されます。

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

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “REMOVE メソッド” (92 ページ)

REPLACE メソッド

指定したキーに関連付けられたデータを新しいデータに置き換えます。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: **keyvalue**”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

要件 KEY:*keyvalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: **datavalue**”ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

要件 DATA:*datavalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

詳細

次の 2 つの方法のいずれかで REPLACE メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、次のコードに示すように、キーおよびデータ項目を定義してから REPLACE メソッドを使用する方法です。この例では、キー'Rottwlr'のデータが'1st'から'2nd'に変更されます。

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

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

data _null_;
  length brd $12;
  length plc $8;
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.show');
    rc = h.defineKey('brd');
    rc = h.defineData('brd', 'plc');
    rc = h.defineDone();
  end;
  /* Specify the key and new data value */
  brd = 'Rottwlr';
  plc = '2nd';
  /* Call the REPLACE method to replace the data value */
  rc = h.replace();
  /* Write the hash table to the data set. */
  rc = h.output(dataset: 'work.show');
run;

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

2 つ目は、次のコードに示すように、ショートカットを使用して、REPLACE メソッド呼び出しでキーおよびデータを直接指定する方法です。

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

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

注: ハッシュオブジェクトの REPLACE メソッドは、各キーごとに 1 つずつ項目があるハッシュテーブルと一緒に使用するためのものです(**MULTIDATA: 'NO'**)。一方、REPLACEDUP メソッドは、各キーごとに複数のデータ項目があるハッシュテーブルと一緒に使用するためのものです(**MULTIDATA: 'YES'**)。REPLACE メソッドを呼び出し、**multidata:'y'**オプションを使用してハッシュオブジェクトを宣言した場合、現在のキーに対するすべてのデータ項目が、新しいデータに置き換えられます。前のリリースでは、項目は置き換えられず、新しいデータが現在のキーに追加されます。MULTIDATA オプションの詳細については、“DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)を参照してください。

注: REPLACE メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACE メソッドでは、データ変数の値はデータ項目の値に置き換えられません。ハッシュオブジェクトの値のみが置き換えられます。

比較

REPLACE メソッドでは、指定したキーに関連付けられたデータが新しいデータに置き換えられます。REPLACEDUP メソッドでは、現在のキーの現在のデータ項目に関連付けられたデータが新しいデータに置き換えられます。

関連項目:

- “ハッシュオブジェクトのデータの置換と削除” (12 ページ)

メソッド:

- “DEFINEDATA メソッド” (56 ページ)
- “DEFINEKEY メソッド” (59 ページ)
- “REPLACEDUP メソッド” (99 ページ)

REPLACEDUP メソッド

現在のキーの現在のデータ項目に関連付けられたデータを新しいデータに置き換えます。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して現在のキーに定義されたデータ変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで REPLACEUP メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、データ項目を定義してから、REPLACEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REPLACEDUP メソッド呼び出しでデータを直接指定する方法です。

注: REPLACEDUP メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACEDUP メソッドでは、データ変数の値はデータ項目の値に置き換えられません。ハッシュオブジェクト内の値のみが置換されます。

比較

REPLACEDUP メソッドでは、現在のキーの現在のデータ項目に関連付けられたデータが新しいデータに置き換えられます。REPLACE メソッドでは、指定したキーに関連付けられたデータが新しいデータに置き換えられます。

例: 現在のキーのデータの置換

この例では、複数のキーに複数のデータ項目があるハッシュオブジェクトを作成します。重複データ項目が見つかった場合、データ項目の値に 300 が加算されます。

```
data testdup;
  length key data 8;
  input key data;
  datalines;
  1 10
  2 11
  1 15
```

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

次の行が SAS ログに書き出されます。

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

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “REPLACE メソッド” (97 ページ)

RESET_DUP メソッド

DO_OVER メソッド使用時にポインタをキーの重複リストの最初にリセットします。

適用対象: ハッシュオブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動し続けます。

反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

サンプルは DO_OVER メソッド例 (62 ページ)を参照してください。

関連項目:

メソッド:

- “DO_OVER メソッド” (61 ページ)

SETCUR メソッド

反復を開始するキー項目を指定します。

適用対象: ハッシュ反復子オブジェクト

構文

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

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

KEY: '*keyvalue*'
反復の開始キーとしてのキー値を指定します。

詳細

ハッシュ反復子を使うと、ハッシュオブジェクト内のどの項目でも反復を開始できます。SETCUR メソッドは、反復の開始キーを設定します。開始項目の指定には、KEY オプションを使用します。

例: 開始キー項目の指定

次の例では、天文学データを含むデータセットを作成します。最初の項目や最後の項目からではなく、RA= 18 31.6 から反復を開始します。データがハッシュオブジェクト内に読み込まれ、反復を開始するのに SETCUR メソッドが使用されます。*ordered* 引数タグが YES に設定されているため、出力が昇順で並べ替えられています。

```
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
;
```

次のコードでは、反復の開始キーが**'18 31.6'**に設定されます。

```
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;
```

次の行が SAS ログに書き出されます。

```
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
```

最初の項目または最後の項目から反復を開始するには、FIRST メソッドまたは LAST メソッドをそれぞれ使用します。

関連項目:

- “ハッシュ反復子オブジェクトの使用 ” (16 ページ)

メソッド:

- “FIRST メソッド” (69 ページ)
- “LAST メソッド” (75 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

SUM メソッド

ハッシュテーブルから特定のキーの集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: ハッシュオブジェクト

構文

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

必須引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

SUM: *variable-name*
特定のキーの現在の集計値を保存する DATA ステップ変数を指定します。

詳細

ハッシュオブジェクトからキーの集計値を取得するには、SUM メソッドを使用します。詳細については、“キー集計の維持” (9 ページ)を参照してください。

比較

SUM メソッドでは、1 つのキーにつき 1 つのデータ項目しか存在しないときに、特定のキーの集計値を取得します。SUMDUP メソッドでは、1 つのキーにつき複数のデータ項目が存在するときに、現在のキーの現在のデータ項目について集計値を取得します。

例: 特定のキーのキー集計値の取得

次の例では、2 つのキー K=99 および K=100 を指定し、SUM メソッドを使用して各キーの集計値を取得します。

```
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;
```

最初の PUT ステートメントは k=100 の集計値を出力します。

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

2 番目の PUT ステートメントは k=99 の集計値を出力します。

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

関連項目:

メソッド:

- “ADD メソッド” (42 ページ)
- “FIND メソッド” (64 ページ)
- “CHECK メソッド” (43 ページ)
- “DEFINEKEY メソッド” (59 ページ)
- “REF メソッド” (90 ページ)
- “SUMDUP メソッド” (106 ページ)

演算子:

- “_NEW_演算子、ハッシュオブジェクトおよびハッシュ反復子オブジェクト” (76 ページ)

ステートメント:

- “DECLARE ステートメントのハッシュオブジェクトとハッシュ反復子オブジェクト” (47 ページ)

SUMDUP メソッド

現在のキーの現在のデータ項目について集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: ハッシュオブジェクト

構文

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

引数

rc

メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

SUM: ***variable-name***
現在のキーの現在のデータ項目について取得した集計値を保存する DATA ステップ変数を指定します。

詳細

キーに複数のデータ項目がある場合、ハッシュオブジェクトからキーの集計値を取得するには、SUMDUP メソッドを使用します。詳細については、“キー集計の維持” (9 ページ)を参照してください。

比較

SUMDUP メソッドでは、1 つのキーにつき複数のデータ項目が存在するときに、現在のキーの現在のデータ項目について集計値を取得します。SUM メソッドでは、1 つのキーにつき 1 つのデータ項目しか存在しないときに、特定のキーの集計値を取得します。

例: 集計値の取得

次の例では、SUMDUP メソッドを使って、現在のデータ項目の集計値を取得します。この例は、HAS_PREV メソッドと FIND_PREV メソッドを使うと、リストの前方にループできることも示しています。FIND_PREV メソッドは、現在のリスト項目については FIND_NEXT メソッドと同様の処理を行います。異なるのは、複数の項目リストで前方に移動する点です。

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

```
      do while(r ne 0);
        rc = h.find_next();
        rc = h.find_prev();
        rc = h.find_next();
        h.has_next(result: r);
      end;
    end;
 end;
i = 0;
do key = 1 to 5;
  rc = h.find();
  if (rc = 0) then do;
     h.sum(sum: sum);
     put key= data= sum=;
     h.has_next(result: r);
     do while(r ne 0);
       rc = h.find_next();
       h.sumdup(sum: sum);
       put 'dup ' key= data= sum=;
       h.has_next(result: r);
     end;
    end;
end;
run;
```

次の行が SAS ログに書き出されます。

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

10、15 および 5(この順序で保存されているものとします)の 3 つのデータ値を持つキー 1 を例にとって、この処理を見てみます。

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

最初の **do key = 1 to 5;**ループのデータリストを移動中、最初の FIND メソッド呼び出しでデータ項目 10 のキー集計値が 1 に設定されます。最初の FIND_NEXT メソッド呼び出しでデータ項目 15 のキー集計値が 1 に設定されます。次の FIND_PREV メソッド呼び出しでは、データ項目 10 に戻り、キー集計値が 2 に増分されます。さらに、最後の FIND_NEXT メソッド呼び出しで、データ項目 15 のキー集計値が 2 に設定されます。ループの次の反復では、データ項目 5 のキー集計値が 1 に設定され、データ項目 15 のキー集計値が 3 に設定されます。最後に、データ項目 5 のキー集計値がキー集計値が 2 に増分されます。

この例では、リストに以前のエントリが存在していることがわかっているため、FIND_PREV メソッドを呼び出す前に HAS_PREV メソッドを呼び出していません。存在しなかった場合は、ループに入りません。

このことから、重複処理によっては特殊なメソッドが必要なこともわかります。(この場合、SUMDUP メソッドが SUM メソッドと同様の処理を、現在のリスト項目のキー集計値について行います。)

関連項目:

- “重複するキーとデータペア” (6 ページ)

メソッド:

- “SUM メソッド” (104 ページ)

3 章

Java オブジェクトの言語要素のディクショナリ

カテゴリ別の Java オブジェクトメソッド

Java オブジェクトメソッドには 5 つのカテゴリがあります。

表 3.1 カテゴリ別の Java オブジェクトメソッド

カテゴリ	説明
削除	Java オブジェクトを削除できます。
例外	例外情報を収集して例外をクリアできます。
フィールド参照	Java オブジェクトの静的なインスタンスフィールドまたは静的でないインスタンスフィールドの値を返したり設定したりできます。
メソッド参照	静的な Java メソッドと静的でない Java メソッドにアクセスできます。
出力	出力先に Java 出力を即座に送信できます。

次の表に、Java オブジェクトメソッドの概要を示します。詳細については、各メソッドの辞書エントリを参照してください。

カテゴリ	言語要素	説明
削除	DELETE メソッド、Java オブジェクト (p. 119)	Java オブジェクトを削除します。
出力	FLUSHJAVAOUTPUT メソッド (p. 125)	出力先に Java 出力を送信するように指定します。
フィールド参照	GET*type*FIELD メソッド (p. 126)	Java オブジェクトの静的でないフィールドの値を返します。
	GETSTATIC*type*FIELD メソッド (p. 128)	Java オブジェクトの静的なフィールドの値を返します。
	SET*type*FIELD メソッド (p. 132)	Java オブジェクトの静的でないフィールドの値を変更します。
	SETSTATIC*type*FIELD メソッド (p. 134)	Java オブジェクトの静的フィールドの値を変更します。
メソッド参照	CALL*type*METHOD メソッド (p. 112)	静的でない Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。
	CALLSTATIC*type*METHOD メソッド (p. 115)	静的な Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。
例外	EXCEPTIONCHECK メソッド (p. 120)	メソッド呼び出し時に例外が発生したかどうかを確認します。
	EXCEPTIONCLEAR メソッド (p. 121)	現在、発生している例外をクリアします。
	EXCEPTIONDESCRIBE メソッド (p. 123)	デバッグログのオン/オフを切り替え、例外情報を出力します。

ディクショナリ

CALL*type*METHOD メソッド

静的でない Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。

カテゴリ: メソッド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
静的でない Java メソッドの結果型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
結果型に BOOLEAN を指定します。

BYTE
結果型に BYTE を指定します。

CHAR
結果型に CHAR を指定します。

DOUBLE
結果型に DOUBLE を指定します。

FLOAT
結果型に FLOAT を指定します。

INT
結果型に INT を指定します。

LONG
結果型に LONG を指定します。

SHORT
結果型に SHORT を指定します。

STRING
結果型に STRING を指定します。

VOID
結果型に VOID を指定します。

参照項目: “型の問題” (22 ページ)

method-name
静的でない Java メソッドの名前を指定します。

要件 メソッド名は一重引用符または二重引用符で囲む必要があります。

method-argument
メソッドに渡すパラメータを指定します。

return-value
メソッドで値を返す場合の戻り値を指定します。

詳細

Java オブジェクトをインスタンス化したら、CALL*type*METHOD メソッドを使用した Java オブジェクトでのメソッド呼び出しによって、静的でない Java メソッドにアクセスできます。

注: type 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

静的でない Java メソッドには CALL*type*METHOD メソッドを使用します。Java メソッドが静的な場合は、CALLSTATIC*type*METHOD メソッドを使用します。

例: フィールドの値の設定と取得

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、CALLC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “CALLSTATICtypeMETHOD メソッド” (115 ページ)

CALLSTATIC*type*METHOD メソッド

静的な Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。

カテゴリ: メソッド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
静的な Java メソッドの結果型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
結果型に BOOLEAN を指定します。

BYTE
結果型に BYTE を指定します。

CHAR
結果型に CHAR を指定します。

DOUBLE
結果型に DOUBLE を指定します。

FLOAT
結果型に FLOAT を指定します。

INT
結果型に INT を指定します。

LONG
結果型に LONG を指定します。

SHORT
結果型に SHORT を指定します。

STRING
結果型に STRING を指定します。

VOID
結果型に VOID を指定します。

参照項目: “型の問題” (22 ページ)

method-name
静的な Java メソッド名を指定します。

要件 メソッド名は一重引用符または二重引用符で囲む必要があります。

method-argument
メソッドに渡すパラメータを指定します。

return-value
メソッドで値を返す場合の戻り値を指定します。

詳細

Java オブジェクトをインスタンス化したら、CALLSTATIC*type*METHOD メソッドを使用した Java オブジェクトでのメソッド呼び出しによって、静的な Java メソッドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

静的な Java メソッドには CALLSTATIC*type*METHOD メソッドを使用します。Java メソッドが静的でない場合は、CALL*type*METHOD メソッドを使用します。

例: 静的なフィールドの設定と取得

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、CALLSTATIC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

```
d=3.14159
```

関連項目:

メソッド:

- “CALLtypeMETHOD メソッド” (112 ページ)

DECLARE ステートメント、Java オブジェクト

Java オブジェクトを宣言します。Java オブジェクトのインスタンスを作成し、データを初期化します。

別名: DCL

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

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

引数

object-reference
Java オブジェクトのオブジェクト参照名を指定します。

java-class
インスタンス化する Java クラスの名前を指定します。

要件 Java クラス名は二重引用符または一重引用符で囲む必要があります。

Java パッケージのパスを指定する場合、パスにはピリオド(.)ではなくフォワードスラッシュ(/)を使用する必要があります。たとえば、*"java.util.Hashtable"*は正しいクラス名ではありません。正しいクラス名は*"java/util/Hashtable"*です。

argument
Java オブジェクトのインスタンスの作成に使用する情報を指定します。*argument* の有効な値は Java オブジェクトによって異なります。

参照項目: “DECLARE ステートメントを使用した Java オブジェクトのインスタンス生成(フォーム 2)” (118 ページ)

詳細

基本

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

詳細については、“DATA ステップコンポーネントオブジェクトについて” (2 ページ)を参照してください。

Java オブジェクトの宣言(フォーム 1)

DECLARE ステートメントを使用して Java オブジェクトを宣言します。

```
declare javaobj j;
```

DECLARE ステートメントは、オブジェクト参照 J が Java オブジェクトであることを SAS に示します。

新しい Java オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。たとえば、次のコード行では、_NEW_演算子で Java オブジェクトが作成され、オブジェクト参照 J に割り当てられます。

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

DECLARE ステートメントを使用した Java オブジェクトのインスタンス生成(フォーム 2)

DECLARE ステートメントと_NEW_演算子を使用して Java オブジェクトを宣言し、インスタンス化する 2 ステップのプロセスの代わりに、DECLARE ステートメントを使用して、Java オブジェクトを 1 つのステップで宣言し、インスタンス化することができます。たとえば、次のコード行では、DECLARE ステートメントで Java オブジェクトが宣言およびインスタンス化され、Java オブジェクトがオブジェクト参照 J に割り当てられます。

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

前述のコード行は、次のコードを使用するのと同等です。

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

*コンストラクタ*は、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用できるメソッドです。たとえば、次のコード行では、DECLARE ステートメントで Java オブジェクトが宣言およびインスタンス化され、Java オブジェクトがオブジェクト参照 J に割り当てられます。Java オブジェクトコンストラクタで唯一必要な引数は、インスタンス化する Java クラスの名前です。その他の引数はすべて Java クラス自体のコンストラクタ引数です。次の例では、Java クラス名 **testjavaclass** がコンストラクタで、値 **100** と **.8** はコンストラクタ引数です。

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

比較

Java オブジェクトを宣言し、インスタンスを作成するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例

例 1: DECLARE ステートメントと_NEW_演算子を使用した Java オブジェクトの宣言とインスタンス化

次の例では、単純な Java クラスが作成されます。DECLARE ステートメントと_NEW_演算子を使用して、このクラスのインスタンスを作成します。

```
/* 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;
```

例 2: DECLARE ステートメントを使用した Java オブジェクトの作成とインスタンス化

次の例では、ハッシュテーブルの Java クラスが作成されます。DECLARE ステートメントを使用して、容量および負荷係数を指定し、このクラスのインスタンスを作成します。この例では、DATA ステップの唯一の数値型が Java のデータ型 DOUBLE と同等であるため、ラッパークラス **mhash** が必要です。

```
/* 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;
```

関連項目:

演算子:

- “_NEW_演算子の Java オブジェクト” (130 ページ)

DELETE メソッド、Java オブジェクト

Java オブジェクトを削除します。

カテゴリ: 削除

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

object.**DELETE**( );

引数

object
Java オブジェクト名を指定します。

詳細

DATA ステップのコンポーネントオブジェクトは、DATA ステップの終了時に自動的に削除されます。他の Java オブジェクトコンストラクタでオブジェクト参照変数を再利用する場合は、DELETE メソッドで Java オブジェクトを削除する必要があります。

削除した Java オブジェクトを使用しようとすると、エラーがログに書き込まれます。

EXCEPTIONCHECK メソッド

メソッド呼び出し時に例外が発生したかどうかを確認します。

カテゴリ: 例外

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

status
返される例外ステータスを指定します。

ヒント Java から返されるステータス値のデータ型は DOUBLE です。これは SAS の数値データ値に対応します。

詳細

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

メソッド呼び出し時に例外が発生したかどうかを確認するには、EXCEPTIONCHECK メソッドを使います。例外が発生する可能性のある Java メソッドを呼び出した後は、毎回 EXCEPTIONCHECK メソッドを呼び出すのが理想的です。

例: 例外の確認

次の例では、例外が発生するメソッドが Java クラスに含まれています。DATA ステップはそのメソッドを呼び出し、例外の有無を確認します。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

```
exception
```

関連項目:

メソッド:

- “EXCEPTIONCLEAR メソッド” (121 ページ)
- “EXCEPTIONDESCRIBE メソッド” (123 ページ)

EXCEPTIONCLEAR メソッド

現在、発生している例外をクリアします。

カテゴリ: 例外

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

object.**EXCEPTIONCLEAR**( );

引数

object
Java オブジェクト名を指定します。

詳細

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

例外が発生するメソッドを呼び出す場合は、呼び出し後に例外の有無を確認することをお勧めします。例外が発生した場合は、適切な操作を実行してから、EXCEPTIONCLEAR メソッドを使って例外をクリアします。

例外が発生していない場合、このメソッドには何の効果もありません。

例

例 1: 例外の確認とクリア

次の例では、例外が発生するメソッドが Java クラスに含まれています。DATA ステップでこのメソッドを呼び出した後、例外をクリアします。

```
/* 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;
```

次の行が SAS ログに書き出されます。

```
exception
no exception
```

例 2: 外部ファイル読み込み時の例外の確認

次の例では、Java IO クラスを使用して DATA ステップから外部ファイルを読み込みます。この Java コードは **DataInputStream** のラッパークラスを作成し、**FileInputStream** をコンストラクタに渡せるようにします。コンストラクタが実際に取得するのは、**FileInputStream** の親である **InputStream** です。現在のメソッドルックアップはスーパークラスルックアップを実行できるほど堅牢でないため、ラッパーが必要です。

```
/* Java code */
public class myDataInputStream extends java.io.DataInputStream
{
```

```
  myDataInputStream(java.io.FileInputStream fi)
  {
    super(fi);
  }
}
```

作成したラッパークラスを使って外部ファイルの **DataInputStream** を作成し、ファイルの終端に達するまでファイルを読み込みます。EXCEPTIONCHECK メソッドを使って **readInt** メソッドでの **EOFException** 発生を判定し、入力ループを終了させることができます。

```
/* 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;
```

関連項目:

メソッド:

- “EXCEPTIONCHECK メソッド” (120 ページ)
- “EXCEPTIONDESCRIBE メソッド” (123 ページ)

EXCEPTIONDESCRIBE メソッド

デバッグログのオン/オフを切り替え、例外情報を出力します。

カテゴリ: 例外

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

status
デバッグログのオン/オフを指定します。**status** 引数は次のいずれかの値です。

0
デバッグログをオフに指定します。

1
デバッグログをオンに指定します。

デフォルト 0(オフ)

ヒント Java から返されるステータス値のデータ型は DOUBLE です。これは SAS の数値データ値に対応します。

詳細

EXCEPTIONDESCRIBE メソッドは、例外のデバッグログのオンとオフを切り替えます。例外のデバッグログがオンの場合、例外情報が JVM の標準出力に出力されます。

注: デフォルトでは、JVM の標準出力は SAS ログにリダイレクトされます。

例: 例外情報の標準出力への出力

次の例では、例外情報は標準出力に出力されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “EXCEPTIONCHECK メソッド” (120 ページ)
- “EXCEPTIONCLEAR メソッド” (121 ページ)

FLUSHJAVAOUTPUT メソッド

出力先に Java 出力を送信するように指定します。

カテゴリ: 出力

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

object.**FLUSHJAVAOUTPUT**( );

引数

object
Java オブジェクト名を指定します。

詳細

SAS ログに対する Java 出力は、DATA ステップの終了時にフラッシュされます。FLUSHJAVAOUTPUT メソッドを使用した場合、Java 出力は DATA ステップの実行中に発行されたすべての 出力の後に表示されます。

例: Java 出力の表示

次の例では、DATA ステップの完了後に"In Java class"行が書き込まれます。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

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

FLUSHJAVAOUTPUT メソッドを使用した場合、Java 出力が実行順に SAS ログに書き込まれます。

```
/* 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;
```

次の行が SAS ログに書き込まれます。

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

関連項目:

"Java 標準出力" (27 ページ)

GET*type*FIELD メソッド

Java オブジェクトの静的でないフィールドの値を返します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題” (22 ページ)

field-name
Java フィールド名を指定します。

要件 フィールド名は一重引用符または二重引用符で囲む必要があります。

value
返されたフィールド値を受け取るする変数名を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。GET*type*FIELD メソッドでは、静的でないフィールドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

GET*type*FIELD メソッドは、Java オブジェクトの静的でないフィールドの値を返します。静的なフィールドの値を返すには、GETSTATIC*type*FIELD メソッドを使用します。

例: 静的でないフィールドの値の取得

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、GET*type*FIELD メソッドを使用してフィールドの値が変更および取得されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “GETSTATICtypeFIELD メソッド” (128 ページ)
- “SETtypeFIELD メソッド” (132 ページ)

GETSTATIC*type*FIELD メソッド

Java オブジェクトの静的なフィールドの値を返します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題” (22 ページ)

field-name
Java フィールド名を指定します。

要件 フィールド名は一重引用符または二重引用符で囲む必要があります。

value
返されたフィールド値を受け取るする変数名を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。GETSTATIC*type*FIELD メソッドでは、静的なフィールドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

GETSTATIC*type*FIELD メソッドは、Java オブジェクトの静的なフィールドの値を返します。静的でないフィールドの値を返すには、GET*type*FIELD メソッドを使用します。

例: 静的なフィールドの値の取得

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、GETSTATIC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “GETtypeFIELD メソッド” (126 ページ)
- “SETSTATICtypeFIELD メソッド” (134 ページ)

_NEW_演算子の Java オブジェクト

Java オブジェクトのインスタンスを作成します。

該当要素: DATA ステップ

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object-reference
Java オブジェクトのオブジェクト参照名を指定します。

java-class
インスタンス化する Java クラスの名前を指定します。

要件 Java クラス名は一重引用符または二重引用符で囲む必要があります。

argument
Java オブジェクトのインスタンスの作成に使用する情報を指定します。*argument* の有効な値は Java オブジェクトによって異なります。

詳細

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

_NEW_演算子を使用して Java オブジェクトをインスタンス化する場合は、最初に DECLARE ステートメントを使用して Java オブジェクトを宣言する必要があります。たとえば、次のコード行では、DECLARE ステートメントは、オブジェクト参照 J が Java オブジェクトであることを SAS に示します。_NEW_演算子で Java オブジェクトが作成され、オブジェクト参照 J に割り当てられます。

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

注: DECLARE ステートメントを使用して Java オブジェクトの宣言とインスタンス化を 1 つのステップで実行できます。

コンストラクタは、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用するメソッドです。たとえば、次のコード行では、_NEW_演算子で Java オブジェクトがインスタンス化され、オブジェクト参照 J に割り当てられます。Java オブジェクトコンストラクタで唯一必要な引数は、インスタンス化する Java クラスの名前です。その他の引数はすべて Java クラス自体のコンストラクタ引数です。次の例では、Java クラス名 **testjavaclass** がコンストラクタで、値 **100** と **.8** はコンストラクタ引数です。

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

定義済みの DATA ステップコンポーネントオブジェクトおよびコンストラクタの詳細については、“DATA ステップコンポーネントオブジェクトについて” (2 ページ)を参照してください。

比較

Java オブジェクトを宣言し、インスタンスを作成するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例: _NEW_演算子を使用した Java クラスのインスタンス化と初期化

次の例では、ハッシュテーブルの Java クラスが作成されます。_NEW_演算子を使用して、容量および負荷係数を指定し、このクラスのインスタンスを作成します。この例では、DATA ステップの唯一の数値型が Java のデータ型 DOUBLE と同等であるため、ラッパークラス **mhash** が必要です。

```
/* 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;
```

関連項目:

ステートメント:

- “DECLARE ステートメント、Java オブジェクト” (117 ページ)

SET*type*FIELD メソッド

Java オブジェクトの静的でないフィールドの値を変更します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題” (22 ページ)

field-name
Java フィールド名を指定します。

要件 フィールド名は一重引用符または二重引用符で囲む必要があります。

value
フィールドの値を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。SET*type*FIELD メソッドを使用すると、静的でないフィールドの値を変更できます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

SET*type*FIELD メソッドは、Java オブジェクトの静的でないフィールドの値を変更します。静的フィールドの値を変更するには、SETSTATIC*type*FIELD メソッドを使用します。

例: 静的でないフィールドを持つ Java クラスの作成

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、SET*type*FIELD メソッドでフィールド値が設定されてから、フィールド値が取得されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “GETtypeFIELD メソッド” (126 ページ)
- “SETSTATICtypeFIELD メソッド” (134 ページ)

SETSTATIC*type*FIELD メソッド

Java オブジェクトの静的フィールドの値を変更します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

制限事項: このメソッドは、CAS Server ではサポートされていません。

構文

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

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題” (22 ページ)

field-name
Java フィールド名を指定します。

要件 フィールド名は一重引用符または二重引用符で囲む必要があります。

value
フィールドの値を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。SETSTATIC*type*FIELD メソッドを使用すると、静的フィールドの値を変更できます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題” (22 ページ)を参照してください。

比較

SETSTATIC*type*FIELD メソッドは、Java オブジェクトの静的フィールドの値を変更します。静的でないフィールドの値を変更するには、SET*type*FIELD メソッドを使用します。

例: 静的フィールドを持つ Java クラスの作成

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、SETSTATIC*type*FIELD メソッドでフィールド値が設定されてから、フィールド値が取得されます。

```
/* 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;
```

次の行が SAS ログに書き出されます。

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

関連項目:

メソッド:

- “GETSTATICtypeFIELD メソッド” (128 ページ)
- “SETtypeFIELD メソッド” (132 ページ)

推奨資料

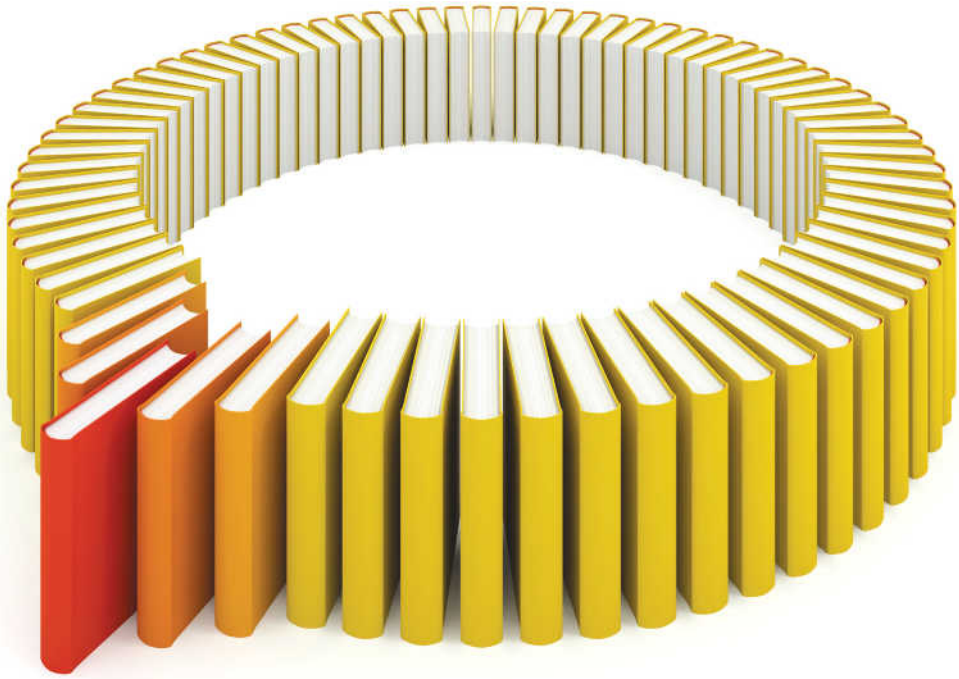
本書の内容に関連する推奨される参考文献のリストを次に示します。

- *SAS Viya Data Set Options: Reference*
- *SAS Hash Object Programming Made Easy*
- *SAS Viya ステートメント: リファレンス*

SAS 刊行物の一覧については、 sas.com/store/books から入手できます。必要な書籍についての質問は SAS 担当者までお寄せください:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
電話: 1-800-727-0025
ファクシミリ: 1-919-677-4444
メール: sasbook@sas.com
Web アドレス: sas.com/store/books

キーワード



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.