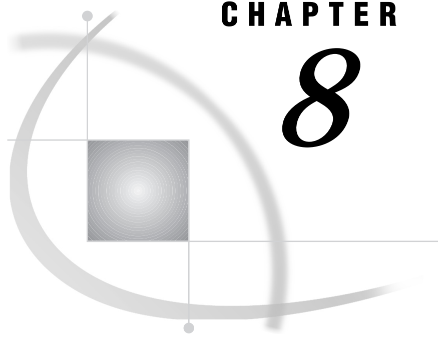


CHAPTER
8

# SAS Object-Oriented Programming Concepts

*Introduction* 94
*Object-Oriented Development and the SAS Component Object Model* 95
*Classes* 96
  *Relationships among Classes* 96
    *Inheritance* 96
    *Instantiation* 97
  *Types of Classes* 97
    *Abstract Classes* 97
    *Models and Views* 98
    *Metaclasses* 98
  *Defining Classes* 98
    *Generating CLASS Entries from CLASS Blocks* 99
    *Generating CLASS Blocks from CLASS Entries* 99
    *Referencing Class Methods or Attributes* 99
  *Instantiating Classes* 100
*Methods* 100
  *Defining Method Scope* 102
  *Defining Method Names and Labels* 102
    *Specifying a Name That Is Different from the Label* 102
    *Using Underscores in Method Names* 103
  *Specifying Parameter Types and Storage Types* 103
  *Passing Objects as Arguments for Methods* 104
  *Returning Values From Methods* 105
  *Method Signatures* 105
    *Signature Strings (SIGSTRINGs)* 106
    *How Signatures Are Used* 107
    *Altering Existing Signatures* 107
  *Forward-Referencing Methods* 107
  *Overloading Methods* 108
    *Example: Different Parameter Types* 108
    *Example: Different Numbers of Parameters* 109
    *Defining One Implementation That Accepts Optional Parameters* 111
    *Overloading and List, Object, and Numeric Types* 111
  *Overriding Existing Methods* 111
  *Defining Constructors* 112
    *Overloading Constructors* 112
    *Overriding the Default Constructor* 113
    *Calling Constructors Explicitly* 113
    *Specifying That a Method Is Not a Constructor* 114
  *Implementing Methods Outside of Classes* 115
  *Method Metadata* 115

*Attributes* **116**
*Creating Attributes Automatically* **116**
*Specifying Where an Attribute Value Can Be Changed* **117**
*Setting Initial Values and the List of Valid Values* **117**
*Associating Custom Access Methods with Attributes* **118**
*Linking Attributes* **118**
*Attribute Metadata* **119**
*Accessing Object Attributes and Methods With Dot Notation* **119**
*Syntax* **119**
*Using Nested Dot Notation* **120**
*Examples* **121**
*What Happens When Attribute Values Are Set or Queried* **122**
*Setting Attribute Values* **123**
*Querying Attribute Values* **124**
*Events and Event Handlers* **125**
*System Events* **126**
*Defining and Sending Events* **126**
*Defining Event Handlers* **126**
*Example* **126**
*Event and Event Handler Metadata* **128**
*Interfaces* **128**
*Defining Interfaces* **129**
*Example* **129**
*Converting Version 6 Non-Visual Classes to Version 8 Classes* **131**
*Removing Global Variables* **132**
*Declaring Variables* **133**
*Converting Labels and LINK Statements* **133**
*Converting CALL SEND to Dot Notation* **134**
*Converting Class Definitions with CREATESCL* **134**
*Using Instance Variables* **135**

# Introduction

*Object-oriented programming* (OOP) is a technique for writing computer software. The term *object oriented* refers to the methodology of developing software in which the emphasis is on the data, while the procedure or program flow is de-emphasized. That is, when designing an OOP program, you do not concentrate on the order of the steps that the program performs. Instead, you concentrate on the data in the program and on the operations that you perform on that data.

Advocates of object-oriented programming claim that applications that are developed using an object-oriented approach

- are easier to understand because the underlying code maps directly to real-world concepts that they seek to model
- are easier to modify and maintain because changes tend to involve individual objects and not the entire system
- promote software reuse because of modular design and low interdependence among modules
- offer improved quality because they are constructed from stable intermediate classes
- provide better scalability for creating large, complex systems.

*Object-oriented application design* determines which operations are performed on which data, and then groups the related data and operations into categories. When the

design is implemented, these categories are called *classes*. A class defines the data and the operations that you can perform on the data. In SCL, the data for a class is defined through the class's *attributes*, *events*, *event handlers*, and *interfaces*. (Legacy classes store data in *instance variables*.) The operations that you perform on the data are called *methods*.

*Objects* are data elements in your application that perform some function for you. Objects can be *visual* objects that you place on the frame—for example, icons, push buttons, or radio boxes. Visual objects are called *controls*; they display information or accept user input.

Objects can also be *nonvisual* objects that manage the application behind the scenes; for example, an object that enables you to interact with SAS data sets may not have a visual representation but still provides you with the functionality to perform actions on a SAS data set such as accessing variables, adding data, or deleting data. An *object* or *component* is derived from, or is an *instance* of, a class. The terms object, component, and instance are interchangeable.

Software objects are self-contained entities that possess three basic characteristics:

behavior
: a collection of operations that an object can perform on itself or on other objects. *Methods* define the operations that an object can perform. For example, you can use the _onGeneric method in **sashelp.classes.programHalt.class** to trap all generic errors.

state
: a collection of attributes and their current values. Two of the programHalt component's attributes are **stopExecution** (which determines whether the program continues to execute after the program halt occurs) and **dump** (which contains the program-halt information). You can set these values through SCL.

identity
: a unique value that distinguishes one object from another. This identifier is referred to as its *object identifier*. The object identifier is created by SCL when you instantiate an object with the _NEW_ operator. This identifier is also used as the first-level qualifier in SCL dot notation.

This chapter describes how object-oriented techniques and related concepts are implemented in SCL.

# Object-Oriented Development and the SAS Component Object Model

The SAS Component Object Model (SCOM) provides a flexible framework for SCL component developers. With SCOM, you can develop model components that communicate with viewer components that are built with other SAS software (such as SAS/AF and WebAF) or with software from other vendors.

A component in SCOM is a self-contained, reusable object that has specific properties, including

- a set of attributes and methods
- a set of events that the object sends
- a set of event handlers that execute in response to various types of events
- a set of supported or required interfaces.

With SCL, you can design components that communicate with each other, using any of the following processes:*

* Drag and drop operations can be defined only through SAS/AF software, not through SCL.

Attribute linking
: enabling a component to change one of its attributes when the value of another attribute is changed.

Model/view communication
: enabling a view (typically a visual control) to communicate with a model, based on a set of common methods that are defined in an interface.

Event handling
: enabling a component to send an event that another component can respond to by using an associated event handler.

Classes form the foundation of the SCOM architecture by defining these attributes, methods, events, event handlers and interfaces. There are two ways to construct a class that uses the SAS Component Object Model:

- You can build a class with the Class Editor that is available in SAS/AF software.
- You can use SCL class syntax to construct a class.

This chapter provides detailed information about using SCL to create and modify classes.

# Classes

A *class* defines a set of data and the operations you can perform on that data. *Subclasses* are similar to the classes from which they are derived, but they may have different properties or additional behavior. In general, any operation that is valid for a class is also valid for each subclass of that class.

## Relationships among Classes

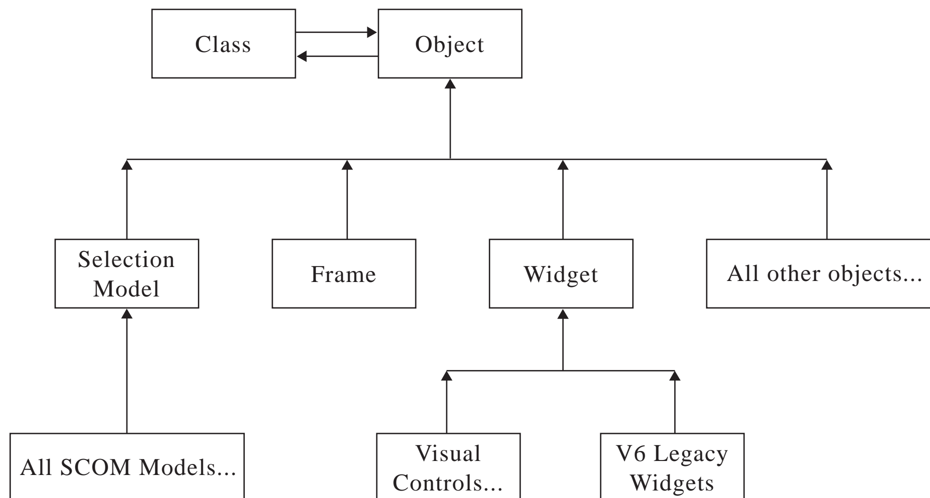
Classes that you define with SCL can support two types of relationships:

- inheritance
- instantiation.

### Inheritance

Generally, the attributes, methods, events, event handlers, and interfaces that belong to a parent class are automatically inherited by any class that is created from it. One metaphor that is used to describe this relationship is that of the *family*. Classes that provide the foundation for other classes are called *parent* classes, and classes that are derived from parent classes are *child* classes. When more than one class is derived from the same parent class, these classes are related to each other as *sibling* classes. A *descendent* of a class has that class as a parent, either directly or indirectly through a series of parent-child relationships. In object-oriented theory, any subclass that is created from a parent class *inherits* all of the characteristics of the parent class that it is not specifically prohibited from inheriting. The chain of parent classes is called an *ancestry*.

**Figure 8.1** Class Ancestry




Whenever you create a new class, that class inherits all of the properties (attributes, methods, events, event handlers, and interfaces) that belong to its parent class. For example, the Object class is the parent of all classes in SAS/AF software. The Frame and Widget classes are subclasses of the Object class, and they inherit all properties of the Object class. Similarly, every class you use in a frame-based application is a descendent of the Frame, Object, or Widget class, and thus inherits all the properties that belong to those classes.

## Instantiation

In addition to the inheritance relationship, classes have an *instantiation* or an "is a" relationship. For example, a frame is an instance of the Frame class; a radio box control is an instance of the Radio Box Control class; and a color list object is an instance of the Color List Model class.

All classes are instances of the Class class. The Class class is a metaclass. A *metaclass* collects information about other classes and enables you to operate on other classes. For more information about metaclasses, see "Metaclasses" on page 98.

# Types of Classes

Some SAS/AF software classes are specific types of classes.

- Abstract classes
- Models and views
- Metaclasses.

## Abstract Classes

*Abstract classes* group attributes and methods that are common to several subclasses. These classes themselves cannot be instantiated; they simply provide functionality for their subclasses.

The Widget class in SAS/AF software is an example of an abstract class. Its purpose is to collect properties that all widget subclasses can inherit. The Widget class cannot be instantiated.

## Models and Views

In SAS/AF software, components that are built on the SAS Component Object Model (SCOM) framework can be classified either as *views* that display data or as *models* that provide data. Although models and views are typically used together, they are nevertheless independent components. Their independence allows for customization, flexibility of design, and efficient programming.

*Models* are non-visual components that provide data. For example, a Data Set List model contains the properties for generating a list of SAS data sets (or tables), given a specific SAS library. A model may be attached to multiple views.

*Views* are components that provide a visual representation of the data, but they have no knowledge of the actual data they are displaying. The displayed data depends on the state of the model that is connected to the view. A view can be attached to only one model at a time.

It may be helpful to think of model/view components as client/server components. The view acts as the client and the model acts as the server.

For more information on interfaces, see “Interfaces” on page 128. For more information on implementing model/view communication, refer to *SAS Guide to Applications Development* and to the SAS/AF online Help.

## Metaclasses

As previously mentioned, the Class class (`sashelp.fsp.Class.class`) and any subclasses you create from it are metaclasses. *Metaclasses* enable you to collect information about other classes and to operate on those classes.

Metaclasses enable you to make changes to the application at run time rather than only at build time. Examples of such changes include where a class’s methods reside, the default values of class properties, and even the set of classes and their hierarchy.

Metaclasses also enable you to access information about parent classes, subclasses, and the methods and properties that are defined for a class. Specifically, through methods of the Class class, you can

- retrieve information about an application, such as information about the application’s structure, which classes are being used, and which legacy classes use particular instance variables. Each class has a super class that is accessed by the _getSuper method. Every class also maintains a list of subclasses that is accessed with the _getSubclassList and _getSubclasses methods.
- list the instances of a class and process all of those instances in some way. Each class maintains a list of its instances. You can use _getInstanceList and _getInstances to process all the instances.
- create objects and classes at run time with the _new method. Instances of the metaclass are other classes.

For more information about metaclasses, see the Class class in the SAS/AF online Help.

---

## Defining Classes

You can create classes in SCL with the CLASS block. The CLASS block begins with the CLASS statement and ends with the ENDCLASS statement:

<**ABSTRACT**> **CLASS** *class-name*<EXTENDS *parent-class-name*><br>
<SUPPORTS *supports-interface-clause*><br>
<REQUIRES *requires-interface-clause*><br>
< / (*class-optional-clause*)><br>
<(*attribute-statements*)>

<(*method-declaration-statements*)>
<(*method-implementation-blocks*)>
<(*event-declaration-statements*)>
<(*eventhandler-declaration-statements*)>

**ENDCLASS**;

The CLASS statement enables you to define attributes, methods, events, and event handlers for a class and to specify whether the class supports or requires an interface. The remaining sections in this chapter describe these elements in more detail.

The EXTENDS clause specifies the parent class. If you do not specify an EXTENDS clause, SCL assumes that **`sashelp.fsp.object.class`** is the parent class.

Using the CLASS block instead of the Class Editor to create a class enables the compiler to detect errors at compile time, which results in improved performance during run time.

For a complete description of the CLASS statement, see “CLASS” on page 277. For a description of using the Class Editor to define classes, refer to *SAS Guide to Applications Development*.

## Generating CLASS Entries from CLASS Blocks

Suppose you are editing an SCL entry in the Build window and that the entry contains a CLASS block. For example:

```
class Simple extends myParent;
  public num num1;
  M1: method n:num return=num / (scl='work.a.uSimple.scl');
  M2: method return=num;
    num1 = 3;
    dcl num n = M1(num1);
    return (n);
  endmethod;
endclass;
```

To generate a CLASS entry from the CLASS block, issue the SAVECLASS command or select

File ▶ Save as class...

Generating the CLASS entry from the CLASS block is equivalent to using the Class Editor to create a CLASS entry interactively.

## Generating CLASS Blocks from CLASS Entries

The CLASS block is especially useful when you need to make many changes to an existing class. To make changes to an existing class, use the CREATESCL function to write the class definition to an SCL entry. You can then edit the SCL entry in the Build window. After you finish entering changes, you can generate the CLASS entry by issuing the SAVECLASS command or selecting

File ▶ Save as class...

For more information, see “CREATESCL” on page 316.

## Referencing Class Methods or Attributes

Any METHOD block in a class can refer to methods or attributes in its own class without specifying the _SELF_ system variable (which contains the object identifier for

the class). For example, if method M1 is defined in class X (and it returns a value), then any method in class X can refer to method M1 as follows:

```
n=M1();
```

You do not need to use the _SELF_ system variable:

```
n=_SELF_.M1();
```

Omitting references to the _SELF_ variable (which is referred to as *shortcut syntax*) makes programs easier to read and maintain. However, if you are referencing a method or attribute that is not in the class you are creating, you must specify the object reference.

## Instantiating Classes

To instantiate a class, declare a variable of the specific class type, then use the _NEW_ operator. For example:

```
dcl mylib.classes.collection.class C1;
C1 = _new_ Collection();
```

You can combine these two operations as follows:

```
dcl mylib.classes.collection.class C1 = _new_ Collection();
```

The _NEW_ operator combines the actions of the LOADCLASS function, which loads a class, with the _new method, which initializes the object by invoking the object's _init method.

You can combine the _NEW_ operator with the IMPORT statement, which defines a search path for references to CLASS entries, so that you can refer to these entries with one or two-level names instead of having to use a four-level name in each reference.

For example, you can use the following statements to create a new collection object called C1 as an instance of the collection class that is stored in **mylib.classes.collection.class**:

```
   /* Collection class is defined in    */
   /* the catalog MYLIB.MYCAT       */
import mylib.mycat.collection.class;
   /* Create object C1 from a collection class  */
   /* defined in MYLIB.MYCAT.COLLECTION.CLASS */
declare Collection C1=_new_ Collection();
```

For more information, see "_NEW_" on page 563 and "LOADCLASS" on page 525.

# Methods

*Methods* define the operations that can be executed by any component that you create from that class. In other words, methods are how classes (and instances of those classes) do their work.

Methods can be *declared* in CLASS blocks. To declare a method, include the following METHOD statement in your CLASS block:

*label* : <*scope*> METHOD <*parameter-list*></(*method-options*)>;

The statements that implement the method can either follow the declaration, or they can reside in a separate SCL entry.

Methods are *implemented* in METHOD blocks. A METHOD block begins with the METHOD statement, includes the SCL code that implements the method, and then ends with the ENDMETHOD statement.

*label* : <*scope*> METHOD <*parameter-list*>
<OPTIONAL=*parameter-list*>
<ARGLIST=*parm-list-id* |REST=*rest-list-id*>
RETURN=*limited-data-type*
</ (*method-options*)>;

. . .*SCL statements that implement the method*. . .

ENDMETHOD;

If your program is an object-oriented program, the METHOD blocks are contained either in the CLASS block or in a USECLASS block that is stored in a separate SCL entry from the CLASS block. To store the method implementation in a separate SCL entry, when you declare the method in the CLASS block, you specify (with the SCL=*entry-name* option) the name of another SCL entry that contains the method implementation.

For example, the Add method can be implemented in the CLASS block as follows:

```
class Arithmetic;
  add: method n1 n2:num;
     return(n1 + n2);
  endmethod;
endclass;
```

If you want to implement the Add method in a separate SCL entry, then the CLASS block would contain only the method declaration:

```
class Arithmetic;
  add: method n1 n2:num / (scl='work.a.b.scl');
endclass;
```

The **work.a.b.scl** entry would contain a USECLASS block that implements the Add method:

```
useclass Arithmetic;
  add: method n1 n2: num;
    return (n1 + n2);
  endmethod;
enduseclass;
```

See “METHOD” on page 540 for a complete description of implementing methods with the METHOD statement. See Chapter 2, “The Structure of SCL Programs,” on page 9; “Implementing Methods Outside of Classes” on page 115; and “USECLASS” on page 698 for more information about implementing methods in USECLASS blocks.

*Note:* The method options that you specify in the CLASS block can also be specified in the USECLASS block. Any option that is included in the CLASS block and is used to specify a nondefault value must be repeated in the USECLASS block. For example, if you specify **State='O'** or **Signature='N'** in the CLASS block, then you must repeat those options in the USECLASS block. However, the SCL option will be ignored in the USECLASS block. △

For compatibility with Version 6, you can also define METHOD blocks in a separate SCL entry outside of CLASS and USECLASS blocks. However, such an application is not a strictly object-oriented application. For these methods, SCL will not validate method names and parameter types during compile time. See “Defining and Using

Methods" on page 13 for more information about methods that are not declared or implemented within a class.

## Defining Method Scope

SCL supports variable method scope, which gives you considerable design flexibility. Method scope can be defined as Public, Protected, or Private. The default scope is Public. In order of narrowing scope,

- Public methods can be accessed by any other class and are inherited by subclasses.
- Protected methods can be accessed only by the same class and its subclasses; they are inherited by subclasses.
- Private methods can be accessed only by the same class and are not inherited by subclasses.

For example, the Scope class defines two public methods (m1 and m4), one private method (m2), and one protected method (m3):

```
class Scope;
  m1: public method n:num return=num
      /(scl='work.a.uScope.scl');
  m2: private method :char;
      /(scl='work.b.uScope.scl');
  m3: protected method return=num;
      num = 3;
      dcl num n = m1(num);
      return(n);
  endmethod;
  m4: method
      /(scl='work.c.uScope.scl');
endclass;
```

By default, method m4 is a public method.

## Defining Method Names and Labels

Method names can be up to 256 characters long. Method labels can be up to 32 characters long. The name of a method should match its label whenever possible.

*Note:* A method that has the same name as the class that contains it is called a *constructor*. See "Defining Constructors" on page 112 for more information. △

### Specifying a Name That Is Different from the Label

If you need the method name to be different from the method label, you must specify either the METHOD or LABEL option in the METHOD statement. These options are mutually exclusive.

*Note:* In dot notation, always use the method name. When implementing the method, always use the method label. △

For example, a label of MyMethod may be sufficient, but if you want the method name to be MySortSalaryDataMethod, you can declare the method as follows:

```
class a;
MyMethod: public method sal:num
   /(Method='MySortSalaryDataMethod', SCL='work.a.a.scl');
```

```
endclass;
```

When you implement the method in **work.a.a.scl**, you identify the method by using the method label as follows:

```
useclass a;
   MyMethod: public method sal:num;
      ...SCL statements...
   endmethod;
enduseclass;
```

You would reference this method in dot notation by using the method name as follows:

```
obj.MySortSalaryDataMethod(n);
```

Alternatively, you can specify the LABEL option. For example, to specify a method name of Increase and a method label of CalculatePercentIncrease, you could declare the method as follows:

```
class a;
  Increase: public method
    /(Label='CalculatePercentIncrease', SCL='work.a.b.scl');
endclass;
```

As in the previous example, you use the method label when you implement the method, and you use the method name when you refer to the method in dot notation. In **work.a.b.scl**, you would implement the method as follows:

```
useclass a;
  CalculatePercentIncrease: public method;
     ...SCL statements...
  endmethod;
enduseclass;
```

You would reference the method in dot notation as follows:

```
obj.Increase();
```

### Using Underscores in Method Names

In Version 6, SAS/AF software used underscores to separate words in method names (for example, _set_background_color_). The current convention is to use a lowercase letter for the first letter and to subsequently uppercase the first letter of any joined word (for example, _setBackgroundColor).

The embedded underscores have been removed to promote readibility. However, for compatibility, the compiler recognizes _set_background_color_ as equivalent to _setBackgroundColor. All Version 6 code that uses the old naming convention in CALL SEND or CALL NOTIFY method invocations will still function with no modification.

Although it is possible for you to name a new method using a leading underscore, you should use caution when doing so. Your method names may conflict with future releases of SAS/AF software if SAS Institute adds new methods to the parent classes.

## Specifying Parameter Types and Storage Types

When you define a method parameter, you must specify its data type. Optionally, you can also specify its storage type: input, output, or update. The storage type determines how methods can modify each other's parameters:

input
: The values of the caller's parameters are *copied into* the corresponding parameters in the called method. When the called method's ENDMETHOD statement is executed, any updated values are *not copied out* to the caller's parameters.

output
: The values of the caller's parameters are *not copied into* the corresponding parameters in the called method. When the called method's ENDMETHOD statement is executed, any updated values are *copied out* to the caller's parameters.

update
: The values of the caller's parameters are *copied into* the corresponding parameters in the called method. When the called method's ENDMETHOD statement is executed, any updated values are *copied out* to the caller's parameters.

The default parameter storage type is update.

You use the colon (:) delimiter to specify both the storage type and the data type for each method parameter:

*variables*<:*storage*>:*type*

In the following example, the TypeStore class defines four methods:

```
import sashelp.fsp.collection.class;
class TypeStore;
  m1: method n:num a b:update:char return=num
      /(scl = 'work.a.uType.scl');
  m2: method n:output:num c:i:char
      /(scl = 'work.b.uType.scl');
  m3: method s:i:Collection
      /(scl = 'work.c.uType.scl');
  m4: method l:o:list
      /(scl = 'work.d.uType.scl');
endclass;
```

The parameter storage type and data type for each method are as follows:

| Method | Parameter | Data Type | Storage |
|---|---|---|---|
| m1 | n | numeric | update |
| | a | character | update |
| | b | character | update |
| m2 | n | numeric | output |
| | c | character | input |
| m3 | s | Collection class | input |
| m4 | l | list | output |

*Note:* If you specify the storage type for a parameter in the CLASS block, then you must also specify the storage type in the USECLASS block. △

## Passing Objects as Arguments for Methods

An object can be declared as an INTERFACE object, a CLASS object, or a generic OBJECT. If you declare an object as a generic OBJECT, then the compiler cannot validate attributes or methods for that object. Validation is deferred until run time.

Any error that results from using incorrect methods or attributes for the generic object will cause the program to halt. For example, if you pass a listbox class to a method that is expecting a collection object, the program will halt.

Object types are treated internally as numeric values. This can affect how you overload methods. See "Overloading and List, Object, and Numeric Types" on page 111 for more information.

## Returning Values From Methods

When you declare or implement a method, you can specify the data type of the return value with the RETURN option. If the method has a RETURN option, then the method implementation must contain a RETURN statement. The method's RETURN statement must specify a variable, expression, or value of the same type. In the following example, method m1 returns a numeric value:

```
class mylib.mycat.myclass.class;
  /* method declaration */
  m1: method n:num c:char return=num;
     /* method implementation */
     return(n+length(c));
  endmethod;
endclass;
```

## Method Signatures

A method's *signature* is a set of parameters that uniquely identifies the method to the SCL compiler. Method signatures enable the compiler to check method parameters at compile time and can enable your program to run more efficiently. All references to a method must conform to its signature definition. Overloaded methods must have signatures. (See "Overloading Methods" on page 108.)

A signature is automatically generated for each Version 8 method unless you specify **Signature='N'** in the method's option list. By default, **Signature='Y'** for all Version 8 methods. When you edit a class in the Build window, a signature is generated for each method that is declared in that class when you issue the SAVECLASS command or select

File ► Save as class...

For all Version 6 methods, the default is **Signature='N'**. See "Converting Version 6 Non-Visual Classes to Version 8 Classes" on page 131 for information about adding signatures to Version 6 methods.

For example, the following method declarations show methods that have different signatures:

```
Method1: method name:char  number:num;
Method2: method number:num name:char;
Method3: method name:char;
Method4: method return=num;
```

Each method signature is a unique combination, varying by argument number and type:

- The first signature contains a character argument and a numeric argument.
- The second signature contains a numeric argument and a character argument.
- The third signature contains a single character argument.
- The fourth signature contains no arguments.

These four method signatures have the following sigstrings (see “Signature Strings (SIGSTRINGs)” on page 106):

```
Method1 sigstring: (CN)V
Method2 sigstring: (NC)V
Method3 sigstring: (C)V
Method4 sigstring: ()N
```

The order of arguments also determines the method signature. For example, the getColNum methods below have different signatures — (CN)V and (NC)V — because the arguments are reversed. As a result, they are invoked differently, but they return the same result.

```
/* method1 */
getColNum: method colname:char number:update:num;
  number = getnitemn(listid, colname, 1, 1, 0);
endmethod;

/* method2 */
getColNum: method number:update:num colname:char;
  number = getnitemn(listid, colname, 1, 1, 0);
endmethod;
```

You can also use the Class Editor to define method signatures. See *SAS Guide to Applications Development* for more information.

## Signature Strings (SIGSTRINGs)

Signatures are usually represented by a shorthand notation, called a *sigstring*. This sigstring is stored in the method metadata as SIGSTRING.

A sigstring has the following compressed form:

(<*argument-type-1 argument-type-2...argument-type-n*>)*return-type*

Each argument type can be one of the following:

N — Numeric

C — Character string

L — SCL list

O — Generic object

O:<*class-name*>; — Specific class. The class name should be preceded by **O:** and followed by a semi-colon.

*Return-type* can be any of the above types, or **V** for void, which specifies that the method does not return a value. The return type cannot be an array.

Arrays are shown by preceding any of the above types with a bracket ( [ ). For example, a method that receives a numeric value and an array of characters and returns a numeric value would have the signature **(N[C)N**.

Here are some examples of method signatures:

- A method that does not receive any parameters and does not return a value: **()V**. This sigstring is the default signature.
- A method that returns a numeric value and that requires three parameters, numeric, character, and list: **(NCL)N**.
- A method that does not have a return value and that requires an object of type ProgramHalt and a numeric value:

```
(O:sashelp.classes.programHalt.class;N)V
```

- A method that returns a character value and receives a generic object and a character value: **(OC)C**.

*Note:* Although the return type is listed as part of the *sigstring*, it is not used by SCL to identify the method. Therefore, it is recommended that you do not define methods that differ only in return type. See "Overloading Methods" on page 108 for more information. △

## How Signatures Are Used

Signatures are most useful when SCL has to distinguish among the different forms of an overloaded method. The SCL compiler uses signatures to validate method parameters. When you execute your program, SCL uses signatures to determine which method to call.

For example, suppose your program contains the following class:

```
class Sig;
      /* Signature is (N)C */
  M1: method n:num return=char /(scl='work.a.uSig.scl');
      /* Signature is ([C)V */
  M1: private method n(*):char /(scl='work.a.uSig.scl');
      /* Signature is ()V */
  M1: protected method /(scl='work.a.uSig.scl');
      /* Signature is (NC)V
  M1: method n:num c:char /(scl='work.a.uSig.scl');
endclass;
```

Suppose also that your program calls M1 as follows:

```
dcl char ch;
ch = M1(3);
```

SCL will call the method with the signature (N)C. If your program calls M1 like this:

```
M1();
```

SCL will call the method with the signature ()V.

## Altering Existing Signatures

After defining a signature for a method and deploying the class that contains it for public use, you should not alter the signature of the method in future versions of the class. Doing so could result in program halts for users who have already compiled their applications. Instead of altering an existing signature, you should overload the method to use the desired signature, leaving the previous signature intact.

# Forward-Referencing Methods

Within a CLASS block, if a method invokes a another method within that same class, then either the second method must be implemented before the first, or the second method must be declared with the **Forward='Y'** option.

*Note:* Any methods that are forward-referenced must be implemented in the class in which they are declared. △

In the following example, m1 calls m2, so the compiler needs to know the existence of m2 before it can compile m1.

```
class mylib.mycat.forward.class;
  m2: method n:num c:char return=num / (forward='y');
```

```
  m1: method n1 n2:num mylist:list return=num;
      dcl num listLen = listlen(mylist);
      dcl num retVal;
      if (listLen = 1) then
         retVal=m2(n1,'abc');
      else if (listLen = 2) then
         retVal=m2(n2,'abc');
  endmethod;
  m2:method n:num c:char return=num;
      return(n+length(c));
  endmethod;
endclass;
```

## Overloading Methods

You can overload methods only for Version 8 classes. Method overloading is the process of defining multiple methods that have the same name, but which differ in parameter number, type, or both. Overloading methods enables you to

- use the same name for methods that are related conceptually.
- create methods that have optional parameters.

All overloaded methods must have method signatures because SCL uses the signatures to differentiate between overloaded methods. If you call an overloaded method, SCL checks the method arguments, scans the signatures for a match, and executes the appropriate code. A method that has no signature cannot be overloaded.

If you overload a method, and the signatures differ only in the return type, the results are unpredictable. The compiler will use the first version of the method that it finds to validate the method. If the compiler finds the incorrect version, it generates an error. If your program compiles without errors, then when you run the program, SCL will execute the first version of the method that it finds. If it finds the incorrect version, SCL generates an error. If it finds the correct version, your program might run normally.

Each method in a set of overloaded methods can have a different scope, as well. However, the scope is not considered part of the signature, so you may not define two methods that differ only by scope. (See “Defining Method Scope” on page 102.)

### Example: Different Parameter Types

Suppose you have the following two methods, where each method performs a different operation on its arguments:

```
CombineNumerics: public method a :num b :num
                 return=num;
                 endmethod;
CombineStrings:  public method c :char d :char
                 return=char;
                 endmethod;
```

Assume that CombineNumerics adds the values of A and B, whereas CombineStrings concatenates the values of C and D. In general terms, these two methods combine two pieces of data in different ways based on their data types.

Using method overloading, these methods could become

```
Combine: public method a :num  b :num
         return=num;
         endmethod;
Combine: public method c :char d :char
```

```
return=char;
endmethod;
```

In this case, the Combine method is overloaded with two different parameter lists: one that takes two numeric values and returns a numeric value, and another that takes two character parameters and returns a character value.

As a result, you have defined two methods that have the same name but different parameter types. With this simple change, you do not have to worry about which method to call. The Combine method can be called with either set of arguments, and SCL will determine which method is the correct one to use, based on the arguments that are supplied in the method call. If the arguments are numeric, SCL calls the first version shown above. If the arguments are character, SCL calls the second version. The caller can essentially view the two separate methods as one method that can operate on different types of data.

Here is a more complete example that shows how method overloading fits in with the class syntax. Suppose you create X.SCL and issue the SAVECLASS command, which generates the X class. (Although it is true here, it is not necessary that the class name match the entry name.)

```
class X;

Combine: public method a:num b:num  return=num;
   dcl num value;
   value = a + b;
   return value;
endmethod;

Combine: public method a:char b:char return=char;
   dcl char value;
   value = a || b;
   return value;
endmethod;

endclass;
```

You can then create another entry, Y.SCL. When you compile and execute Y.SCL, it instantiates the X class and calls each of the Combine methods.

```
import X.class;
init:
   dcl num n;
   dcl char c;
   dcl X xobject = _new_ X();
   n = xobject.Combine(1,2);
   c = xobject.Combine("abc","def");
   put n= c=;
```

The PUT statement produces

```
n=3 c=abcdef
```

## Example: Different Numbers of Parameters

Another typical use of method overloading is to create methods that have optional parameters.

*Note:* This example shows two implementations of an overloaded method that each accept different numbers of parameters. "Defining One Implementation That Accepts Optional Parameters" on page 111 describes how to use the OPTIONAL option to create a method with one implementation that accepts different numbers of parameters. △

For example, suppose we have a method that takes a character string and a numeric value, where the numeric value is used as a flag to indicate a particular action. The method signature would be (CN)V.

```
M: public method c :char f :num;
   if (f = 1) then
     /* something */
   else if (f = 2)
     /* something else */
   else
     /* another thing */
endmethod;
```

If method M is usually called with the flag equal to one, you can overload M as (C)V, where that method would simply include a call to the original M. The flag becomes an optional parameter.

```
M:  public method c: char;
  M(c, 1);
endmethod;
```

When you want the flag to be equal to one, call M with only a character string parameter. Notice that this is not an error. Method M can be called with either a single character string, or with a character string and a numeric — this is the essence of method overloading. Also, the call **`M(c,1);`** is not a recursive call with an incorrect parameter list. It is a call to the original method M.

This example can also be turned around for cases with existing code. Assume that we originally had the method M with signature (C)V and that it did all the work.

```
M: public method c: char;
   /* A lot of code for processing C. */
endmethod;
```

Suppose you wanted to add an optional flag parameter, but did not want to change the (possibly many) existing calls to M. All you need to do is overload M with (CN)V and write the methods as follows:

```
M:  public method c: char f: num;
   Common(c, f);
endmethod;

M:  public method c: char;
   Common(c, 0);
endmethod;

Common:  public method c: char f: num;
   if (f) then
      /* Do something extra. */
/* Fall through to same old code for */
/* processing S.                     */
endmethod;
```

Notice that when you call M with a single character string, you get the old behavior. When you call M with a string and a (non-zero) flag parameter, you get the optional behavior.

## Defining One Implementation That Accepts Optional Parameters

You can use the OPTIONAL option to create an overloaded method with only one implementation that will accept different numbers of parameters, depending on which arguments are passed to it.

In the following example, the method M1 will accept from two to four parameters:

```
class a;
M1: public method p1:input:num p2:output:char
    optional=p3:num p4:char
    / (scl='mylib.classes.old.scl');
endclass;
```

SCL will generate three signatures for this method:

(NC)V

(NCN)V

(NCNC)V

## Overloading and List, Object, and Numeric Types

Lists and objects (variables declared with either the OBJECT keyword or a specific class name) are treated internally as Numeric values. As a result, in certain situations, variables of type List, Numeric, generic Object, and specific class names are interchangeable. For example, you can assign a generic Object or List to a variable that has been declared as Numeric, or you can assign a generic Object to a List. This flexibility enables Version 6 programs in which list identifiers are stored as Numeric variables to remain compatible with Version 8.

The equivalence between objects, lists, and numeric variables requires that you exercise caution when overloading methods with these types of parameters. When attempting to match a method signature, the compiler first attempts to find the best possible match by matching the most parameter types exactly. If no exact match can be found, the compiler resorts to using the equivalence between List, generic Object, and Numeric types.

For example, suppose you have a method M with a single signature (L)V. If you pass a numeric value, a list, or an object, it will be matched, and method M will be called. If you overload M with signature (N)V, then Numeric values will match the signature (N)V, and List values will match the signature (L)V. However, List values that are undeclared or declared as Numeric will now match the wrong method. Therefore, you must explicitly declare them with the LIST keyword to make this example work correctly. Also, if you pass an object, it will match both (L)V and (N)V, so the compiler cannot determine which method to call and will generate an error message.

# Overriding Existing Methods

When you instantiate a class, the new class (or subclass) inherits the methods of the parent class. If you want to use the signature of one of the parent's methods, but you want to replace the implementation with your own implementation, you can *override* the parent's method. To override the implementation of a method, specify **`State='O'`** in the method declaration and in the method implementation. Here is an example for a class named State:

```
class State;
  _init: method / (state='o');
    _super();
  endmethod;
endclass;
```

## Defining Constructors

*Constructors* are methods that are used to initialize an instance of a class. The Object class provides a default constructor that is inherited for all classes. Unless your class requires special initialization, you do not need to create a constructor.

Each constructor has the following characteristics:

- It has the same name as the class in which it is declared.
- It is run automatically when the class is instantiated with the _NEW_ operator. If you do not create your own constructor, the default constructor is executed.

  *Note:* Using the _NEW_ operator to instantiate a class is the only way to run constructors. Unlike other user-defined methods, you cannot execute constructors using dot notation. If you instantiate a class in any way other than by using the _NEW_ operator (for example, with the _NEO_ operator), constructors are not executed. △
- It is intended to run as an initializer for the instance. Therefore, only constructors can call other constructors. A constructor cannot be called from a method that is not a constructor.
- It cannot return a value; it must be void a method. The _NEW_ operator returns the value for the new instance of the class; it cannot return a value from an implicitly called constructor.

For example, you could define a constructor X for class X as follows:

```
class X;
  X: method n: num;
     put 'In constructor, n=';
  endmethod;
endclass;
```

You can instantiate the class as follows:

```
init:
  dcl X x = _new_ X(99);
return;
```

The constructor is run automatically when the class is instantiated. The argument to _NEW_, 99, is passed to the constructor. The output is

```
In constructor, n=99
```

### Overloading Constructors

Like other methods, constructors can be overloaded. Any void method that has the same name as the class is treated as a constructor. The _NEW_ operator determines which constructor to call based on the arguments that are passed to it. For example, the Complex class defines two constructors. The first constructor initializes a complex number with an ordered pair of real numbers. The second constructor initializes a complex number with another complex number.

```
class Complex;
  private num a b;

  Complex: method r1: num r2: num;
    a = r1;
    b = r2;
  endmethod;

  Complex: method c: complex;
    a = c.a;
    b = c.b;
  endmethod;
endclass;
```

This class can be instantiated with either of the following statements:

```
dcl Complex c  = _new_(1,2);
dcl Complex c2 = _new_(c);
```

These statements both create complex numbers. Both numbers are equal to **`1 + 2i`**.

## Overriding the Default Constructor

The default constructor does not take any arguments. If you want to create your own constructor that does not take any arguments, you must explicitly override the default constructor. To override the default constructor, specify **`State='o'`** in the method options list.

```
class X;
   X: method /(state='o');
      ...SCL statements to initialize class X...
   endmethod;
endclass;
```

## Calling Constructors Explicitly

Constructors can be called explicitly only from other constructors. The _NEW_ operator calls the first constructor. The first constructor can call the second constructor, and so on.

When a constructor calls another constructor within the same class, it must use the _SELF_ system variable. For example, you could overload X as follows:

```
class X;
  private num m;

  X: method n: num;
     _self_(n, 1);
  endmethod;

  X: method n1: num n2: num;
     m = n1 + n2;
  endmethod;

endclass;
```

The first constructor, which takes one argument, calls the second constructor, which takes two arguments, and passes in the constant **`1`** for the second argument.

The following labeled section creates two instances of X. In the first instance, the **m** attribute is set to **3**. In the second instance, the **m** attribute is set to **100**.

```
init:
  dcl X x  = _new_ X(1,2);
  dcl X x2 = _new_ X(99);
return;
```

Constructors can call parent constructors by using the _SUPER operator. For example, suppose you define class X as follows:

```
class X;
  protected num m;

  X: method n: num;
     m = n * 2;
  endmethod;

endclass;
```

Then, you create a subclass Y whose parent class is X. The constructor for Y overrides the default constructor for Y and calls the constructor for its parent class, X.

```
class Y extends X;
  public num p;

  Y: method n: num /(state='o');
     _super(n);
     p = m - 1;
  endmethod;

endclass;
```

You can instantiate Y as shown in the following labeled section. In this example, the constructor in Y is called with argument **10**. This value is passed to the constructor in X, which uses it to initialize the **m** attribute to **20**. Y then initializes the **p** attribute to **19**.

```
init:
  dcl Y y = _new_ Y(10);
  put y.p=;
return;
```

The output would be:

```
y.p=19
```

*Note:* As with other overridden methods that have identical signatures, you must explicitly override the constructor in Y because there is a constructor in X that has the same signature. △

## Specifying That a Method Is Not a Constructor

The compiler automatically treats as a constructor any void method that has the same name as the class. If you do not want such a method to be treated as a constructor, you can specify **constructor='n'** in the method declaration.

```
class X;
  X: method /(constructor='n');
     put 'Is not constructor';
```

```
    endmethod;
  endclass;

  init:
    dcl X x = _new_ X();
    put 'After constructor';
    x.x();
  return;
```

This will result in the following output:

```
After constructor
Is not constructor
```

## Implementing Methods Outside of Classes

You can define the implementation of methods outside the SCL entry that contains the CLASS block that defines the class. This feature enables multiple people to work on class methods simultaneously.

To define class methods in a different SCL entry, use the USECLASS statement block. The USECLASS block binds methods that it contains to the class that is specified in the USECLASS statement. The USECLASS statement also enables you to define implementations for overloading methods. (See "Overloading Methods" on page 108. )

Method implementations inside a USECLASS block can include any SCL functions and routines. However, the only SCL statements that are allowed in USECLASS blocks are METHOD statements.

The USECLASS block binds the methods that it contains to a class that is defined in a CLASS statement block or in the Class Editor. Therefore, all references to the methods and the attributes of the class can bypass references to the _SELF_ variable completely as long as no ambiguity problem is created. Because the binding occurs at compile time, the SCL compiler can detect whether an undefined variable is a local variable or a class attribute. See also "Referencing Class Methods or Attributes" on page 99.

## Method Metadata

SCL stores metadata for maintaining and executing methods. You can query a class (or a method within a class) to view the method metadata. For example, to list the metadata for a specific method, execute code similar to the following:

```
init:
   DCL num rc metadata;
   DCL object obj;

   obj=loadclass('class-name');

   /* metadata is a numeric list identifier */
   rc=obj._getMethod('getMaxNum',metadata);
   call putlist(metadata,'',2);
return;
```

# Attributes

*Attributes* are the properties that specify the information associated with a component, such as its name, description, and color. Attributes determine how a component will look and behave. For example, the Push Button Control has an attribute named **label** that specifies the text displayed on the button. You can create two instances of the Push Button Control on your frame and have one display "OK" and the other display "Cancel," simply by specifying a different value for the **label** attribute of each instance.

You can define attributes with attribute statements in CLASS blocks:

*scope data-type attribute-name*/(*attribute-options*);

Attribute names can be up to 256 characters long.

Like methods, attributes can have public, private, or protected scope. The scope works the same for attributes as it does for methods. See "Defining Method Scope" on page 102 for more information.

Examples of attribute options include the attribute description, whether the attribute is editable or linkable, custom access methods that are to be executed when the attribute is queried or set, and whether the attribute sends events.

If an attribute is editable, you can use the Editor option to specify the name of the FRAME, SCL, or PROGRAM entry that will be used to edit the attribute's value. This entry is displayed and executed by the Properties window when the ellipsis button (...) is selected.

To specify an attribute's category, use the Category attribute option. The category is used for grouping similar types of options in the Class Editor or for displaying related attributes in the Properties window. You can create your own category names. Components that are supplied by SAS may belong to predefined categories.

## Creating Attributes Automatically

With the Autocreate option, you can control whether storage for list attributes and class attributes is automatically created when you instantiate a class. By default, **Autocreate='Y'**, which means that SCL automatically uses the _NEW_ operator to instantiate class attributes and calls the MAKELIST function to create the list attributes.

*Note:* Even when **Autocreate='Y'**, storage is not created for generic objects because the specific class is unknown. △

If you specify **Autocreate='N'**, then storage is not automatically created, and it is your responsibility to create (and later destroy) any list attributes or class attributes after the class is instantiated.

```
import sashelp.fsp.collection.class;
class myAttr;
  public list myList / (autocreate='N');
  public list listTwo;   /* created automatically */
  public collection c1;  /* created automatically */
  public collection c2 / (autocreate='N');
endclass;
```

## Specifying Where an Attribute Value Can Be Changed

An attribute's scope and the value of its Editable option determines which methods can change an attribute's value.

- If the scope is public and **Editable='Y'**, then the attribute can be accessed (both queried and set) from any class method as well as from a frame SCL program.
- If the scope is public and **Editable='N'**, then the attribute can only be queried from any class method or frame SCL program. However, only the class or subclasses of the class can modify the attribute value.
- If the scope is protected and **Editable='N'**, then the class and its subclasses can query the attribute value, but only the class itself can set or change the value. A frame SCL program cannot set or query the value.
- If the scope is private and **Editable='N'**, then the attribute value can be queried only from methods in the class on which it is defined, but it cannot be set by the class. Subclasses cannot access these attributes, nor can a frame SCL program. This combination of settings creates a private, pre-initialized, read-only constant.

## Setting Initial Values and the List of Valid Values

Unless you specify otherwise, all numeric attributes are initialized to missing values, and all character attributes are initialized to blank strings. You can use the initialValue attribute option to explicitly initialize an attribute. For example:

```
class myAttr;
   public num n1 / (initialvalue = 3);
   public list list2 / (initialvalue = {1, 2, 'abc', 'def'};
endclass;
```

Explicitly initializing attribute values improves the performance of your program.

You can use the ValidValues attribute option to specify a list of values that the attribute can have. This list is used as part of the validation process that occurs when the value is set programmatically by using either dot notation or the _setAttributeValue method.

If you specify the ValidValues option and the InitialValue option, the value that you specify with the InitialValue option must be included in the values that you specify with the ValidValues option.

In the list of valid values, you can use blanks to separate values, or, if the values themselves contain blanks, use a comma or a slash (/) as the separator. For example:

```
class business_graph_c;
  public char statistic
         / (ValidValues='Frequency/Mean/Cumulative Percent',
            InitialValue='Mean');
  public char highlightEnabled
         / (ValidValues='Yes No',
            InitialValue='Yes');
 endclass;
```

You can also specify an SCL or SLIST entry to validate values. For more information on how to use an SCL entry to perform validation, refer to *SAS Guide to Applications Development*.

## Associating Custom Access Methods with Attributes

A custom access method (CAM) is a method that is executed automatically when an attribute's value is queried or set using dot notation. When you query the value of an attribute, SCL calls the _getAttributeValue method. When you set the value of an attribute, SCL calls the _setAttributeValue method. These methods are inherited from the Object class.

You can use the getCAM and setCAM attribute options to specify additional methods that you want _getAttributeValue or _setAttributeValue to execute. For example:

```
class CAM;
  public char A / (getCAM='M1');
  public num B / (setCAM='M2');
  protected M1: method c:char;
    put 'In M1';
  endmethod;
  protected M2: method b:num;
    put 'In M2';
  endmethod;
endclass;
```

When the value of A is queried, _getAttributeValue is called, then M1 is executed. When the value of B is set, _setAttributeValue is called, then M2 is executed.

CAMs always have a single signature and cannot be overloaded. The CAM signature contains a single argument that is the same type as its associated attribute. A CAM always returns a numeric value.

You should never call a CAM directly; instead, use the _getAttributeValue or _setAttributeValue methods to call it automatically. To prevent CAMs from being called directly, it is recommended that you define them as protected methods.

## Linking Attributes

Attribute linking enables one component to automatically upate the value of one of its attributes when the value of another component attribute is changed. You can link attributes between components or within the same component. Only public attributes are linkable.

To implement attribute linking, you need to identify attributes as either source attributes or target attributes. You can identify source and target attributes either in the Properties window or with SCL. To identify an attribute as a source attribute with SCL, specify **SendEvent='Y'** in the attribute's option list. To identity an attribute as a target attribute, specify **Linkable='Y'** in the attribute's option list.

You can then link the attributes (specify the **LinkTo** option) in the Properties window.

When **SendEvent='Y'**, SAS/AF software registers an event on the component. For example, the **textColor** attribute has an associated event named "textColor Changed". You can then register an event handler to trap the event and to conditionally execute code when the value of the attribute changes.

If you change the **SendEvent** value from **'Y'** to **'N'**, and if **Linkable='Y'**, then you must send the "*attributeName* Changed" event programmatically with the attribute's setCAM in order for attributes that are linked to this attribute to receive notification that the value has changed. If the linked attributes do not receive this event, attribute linking will not work correctly. In the previous example, the setCAM for the **textColor** attribute would use the _sendEvent method to send the "textColor Changed" event.

Refer to *SAS Guide to Applications Development* for more information on attribute linking.

## Attribute Metadata

SCL uses a set of attribute metadata to maintain and manipulate attributes. This metadata exists as a list that is stored with the class. You can query a class (or an attribute within a class) with specific methods to view attribute metadata. To list the metadata for a specific attribute, execute code similar to the following:

```
init:
   DCL num rc;
   DCL list metadata;
   DCL object obj;

   obj=loadclass('class-name');

   rc=obj._getAttribute('attribute-name',metadata);
   call putlist(metadata,'',3);
return;
```

# Accessing Object Attributes and Methods With Dot Notation

SCL provides dot notation for directly accessing object attributes and for invoking methods instead of using the SEND and NOTIFY routines. Thus, dot notation provides a shortcut for invoking methods and for setting or querying attribute values. Using dot notation reduces typing and makes SCL programs easier to read.

Using dot notation enhances run-time performance if you declare the object used in the dot notation as an instance of a predefined class instead of declaring it as a generic object. The object's class definition is then known at compile time, enabling the SCL compiler to verify the method and to access attributes at that time. Moreover, since dot notation checks the method signature, it is the only way to access an overloaded method. SEND does not check method signatures. It executes the first name-matched method, and the program might halt if the method signature does not match.

## Syntax

The syntax for dot notation is as follows:

```
object.attribute
```

or

```
object.method(<arguments>)
```

Where

*object*
: specifies an object or an automatic system variable (for example, _SELF_). An object must be a component in a FRAME entry or a variable that is declared as an Object type in the SCL program. Automatic system variables like _SELF_ are declared internally as Object type, so they do not have to be declared explicitly as such in a program.

*attribute*
: specifies an object attribute to assign or query. It can be of any data type, including Array. If the attribute is an array, use the following syntax to reference its elements:

```
object.attributeArray[i]
```

You can also use parentheses instead of brackets or braces when referencing the array elements. However, if you have declared the object as a generic object, the compiler interprets it as a method name rather than an attribute array. If you have declared a type for the object, and an attribute and method have the same name, the compiler still interprets the object as a method. To avoid this ambiguity, use brackets when referencing attribute array elements.

*method*
specifies the name of the method to invoke. If an object is declared with a specific class definition, the compiler can perform error checking on the object's method invocations.

If the object was declared as a generic object (with the OBJECT keyword), then the method lookup is deferred until run time. If there is no such method for the object, the program halts. If you declare the object with a specific definition, errors such as this are discovered at compile time instead of at run time.

*arguments*
are the arguments passed to the method. Enclose the arguments in parentheses. The parentheses are required whether or not the method needs any arguments.

You can use dot notation to specify parameters to methods. For example:

```
return-value = object.method (object.id);
```

However, if you use dot notation to specify an update or output parameter, then SCL executes the _setAttributeValue method, which may produce side effects. See "What Happens When Attribute Values Are Set or Queried" on page 122 for more information.

Some methods may be defined to return a value of any SCL type. You can access this returned value by specifying a variable in the left side of the dot notation. For example:

```
return-value = object.method (<arguments>);
```

or

```
if ( object.method (<arguments>) ) then ...
```

The return value's type defaults to Numeric if it is not explicitly declared. If the declared type does not match the returned type, and the method signature is known at compile time, the compiler returns an error. Otherwise, a data conversion might take place, or the program will halt at run time.

If you override an object's INIT method, you must call _SUPER._INIT before you can use dot notation to set attribute values or to make other method calls.

Dot notation is not available in the INPUT and PUT functions.

By default, your application halts execution if an error is detected in the dot notation that is used in the application. You can control this behavior with the HALTONDOTATTRIBUTE or NOHALTONDOTATTRIBUTE option in the CONTROL statement. See "CONTROL" on page 302 for more information.

## Using Nested Dot Notation

You can also use dot notation in nested form. For example,

```
value = object.attribute1.method1().attribute2;
```

is equivalent to the following:

```
dcl object object1 object2;
object1 = object.attribute1;  /* attribute1 in object
```

```
                                   is of OBJECT type */
object2 = object1.method1();  /* method1 in object1
                                   returns an object */
value   = object2.attribute2; /* assign the value of
                                   attribute2 in object2
                                   to the variable
                                   'value'. */
```

You can also specify the nested dot notation as an l-value. For example,

```
object.attribute1.method1().attribute2 = value;
```

is equivalent to the following:

```
dcl object object1 object2;

object1 = object.attribute1;
object2 = object1.method1();
object2.attribute2 = value; /* assume 'value' has
                                 been initialized.
                                 This would set
                                 attribute2 in object2
                                 to the value */
```

## Examples

An application window contains a text entry control named clientName. The following examples show how to use dot notation to invoke methods and to query and assign attribute values. For example, the following statement uses dot notation to invoke the _gray method of the control:

```
clientName._gray();
```

This is equivalent to

```
call send('clientName','_gray');
```

You can change the text color to blue, using dot notation to set the value of its **textColor** attribute:

```
name.textColor='blue';
```

You can also use dot notation to query the value of an attribute. For example:

```
color=clientName.textColor;
```

You can use dot notation in expressions. You can use a method in an expression only if the method can return a value via a RETURN statement in its definition. For example, suppose you create a setTable method, which is a public method and accepts an input character argument (the name of a SAS table). The method determines whether a SAS table exists and uses the RETURN statement to pass the return code from the EXIST function.

```
setTable: public method dsname:i:char(41) return=num;
  rc = exist(dsname, 'DATA');
  return rc;
endmethod;
```

Then you could use a statement like the following to perform actions that depend on the value that the setTable method returned.

```
if (obj.setTable('sasuser.houses')) then
   /*  the table exists, perform an action */
else
   /*  the table doesn't exist,    */
   /*  perform another action      */
```

The next example shows how to use dot notation with an object that you create in an SCL program. Suppose class X is saved in the entry X.SCL, and the INIT section is saved in the entry Y.SCL.

```
class x;
  public num n;
  m: public method n1: num n2: num return=num;
    dcl num r;
    r = n1 + n2;
    /* return sum of n1 and n2 */
    return r;
 endmethod;
 m: public method c1: char c2:char return=char;
    dcl num s;
    /* concatenate c1 and c2 */
    s = c1 || c2;
    return s;
 endmethod;
endclass;

init:
  dcl x xobj = _new_ x();
  dcl num n;
  dcl string s;
  n = xobj.m(99,1);
  s = xobj.m("abc","def");
  put n= s=;
  return;
```

If you compile and run Y.SCL, it produces

```
n=100 s=abcdef
```

## What Happens When Attribute Values Are Set or Queried

When you use dot notation to change or query an attribute value, SCL translates the statement to a _setAttributeValue method call (to change the value) or to a _getAttributeValue method call (to query the value). As a result, defining the attribute with a getCAM or setCAM method could produce side effects.

When you use dot notation to specify a parameter to a method, SCL executes the _setAttributeValue method if the parameter is an update or output parameter. SCL executes the _getAttributeValue method if the parameter is an input parameter. However, if the object is declared as a generic object or if the method does not have a signature, then all of the method's parameters are treated as update parameters. Therefore, SCL will execute the _setAttributeValue method even if the parameter is an input parameter, which could execute a setCAM method and send an event.

*Note:* If you use dot notation to access a class attribute, program execution halts if any error is detected while the _getAttributeValue or _setAttributeValue method is running. Explicitly invoking the _getAttributeValue or _setAttributeValue method allows the program to control the halt behavior. The _getAttributeValue or _setAttributeValue method also enables you to check the return code from the method. For example:

```
rc = obj._setAttributeValue ('abc');
if ( rc ) then do;
    /* error detected in the _setAttributeValue method */
    ...more SCL statements...
    end;
```

△

## Setting Attribute Values

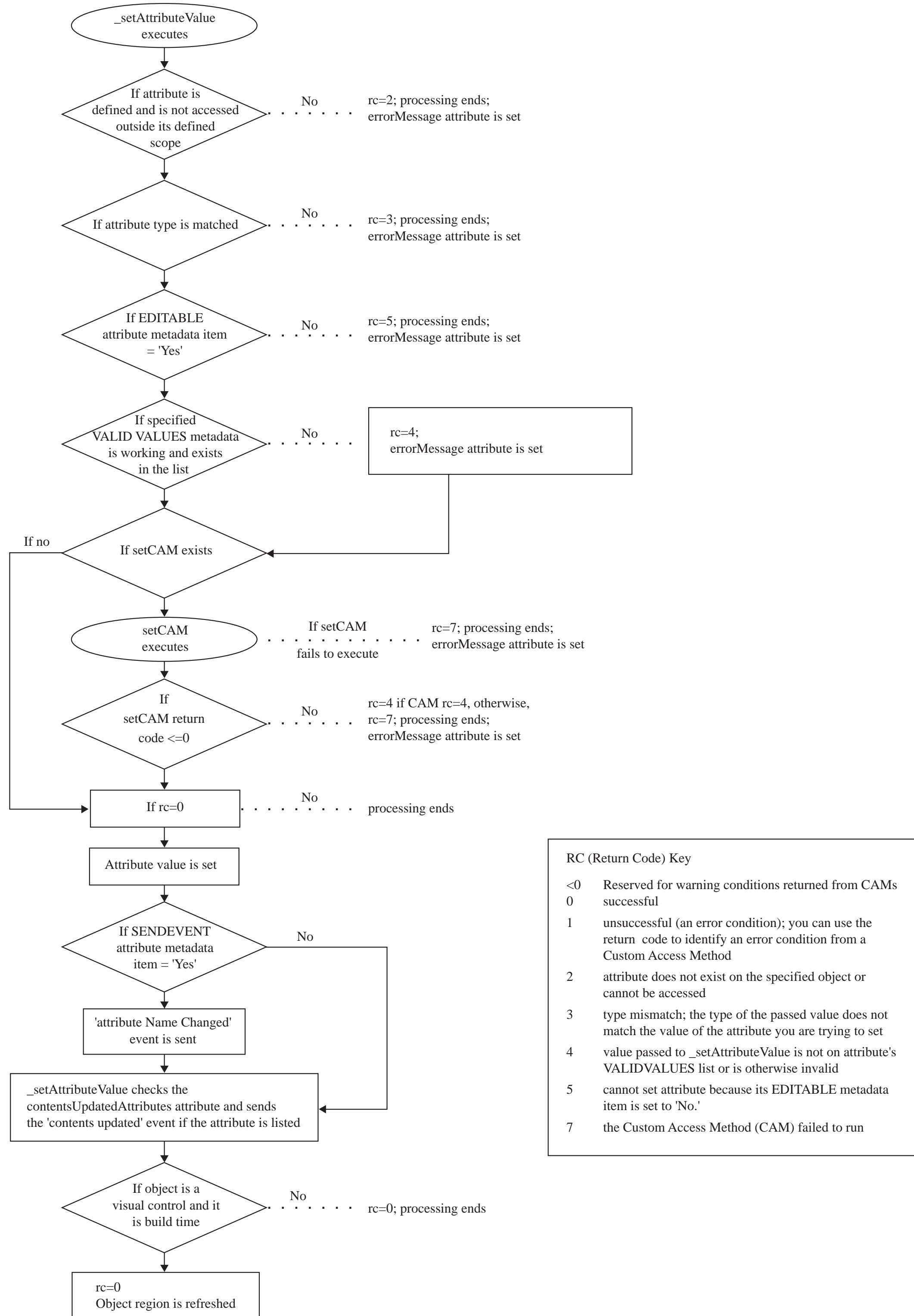
When you use dot notation to set the value of an attribute, SCL follows these steps:

1 Verify that the attribute exists.
2 Verify that the type of the attribute matches the type of the value that is being set.
3 Check whether the attribute value is in the ValidValues list. If the ValidValues metadata is an SCL entry, it is executed first to get the list of values to check the attribute value against.
4 Run the setCAM method, if it is defined, which gives users a chance to perform additional validation and to process their own side effects.

   *Note:* If the Editable metadata is set to **No**, the custom set method is not called (even if it was defined for the attribute). △

5 Store the object's value in the attribute.
6 Send the "*attributeName* Changed" event if the SendEvent metadata is set to **Yes**.
7 sends the "contents Updated" event if the attribute is specified in the object's **contentsUpdatedAttributes** attribute. This event notifies components in a model/view relationship that a key attribute has been changed.

**Figure 8.2** Flow of Control for _setAttributeValue




# Querying Attribute Values

When you use dot notation to query the value of an attribute, SCL follows these steps:

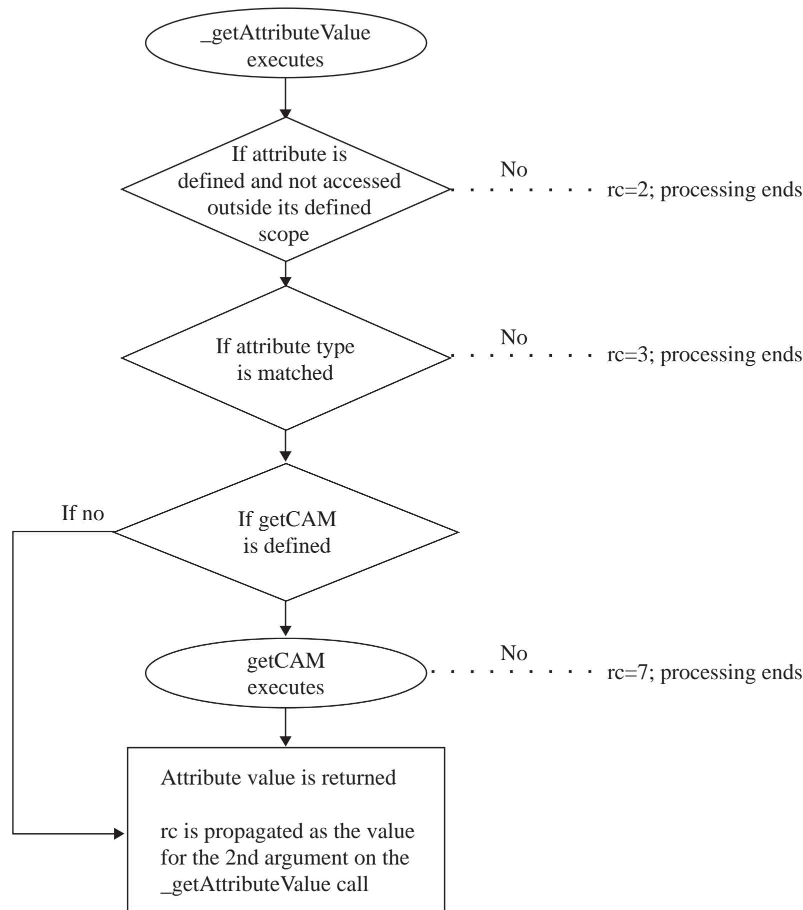
1 Execute the getCAM method to determine the attribute value, if a getCAM method has been defined.

2 Return the attribute value, if a value has been set.

3 Return the initial class value, if no attribute value has been set.

The following figure shows this process in detail.

**Figure 8.3** Flow of Control for _getAttributeValue




# Events and Event Handlers

Events alert applications when there is a change of state. Events occur when a user action takes place (such as a mouse click), when an attribute value is changed, or when a user-defined condition occurs. Events are essentially generic messages that are sent to objects from the system or from SCL applications. These messages usually direct an object to perform some action such as running a method.

Event handlers are methods that listen for these messages and respond to the them. Essentially, an event handler is a method that determines which method to execute after the event occurs.

SCL supports both system events and user-defined events.

## System Events

System events include user interface events (such as mouse clicks) as well as “attribute changed” events that occur when an attribute value is updated. SCL automatically defines system events for component attributes when those attributes are declared.

SCL can also automatically send system events for you when a component’s attribute is changed. If you want “attribute changed” events to be sent automatically, specify **`SendEvent='Y'`** in the options list for the attribute.

If you want an action to be performed when the system event occurs, then you need to define the event handler that you want to be executed when the event occurs. You define event handlers for system events in the same way that you define them for user-defined events. See “Defining Event Handlers” on page 126 for more information.

## Defining and Sending Events

You can create user-defined events through the Properties window in the Class Editor or with event declaration statements in CLASS blocks.

EVENT *event-name*</(*event-options*)>;

Event names can be up to 256 characters long.

For the event options, you can specify the name of the method that handles the event and when an object should send the event. Events can be sent automatically either before (specify **`Send='Before'`**) or after (**`Send='After'`**) a method executes or they can be programmed manually (**`'Manual'`**) with SCL. New events default to **`'After'`**. You must specify a method name for events that are to be sent automatically.

After an event is defined, you can use the _sendEvent method to send the event:

*object*._sendEvent("*event-name*"<, *event-handler-parameters*>);

For a complete description of _sendEvent, refer to the SAS/AF online Help.

## Defining Event Handlers

You can define event handlers with event handler declaration statements in CLASS blocks.

EVENTHANDLER *event-handler-name*</(*event-handler-options*)>;

As part of the event handler options, you can specify the name of the event, the name of the method that handles the event, and the name of the object that generates the event (the sender). As the sender, you can specify **`'_SELF_'`** or **`'_ALL_'`**. When **`Sender='_SELF_'`**, the event handler listens only to events from the class itself. When **`Sender='_ALL_'`**, the event handler listens to events from any other class.

Using the _addEventHandler method, you can dynamically add a sender to trigger the event. For a complete description of _addEventHandler, refer to the SAS/AF online Help.

For more information about defining event handlers, see “CLASS” on page 277.

## Example

The following class defines one user-defined event, myEvent, and the event handler for this event, M2. When this class is created, SCL also assigns the system event name “n Changed” for the attribute **`n`** and registers the event name with the component.

```
class EHclass;
  public num n;   /* system event */
  event 'myEvent' / (method='M2');
  eventhandler M1 / (sender = '_SELF_',
                     event = 'n Changed');
  eventhandler M2 / (sender = '_SELF_',
                     event = 'myEvent');

  M1: method a:list;
    put "Event is triggered by attribute n";
  endmethod;

  M2: method a:string n1:num n2:num;
    put "Event is triggered by _sendEvent";
    put a= n1= n2=;
  endmethod;
endclass;
```

When the value of the attribute **n** is changed, the system automatically sends the “n Changed” event, and method M1 is executed. Method M2 is not executed until myEvent is sent with the _sendEvent method.

The next class, EHclass1, defines a second event handler, M3, that is also executed when myEvent is sent.

```
class EHclass1;
  /* Sender='*' means that the sender */
  /* is determined at run time.       */
  eventhandler M3 / (sender = '*', event='myEvent');
  M3: method a:string n1:num n2:num;
    put "Event myEvent is defined in another class";
    put "that is triggered by _sendEvent.";
    put a= n1= n2=;
  endmethod;
endclass;
```

In the following program, the system event “n Changed” is triggered when the value of the **n** attribute is modified. The user-defined event myEvent is triggered with the _sendEvent method.

```
import work.a.EHclass.class;
import work.a.EHclass1.class;
init:
  dcl EHclass obj = _new_ EHclass();
  dcl EHclass1 obj1 = _new_ EHclass1();

  /* Trigger the system event. */
  obj.n = 3;

  /* Trigger the user-defined event. */
  obj._sendEvent("myEvent", 'abc', 3, 4);
return;
```

The order in which the two classes are instantiated determines the order in which the event handlers for myEvent are executed. EHclass is instantiated first, so when myEvent is sent, event handler M2 is executed first, followed by the event handler defined in EHclass1, M3.

The output from this test program is

```
Event is triggered by attribute n
Event is triggered by _sendEvent
a=abc n1=3 n2=4
Event myEvent is defined in another class
that is triggered by _sendEvent.
a=abc n1=3 n2=4
```

## Event and Event Handler Metadata

Events and event handlers are implemented and maintained with metadata. This metadata exists as a list that is stored with the class. You can query a class (or an event within a class) to view the event and event handler metadata. To list the metadata for the an event, execute code similar to the following:

```
init:
   DCL num rc;
   DCL list metadata;
   DCL object obj;

   obj=loadclass('class-name');

   rc=obj._getEvent('event-name',metadata);
   call putlist(metadata,'',3);
   rc=obj._getEventHandler('_self_','event-handler-name',
                            '_refresh',metadata);
   call putlist(metadata,'',3);
return;
```

# Interfaces

Interfaces are groups of method declarations that enable classes to possess a common set of methods even if the classes are not related hierarchically. An interface is similar to a class that contains only method declarations.

A class can either *support* or *require* an interface. A class that supports an interface must implement all of the methods in the interface. A class that requires an interface can invoke any of the methods in the interface.

Suppose you have the following interface:

```
interface I1;
  M1: method;
endinterface;
```

If class A supports the interface, then it must implement the method M1:

```
class A supports I1;
  M1: method;
    put 'Implementation of M1';
  endmethod;
endclass;
```

Class B requires the interface, which means that it can invoke the methods declared in the interface.

```
class B requires I1;
  M2: method;
```

```
    dcl I1 myObj = _new_ I1;
    myObj.M1();
  endmethod;
endclass;
```

Interfaces are especially useful when you have several unrelated classes that perform a similar set of actions. These actions can be declared as methods in an interface, and each class that supports the interface provides its own implementation for each of the methods. In this way, interfaces provide a form of multiple inheritance.

A class can be defined to support or require one or more interfaces. If two components share an interface, they can indirectly call each others' methods via that interface. For example, model/view component communication is implemented with the use of interfaces. The model typically *supports* the interface, whereas the view *requires* the same interface. The interfaces for the components must match before a model/view relationship can be established. A class stores interface information as a property to identify whether it supports or requires an interface. Refer to *SAS Guide to Applications Development* for more information about model/view communication.

Although classes that support or require an interface are often used together, they are still independent components and can be used without taking advantage of an interface.

## Defining Interfaces

You define interfaces with the INTERFACE statement block:

INTERFACE *interface-name*
    <EXTENDS *interface-name*>
    </ (*interface-optional-clause*)>;

<*limited-method-declaration-statements*>

ENDINTERFACE;

For more information about defining interfaces, see Chapter 9, "Example: Creating An Object-Oriented Application," on page 137 and "INTERFACE" on page 486.

## Example

The following INTERFACE block declares two methods for reading and writing data.

```
interface Reader;
  Read: method return=string;
  Write: method data:string;
endinterface;
```

Only the method declarations are given in the INTERFACE block. The method implementations are given in any class that supports the interface.

For example, the Lst and Ddata classes both implement the Read and Write methods. These classes *support* the Reader interface. In the Lst class, these methods read and write items from and to an SCL list.

```
class Lst supports Reader;
  dcl list L;
  dcl num cur n;

  /* Override the class constructor. */
  /* Create a new list.              */
  Lst: method/(state='o');
```

```
    L = makelist();
    cur = 0;
    n = 0;
  endmethod;

  Read method: return=string;
    if (cur >= n) then do;
      put 'End of file';
      return "";
      end;
    else do;
      cur + 1;

      /* Get the current item from the list. */
      return getitemc(l,cur);
    end;
  endmethod;

  Write method: c:string;
    n + 1;

    /* Insert a new item into the list. */
    insertc(l,c,-1);
  endmethod;

endclass;
```

The method implementations in the Ddata class read and write data from and to a SAS table.

```
class Ddata supports Reader;
  protected num fid;
  protected num obs n;

  /* Override the class constructor. */
  /* Use the open function to open a SAS table. */
  Ddata: method name: string mode: string;
    fid = open(name, mode);
    obs = 0;
    n = 0;
  endmethod;

  Read method: return=string;
    if (obs >= n) then do;
      put 'End of file';
      return "";
    end;
    else do;
      dcl string c;

      /* Fetch an observation from the table. */
      obs + 1;
      fetchobs(fid, obs);

      /* Get the contents of table column 1. */
```

```
      c = getvarc(fid, 1);
      return c;
    end;
  endmethod;

  Write method: c:string;
    dcl num rc;

    /* Add a new row to the table and         */
    /* write the contents of C into column 1. */
    append(fid);
    call putvarc(fid, 1, c);
    rc = update(fid);
    n + 1;
  endmethod;

endclass;
```

Using the interface, you can read and write data without knowing the data source. In the following example, the Read class implements method M, which calls the method that was declared in the Reader interface. The interface determines which method implementation is executed.

```
class Read;
  M: method r:Reader;

    /* Write a string to the data source, */
    /* then read it back.                 */
    r.write("abc");
    put r.read();
  endmethod;
endclass;
```

The following labeled program section reads and writes data to both a list and a SAS table. This code passes a Lst class and a Ddata class to the Read class, which treats the list and the table in the same way. The data is read and written transparently. The Read class does not need to know what the actual implementation of the Reader is — it only needs to know the interface.

```
init:
  dcl Lst L = _new_ Lst();
  dcl Ddata D = _new_Ddata("test","un");
  dcl read R = _new_ read();

  R.M(L);
  R.M(D);
return;
```

## Converting Version 6 Non-Visual Classes to Version 8 Classes

You do not need to convert Version 6 classes to Version 8 classes in order to run programs from the previous versions. Version 6 classes are automatically loaded into Version 8 formats when they are instantiated. Existing Version 6 SCL programs should run normally in Version 8 environments.

However, you can use Version 8 SAS Component Object Model (SCOM) features to make your programs more object-oriented, easier to maintain, and more efficient. Using SCOM features also enables you to reuse model classes in the future development of client/server applications.

To convert Version 6 model classes to Version 8 classes, you must modify the method implementation files and regenerate the class files. To modify the method implementation files, follow these steps:

1 Remove global variables. Declare them as private attributes or, if they are referenced in only one method, declare them as local variables within that method. See “Removing Global Variables” on page 132 for more information.
2 Declare all variables. See “Declaring Variables” on page 133 for more information.
3 Convert labels to method names and convert LINK statements to method calls. Declare the labeled sections as private methods. If necessary, specify the **Forward='Y'** option for the method. See “Converting Labels and LINK Statements” on page 133 for more information.
4 Convert CALL SEND statements to dot notation. See “Converting CALL SEND to Dot Notation” on page 134 for more information.

To regenerate the class files, follow these steps:

1 Use CREATESCL to convert Version 6 class files to Version 8 class files. See “Converting Class Definitions with CREATESCL” on page 134 for more information.
2 Convert instance variables to attributes, if appropriate. See “Using Instance Variables” on page 135 for more information.
3 Make sure signatures are generated for all methods. The best way to ensure that signatures are generated is to delete the method declarations from the class files and to replace them with the METHOD blocks from the method implementation files.
4 Change the class names specified in the CLASS statements if you do not want to overwrite the existing Version 6 classes.
5 Issue the SAVECLASS command to generate the new Version 8 class.

## Removing Global Variables

Remove all global variables from the Version 6 method implementation entries. Convert them either to local variables through DECLARE or to private attributes in the class definition file. For example, suppose that a Version 6 method implementation file contains the variables N1, N2, C1 and C2 as shown:

```
length n1 n2 8;
length c1 c2 $200;
```

In this example, four attributes need to be added to **mylib.classes.newclass.scl**, as follows:

```
Private num n1;
Private num n2;
Private char c1;
Private char c2;
```

After the attributes are added, issue the SAVECLASS command to generate the new class.

## Declaring Variables

Declare all of the variables in your program. Lists should be declared with the LIST keyword rather than allowing them to default to a Numeric type. Objects should be declared either as generic objects (with the OBJECT keyword) or as specific class objects. You can use dot notation with an object only if it is declared as an object. Using specific LIST and object declarations can avoid problems with overloading methods. For more information, see "Overloading and List, Object, and Numeric Types" on page 111.

Whenever possible, classes should be declared with a specific class declaration such as

```
dcl work.a.listbox.class lboxobj;
```

Try to avoid using generic object declarations such as

```
dcl object lboxobj;
```

Also, the compiler cannot check method signatures or validate methods and attributes if it does not know the specific class type. If the compiler is not able to do this checking and validation at compile time, then SCL must do it at run time, which makes your program less efficient.

For example, assume that you declare a generic object named SomeC that has a method Get, which returns a numeric value. You also declare a class named XObj that has a method M, which is overloaded as (N)V and (C)V. Suppose you need to pass the return value of Get to the M method:

```
dcl object SomeC = _new_ someclass.class();
dcl work.a.xclass.class XObj = _new_ xclass.class();
XObj.M(SomeC.Get());
```

SomeC is declared as a generic object, so the compiler cannot determine what object it contains at compile time. Even though there is a specific object assignment to SomeC, the compiler cannot guarantee what type it will contain at any given point, because the value could be changed elsewhere when the program runs.

Therefore, the compiler cannot look up the Get method to find that it returns a Numeric value, and it cannot determine which method M in Xclass to call. This method validation must be deferred until run time, when the return type of the Get method will be known (because the actual call will have taken place and the value will have been returned).

The problem can be remedied by declaring SomeC as a specific object:

```
dcl someclass SomeC = _new_ someclass.class();
```

If this is not possible, then you could declare a Numeric variable to hold the result of the Get method, as shown in this example:

```
dcl object SomeC = _new_ someclass.class();
dcl xclass XObj = _new_ xclass.class();
dcl num n;
n = SomeC.Get();
XObj.M(n);
```

Even though the compiler cannot validate the Get method for the SomeC class, it can validate the method name and parameter type for XObj.

## Converting Labels and LINK Statements

The next step is to remove all link labels from the Version 6 method implementation catalog entries. Convert them to private methods in the class definition file, and convert

the link to a method call. For example, suppose that **myclass.classes.old.scl** contains the following:

```
m1: method;
   link a1;
endmethod;

a1:
  ...SCL statements...
return;
```

To change the labeled section to a private method in **mylib.classes.newclass.scl**, add the following:

```
a1: Private method;
   ...SCL statements...
endmethod;
```

If needed, you can also add parameters to the method. To change the link to a method call, change the following:

```
m1: method;
   a1();
endmethod;
```

In the old entry, the A1 labeled section is after the M1 method. In the new entry, the labeled section has been converted to a method. However, you cannot call a method before it is declared. To fix this problem, you must either move the A1 method before the M1 method, or you can declare A1 with the **Forward='Y'** option:

```
a1: Private method / (Forward='y');
   ...SCL statements...
endmethod;
```

## Converting CALL SEND to Dot Notation

The final step in modifying your method implementation files is converting CALL SEND statements to METHOD calls that use dot notation.

*Note:* To use dot notation, the method that you specify must have a signature. Therefore, you cannot convert CALL SEND statements to dot notation unless your class files have been converted to Version 8 class files. Also, the object that you specify should be declared as a specific class type to enable the compiler to validate method parameters. △

For example, suppose that a Version 6 program contains the following line:

```
call send(obj1,'m1',p1);
```

Converting this line to dot notation results in

```
obj1.m1(p1);
```

## Converting Class Definitions with CREATESCL

Assume that the Version 6 class is **mylib.classes.oldclass.class** and that the method implementation file is **mylib.classes.old.scl**.

1 Use CREATESCL to create an SCL entry that contains the following SCL statements:

```
Init:
   rc=createscl('mylib.classes.oldclass.class',
                'mylib.classes.newclass.scl');
return;
```

2 Issue the SAVECLASS command to generate the Version 6 class file **mylib.classes.newclass.class**.

3 Open this entry in the Build window and modify the class definition as needed. Reissue the SAVECLASS command to generate the new class file in Version 8 format.

## Using Instance Variables

The object model in Version 6 uses instance variables. In Version 8, instance variables have been replaced with attributes.

When a class is loaded, the class loader automatically converts Version 6 formats to the Version 8 format. This process includes converting instance variables to public or private attributes with the option IV, which specifies the name of the Version 6 instance variable.

In the following example, the Version 6 instance variable ABC is converted to the Version 8 attribute **abc**.

```
class IVclass;
  public char abc / (iv='ABC');
endclass;
```

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Component Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999.

**SAS® Component Language: Reference, Version 8**

Copyright © 1999 by SAS Institute Inc., Cary, NC, USA.

ISBN 1–58025–495–0

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

**U.S. Government Restricted Rights Notice.** Use, duplication, or disclosure of the software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227–19 Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The Institute is a private company devoted to the support and further development of its software and related services.