

Guide to SAS/AF[®] Applications Development



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2011. *Guide to SAS/AF® Applications Development*. Cary, NC: SAS Institute Inc.

Guide to SAS/AF® Applications Development

Copyright © 2011, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America.

For a hardcopy book: 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.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

This content was previously published as *SAS® Guide to Applications Development, Second Edition*.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this 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 electronic book, July 2011

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at support.sas.com/publishing or call 1-800-727-3228.

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.

Contents

<i>About This Book</i>	<i>vii</i>
<i>Recommended Reading</i>	<i>ix</i>

PART 1 Introducing the SAS/AF Development Environment 1

Chapter 1 • The Development Environment	3
What Is SAS/AF Software?	3
Chapter 2 • An Applications Development Methodology	5
Using SAS Tools to Develop Applications	5
Steps to Developing SAS/AF Applications	5

PART 2 Developing Applications 11

Chapter 3 • Tools for the Applications Developer	13
About the SAS/AF Development Environment	13
Build Window	15
Components Window	15
Components	16
Properties Window	17
Source Window	17
Chapter 4 • Developing Frames with Ready-Made Objects	19
Introduction	19
Working with Frames	20
Selecting Components	23
Re-using Components	24
Defining Attachments to Enable Automatic Component Resizing	25
Chapter 5 • Communicating with Components	27
Introduction	27
Attribute Linking	28
Model/View Communication	29
Drag and Drop Communication	32
Chapter 6 • Adding SAS Component Language Programs to Frames	35
Introduction	35
Working with Frames and SCL	36
When Frame SCL Is Not Required	37
When Frame SCL Is Required	37
Constructing a Frame SCL Program	37
Controlling the Execution of SCL Programs	41
Calling Other Entries and Opening Windows	44
Compiling and Testing Applications	46

Debugging and Optimizing Your Applications	47
Saving and Storing Frame SCL Programs	49
Chapter 7 • Adding Application Support Features	51
Implementing Custom Menus	51
Adding Online Help to Your Applications	53
Chapter 8 • Deploying Your Applications	57
Application Deployment Issues	57
Migrating Your Application from Testing to Production	58
Configuring a SAS Session for Your Application	60
Enabling the Application for Your Users	63
PART 3 Developing Custom Components 65	
Chapter 9 • Tools for the Component Developer	67
Introduction	67
Class Editor	67
Resource Editor	68
Interface Editor	68
Source Window	69
Other Development Tools and Utilities	70
Chapter 10 • SAS Object-Oriented Programming Concepts	71
Introduction	71
Object-Oriented Development and the SAS Component Object Model	73
Classes	73
Methods	77
Attributes	84
Events	90
Event Handlers	92
Interfaces	93
Chapter 11 • Developing Components	95
Creating Your Own Components	95
Creating a Class with SCL	100
Using SCL to Instantiate Classes	105
Chapter 12 • Managing Methods	109
Implementing Methods with SCL	109
Overriding Frame Methods to Augment SCL Labeled Sections	113
Chapter 13 • Managing Attributes	115
Specifying a Component's Default Attribute	115
Validating the Values of Character Attributes	116
Assigning an Editor to an Attribute	117
Adding a Custom Attributes Window to a Component	119
Assigning a Custom Access Method (CAM) to an Attribute	121
Using the Refresh Attributes Event	125
Using List Attributes	127
Chapter 14 • Adding Communication Capabilities to Components	129
Introduction	129
Enabling Attribute Linking	130

Implementing Model/View Communication	133
Enabling Drag and Drop Functionality	139
Modifying or Adding Event Handling	148
Chapter 15 • Deploying Components	151
Introduction	151
Managing Classes with Resources	152
Making a New Class Available for Use	154
Generating Class Documentation with GenDoc	156
PART 4 Appendixes 161	
Appendix 1 • Flow of Control	163
How SCL Programs Execute for FRAME Entries	163
Flow of Control for Frame SCL Entries	169
Appendix 2 • Adding Attachments to Frame Controls	179
Introduction	179
Selecting the Attachment Mode	180
Initiating Define Attachment Mode	181
Selecting the Direction and Type for the Attachment	182
Making the Attachments	186
Defining Attachments to Sibling Components	188
Defining Attachments to Components That Have Borders	189
Moving Multiple Components That Include Attachments	190
Restricting Component Size	190
Changing and/or Deleting an Attachment	192
Displaying Attachments	192
Situations in Which an Attachment Is Ignored	193
Errors and Error Handling	193
Tips for Using Attachments	194
Appendix 3 • Working with SAS/AF Keys in the SAS Registry	197
About the SAS Registry	197
SAS Registry Editor	197
Modifying the Registry Settings for SAS/AF Software	198
Glossary	201
Index	209

About This Book

Audience

SAS/AF software provides IS and IT developers the tools that they need to build enterprise-wide applications that can be integrated with the features and functionality of other SAS software products. Whether you are a component designer who defines the architecture and creates class libraries, or an application developer who uses the supplied components to create applications, this book provides a single reference to begin developing applications with SAS software.

Requirements

To use the SAS/AF development environment requires a graphics display device capable of displaying the Graphics Editor in SAS/GRAPH output.

You can run frame-based applications on nongraphics devices, but any graphics objects in the frame will not be displayed, except for control objects (which are displayed as question marks (?) by default). Text-based frames are those that do not use SAS/GRAPH output, SAS/GRAPH fonts, graphics control boxes, or special region outlining and titles. For detailed requirements and the most current information, see the system requirements sheet that is shipped with SAS/AF software.

To create and run applications in FRAME entries, you need the following SAS software products:

- Base SAS software
- SAS/AF software (required only for creating Frame applications; not required for running Frame applications)
- SAS/GRAPH software (required for creating or displaying graphics objects or images, for using SAS/GRAPH fonts, and for printing some frame objects)

Recommended Reading

Other resources that may be useful to you:

- The *SAS Component Language: Reference* provides detailed reference information about the statements, functions, and other elements of SCL. Available from the SAS/AF Product Documentation page at support.sas.com/documentation/onlinedoc/af/.
- The online Help for SAS/AF software. The Help system provides a complete Component Reference, enabling you to quickly view class and property information.
- *Getting Started with SAS/AF and Frames* provides a more basic introduction to using FRAME entries and SCL. Available from the SAS/AF Product Documentation page at support.sas.com/documentation/onlinedoc/af/.
- SAS offers instructor-led training and self-paced e-learning courses to help you get started with SAS/AF. For more information about the courses available, see support.sas.com/training.

For a complete list of SAS publications, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Publishing Sales Representative:

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

x *Recommended Reading*

Part 1

Introducing the SAS/AF Development Environment

<i>Chapter 1</i>	
The Development Environment	3
<i>Chapter 2</i>	
An Applications Development Methodology	5

Chapter 1

The Development Environment

What Is SAS/AF Software?	3
Overview of SAS/AF Software	3
SAS Component Language (SCL)	3
Mainframe Support Issues	3

What Is SAS/AF Software?

Overview of SAS/AF Software

SAS/AF software is a set of development tools to help you create customized applications. With its interactive development environment and rich set of object-oriented classes, you can rapidly develop and deploy portable, GUI applications that take advantage of other SAS software products. Central to SAS/AF development is the *frame*, which is an application window that contains the interface elements, such as fields, buttons, and tables. Although the many built-in SAS/AF components are flexible, you can extend them using SAS Component Language (SCL).

SAS Component Language (SCL)

SCL is the programming language that controls SAS/AF applications. SCL programs are stored in separate SCL entries that can be accessed by more than one FRAME entry. This means that an SCL program can be written once and used many times.

SCL provides complete object-oriented programming constructs for creating entire object-oriented applications in SCL and for creating and scripting objects with the SAS Component Object Model (SCOM).

For more information about SCOM, see [“Object-Oriented Development and the SAS Component Object Model” on page 73](#). For more information about SCL, see the *SAS Component Language: Reference*.

Mainframe Support Issues

SAS/AF software does not support a mainframe build-time environment for FRAME entries. However, mainframe SAS/AF developers can create and modify SCL programs as well as build PROGRAM entries and full-screen applications.

Because SAS/AF applications are stored in SAS catalogs, they are portable to all SAS software platforms. If your site primarily uses a mainframe, you can develop frame-

based applications on a PC running Windows, and then port the application to your mainframe. Users who run GUI applications on character-based display devices will see widgets represented as characters that are typical for that host environment.

For more information, see [“Porting Your Application to Different Operating Environments” on page 59](#). For more information about the behavior of classes in specific host environments, see the *SAS/AF Component Reference* in the online Help.

Chapter 2

An Applications Development Methodology

Using SAS Tools to Develop Applications	5
Steps to Developing SAS/AF Applications	5
Step 1: Analyze the Problem or Business Process	5
Step 2: Set Up the Development Environment	6
Step 3: Design and Develop Components	7
Step 4: Develop the Application	8
Step 5: Compile and Test Your Application	8
Step 6: Deploy Your Application	9

Using SAS Tools to Develop Applications

The first step in building SAS applications is to determine the tool that is most appropriate for your situation.

You can use SAS/AF software

- to extend the functionality of another SAS software product
 - to develop an enterprise-wide application that might involve custom components and programming
-

Steps to Developing SAS/AF Applications

Step 1: Analyze the Problem or Business Process

Begin a development project by conducting an analysis to discover the key issues of the problem. Such an analysis enables you to understand and describe the problem, which promotes the mapping of the issues and their relationships to your application's components. The purpose of object-oriented analysis, then, is

- to define the boundaries of the problem domain so that you know what the application will do (and what it will not do)
- to describe the problem domain in terms of objects and classes, and to determine what services the objects and classes must provide

- to identify relationships between the objects in the problem domain, especially from the perspective of different types of users

When you model a problem or business process, it is important to consider all potential users of the application. Such consideration promotes cross-functional input and a more complete understanding of the problem. The terminology and concepts that describe the problem lead to new issues that enable you to better meet the needs of the users.

For example, consider the problems and business processes that underlie a financial application. Users might include executives, managers, accountants, clerks, and auditors. Concepts such as “balance sheet” and “invoice” can be easily understood, but a process such as “consolidate balance sheets” can mean different things to different user groups. Executives might be interested in the bottom line, whereas auditors might focus more on the details of the consolidation process.

The outcome of an analysis phase is typically a requirements document. In some cases, it might be important to prepare simple class diagrams that provide an overview of what each object should do. There are many formal object-oriented methodologies as well as a number of commercial products that can assist you in the design process.

Step 2: Set Up the Development Environment

Considerations When You Are Planning a New Application

As you begin planning a new SAS/AF application, consider the following:

- Will you need to create a common SAS library or use a combination of new and existing libraries to access the application, its classes, and the data that it requires?
- How will you reference SAS catalogs and catalog entries during the development and deployment of your application?
- What SAS software products will end users need to license?

Creating Common SAS Libraries and Catalogs

You might consider setting up a common SAS library and appropriate catalogs for an applications development project. Often, this makes it easier for your development team to locate, share, and store the SAS catalog entries that your application requires.

You can store a mixture of entry types in a SAS catalog. The following table lists the common entry types that you are likely to use in your SAS/AF applications.

Table 2.1 SAS/AF Entry Types

Entry Type	Description
CLASS	Stores the properties that define a class.
FRAME	Stores the properties that define a frame.
INTERFACE	Stores the definition of an interface, which is a group of abstract methods that are shared by related classes.
KEYS	Stores function key settings.
PMENU	Stores the code that defines a menu.

Entry Type	Description
RANGE	Stores the definition of a range of values. Range entries can define ranges for both numeric text-entry fields and critical success factor (CSF) controls.
RESOURCE	Stores the definition of a resource, which typically includes a group of related classes.
SCL	Stores SCL program code.

You can also use library or catalog concatenation so that you can access entries in multiple libraries or catalogs by specifying a single library reference name (libref) or a catalog reference name (catref). If you create a catalog concatenation, you can specify the catref in any context that accepts a simple, nonconcatenated catref.

For example, you might need to access entries in several SAS catalogs that might be designated as the development, test, and production areas. The most efficient way to access the information is to logically concatenate the catalogs, which allows access to the information without creating a new catalog. The following statement assigns the catref **app** to development, test, and production catalogs:

```
catname app (corp.app mis.testapp projects.newapp);
```

In this example, **corp.app** is the production environment, **mis.testapp** is the testing catalog, and **projects.newapp** is the development area.

Catalog concatenation is useful when you need to migrate catalog entries — particularly those that have hardcoded SCL entry names such as classes — between different environments. For details on catalog concatenation, see the *SAS Language Reference: Concepts*. For information about the use of libraries and catalogs when you deploy applications, see “[Step 6: Deploy Your Application](#)” on page 9.

Using Existing SAS Components

Your organization might already have a number of existing custom SAS components available. If you plan to use any of these custom components in your new application, then you should consider how the classes are grouped and made available in a resource. For more information, see “[Deploying Components](#)” on page 151 .

Licensing SAS Software Products

When you create an application, you can use the functionality of different SAS software products, or you can develop objects (such as a graph) that require other SAS software products to operate. In either case, you must ensure that your end users license all the SAS software products that they need for your application to run.

Step 3: Design and Develop Components

The object-oriented design phase of the development life cycle is often focused on the work of component developers or “class writers.” As a component developer, you can define how the elements of the application work. During this phase, you can detail and describe

- the services provided by each class.
- the actual name, data type, and default values of class attributes.
- any class relationships — especially inheritance.

- the business rules that govern object behavior.
- other features that are specific to SAS software. See [“SAS Object-Oriented Programming Concepts” on page 71](#) for details.

You can construct classes in terms of their attributes and behaviors. For complete information about using SAS/AF software to design and implement classes, see Part 3: Developing Custom Components. For information about making the components that you create available for use, see [“Deploying Components” on page 151](#).

Step 4: Develop the Application

Overview of Developing Applications

This phase of the development life cycle is where application developers build the system or application, using the objects that have been created by component developers. For complete information about using SAS/AF software to develop applications, see Part 2: Developing Applications.

Designing the User Interface

You can use any one of a number of modeling tools and flow chart techniques to help determine how many windows your application will need, as well as how the user should interact with each window and how the windows should be connected to each other.

Consider the tasks that a user needs to perform in a window, and a frame that enables the user to complete those tasks. For information about building frames, see [“Working with Frames” on page 20](#).

Adding Communication Between Frames and Components

Your application design might specify that the components on a frame, or even the frames themselves, need to work with each other. For example, you might want a graphic object to display a specific graph when a user selects a graph from a list box. Or, you might want the user to open another frame when a push button is selected. You can add component and frame communication either by using SAS Component Language (SCL) programs or by taking advantage of the features included in components that have been built using the SAS Component Object Model (SCOM) architecture.

For more information about SCL, see [“Adding SAS Component Language Programs to Frames” on page 35](#). For more information about SCOM, see [“Communicating with Components” on page 27](#).

Designing for Reuse

If you notice that several of your frames share common traits, you can create a composite object that includes those traits. For example, you could create a dialog box frame that included OK, Cancel, and Help buttons. For more information about composites, see [“Re-using Components” on page 24](#).

Step 5: Compile and Test Your Application

Testing SCL Programs with Frames

When you use SCL programs with the frames of your application, you must compile and test your frames. You can perform debugging and unit testing as well. Once all of the individual pieces are working, you might also consider assembling all application components and performing integrated testing.

For information about compiling and testing, see [“Compiling and Testing Applications” on page 46](#).

Testing the Usability of Your Application

Although your application enables users to perform all the tasks that they need to complete, it might still be difficult to learn or to use. Just as the early analysis phase encourages input from users, you might want to revisit the design by having users test the application during its development. Usability testing enables you to enhance or correct usability issues that might arise from difficult concepts or business processes. There are many different types of usability testing, ranging from user exploration of the application to formal testing that assesses validity and usability.

Step 6: Deploy Your Application

Once you have designed, created, and tested your application, you are ready to deploy your application to users. Deploying applications involves many considerations, including how the application should appear when it opens, whether the application will run on a stand-alone machine or a network, and what should happen when the application closes.

For more information about application deployment issues, see [“Deploying Your Applications” on page 57](#).

Part 2

Developing Applications

<i>Chapter 3</i>	
Tools for the Applications Developer	<i>13</i>
<i>Chapter 4</i>	
Developing Frames with Ready-Made Objects	<i>19</i>
<i>Chapter 5</i>	
Communicating with Components	<i>27</i>
<i>Chapter 6</i>	
Adding SAS Component Language Programs to Frames	<i>35</i>
<i>Chapter 7</i>	
Adding Application Support Features	<i>51</i>
<i>Chapter 8</i>	
Deploying Your Applications	<i>57</i>

Chapter 3

Tools for the Applications Developer

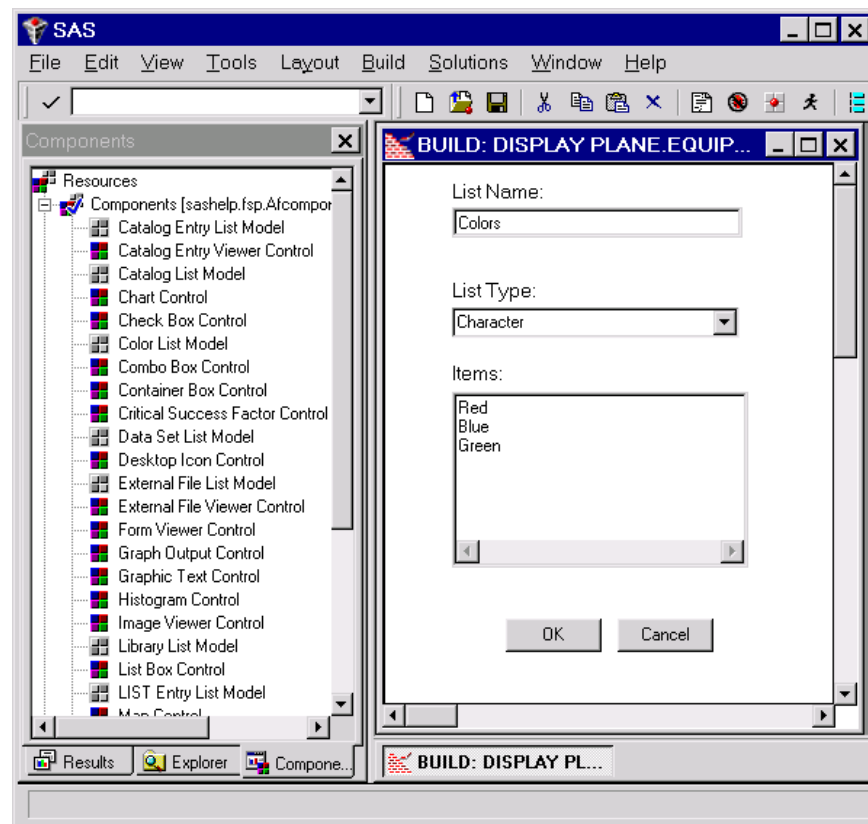
About the SAS/AF Development Environment	13
Build Window	15
Components Window	15
Components	16
Properties Window	17
Source Window	17

About the SAS/AF Development Environment

As an applications developer, you create the frames that define application windows. You can use the following tools from the SAS/AF development environment to create frames:

- the Build window
- the Properties window
- the Components window
- the Components themselves
- the Source window

When you create or open a FRAME entry, SAS/AF software opens its build-time environment, which includes specific windows.

Display 3.1 SAS/AF Build-Time Environment

Not every build-time environment window opens when you create or open a FRAME entry. Some windows must be opened by the user.

You can open build-time environment windows in several ways.

Make a selection from a menu.

The menu of any active build-time window provides several options. Many options are duplicated on pull-down and pop-up menus.

Make a selection from a pop-up menu.

A pop-up menu is available when the frame or any component on the frame is selected. (Usually, you will not see any visual cues to indicate where pop-up menus can be displayed.)

Enter a command at the command prompt.

Users who are more comfortable with command-driven processes can use the command line or the command window. Refer to the SAS/AF online Help for a list of valid commands.

Click an icon on the toolbar.

The toolbar includes icons for the Properties window, the Components window, and the Source window.

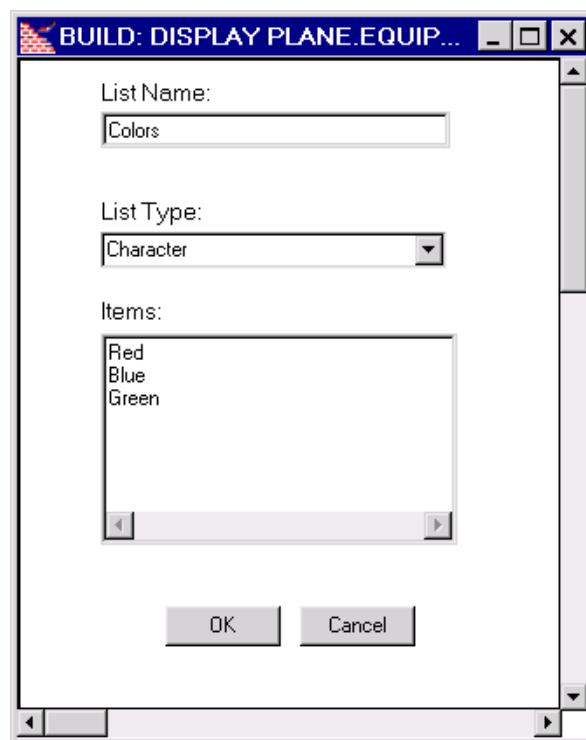
Press a function key that has an assigned command.

You might find it useful to assign commands to function keys to create shortcuts to both command- and menu-driven actions.

Build Window

The Build window enables you to drop components onto a frame and then to manipulate the layout and appearance of your application's user interface. Menus provide access to commands that enable you to set component properties, to edit and compile SCL programs, and to access other tools in the development environment. When the Build window opens, it displays an instance of the Frame class (or a subclass of the Frame class). When you save a frame, the contents of the Build window are stored in a FRAME entry of a SAS catalog.

Display 3.2 SAS/AF Build Window

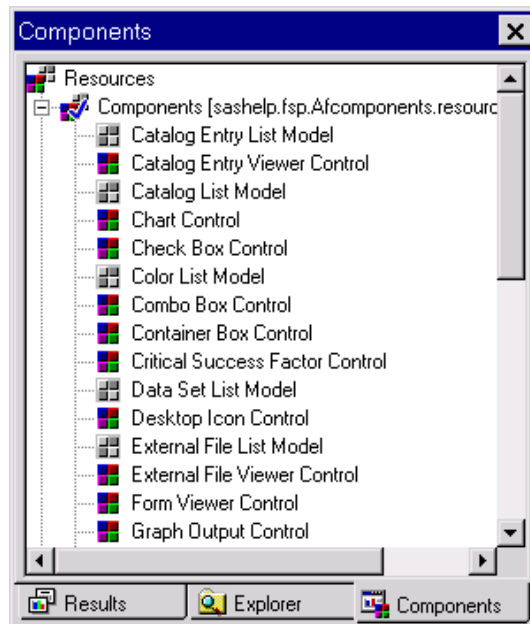


The Build window displays both horizontal and vertical scroll bars.

The Build window appears when you open an existing FRAME entry or create a new FRAME entry. For more information about the Build window, see “Working with Frames” in the SAS/AF online Help.

Components Window

The Components window enables you to view and select components that you can add to a frame at build time. There are a number of ways to add a component to a frame. For example, you can select the component from the Components window, and then drag the component and drop it onto a frame. You can also double-click a component in the Components window to make that component appear on a frame.

Display 3.3 SAS/AF Components Window

With the Components window, you can also

- manage resource entries for maintaining class libraries.
- add to the classes or resources that appear in the window.
- access Help on classes and resources. To access Help, right-click on a component and select **Help on Class**.

By default, the Components window appears when you open a FRAME entry (although you can prevent this by editing the **AutoOpen** property for the Components window in the SAS Registry). You can also click the Components window icon on the toolbar or select **View** ⇒ **Component Window** to open the Components window.

For more information about the Components window, see “Customizing the Components Window” in the SAS/AF online Help. For more information about the SAS Registry, see “Modifying SAS/AF Items in the SAS Registry” in the SAS/AF online Help.

Components

Components themselves are tools that you can use to design application windows. SAS/AF software provides ready-made components that are stored in **sashelp.fsp.afcomponents.resource**. These ready-made components appear in the Components window by default.

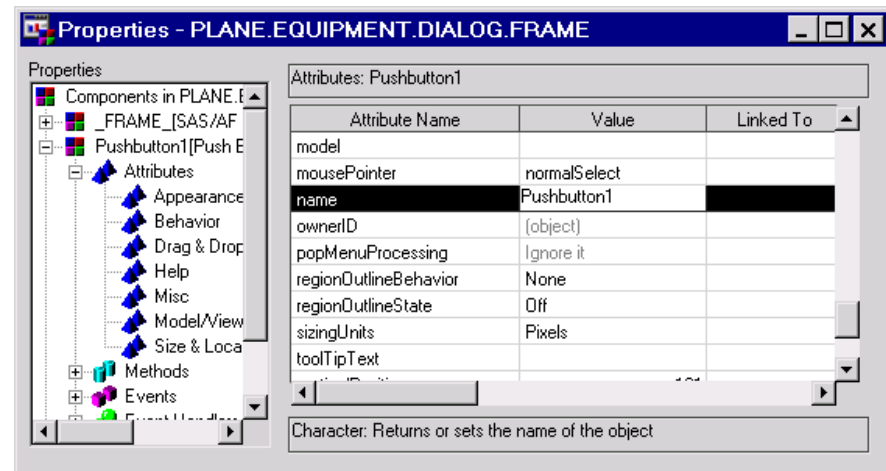
Your organization might also need to create custom components to meet specific application needs.

For information about selecting the appropriate component, see “[Selecting Components](#)” on page 23. For information about creating custom components, see “[Developing Components](#)” on page 95.

Properties Window

The Properties window enables you to set component properties at build time. Properties include the attributes, methods, events, and event handlers of a frame or a component. You can view, edit, or add properties with this window.

Display 3.4 Properties Window



You can open the Properties window once you create or open a frame in the Build window. Click the Properties window icon on the toolbar or select **View ⇒ Properties Window**.

For more information about working with the Properties window, see “Working with Component Properties” in the SAS/AF online Help.

Source Window

The Source window provides a text editor for creating and editing SAS Component Language (SCL) programs.

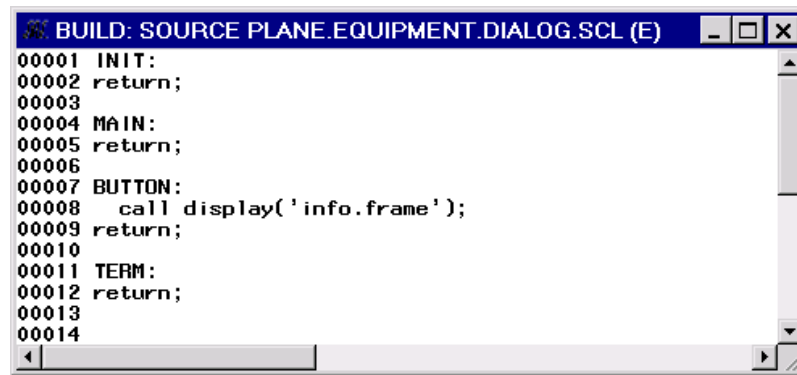
You can edit a frame's SCL entry by selecting **View ⇒ Frame SCL** or by selecting **Frame SCL** from the frame's pop-up menu.

You can also open the Source window by

- double-clicking an existing SCL entry in the SAS Explorer
- creating a new SCL entry via the BUILD command
- clicking the Source window icon on the toolbar
- issuing the SOURCE command from a FRAME or PROGRAM entry

Note: Avoid opening frame SCL entries in a Source window or by double-clicking them in the SAS Explorer. Instead, open a frame SCL entry from within its respective frame. A frame SCL entry must be compiled along with its respective frame. If you open a frame SCL entry outside of its frame and inadvertently compile the frame SCL entry, you will produce errors.

Display 3.5 SAS/AF Build Window

A screenshot of a SAS/AF Build Window. The window title bar reads "BUILD: SOURCE PLANE.EQUIPMENT.DIALOG.SCL (E)". The main area contains the following SCL code:

```
00001 INIT:  
00002 return;  
00003  
00004 MAIN:  
00005 return;  
00006  
00007 BUTTON:  
00008   call display('info.frame');  
00009 return;  
00010  
00011 TERM:  
00012 return;  
00013  
00014
```

The window has a standard Windows-style border with minimize, maximize, and close buttons in the top right corner. A vertical scrollbar is on the right side, and a horizontal scrollbar is at the bottom.

For information about SCL, see [“Adding SAS Component Language Programs to Frames”](#) on page 35.

Chapter 4

Developing Frames with Ready-Made Objects

Introduction	19
Working with Frames	20
Opening an Existing FRAME Entry or Creating a New FRAME Entry	20
Frame Types	20
Frame Usage Tips	22
Specifying the Frame SCL Entry	22
Compiling Frames	22
Selecting Components	23
Re-using Components	24
Combining Components to Create Composites	24
Subclassing	25
Writing Methods	25
Defining Attachments to Enable Automatic Component Resizing	25

Introduction

You can quickly develop GUIs by building frames and adding ready-made objects (components) to your frames. Building a frame-based application typically consists of the following steps:

1. Create a frame (or a set of frames) to serve as the user interface for your application.
2. Add components to each frame by dragging a selected component from the Components window and dropping it onto the frame.
3. Modify the properties of each component with the Properties window.
4. Add frame SCL as needed to incorporate the necessary logic or business rules behind the application.
5. Save, compile, and test the frame or frames.

Note: SAS/AF software supports a native look and feel for controls (visual components) on a frame. This capability enables you to take advantage of best practices for user interface design for those platforms on which you implement your application.

Working with Frames

Opening an Existing FRAME Entry or Creating a New FRAME Entry

Frames provide the user interface to SAS/AF applications. Frames are stored in a SAS catalog entry of type FRAME. A frame is also an instantiation of the Frame class or one of its subclasses.

The SAS/AF application development environment provides several ways to open an existing FRAME entry or to create a new one:

From the Explorer window

- To open an existing frame, select a catalog, and then double-click on the FRAME entry you want to open.
- To create a new frame, select the catalog in which you want to store the FRAME entry, and then select **File** ⇒ **New** and select **Frame** from the New Entry dialog box.

From the Command prompt

To open an existing frame or to create a new one, enter **build libref.catalog.framename.frame**, where *framename* is the name of the FRAME entry.

From the SAS/AF software development environment

To create a new frame from the SAS/AF development environment, select **File** ⇒ **New**.

From the Program Editor using PROC BUILD

To open an existing frame or to create a new one, submit **proc build c=libref.catalog.framename.frame; run;** where *framename* is the name of the FRAME entry.

The BUILD command has advantages over the BUILD procedure in that the BUILD command enables you to open more than one BUILD session at a time and allows you to submit other SAS procedures while the BUILD session is active.

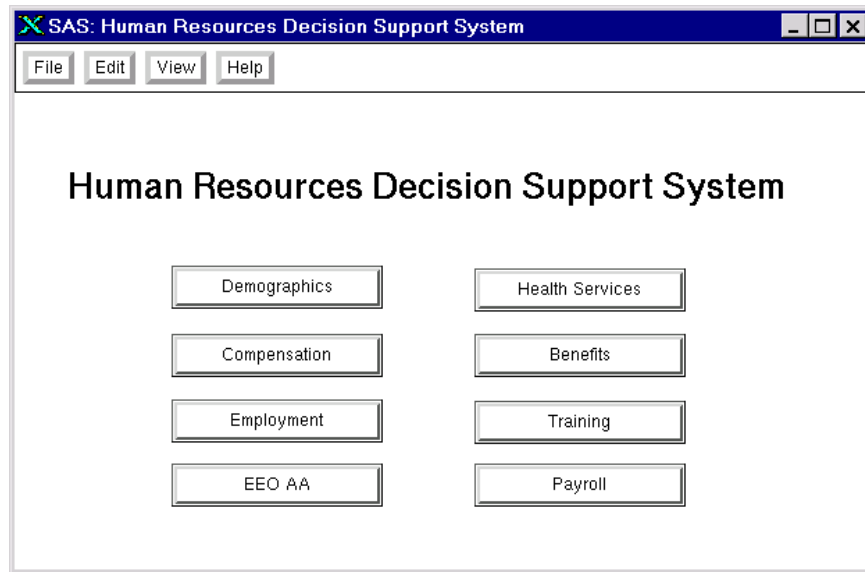
For detailed information about working with frames, see the SAS/AF online Help.

Frame Types

You can create two types of frames: standard or dialog. Standard frames define an application's primary window or windows. Primary windows enable you to access secondary windows that are needed to enter, edit, collect, or display information. For example, a primary window of a word processor might include all the menus or buttons used to access the commands or other windows that are associated with the program.

The following application main menu is an example of a typical frame that was created using SAS/AF software.

Display 4.1 Example of a Standard Frame



A dialog frame is used to create secondary windows such as the dialog boxes, palette windows, and message boxes that are accessed from a primary window. Secondary windows are useful for specifying parameters or options, for displaying error or information messages, and for providing a place to collect user input. Dialog frames cannot be resized by users.

Display 4.2 Example of a Dialog Frame



In your SCL programs, you can use the DIALOG function to open a frame as a modal window. For more information, see [“Calling Dialog Boxes from an Application”](#) on page 45.

Note: All new frames are of type standard by default.

To specify whether a frame should be of type standard or dialog, set the frame's **type** attribute in the Properties window.

Frame Usage Tips

The following table provides general information about working with frames. For more information, see the SAS/AF online Help.

Table 4.1 Common Frame Tasks

Task	Solution
Setting frame properties	Use the Properties window and select the <code>_FRAME_</code> object.
Adding components to a frame	Drag and drop components onto a frame from the Components window. Or, select a component in the Components window, and then double-click in the frame where you want to place the component.
Positioning frame components	Use the mouse to drag and position components. Use the Properties window to set component attributes that specify the component's position on a frame.
Adding a menu to a frame	Use the Properties window to set the frame's pmenuEntry attribute. You can create a new menu with the PMENU procedure.
Storing a frame	Save the frame to a specific SAS catalog by selecting Save or Save as from the File menu. Frames are stored in FRAME catalog entries.

Specifying the Frame SCL Entry

Frame SCL entries provide the SCL source code to control a frame. The name of the SCL entry that controls a FRAME entry is assigned through the frame's **SCLEntry** attribute. By default, the **SCLEntry** attribute identifies the frame's SCL source as `*.SCL`, where the asterisk (*) represents the name of the FRAME entry.

For more information about frame SCL, see [“Adding SAS Component Language Programs to Frames” on page 35](#).

Compiling Frames

If your frame has an associated frame SCL entry, you must compile the frame in order to associate the frame SCL with your frame.

To compile a frame (and its frame SCL entry), select **Build** ⇌ **Compile** from the Frame entry's Build window.

For more information about compiling frames, see [“Compiling and Testing Applications” on page 46](#).

Selecting Components

A well-designed user interface is based on a development process that focuses on users and their tasks.

The components that you choose to place on a frame are determined by the tasks that you want your users to perform. The following table describes common actions that your applications might require for users to complete specific tasks. It also suggests a SAS/AF component that you could use.

In some cases, multiple components are listed because more than one component might suit your needs. For more information about any of the components, see the “SAS/AF Component Reference” in the SAS/AF online Help.

Table 4.2 *User Tasks and Components*

If you want to...	then use a...
enable a user to indicate a yes/no, true/false, or on/off option	checkbox control or radio box control
enable a user to make choices	checkbox control combo box control list box control spin box control
visually group multiple components together	container box control
display an existing icon on your frame	desktop icon control or push button control
display an existing graph on your frame	graph output control
enable a user to execute a command	push button control
enable a user to choose only one item in a group of possible selections	radio box control
enable a user to enter a limited amount of text	text entry control
enable a user to enter a large amount of text	text pad control
provide a label for other components or include an uneditable line of text on your frame	text label control
include a chart or graph on your frame	chart control critical success factor control histogram control map control pie control scatter control

If you want to...	then use a...
provide a list of available color choices*	color list model
retrieve or access information about SAS software files and storage utilities*	catalog entry list model catalog list model data set list model library list model SAS file list model variable list model variable values list model
browse or edit data	form viewer control table viewer control SAS data set model
access information about file and storage utilities that were not created by or are not part of SAS software*	external file list model
access the items contained in a list*	LIST entry list model SLIST entry list model

* If you want to display this information in a frame, then you must use a view (such as a list box) to show the model. Models are referred to as non-visual components. They provide data that is typically used along with a visual control. For more information about models and views, see [“Model/View Communication”](#) on page 29.

Re-using Components

Combining Components to Create Composites

Composites are custom components that consist of at least two existing components. You should consider creating a composite if you find that you frequently place the same few components together on different frames to accomplish a specific task. For example, if you often enable your users to type in a file path value or select that file path by clicking a browse button, you might consider creating a composite that includes a text field and a browse button.

If you often enable users to make a color selection from a list box, you might consider creating a composite that includes a list box and a color list.

You can save a composite if you think you might want to use it again (either in the current application or in another application). The process of saving a composite actually prompts you to create a new class. To use the composite again, you would create an instance of the new composite class.

Composites can be a combination of

- two or more visual controls, such as a check box control and a container box control

- a visual control, such as a list box control, and a non-visual component, such as a color list model

For step-by-step instructions on creating a composite, see “Creating Composite Controls” in the SAS/AF online Help.

Subclassing

If you find that you consistently modify an existing component for use in one or many applications, you might want to make a subclass of that component's class. For example, if you want the default text in a push button control to be **OK** instead of **Button**, then you could subclass the push button control.

For more information about subclassing, see “Creating Your Own Components” on page 95.

Writing Methods

You can write new methods that override or are added to an existing class. The method writing process enables you to reuse an existing class by modifying it slightly for your current needs.

For more information, see “Managing Methods” on page 109.

Defining Attachments to Enable Automatic Component Resizing

An attachment is a connection between two or more components that controls the placement or size of those components with respect to each other and to the frame itself.

Attachments enable you to define the spatial relationships between components and/or between components and frames, as well as to provide support for automatic component resizing. SAS software applications that have attachments automatically adjust component sizing and spacing when

- a user resizes the window that contains the component(s).
- the window that contains the component(s) is a different size at initialization time than it was when the FRAME entry was saved.
- the application is ported to an environment that has different window sizes.
- the application is displayed on a device that has a different resolution than the device on which it was developed.

Note: Attachments are not required, but they should be used to ensure accurate component sizing.

To define attachments, follow these steps:

1. Select a component or components.
2. Select the appropriate attachment mode. To do this, select **Layout** ⇒ **Attach** ⇒ **Attach Current Region** or **Layout** ⇒ **Attach** ⇒ **Attach Child Region**.
3. Initiate define attachment mode. To do this, select **Layout** ⇒ **Attach** ⇒ **Define Attachment**.
4. Select the attachment direction and type from the Define Attachments window.

5. Drag out your attachments between components or between components and the frame.
6. Click **OK** in the Define Attachments window. This ends define attachment mode and saves your attachments.

For more information about attachments, see [“Adding Attachments to Frame Controls” on page 179](#).

Chapter 5

Communicating with Components

Introduction	27
Attribute Linking	28
Introduction	28
Determining When to Use Attribute Linking	28
Establishing an Attribute Link	28
Example	29
Model/View Communication	29
Introduction	29
Determining When to Use Model/View Communication	30
Examples	31
Drag and Drop Communication	32
Introduction	32
Determining When to Use Drag and Drop Communication	32
Which Components Support Drag and Drop?	32
Defining Drag and Drop Sites	32
Tips for Defining Drag and Drop Sites	33
Example	33

Introduction

The components in your applications must have a mechanism through which they can communicate with each other. For example, if a frame contains a list box along with another object whose contents are updated based on an item that is selected from the list box, the list box must communicate to the other object that an item was selected and which item was selected.

The SAS Component Object Model (SCOM) provides this communication, and makes it possible for components to communicate with each other without you adding any programming code.

As an applications developer, you can take advantage of SCOM with

- attribute linking
- model/view communication
- drag and drop communication

Attribute Linking

Introduction

Attribute linking enables components to interact without the need for any SAS Component Language (SCL) code. Instead, interactions are specified with the Properties window.

Attribute linking involves setting the *Link To* values of a component's attributes. For example, if you want the text value that is entered into a text field to update a graph output control, link the graph output control's **graph** attribute to the text entry control's **text** attribute. You can define attribute links between attributes on the same component or between different components on the same frame.

Determining When to Use Attribute Linking

Use attribute linking when you

- want one component to access the value of another component without any SCL code.
- want to maintain consistency between components. For example, if you change the background color of a push button, you may want every push button on the frame to use the new background color.
- define component communication for objects at build time.

Establishing an Attribute Link

To establish an attribute link between two attributes, you define the link on the attribute that you want to receive the new value. Attribute linking includes the following steps:

1. Identify the component whose attribute or behavior you want to dynamically change. For example, you may want to change whether a text entry control is enabled or disabled.
2. In the Properties window Attribute table, find the component attribute that you want to change. In the case of our text entry control, it would be the **enabled** attribute.
3. Define the link on that attribute by specifying a value in its **Link To** cell. For example, you might link the text entry control's **enabled** attribute to the **selected** attribute of a check box control that is also present on the frame.

An attribute link consists of a *componentName/attributeName* pair where

componentName

refers to the component that owns the above attribute

attributeName

refers to the attribute that contains the value that you want your linked attribute to contain

Example

Suppose your application prompts a user to choose whether or not they receive e-mail messages. Your frame might include a check box labeled “Do you have an e-mail account?” If the user clicks the check box, then a text label and text entry field are enabled so that the user can enter an e-mail address.

You can complete this type of component communication with multiple attribute links. The following steps detail the process:

1. Create a new FRAME entry.
2. Place a check box control on the frame and change its `label` attribute to “Do you have an e-mail account?”

You can use the Properties window to change the `label` attribute.

3. Place a text label and a text entry control on the frame.

The text label should be positioned on the left of the text entry and should be labeled “e-mail address:”

4. Link the `enabled` attribute of the text entry control to the `selected` attribute of the check box.
5. Link the `enabled` attribute of the text label object to the `enabled` attribute of the text entry object.
6. Test the frame by selecting **Build** ⇨ **Test**.

Note: This example is used to describe attribute linking only. Other tasks would be necessary to complete this frame and make it function properly within an application.

When you select the check box, the text label and text entry objects become available. When you deselect the check box, the text label and text entry objects become grayed and are therefore unavailable.



Model/View Communication

Introduction

Components are often separated into two categories: those that display data (viewers) and those that provide access to data (models). These different kinds of components

break the complex process of displaying data into logical parts. See “[Models and Viewers](#)” on page 76 for more information about model/view architecture.

SAS/AF software provides an easy way for you to add model components and viewer components to your applications. You can set model/view component communication during application build time (that is, within the Build window) by

- dragging a model onto a viewer
- setting the `model` attribute of a viewer in the Properties window

For step-by-step information, see “Assigning Models to Viewers” in the SAS/AF online Help.

Determining When to Use Model/View Communication

Use model/view communication when you want to attach a non-visual component (model) to a visual component (viewer) to display specific data.

The following table lists the default models and viewers that SAS/AF software provides. It also shows which models and viewers can be used together. Within the corresponding sections of the table, any model on the left can be used with any viewer on the right. The appropriate model/view combination depends on your final goal.

Table 5.1 Models and Viewers

Use any of these models...	with any of these viewers...
Catalog Entry List	Combo Box
Catalog List	List Box
Color List	Radio Box
Data Set List	Spin Box
External File List	
Library List	
LIST Entry List	
SAS File List	
SLIST Entry List	
Variable List	
Variable Values List	
SAS Data Set	Form Viewer
SCL List	Table Viewer

Note: You can create and/or customize models and viewers within a SAS/AF application if the models and viewers that are provided do not meet the needs of your application. For information on creating a new model or viewer, see “[Implementing Model/View Communication](#)” on page 133. In addition, if you plan to use a model/view pair regularly, you might want to create a composite. For more information on creating a composite, see “Creating Composite Controls” in the SAS/AF online Help.

Examples

You can use model/view communication to display a list of color choices in a list box. Additionally, you can set this model/view component communication by dragging and dropping components, or by using the Properties window.

To set model/view communication by dragging and dropping components:

1. Create or open a FRAME entry.
2. Drag a List Box control from the Components window and drop it onto the frame.
3. Drag the Color List model from the Components window and drop it onto the List Box.

To set model/view communication with the Properties window:

1. Create or open a FRAME entry.
2. Drag a List Box control from the Components window and drop it onto the frame.
3. Drag a Color List model from the Components window and drop it onto the frame.
4. Open the Properties window.
5. In the Properties window, open the Attributes table for the List Box control.
6. Set the `model` attribute value to the Color List model.

The above examples show that some models can return information at build time. The list box that serves as the viewer is automatically populated with values when the model is attached.

The following example shows that other models return information only at run time. In this situation, the model information is not displayed in the viewer during build time.

1. Create or open a FRAME entry.
2. Place a Combo Box control and a Catalog List model on the frame.
3. Open the Properties window.
4. In the Properties window, open the Attributes table for the Combo Box control.
5. Set the `model` attribute value to the Catalog List model.
6. In the Properties window, open the Attributes table for the Catalog List model.
7. Set the `library` attribute value to an existing library.
8. Select **Build** ⇌ **Test** to run the frame and then select the down arrow in the combo box to see a catalog list.

Drag and Drop Communication

Introduction

Drag and drop communication involves dragging components or objects with a mouse (or other pointing device) and dropping them over other objects *at run time* to perform an action.

A component that can be dragged is called a *drag site*, and a component that can receive the dragged object is called a *drop site*. The data that is transferred between the components is referred to as the *dragInfo* or the *dropInfo*.

For step-by-step information on setting drag and drop component communication, see “Enabling Drag and Drop Functionality” in the SAS/AF online Help.

Determining When to Use Drag and Drop Communication

Use drag and drop communication when

- you want users to be able to transfer data from one component to another by interacting with a frame's components during application run time.
- you cannot easily use attribute linking to transfer data from one component to another during application run time.

Drag and drop functionality varies between different operating systems, and some environments do not support it. If you develop an application that uses drag and drop communication, you may want to provide an alternative process so that the action can be performed in all environments.

Which Components Support Drag and Drop?

Most of the components in `sashelp.fsp.AFComponents.resource` have default drag and drop attribute settings. Exceptions include

- Container Box control
- Critical Success Factor control
- Map control

If you want to change the default drag and drop attribute settings, use the Properties window.

Defining Drag and Drop Sites

Drag and drop sites can be defined for any component that is a subclass of the Frame class or the Widget class, including all visual controls.

In many cases, SAS has already defined components as drag and/or drop sites. You can use those components to perform drag and drop component communication without having to modify them.

To define a component to function as a drag site:

1. Set the **dragEnabled** attribute to **Yes** if you want the component to work as a drag site by default. In most cases, setting this attribute is all you need to do to define a drag site.
2. Specify a data representation and attribute to be passed for the drag site by setting the **dragInfo** attribute.
3. Specify a drag and drop operation for the drag site by setting the **dragOperations** attribute.

To define a component to function as a drop site:

1. Set the **dropEnabled** attribute to **Yes** if you want the component to work as a drop site by default. In most cases, setting this attribute is all you need to do to define a drop site.
2. Specify a data representation and attribute to be passed for the drop site by setting the **dropInfo** attribute.
3. Specify a drag and drop operation for the drop site by setting the **dropOperations** attribute.

Tips for Defining Drag and Drop Sites

When you define drag and drop sites, keep the following tips in mind:

- Any subclass of the Widget or Frame class supports drag and drop component communication.
- Data from one component can be dragged to another component.
- The drop sites and drag sites do not need to reside in the same window.
- Components can act as both drag sites and drop sites.
- Drag sites cannot be dragged outside the SAS software environment unless they have a data representation of `_DND_TXT`.
- Default drag and drop behavior may vary according to the host operating environment on which your application executes.

Example

Use drag and drop communication to cause a graph to appear when you drag a specific graph entry from a list box and drop it onto a Graph Output control.

1. Create or open a FRAME entry.
2. Place a List Box control and a Graph Output control on the frame.
3. Drag and drop the Catalog Entry List model onto the List Box control.
4. Use the Properties window to set the Catalog Entry List model's **catalog** attribute to the name of a catalog that contains graphic entries (for example, SASHELP.EISGRPH). (Optional) If you want the list box to show only one particular type of entry, set the Catalog Entry List model's **typeFilter** attribute to the appropriate entry type.
5. Use the Properties window to set the Graph Output control's **borderStyle** attribute to **Simple**.

6. Use the Properties window to set the List Box control's **dragEnabled** attribute to **Yes** and the Graph Output control's **dropEnabled** attribute to **Yes**.
7. At run time, drag a graphic entry from the list box and drop it onto the Graph Output control. The selected graphic appears in the Graph Output control.

Chapter 6

Adding SAS Component Language Programs to Frames

Introduction	35
Working with Frames and SCL	36
When Frame SCL Is Not Required	37
When Frame SCL Is Required	37
Constructing a Frame SCL Program	37
Overview	37
SCL Labeled Sections	38
SCL Statements	38
SCL Routines and Functions	39
SCL Variables	39
Dot Notation and SCL	40
Controlling the Execution of SCL Programs	41
Introduction	41
Processing Custom Commands in SCL Programs	42
Calling Other Entries and Opening Windows	44
Introduction	44
Opening Other Windows	44
Calling Dialog Boxes from an Application	45
Compiling and Testing Applications	46
Introduction	46
Compiling FRAME Entries in Batch	46
Compiling FRAME Entries Automatically	47
Testing Your Application	47
Debugging and Optimizing Your Applications	47
Introduction	47
Common SCL Debugger Commands	47
Optimizing the Performance of SCL Code	48
Saving and Storing Frame SCL Programs	49

Introduction

SAS Component Language (SCL) is an object-oriented programming language that is designed to facilitate the development of interactive SAS applications. In SAS/AF software, a frame is controlled primarily by an SCL program, although it is possible to

have a fully functional frame that has no associated SCL code. You can add code to a FRAME entry's SCL program to

- validate user input
- calculate field values that are based on user input
- change attribute values of components on a frame at run time
- invoke methods of components on a frame
- define custom error handling and messaging
- provide special user interface features such as menus, graphs, selection lists, font lists, and system information
- link to other SAS catalog entries, including other frames
- submit SAS programs
- read from and write to SAS tables, SAS catalog entries, and external files
- interact with other SAS software as well as software from other vendors

For complete reference information on SCL, refer to the *SAS Component Language: Reference*. For task-oriented information on working with SCL programs, see the SAS/AF online Help.

Working with Frames and SCL

There are several important items to remember when you are working with SCL programs for FRAME entries:

- The SCL for a frame (often referred to as *frame SCL*) is stored in a separate catalog entry of type SCL. If you do not store the SCL entry in the same catalog as the FRAME entry, then the frame's **SCLEntry** attribute must specify the four-level name of the associated SCL entry. See “[Saving and Storing Frame SCL Programs](#)” on page 49 for details.
- The name of the SCL entry that controls a FRAME entry is assigned through the frame's **SCLEntry** attribute. By default, the **SCLEntry** attribute identifies the frame's SCL source as *.SCL, where the asterisk (*) represents the name of the FRAME entry. For example, if your FRAME entry is named MENU.FRAME, specifying *.SCL as the name of the SCL entry identifies MENU.SCL. Thus, the *.SCL designates an SCL entry that has the same name as the FRAME entry and is in the same catalog.

To check the value of the **SCLEntry** attribute, open the Properties window, select the object named **_FRAME_** in the Properties tree, and then scroll down to the **SCLEntry** attribute.

- You can add or edit the SCL program for a frame in the Source window. When a frame is displayed in the Build window, select **View** ⇨ **Frame SCL** or select **Frame SCL** from the Build window's pop-up menu to open the frame's SCL program in the Source window.
- SCL entries must be compiled with their associated FRAME entries before you can test or run them.
- SCL source code is reusable, even if it is specified for a FRAME entry. Since SCL source is stored separately from FRAME entries, you can use the same SCL source for several FRAME entries without having to duplicate the SCL source for each one.

For example, you can develop the prototype FRAME entries MYREPORT1.FRAME, MYREPORT2.FRAME, and MYREPORT3.FRAME. Then, create the SCL source entry MYREPORT.SCL, and identify it as the SCL source entry for each of the three FRAME entries by setting each frame's **SCLEntry** attribute to MYREPORT.SCL in the Properties window.

When Frame SCL Is Not Required

A frame does not require an SCL program. Many components that you can add to a frame are designed to perform tasks without additional SCL code. For example, you can add a push button control to a frame and set its **commandOnClick** attribute to **end**. The END command then executes when a user clicks the push button. You do not need SCL to control this control's behavior.

Frame SCL entries do not have to control every component on a frame. When you create your own components (or subclass those provided by SAS software), you can add methods to perform operations. You implement these methods in an SCL entry that is separate from the SCL entries that are used by your frames. See [“Implementing Methods with SCL” on page 109](#) for more information.

In addition, communication between components in your application is possible without frame SCL. See [“Communicating with Components” on page 27](#) for more information.

When Frame SCL Is Required

You must include an SCL program for a frame in these situations:

- You want to use methods for a component on the frame.
- You need to modify a component's properties at run time. For example, values that are passed to the frame may be applied to change the attribute of a component, or you may change the appearance of a control after a user enters input.
- You need to perform conditional or custom processing for the selections that a user makes or for values that a user enters.

Constructing a Frame SCL Program

Overview

A typical SCL program for a frame consists of

- labeled sections
- statements
- routines and functions
- SCL variables

This section describes how you can combine these elements to create the programs that control SAS/AF frames.

SCL Labeled Sections

A section in SCL begins with a label and ends with a RETURN statement. SCL programs for FRAME entries use reserved sections to process program initialization, main processing, and termination.

```
INIT:
  /* ...statements to initialize the application... */
return;

MAIN:
  /* ...statements to process user input... */
return;

TERM:
  /* ...statements to terminate the application... */
return;
```

In addition, you can add sections labeled with object names for each component that you add to the frame. The labeled section for a component executes when an action is performed on the component, such as selecting the component or changing its value. For example, if you have a push button named `okButton` on your frame, you could have the following labeled section in your SCL:

```
okButton:
  dcl list message={'Are you sure you want to exit?'};
  response=messagebox(message, '!', 'YN', 'Confirm Exit','N',");
  rc=dellist(message);
return;
```

The code in the `okButton` section automatically executes when the push button is selected.

In general,

- INIT executes once before the frame is displayed to the user.
- MAIN executes immediately after the corresponding object-label section has executed. If a corresponding object-label section does not exist, then MAIN executes each time a user activates any object on the frame.
- TERM executes when either your program or the user issues an END command or a CANCEL command.

For a detailed explanation of how labeled sections are processed, see [“How SCL Programs Execute for FRAME Entries”](#) on page 163.

SCL Statements

The labeled sections in frame SCL programs consist of one or more executable statements that control program flow. (Declarative statements, such as DECLARE or ARRAY, can occur anywhere in the program and do not have to exist in labeled sections.)

You can also use the following SAS language statements in SCL programs:

- LENGTH and ARRAY statements
- assignment statements that use standard arithmetic, comparison, logical, and concatenation operators

- IF-THEN/ELSE and SELECT statements
- DO groups and all forms of DO loops
- LINK and RETURN statements
- comment indicators, including `/* comment */` and `* comment;`

For details on SCL statements, refer to the *SAS Component Language: Reference*.

SCL Routines and Functions

SCL provides a rich set of routines and functions that perform some action, such as the DISPLAY routine or the OPENSASFILEDIALOG function. For detailed information about these functions, refer to the *SAS Component Language: Reference*.

In addition, SCL supports nearly all of the functions of the Base SAS language. For details on the Base SAS functions, see SAS Language Reference: Dictionary.

SCL Variables

Each variable used in an SCL entry represents a specific data type. SCL supports the following data types:

Character

declared with the keyword CHAR

Numeric

declared with the keyword NUM

SCL List

declared with the keyword LIST

Object

declared with the keyword OBJECT or with a specific four-level class name

SCL provides several automatically defined system variables, such as

- `_FRAME_`, a generic object that contains the identifier of the current active frame
- `_MSG_`, a character variable that enables you to set the text to display in the frame's message area
- `_STATUS_`, a character variable that you can query to see whether a user has cancelled or ended from a frame so that your application can either halt execution immediately or resume processing.

See the *SAS Component Language: Reference* for a complete list of system variables and their data types.

Automatic system variables like `_FRAME_` and `_STATUS_` are declared internally, so you can use them without having to declare them in your SCL. Objects created on a frame are also declared automatically, and as long as you compile the frame's SCL entry from the frame, the SCL compiler recognizes them.

All other variables must be defined using the DECLARE statement. Consider the following code:

```
DECLARE NUM
  n1 n2,
        NUM          n3 [15],
        CHAR          c1,
        CHAR(10)      c2,
```

```

LIST      myList = {};
OBJECT    obj1 obj2 [5],
classname obj2;

```

- The DECLARE keyword begins the variable declaration statement. You can also use the abbreviation DCL.
- NUM, CHAR, LIST, and OBJECT are reserved keywords that indicate the data type of the variables that follow the keyword.
- The variable declared as `n3 [15]` defines a 15-item array of numeric values.
- CHAR(n) is a notation that enables you to define the length of a character variable, where n, a value up to 32767, is its length. By default, character variables are 200 characters long. In the declaration above, `c1` is a character variable with a length of 200, and `c2` is a character variable with a length of 10.
- OBJECT indicates that the variable contains an object identifier, which enables the SCL compiler to recognize dot notation for method calls or attribute references. The variable declared as `objs [5]` is a five-item array of object identifiers.
- `classname` indicates that the variable contains the object identifier of a specific class such as `sashelp.classes.cataloglist_c.class`.

Dot Notation and SCL

SCL provides dot notation for direct access to component properties. Dot notation is intended to save time, improve code readability, and reduce the amount of coding necessary. It also improves the performance of your applications by providing additional compile-time validation. For example, you can use it to check the method's argument type. You can also use it to execute methods and to access component attributes without having to use SEND or NOTIFY routines.

You can use dot notation in any of the following forms:

```

object.method(<arguments>);
return-value=object.method(<arguments>);
object.attribute=value;
value=object.attribute;
if (object.method(<arguments>)) then ...
if (object.attribute) then ...

```

where

object

specifies an object or an automatic system variable (such as `_CFRAME_`, `_FRAME_`, or `_SELF_`) whose type is object.

method

specifies the name of the method to execute.

arguments

specifies one or more arguments based on the appropriate method signature.

return value

specifies the value returned by a method (if any).

attribute

specifies an attribute that exists on *object*.

value

specifies a value assigned to or queried from the attribute.

Dot notation can be used to set or query attributes. For example, suppose object1 is a text entry control:

```
/*Setting the text color */
  object1.textColor='yellow';

/* Querying the text color.           */
/* object1's textColor attribute is returned */
/* to the local SCL variable 'color'     */
  color = object1.textColor;
```

Dot notation also is used to call methods. For example,

```
object._cursor();
```

is equivalent to

```
call send(object, '_cursor');
```

For compatibility with legacy programs, you can continue to use CALL SEND and CALL NOTIFY to invoke methods.

You can use dot notation to call methods on objects in several ways, provided the method contains the appropriate signature information. For example,

```
/* With a return variable */
  cost=object1.getPrice(itemnum);

/* Without a return variable */
  object1.getPrice(itemnum, cost);

/* In an IF statement */
  if object1.getPrice(itemnum) > 100 then
do...
```

Note: Objects that are created on a frame and automatic system variables (such as `_FRAME_`) are automatically declared as objects. You can refer to these objects in dot notation without having to declare them in your SCL programs. To use dot notation with legacy objects that you add to a frame, you must first select “Use object name as ID in SCL” for those objects in the Properties window.

For more information about dot notation, refer to the *SAS Component Language: Reference* and the SAS/AF online Help.

Controlling the Execution of SCL Programs

Introduction

There are several ways to control application execution in an SCL program for a frame. You can

- conditionally change the program flow, or branch to other sections of SCL
- submit SAS and SQL statements (see the *SAS Component Language: Reference* for details)
- use the SYSTEM function to issue a host operating system command that performs system-specific data management tasks or invokes an application other than SAS

- use the CONTROL statement with the ALWAYS option or the ENTER option to process custom commands

You can use conditional SCL statements to alter the flow of a program. For example, to conditionally return control to the window:

```
object1:
  /* ...SCL statements... */
  if condition
    then return;
  /* ...SCL statements... */
return;
```

You can use the LINK statement to branch to a common labeled section in the same SCL program. For example:

```
object1:
  /* ...SCL statements... */
  link computeValue;
  /* ...SCL statements... */
return;

object2:
  /* ...SCL statements... */
  link computeValue;
return;

computeValue:
  /* ...SCL statements... */
return;
```

The labeled section computeValue may or may not correspond to an object of the same name. Labeled sections are not required for all objects on a frame; likewise, a labeled section can exist with or without an associated frame object. The computeValue section executes each time

- object1 is activated or modified
- object2 is activated or modified
- an object named computeValue is activated or modified (if it exists on the frame)

For additional information on controlling application flow with SCL, refer to the topics on submitting SAS statements and using macro variables in the *SAS Component Language: Reference*.

Processing Custom Commands in SCL Programs

You can add custom command processing to your frame SCL programs to control program flow. A custom command can be any name that does not correspond to an AF window command or SAS global command and that you implement in the MAIN section of your frame SCL program. A user issues commands by typing on a command line, by pressing a function key with a defined command, or by selecting items on the frame, including menu selections or commands that are associated with visual controls.

To implement a custom command, you must

- select a unique command name (that is, one that differs from all AF window commands and SAS global commands).

- add a CONTROL statement to change the default behavior for command processing during SCL program execution. Alternatively, you can set the frame's `commandProcessing` attribute to `Run main`.
- add code to the MAIN section of the frame's SCL program to read the custom command from the command line, process the custom command, and prevent the SAS command processor from evaluating the custom command.

The CONTROL statement controls the execution of labeled sections in an SCL program. Typically, you add this statement to the INIT section of your program.

- Use CONTROL ALLCMDS to execute the MAIN section when a procedure-specific or custom command is issued, or when a user presses the ENTER key, even if an error flag is in effect for an object on the frame.
- Use CONTROL ALWAYS to execute the MAIN section when a custom command is issued or when a user presses the ENTER key, even if an error flag is in effect for an object on the frame.
- Use CONTROL ENTER to execute the MAIN section when a custom command is issued or when a user presses the ENTER key, unless an error flag is in effect for an object on the frame.

You can use the WORD function to return the command for processing, using the syntax

```
command=word(n <, case>);
```

where *command* is the word that is currently in the command buffer, *n* is the position (either 1, 2, or 3) of the word in the command, and *case* is an optional parameter that converts the word to either uppercase ('U') or lowercase ('L').

You can use the NEXTCMD routine to remove the current command from the command buffer, using the syntax

```
call nextcmd();
```

If you do not remove a custom command from the command buffer, SAS issues an error message.

For example, consider a frame that contains a toolbar object on which each button is set to issue a different command. These commands can be a mixture of valid SAS commands or custom commands that you implement in the MAIN section of the frame's SCL program:

```

decl char(8) command;
  INIT:
    control always;
    return;
  MAIN:
    command=word(1, 'u');
    select command;
      when ('CUSTOM1')
        /* ...code to process the CUSTOM1 command... */
        call nextcmd();
      when ('CUSTOM2')
        /* ...code to process the CUSTOM2 command... */
        call nextcmd();
      otherwise;
    end;
  return;

```

Calling Other Entries and Opening Windows

Introduction

You can use SCL to access other entries that you may need for your application, such as frames and other SCL programs. You can use the DISPLAY routine to invoke another SAS catalog entry, including FRAME, SCL, CBT, and PROGRAM entries. For example,

```
call display('sasuser.mycat.myApp.scl');
```

invokes the SCL entry named myApp. The calling program transfers control to myApp and waits for it to finish processing.

SCL also enables you to pass parameters between different entry types. In general, any argument to an SCL function, routine, or method can be

- a constant

```
call display('myFrame.frame', 2);
```

- an SCL variable

```
call display('myApp.scl', myTable);
```

- an object attribute

```
call display('dlg.frame', listBox1.selectedItem);
```

Character constants, numeric constants, and expressions are passed by value. You can use any SCL variable to pass parameters by reference to another entry. To use the DISPLAY function with a return value, you must include an ENTRY statement in the frame SCL of the called FRAME entry. For example, consider the following SCL code:

```
validUser=DISPLAY('password.frame', userid);
```

If PASSWORD.FRAME contained a text entry control named **userPassword** in which a user could provide a valid password, the frame's SCL could contain

```
/* FRAME SCL for mylib.mycat.password.frame */
entry userid:input:num return=num;
term:
  dcl char pwd;
  dcl num isValid;
  /* Assume that the user ID is validated in some */
  /* way to establish a value for the password. */
  if userPassword.text = pwd
    then isValid=1;
    else isValid=2;
return(isValid);
```

For details on passing parameters to other SAS catalog entries, see the ENTRY statement in the *SAS Component Language: Reference*.

Opening Other Windows

Your applications can consist of any number of windows and dialog boxes. Using the DISPLAY routine, you can open other windows in your application as needed.

For example, assume that a frame contains three push button controls named **Rates**, **Lenders**, and **Recalculate**. The SCL entry for that frame can call a FRAME entry named LOANRATES.FRAME when a user clicks the **Rates** button, and it can call a FRAME entry named LENDERINFO.FRAME when a user clicks **Lenders**. The frame SCL also runs another SCL entry when a user clicks **Recalculate**.

```
RATES:
  call display('loanRates.frame');
return;

LENDERS:
  call display('lenderInfo.frame', 2, 'North', listBox1.selectedItem);
return;

RECALCULATE:
  call display('sasuser.myapp.calculate.scl');
return;
```

You can also invoke full-screen applications from a SAS/AF frame application. For example, the following SCL code enables a user to access the FSEDIT window and to browse the RECORDS.CLIENTS table after clicking **Clients** on the active frame:

```
CLIENTS:
  call fsedit('records.clients',",','browse');
return;
```

Calling Dialog Boxes from an Application

You can present information in SAS/AF applications through dialog boxes as well as through standard frames. Your SCL code can call dialog boxes using any of the following:

- the DIALOG routine or function

Much like CALL DISPLAY, the DIALOG function enables you to display a FRAME entry. DIALOG differs from CALL DISPLAY in that it displays the frame as a modal window, which disables all other SAS windows until the dialog frame is closed. For example:

```
CALL DIALOG('verify.frame');
```

or

```
validInput=DIALOG('validate.frame', some-variable);
```

To use the DIALOG function with a return value, you must include an ENTRY statement in the frame SCL of the called FRAME entry.

- the MESSAGEBOX function

You can use the MESSAGEBOX function to display a host message window for informational, warning, or error messages. For example:

```
VALIDATE:
  dcl list msgList={'SAS table not specified.'};
  dcl num rc;

  choice=MessageBox(msgList, '!', 'OC', 'Validation');
  if choice='CANCEL'
    then call execcmd('end');
```

```
rc=dellist(msgList);
return;
```

You can use the MESSAGEBOX function instead of the `_MSG_` automatic variable to display warnings.

- the SCL File or Entry Dialog functions

You can use the OPENSASFILEDIALOG and SAVESASFILEDIALOG functions to manipulate two-level SAS filenames just as you would use DIRLIST. Likewise, you can use the OPENENTRYDIALOG and SAVEENTRYDIALOG functions to manipulate SAS catalog entry names just as you would use CATLIST. For example,

```
selected=OpenEntryDialog('RANGE');
```

opens a catalog entry selector, displays available RANGE entries, and returns the user's selections.

For complete information on these functions, see the *SAS Component Language: Reference*.

Compiling and Testing Applications

Introduction

To compile your SCL program in the build environment, you can select **Build** ⇒ **Compile**. You can also enter the COMPILE command or click the Compile icon on the toolbar. If the SCL entry was opened from within the frame, you can issue the COMPILE command with either the FRAME entry or the SCL entry as the active window, and both entries will be compiled.

You should also consider the following when compiling frames and their SCL entries:

- If your FRAME entry uses an SCL program, you must compile the FRAME entry in order to associate the compiled SCL source with the FRAME entry.
- The compiler produces a list of errors, warnings, and notes in the Log window. Open the Log window by issuing the LOG command or by selecting **View** ⇒ **Log**.

Although you can compile an SCL program independently of its FRAME entry, the compiled code is associated with the FRAME entry only if you compile the SCL source either through the Source window that was opened from the frame or through the Build window that contains the FRAME entry.

You can also compile FRAME entries in batch or automatically.

Compiling FRAME Entries in Batch

You can use the BUILD procedure statement to compile in batch all of the FRAME entries in a catalog. However, you must use the ENTRYTYPE option to specify that the FRAME entries are to be compiled. When you use the COMPILE option without an ENTRYTYPE option, the BUILD procedure compiles only PROGRAM entries. For example, to compile all of the FRAME entries in the PAYROLL catalog (in the library that was assigned the libref DEPT), use the following statements:

```
proc build c=dept.payroll batch;
    compile et=frame;
run;
```


Compiling FRAME Entries Automatically

If you set a frame's `automaticCompile` attribute to `Yes` in the Properties window, the FRAME entry is compiled automatically when you issue the SAVE, END, or TESTAF command from that FRAME entry.

Testing Your Application

SAS/AF software provides a testing mode that is available from within the build environment. To test your application while your frame and SCL entry are open, you can select **Build** ⇒ **Test**. You can also click the TESTAF toolbar button or enter the TESTAF command.

As long as the SCL entry was opened from within the frame, you can issue the TESTAF command with either the FRAME entry or the SCL entry as the active window.

Note: You cannot use the TESTAF test mode if the FRAME entry is stored in a library that is accessed with SAS/SHARE software. Also, the TESTAF command does not process SUBMIT statements in your SCL code.

To test your application outside the build environment, you can do either of the following:

- Open the Explorer window, select the frame to test, and then select **Run** from the pop-up menu.
- Enter the following command:

```
af c=libref.catalog.catalogEntry.entryType
```

Testing outside the build environment has no restrictions or limitations, and frames perform with complete functionality.

Debugging and Optimizing Your Applications

Introduction

You can use the SCL Debugger to debug your frame SCL programs. The SCL Debugger is a powerful interactive utility that can monitor the execution of SCL programs. To use the debugger, issue the DEBUG ON command, select **Build** ⇒ **Debug** ⇒ **Debug On** or click the Debug toolbar button. After you turn on the Debugger, you must recompile your SCL.

See the *SAS Component Language: Reference* for details on using the SCL debugger.

Common SCL Debugger Commands

The following table describes some of the common SCL Debugger commands that you can use to track errors in your SCL programs:

Table 6.1 Common SCL Debugger Commands

Command	Description
BREAK	sets a breakpoint at a particular executable program statement. When the Debugger encounters a breakpoint, program execution is suspended and the cursor is placed at the DEBUG prompt in the Debugger's Message window.
DELETE	removes breakpoints, tracepoints, or watched variables that were previously set by the BREAK, TRACE, and WATCH commands.
DESCRIBE	displays the name, type, length, and class attributes of a variable.
EXAMINE	displays the value of a variable or an object attribute.
GO	continues program execution until a specified statement or until the next breakpoint or end of program is encountered.
JUMP	restarts program execution at a specified executable statement, which may skip intermediate statements.
LIST	displays all of the breakpoints, tracepoints, and watched variables that have been set by the BREAK, TRACE, and WATCH commands.
PUTLIST	displays the contents of an SCL list.
QUIT	terminates the Debugger.
STEP	steps through the program, statement by statement. By default, the ENTER key is set to STEP.
TRACE	sets a tracepoint. When a tracepoint is encountered, the Debugger displays relevant information in the Message window and continues processing.
WATCH	sets a watched variable. If the value of a watched variable is modified by the program, the Debugger suspends execution at the statement where the change occurred. The old and new values of the variable are displayed in the Message window.

Optimizing the Performance of SCL Code

You can optimize the performance of SCL programs in your applications with the SAS/AF SCL analysis tools.

The following table lists the available tools and provides information about when you might want to use each tool.

Table 6.2 SCL Tools

Use the...	when you want to...
Coverage Analyzer	monitor an executing application and access a report that shows which lines of SCL code are not executed during testing.

Use the...	when you want to...
List Diagnostic Utility	monitor an executing application and access reports on the use of SCL lists, including any lists that are left undeleted.
Performance Analyzer	monitor an executing application and identify any bottlenecks in an application's SCL code.
Static Analyzer	access reports that detail SCL code complexity; variable, function, method, and I/O usage; and error and warning messages.

To display a menu of the SCL analysis tools, enter **sclprof** at the command prompt. For detailed information about using these tools, refer to the SAS/AF online Help.

Saving and Storing Frame SCL Programs

There are several important items to remember when you are saving SCL programs for FRAME entries:

- If an SCL entry that is associated with a frame is saved from the Source window, then both the FRAME entry and the SCL entry are saved.
- Although you can compile an SCL program independently of its FRAME entry, the compiled code is stored with the FRAME entry only if you compile the SCL source either through the Source window or through the Build window that contains the FRAME entry. Compiling the FRAME entry always compiles the associated SCL entry.
- The SCL program is automatically compiled and saved when you save the FRAME entry if you set the frame's **automaticCompile** attribute to **Yes**.
- When you compile a FRAME entry, the FRAME entry's SCL source is compiled, and the compiled code is stored in the FRAME entry. Since the compiled SCL program is stored in the FRAME entry, the SCL entry that contains the source statements does not have to exist when you install and run the application.
- If you change the name of a frame that has an associated SCL entry, you must also remember to either
 - change the name of the SCL entry to match the FRAME entry if it is not ***.SCL**, and then recompile the frame; or
 - edit the frame's **SCLEntry** attribute to use the existing frame SCL entry.

Chapter 7

Adding Application Support Features

Implementing Custom Menus	51
Introduction	51
Using the PMENU Procedure to Create Menus	51
Adding Online Help to Your Applications	53
Introduction	53
Adding Help to a Frame	54
Attaching Context-Sensitive Help to Frame Components	54
Calling Help from Your Application	55

Implementing Custom Menus

Introduction

You can create custom menus for your SAS/AF applications using the Base SAS software PMENU procedure.

To add a menu to your application:

1. Create a PMENU catalog entry using PROC PMENU.
2. In the SAS/AF build environment, use the Properties window to set the appropriate frame's **pmenuEntry** attribute to the PMENU catalog entry that you want to display.
3. Set the frame's **forcePmenuOn** attribute to **Yes** and the frame's **bannerType** attribute to **None**.

For additional information about PMENU entries, see the SAS/AF online Help.

Using the PMENU Procedure to Create Menus

To create the structures and commands associated with your menu, you submit a source program that contains PMENU procedure statements. Because the source statements that are used to create a PMENU entry are not accessible in the PMENU catalog entry, you should save the statements in your development catalog as a SOURCE entry.

Consider the following example:

```
proc pmenu catalog=sasuser.corprpts;
  menu main;
```

```

item 'File' menu=menuFile mnemonic='F';
item 'View' menu=menuView mnemonic='V';
item 'Help' menu=menuHelp mnemonic='H';
menu menuFile;
  item 'Open Report' selection=fileOpen mnemonic='O';
  item 'Save Report' selection=fileSave mnemonic='S';
  separator;
  item 'End';
  selection fileOpen 'openrpt';
  selection fileSave 'saverpt';
menu menuView;
  item 'View Table' selection=viewTbl mnemonic='T';
  item 'Options' selection=viewOpts mnemonic='p';
  selection viewTbl 'afa c=mylib.mycat.view.frame';
  selection viewOpts 'afa c=mylib.mycat.options.frame';
menu menuHelp;
  item 'Contents' selection=helpCont mnemonic='C';
  item 'About...' selection=helpAbt mnemonic='A';
  selection helpCont 'wbrowse "http://myintranet.com/help.htm"';
  selection helpAbt 'afa c=mylib.mycat.about.frame';
run;
quit;

```

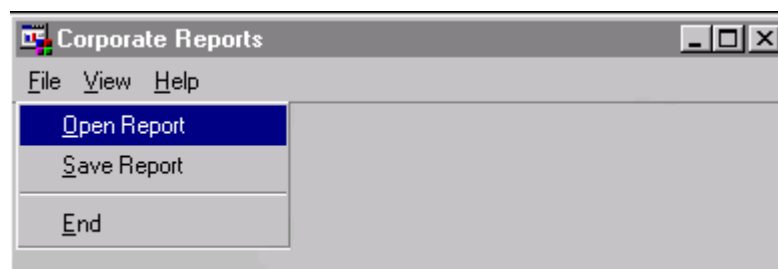
In this example,

- The menu is stored in the catalog named in the PROC PMENU statement.
- The first menu statement names the PMENU catalog entry that will contain the menu, which in this example is **sasuser.corprpts.main.pmenu**.
- Each ITEM statement specifies an item in the menu bar. The value of MENU= corresponds to a subsequent MENU statement.
- Each MENU= option is defined by a MENU statement, where each subsequent ITEM statement specifies the selections that are available under the named menu in the menu bar.
- A separator line can be added to menus by using the SEPARATOR statement.
- ITEM statements with no SELECTION= option indicate that the menu command is equivalent to the value of the ITEM. For example, **item 'End'** in the menuFile menu is the **End** command.
- Each SELECTION= option points to a corresponding SELECTION statement to define the command that is issued by the menu.

Note: To issue multiple commands, separate them with semicolons.

- Commands in a SELECTION statement that do not correspond to a valid SAS command must be implemented and trapped in the frame's SCL program.

The completed menu appears as follows:



In the PMENU example, the menuFile menu contains two SELECTION statements that issue the custom commands **openrpt** and **saverpt**. Your frame SCL program can process commands that are not valid SAS commands if

- the command name is not a valid AF window command or SAS global command
- your frame SCL includes a CONTROL ALWAYS or CONTROL ENTER option
- the MAIN section in your frame SCL performs the processing

For example, the SCL program for the frame associated with MAIN.PMENU could include the following code:

```
dcl char(8) command;
  INIT:
    control always;
    return;

  MAIN:
    command=lowercase(word(1));
    if command='openrpt' then do;
      /* ...SCL to open reports... */
      call nextcmd();
    end;
    else if command='saverpt' then do;
      /* ...SCL to save reports... */
      call nextcmd();
    end;
  return;
```

For complete information on the PMENU procedure, refer to the SAS Procedures Guide.

Adding Online Help to Your Applications

Introduction

You can add online Help to your applications to provide assistance to your users. SAS/AF software supports help in several formats, including HTML-based Web pages and SAS/AF CBT catalog entries. Your applications can also take advantage of context-sensitive help and tooltips.

When you implement a Help system for your application, you should consider

- how much detail you want to add to the help
- whether or not context-sensitive help is required (so that you can make the necessary changes to frame and component properties)
- what functionality the various help commands provide and how you can use them appropriately
- how your help files are structured and stored

Adding Help to a Frame

You can add help to a frame by setting its Help attributes. To display the attributes in the Help category, expand the **Attributes** tree in either the Properties window or the Class Editor and select the **Help** node. You can set the following attributes:

CBTFrameName

Set the CBT frame name that you want to display if you are using SAS/AF CBT entries to deliver your help information. This attribute is valid only if a CBT entry is assigned for the frame's **help** attribute.

help

Set the frame's **help** attribute to specify the type of help and the item to display when help is selected. If you select a SAS catalog entry as the type of help, SAS/AF assumes that you want to display a CBT entry. Other types of help provide different ways of displaying help in a browser. For details on the types of help, see the **help** attribute in the Frame class.

showContextHelp

Set this attribute to toggle the display of the question mark control button (?) in the window border of the frame. Set this attribute to **Yes** if

- your frame's **type** attribute is set to "Dialog"
- you have defined help for components on your frame
- you want the HELPMODE command to recognize context-sensitive help selections

Attaching Context-Sensitive Help to Frame Components

You can add context-sensitive help for components on a frame by using the "What's This" type of help for a component, for status-line messages, and/or for tooltips. You must set the appropriate attribute in the Properties window if you want to specify help for a component. You can set the following attributes:

CBTFrameName

Set the CBT frame name that you want to display if you are using SAS/AF CBT entries to deliver your help information. This attribute is valid only if a CBT entry is assigned for the component's **help** attribute.

help

Set the component's **help** attribute to specify the type of help and the item to display when help is selected. If you select a SAS catalog entry as the type of help, SAS/AF assumes that you want to display a CBT entry. Other types of help provide different ways of displaying help in a browser. For details on the types of help, see the **help** attribute in the Object class.

helpText

Set the component's **helpText** attribute to specify a message to display on the status bar when a user positions the cursor over the component.

toolTipText

Specify the text that you want to appear as a tooltip for the component. A tooltip (also known as "balloon help") is presented as pop-up text when a user positions the cursor over the component for a particular time interval.

Calling Help from Your Application

SAS/AF software provides several commands that you can use to display help for your application. You can add these help commands to a menu item, or you can assign one of them as the command that is executed by a control such as a push button. You can also attach the commands to icon buttons on a toolbar.

The commands include

HELP

The HELP command displays help for the active frame or window based on the values of the frame's Help attributes. The Help topic or topics are displayed in the designated help browser.

The WINDOWHELP command performs the same function as HELP.

HELPMODE

When the HELPMODE command is issued, SAS enters object help detection mode (helpmode). In helpmode, the cursor changes to a question mark (?) and SAS waits for the user to select an object on the active frame. When a user selects an object while in helpmode, SAS displays the help that is defined in that object's **help** attribute.

WBROWSE valid-URL

The WBROWSE command simply displays a valid URL, which can be any page that can be displayed by the associated Web browser.

The HELP command can be used in conjunction with the HELPLOC:// protocol to display a specific Help topic. The HELPLOC:// protocol is a mechanism defined by SAS for displaying help files from within your application. For example, **help helploc://myapp/intro.htm** opens the Help topic that is stored in the HTML file named **intro.htm**.

To specify the location of the help files for your application, you must add the appropriate path in the HELPLOC system option in the configuration file that is used to start your application.

Note: SAS/AF software uses the HELPLOC option to identify the path it searches to locate online Help files. The default path for SAS online Help is a format such as

```
!SASROOT/X11/native_help
```

or

```
!sasroot\core\help
```

If you modify or remove this path from the HELPLOC option in your software configuration file (instead of in your application's configuration file), SAS software cannot locate its associated online Help.

You can add multiple search paths to HELPLOC. The SAS Help facility replaces the HELPLOC:// protocol with a path listed in the HELPLOC option. The paths are searched in the order in which they are listed until the valid Help topic is found or until there are no further paths to search. For example, if your HELPLOC option contains

```
('!sasroot\core\help' '!sasuser\classdoc')
```

you could add a path to the directory in which you stored the help for your application:

```
('f:\apps\myapp\help' '!sasroot\core\help' '!sasuser\classdoc')
```

The SAS Help facility will then search through **f:\apps\myapp\help** to locate a Help topic before it searches the SAS online Help path (**!sasroot\core\help**).

A path specified in the HELPLOC option does not have to be a local directory. It can also be a path on a valid Web server or a path to a network directory. For example, consider a HELPLOC option that is defined as

```
('!sasroot\core\help' 'http://mycompany.com/intranet/apphelp')
```

and a help call from a SAS/AF frame that uses the form

```
help helploc://myapp/intro.htm
```

In this example, the SAS Help facility will first search for the requested Help topic in the SAS online Help path. Then it will send a fully qualified URL to the browser to locate the Help topic at

```
http://mycompany.com/intranet/apphelp/myapp/intro.htm
```

See [“Configuring a SAS Session for Your Application” on page 60](#) for information on customizing the configuration file.

Chapter 8

Deploying Your Applications

Application Deployment Issues	57
Migrating Your Application from Testing to Production	58
Introduction	58
Porting Your Application to Different Operating Environments	59
Configuring a SAS Session for Your Application	60
Introduction	60
Specifying a Custom Configuration File	61
Invoking Applications	62
Enabling the Application for Your Users	63

Application Deployment Issues

After fully testing your SAS/AF application, you can begin to implement and deploy it to your users. Deploying an application requires you to move from the development process of a prototype to assembling all of the parts to create a production application.

You should consider the following issues when planning to deploy a SAS/AF application:

- Is this a stand-alone application that will be installed on individual machines?
- Will this application be installed on a network, with many users accessing it simultaneously?
- Which SAS system options should be set to establish the desired application environment?
- Which windowing features should be in effect?
- When the user closes the application, where should your application return to (the SAS display manager, another application, or the operating environment)?

The basic methodology for deploying an application should include the following steps:

1. Modify the prototype so that only the desired SAS software windows are open when the application starts.
2. Modify the prototype so that the user is returned to the appropriate environment when the application closes.
3. Copy the catalog or catalogs that comprise your application from the testing or prototype location to the production location.

4. Prepare a custom configuration file to initialize SAS for use by your application.
5. Create a command file or icon that launches the application.

Migrating Your Application from Testing to Production

Introduction

Once you have tested your application in a development environment, you should copy it to the area designated as a production location, where you can retest it.

Two procedures are available for copying applications:

- the COPY procedure
- the BUILD procedure.

The COPY procedure is a general-purpose utility for copying SAS libraries from one device to another or from one file to another. You can copy an entire library, or you can select or exclude members of specific types.

For example, to copy an application from a prototyping or test environment on the C: drive to a production environment on the F: drive, you could use this code:

```
libname proto 'c:\test';
libname product 'f:\apps';

proc copy in=proto out=product;
run;
```

If your catalogs are already in the production location, then run this program simply to reduce the size of the catalogs:

```
proc copy in=product out=work;
proc copy in=work out=product;
run;
```

Note: Although you can use operating environment commands to copy SAS files, you should use the COPY procedure instead. After you edit catalogs in the build environment, some entries can become fragmented. When the COPY procedure executes, it reconstructs and reorganizes all SAS files, including data sets and catalogs. The new files often occupy less disk space; it is not unusual for a catalog to shrink by 50% to 75% when the COPY procedure is run on it.

You can copy either an entire catalog or selected entries with the MERGE statement in the BUILD procedure. The general form of the MERGE statement in PROC BUILD is:

```
PROC BUILD C=libref.out-cat;
MERGE C=libref.in-cat options;
```

where

out-cat
names the output catalog.

in-cat
names the input catalog.

options

are options to control which entries are copied. MERGE statement options include

REPLACE	causes entries in out-cat to be replaced with like-named entries from in-cat. The default is not to replace like-named entries.
ENTRYTYPE=	specifies an entry type for processing. If omitted, all entries are processed. If you want to copy entries of more than one type, use multiple MERGE statements. You can abbreviate the ENTRYTYPE= option as ET=.
NOEDIT	specifies that the entries being merged cannot be edited with PROC BUILD. Be sure to save the original copy of the entry for editing.
NOSOURCE	tells the procedure to copy only the compiled portions of FRAME and SCL entries. You can abbreviate the NOSOURCE option as NOSRC. The SCL code itself is removed from SCL and PROGRAM entries.

For complete information about the MERGE statement and about the BUILD procedure, refer to the SAS/AF online Help.

When you move your application from one library or catalog to another, be sure to check for hardcoded librefs. Particular care must be taken to ensure that SCL code and class references that are associated with your components refer to the proper location. See [“Deploying Components” on page 151](#) for more information.

Porting Your Application to Different Operating Environments

SAS/AF software enables you to create an application in one development environment and to port that same application to the operating environments of your users.

Note: Although SAS/AF software does not support a build-time environment for FRAME entries on a mainframe, applications can be developed in desktop environments such as Windows and ported to run on a mainframe.

To port (or “export”) an application to another environment:

1. In the development environment, use the CPORT procedure to export a SAS library or SAS catalog into a transport file. (You can transfer SAS tables directly from one operating environment to another without creating a transport file.)

The basic form of the CPORT procedure is

```
proc cport source-type=libref<.member-name>
  file=transport-fileref;
```

where *source-type* specifies a catalog, data set, or library; *member-type* is required if *source-type* is either CATALOG or DATA; and *transport-fileref* specifies the transport file to write to.

2. Copy the transport file from your development environment to the production or end-user environment(s), using the appropriate operating system commands for a binary file transfer such as FTP.
3. In each environment to which you want to deploy the application, use the CIMPORT procedure to restore the transport file to its original form in the appropriate format for the host operating environment.

The basic form of the CIMPORT procedure is

```
proc cimport destination=libref<.member-name>
             infile=transport-fileref;
```

where *destination* specifies a catalog, data set, or library; *member-type* is required if *destination* is either CATALOG or DATA; and *transport-fileref* specifies the transport file to read.

For details on using the CPORT or CIMPORT procedures, see the Base SAS Procedures Guide.

When you develop a frame for use in other environments, it is recommended that you port the frame to the target platform to test the frame, using the same display that your users will use. The sizing of visual components on a frame can be affected by the host environment due to differences in the system fonts in the development environment and the run-time environments. Currently, font sizes can vary depending on the display device that is used.

It is important to test applications on all display devices that are used in the deployment environments. Also, keep in mind the following restrictions when developing frame-based applications that you plan to deploy in multiple environments:

- Some objects are not supported on all display devices, and some objects behave differently when displayed in different host environments. Refer to the “Component Reference” in the SAS/AF online Help for details.
- Text and graphic alignment may differ between the build-time environment and the run-time environment.
- Certain events or functions that are supported in some environments do not work in others, including mouse movement, drag and drop operations, double-clicking, selecting either mouse button 2 or 3, and others.
- Some display devices have very limited color selections. For example, when a color such as gray is displayed on a terminal that must choose between black and white, some visual objects on the frame may not appear or may disappear into the background of the frame.

For additional information about a specific host operating environment and SAS software, refer to the SAS Companion for that environment.

Configuring a SAS Session for Your Application

Introduction

There are several ways to set up a SAS session to accommodate the needs of the applications you want to deploy. Some applications will be called by users from within a SAS session, which requires that those users have access to the libraries that contain the data sets and catalogs that comprise the application. Other applications may extend or add additional capabilities to existing applications, and you can add program calls from one application to start another.

However, most applications that you deploy, require that you provide a custom invocation of SAS. To start a SAS session, you must have access to the SAS executable file and to a SAS configuration file. (A standard SAS configuration file is supplied with SAS software and is typically located in the same directory as the SAS executable file.) The SAS configuration file specifies

- the value of an environment variable, SASROOT, that contains the name of the directory where SAS software is installed. The SAS executable and configuration files are stored in this directory.
- which subdirectories of the SASROOT directory to search for various SAS products.
- the location of the files that contain warning and error messages.
- the location of the SAS help facility and other modules that are necessary for interactive execution (the SASHELP libref).
- the location of a temporary directory for users' work files (the WORK libref).
- the location of a permanent directory for users' personal files (the SASUSER libref).
- the locations of online Help (HELPLOC) files.
- the settings of other system options, some of which are host-specific.

Specifying a Custom Configuration File

The CONFIG system option specifies the location of the configuration file. You can include SAS system options as part of the SAS command that is used to start a SAS session. The configuration file that is supplied with Version 9 is called SASV9.CFG. Here are two examples, one for a PC environment and the other for UNIX operating environments:

```
sas -config c:\myApp\myConfigFile.cfg
```

```
sas -config /u/myApp/myConfigFile.cfg
```

Note: Your configuration file does not have to be named SASV9.CFG. If you use a generic name for your configuration file, start with a copy of the one that is supplied by SAS and then rename it. If you do not specify a configuration file in the SAS command, the first file named SASV9.CFG in a predefined search path is used. For details about host-specific system options and configuration files, see the SAS Companion for your operating environment.

To create a configuration file, you can

1. Make a copy of the default configuration file.
2. Modify the copy by adding or changing SAS system options.

You can also specify additional windowing environment options in a custom configuration file. For example, consider this configuration file for an application on a Windows PC:

```
-AWSCONTROL TITLE NOSYSTEMMENU MINMAX
-AWSTITLE 'Corporate Reporting'
-NOAWSMENUMERGE
-SASCONTROL NOSYSTEMMENU NOMINMAX
-NOSPLASH
-SET corprept F:\apps
-HELPLOC ('!sasroot\core\help' 'F:\apps\help')
-SPLASHLOC F:\apps\images
-INITCMD "AF C=corprept.APP.MAINMENU.FRAME PMENU=YES"
```

where

AWSCONTROL

specifies application workspace (AWS) features. Precede a feature with NO to disable it.

- TITLE indicates whether the AWS includes a title bar.
- SYSTEMMENU indicates whether the AWS menu appears.
- MINMAX indicates whether the minimize and maximize controls appear.

AWSTITLE

specifies the title for the AWS window if the AWSCONTROL TITLE option is set.

AWSMENUMERGE/NOAWSMENUMERGE

specifies whether the AWS menu items Options, Window, and Help appear.

SASCONTROL

specifies application window features. Precede a feature with NO to disable it.

- SYSTEMMENU indicates whether the AWS menu appears.
- MINMAX indicates whether the minimize and maximize controls appear.

SPLASH/NOSPLASH

specifies whether the SAS software splash screen appears while your application is starting. You can substitute your own bitmap image for the SAS software splash screen. For more information, see the *SAS Companion for the Microsoft Windows Environment*.

SET

creates an environment variable. An environment variable that contains a pathname can be used as a SAS libref. For networked applications, ensure that all SAS files in this path have a read-only attribute.

HELPLOC

specifies the location of online Help files.

SPLASHLOC

specifies the location of a custom splash screen image to display when SAS starts.

Invoking Applications

You can use the INITCMD system option to launch your SAS/AF software application. The main advantage of this option is that SAS display manager windows are not displayed before your primary application window appears. You can add the INITCMD option to

- the SAS command that you use to invoke the SAS session for your application
- the configuration file that is used in conjunction with the SAS command.

The general form of the INITCMD system option is

```
-INITCMD "AF-command<, SAS-windowing-environment-command>; . . . ;
SAS-windowing-environment-command"
```

The INITCMD option

- is specified either as part of the SAS command or in a configuration file
- requires you to specify the DMS option
- must follow the -DMSEXP option (unless -NODMSEXP is specified) if placed in a configuration file

- opens the AF window first
- does not open other SAS windowing environment windows (but they can be opened by the application)
- can issue additional SAS windowing environment commands
- eliminates the need for the DM statement in an autoexec file and does not permit execution of DM statements
- exits SAS when the user exits the application
- is available in most operating environments.

Enabling the Application for Your Users

Once the application is tested and ready for use in the production environment, you must provide a mechanism for users to invoke the application. Typically, you can either create a command file or a program item that contains the SAS command plus a reference to the custom configuration file.

The Windows operating environment has graphical interface features for launching application programs. In such an environment, you can create a program item for your application and make your application accessible to users via a desktop icon or folder.

For some environments, you might want your users to access the application by using a shell script or a batch file. For example, the following shell script starts an application in a UNIX environment:

```

APPS=/u/apps
export APPS
sas -sasuser          /u/apps/sasuser
    -rsasuser
    -autoexec         /u/apps/autoexec.sas
    -printcmd nlp
    -xrm              'SAS.windowWidth: 80'
    -xrm              'SAS.windowHeight: 32'
    -xrm              'SAS.windowUnitType: character'
    -xrm              'SAS.awsResizePolicy: variable'
    -xrm              'SAS.sessionGravity: CenterGravity'
    -initcmd          "af c=corprept.app.MAINMENU.FRAME"

```

You can give users “execute” permission to the script file, and they can simply enter the script's name at a shell prompt to launch the application.

Part 3

Developing Custom Components

<i>Chapter 9</i>	
Tools for the Component Developer	67
<i>Chapter 10</i>	
SAS Object-Oriented Programming Concepts	71
<i>Chapter 11</i>	
Developing Components	95
<i>Chapter 12</i>	
Managing Methods	109
<i>Chapter 13</i>	
Managing Attributes	115
<i>Chapter 14</i>	
Adding Communication Capabilities to Components	129
<i>Chapter 15</i>	
Deploying Components	151

Chapter 9

Tools for the Component Developer

Introduction	67
Class Editor	67
Resource Editor	68
Interface Editor	68
Source Window	69
Other Development Tools and Utilities	70

Introduction

The classes from which components are derived can be designed and managed in the SAS/AF development environment with the following tools:

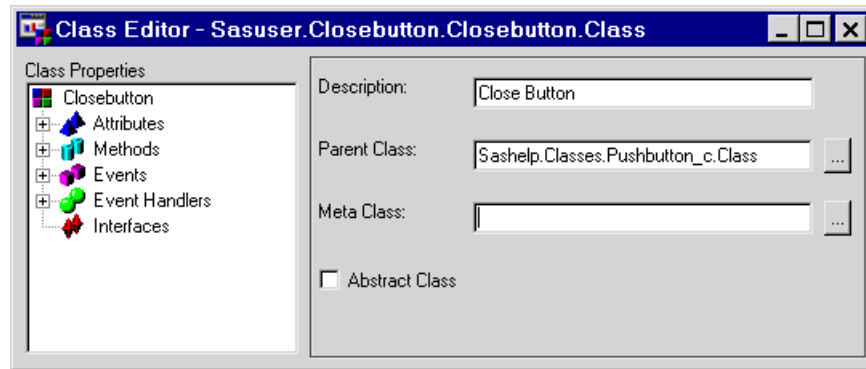
- Class Editor
- Resource Editor
- Interface Editor
- Source window

Other complementary development tools are also available to increase performance, provide source management, and produce class documentation.

Class Editor

SAS/AF software provides a Class Editor for editing existing classes and for subclassing. The Class Editor enables you to modify or add properties, including attributes, methods, events, event handlers, and interfaces. Any time you open a CLASS catalog entry you are using the Class Editor.

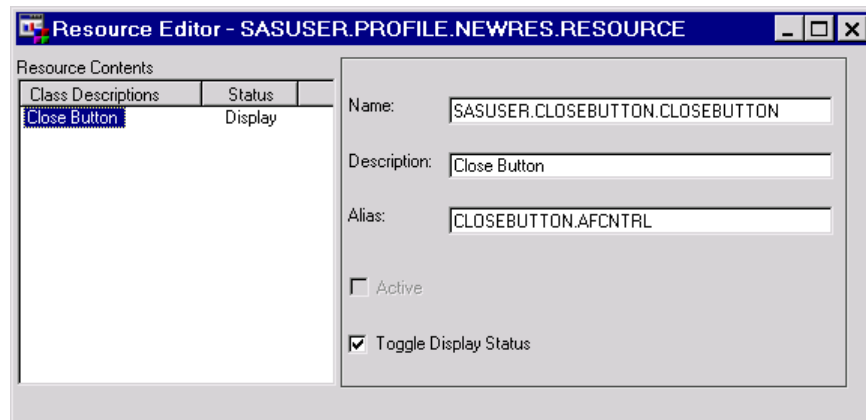
Changing any of the properties or behaviors of a class changes the property or behavior of all instances and children of that class.



To open the Class Editor, double-click an existing class in the SAS Explorer, or create a new class via the SAS Explorer or the BUILD command (for example, `build work.a.a.class`). For more detailed information on using the Class Editor, see “Component Development” or “Class Editor” in the SAS/AF online Help.

Resource Editor

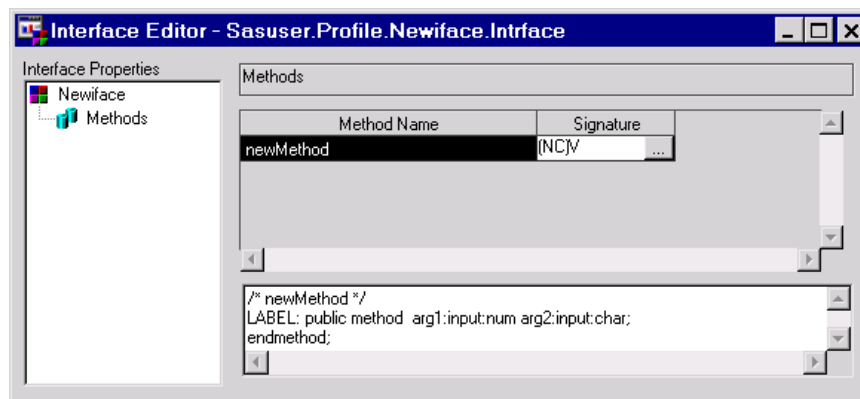
The Resource Editor enables you to group classes into resources. These resources can then be added to the Components window so that you can easily use your classes at design time.



To open the Resource Editor, double-click an existing resource in the SAS Explorer, or create a new resource via the SAS Explorer or the BUILD command (for example, `build work.a.a.resource`). For more information on the Resource Editor, see “Managing Classes with Resources” on page 152, or “Resources” in the SAS/AF online Help.

Interface Editor

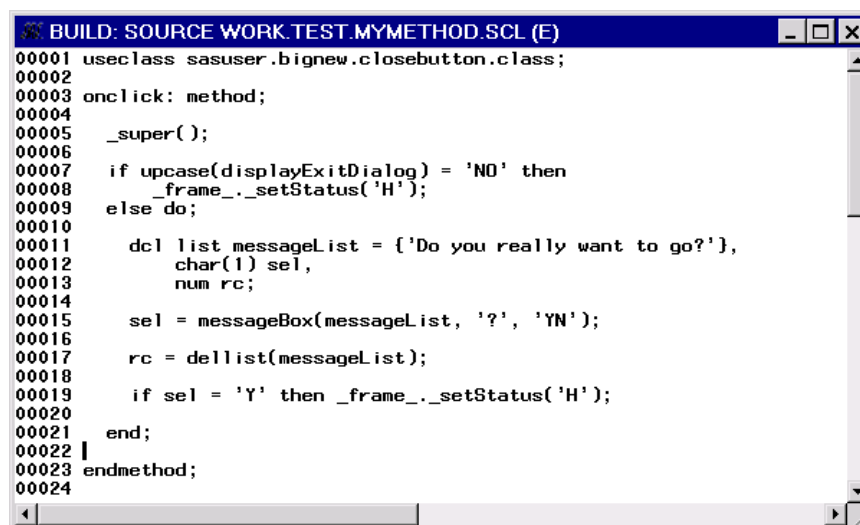
The Interface Editor enables you to build interface objects that consist of method definitions (the method name and method signature). The methods defined in an interface are implemented on the classes that implement that interface. Typically, interfaces are used by models and views to establish model/view communications.



To open the Interface Editor, double-click an existing interface in the SAS Explorer, or create a new interface via the SAS Explorer or the BUILD command (for example, **build work.a.a.interface**). For more information on how to use the Interface Editor, see “Interfaces” in the SAS/AF online Help. For more information on using interfaces, see “Interfaces” on page 93.

Source Window

The Source window provides a text editor for creating and editing SAS Component Language (SCL) programs.



As a component developer, you can open the Source window by

- double-clicking an existing SCL entry in the SAS Explorer
- creating a new SCL entry via the SAS Explorer
- creating a new SCL entry via the BUILD command (for example, **build work.a.a.scl**)
- issuing the SOURCE command from a FRAME or PROGRAM entry
- selecting **Source** from the Class Editor's pop-up menu when any method metadata item is selected.

For more information about editing SCL programs, see [“Implementing Methods with SCL” on page 109](#).

Other Development Tools and Utilities

SAS/AF software provides several tools and diagnostic utilities that extend the applications development environment.

Class Browser

enables you to browse through the class hierarchy, create subclasses, edit classes, copy or delete classes, and view properties information and help for a selected class.

SCL Analysis Tools

help you optimize the performance of SAS Component Language (SCL) code in your applications. Tools include a Coverage Analyzer, a List Diagnostic Utility, a Performance Analyzer, and a Static Analyzer.

Source Control Manager (SCM)

provides source management for applications development with SAS software. SCM is a full-featured source manager.

GenDoc Utility (experimental)

enables you to generate HTML files that document class, interface, resource, and frame entries. For more information on the GenDoc Utility, see [“Generating Class Documentation with GenDoc” on page 156](#).

For more information about these tools, see the SAS/AF online Help.

Chapter 10

SAS Object-Oriented Programming Concepts

Introduction	71
Object-Oriented Development and the SAS Component Object Model	73
Classes	73
Introduction	73
Relationships among SAS/AF Classes	74
Types of Classes	76
Methods	77
Introduction	77
Method Signatures	78
Overloading Methods	80
Method Scope	81
Method Metadata	81
Attributes	84
Introduction	84
Attribute Metadata	84
Attribute Values and Dot Notation	88
Events	90
Introduction	90
Event Metadata	91
Event Handlers	92
Introduction	92
Event Handler Metadata	92
Interfaces	93
Introduction	93
Interface Properties of a Class	93

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 SAS/AF software, 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 in SAS/AF software.

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:

state

a collection of attributes and their current values. Two of a Push Button control's attributes are `label` (the text displayed on the command push button) and `commandOnClick` (the command that executes when the command push button is pressed). You can set these values through the Properties window or through SCL.

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, a Push Button can set its own border style via the `_setBorderStyle` method.

identity

a unique value that distinguishes one object from another. In SAS/AF, this identifier is referred to as its object identifier. The object 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 SAS Component Language (SCL) and in SAS/AF software.

Object-Oriented Development and the SAS Component Object Model

The SAS Component Object Model (SCOM) is an object-oriented programming model that provides a flexible framework for SAS/AF component developers. With SCOM, you can develop plug-and-play components that adhere to simple communication rules, which in turn make it easy to share information between components.

A component in SCOM is a self-contained, reusable object with 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 SCOM, you can design components that communicate with each other, using any of the following processes:

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 control) to communicate with a model based on a set of common methods that are defined in an interface.

Drag and drop operations

enabling information to be transferred from one component to another by defining “drag” attributes on one component and “drop” attributes on the other.

Event handling

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

For more information on component communication with SCOM, see [“Adding Communication Capabilities to Components” on page 129](#).

Classes

Introduction

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 SAS/AF Classes

Introduction

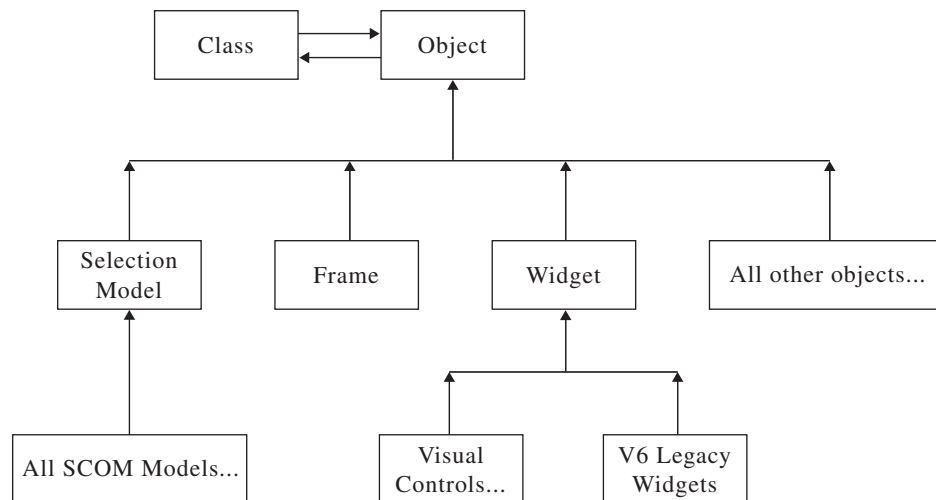
SAS/AF software classes support three types of relationships:

- inheritance
- instantiation
- uses.

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 10.1 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, SAS/AF software 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 76.

Uses

For some classes, a “uses” relationship exists. With a uses relationship, not all of the functionality that a class needs is defined by one class. Instead, a class can use the operations defined by another class. This process is called *delegation*. In SAS/AF software classes, the “uses” relationship is defined in two ways: delegation and composition.

delegation

With delegation, a method is automatically forwarded to a designated object (or list of objects, in some predetermined order) when it is not recognized by the object that initially received the message to execute the method. Each class has an optional delegates list, which is a list of complex attribute names (or, for legacy classes, instance variable names) to which undefined method calls are redirected. The delegates list contains named values; each item name matches the name of a complex attribute. When you call a method on an instance of the class and the method is not found in the object's class nor in any ancestor class, the method call is tried on each complex attribute (or instance variable) named in the delegates list until one of the named objects implements the method or until the list is exhausted. The delegate objects may, in turn, delegate the method. If none of the named objects implements the method, then the parent class's delegates list, if it exists, is tried next, and so on. If no delegates implement the method, there is an error.

Delegation is allowed whenever you call a method using SEND or NOTIFY routines. However, delegation is not allowed when you use dot notation on an object of a specific class. In order to use dot notation and delegation together, you must use a reference to a generic object. For example, the following will work

```
DCL object foo; foo.delegatedMethod();
```

but the following will not

```
DCL lib.cat.someclass.class foo; foo.delegatedMethod();
```

composition

Objects can work together to form composite objects. Collectively, the component objects form a composite object. The relationship that exists between the components and the composite object is called composition. With composition, a class implements a method by explicitly invoking that method on one or more objects; the other objects typically are stored in complex attributes. For example, consider a composite widget that is composed of an input field and three list boxes. The input field and the list boxes are the components. Note that the components may not have any explicit relationship with each other (except for the passive relationship of all being members of the same composite).

In a composite object, methods can be forwarded to individual objects within the composite object.

Some of the composite objects in SAS/AF software include

- Dual Selector Control
- Extended Input Field (legacy class)
- Toolbar (legacy class)

Types of Classes

Introduction

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

- Abstract classes
- Models and viewers
- 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 Viewers

In SAS/AF software, components that are built on the SAS Component Object Model (SCOM) framework can be classified either as viewers that display data or as models that provide data. Although models and viewers 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. Some models may be attached to multiple viewers.

Viewers 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 viewer. A viewer can have only one model attached to it at a time.

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

Models that are built on the SCOM framework are enabled for model/view communication through their support of a specified interface (such as `sashelp.classes.staticStringList.interface`). Likewise, all controls that display items in lists have also been enabled for model/view communication by requiring the use of the same interface.

For more information, see [“Interfaces” on page 93](#), [“Implementing Model/View Communication” on page 133](#), and 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.

Methods

Introduction

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

A method consists of two primary parts, its name and its implementation. You use the method name in SCL programs to manipulate the associated component. For example, to deselect the selections in a list box, you write an SCL statement that invokes the `_deselectAll` method:

```
listbox1._deselectAll();
```

Use caution when adding new methods with a leading underscore character, which typically indicates a method supplied by SAS and may cause conflicts with future releases of SAS/AF software. If you attempt to name a method with both leading and trailing underscores, SAS/AF displays a warning. Embedded underscores themselves are allowed, but they are removed when a leading and trailing underscore trigger the name conversion. For example, the method name `_foo_Test_Foo` is converted to `_fooTestFoo`.

Method names can be up to 256 characters long.

The implementation of the method (where the code exists) is specified in the name of a SAS/AF entry. This entry is usually an SCL entry that is combined with an optional SCL labeled section.

As previously mentioned, all classes inherit methods from their parent classes. Some classes also delegate methods to other classes. In addition, SAS/AF software defines three other types of methods:

automatic methods

are methods that automatically execute when certain events occur; you do not explicitly call automatic methods. Automatic methods can be classified in two major groups:

- those that execute at build time — for example, `_bInit`.

- those that execute at run time (that is, when the FRAME entry is opened by a TESTAF, AF, or AFA command) — for example, `_select`.

For more information about automatic methods, see [“Flow of Control for Frame SCL Entries” on page 169](#).

virtual methods

are methods that exist in a class but have no defined functionality for that class. You add their functionality by overriding the method in a subclass. Virtual methods exist for these reasons:

- The functionality of the method needs to be defined, but its functionality is specific to the subclass.
- The presence of a virtual method serves as a reminder that the method should be included in the subclass and that its functionality should be defined there.

The Object class contains some virtual methods — for example, `_printObject`. Each object can have specific printing needs; therefore, you should define the functionality of `_printObject` at the subclass level. Unless you override the definition of `_printObject`, the method does nothing, but it also does not generate an error.

per-instance methods

are methods that enable you to add to or override a method for a specific object in an application. These methods are not associated with a class but are assigned on a per instance basis. A per-instance method that is bound to an object is not shared with or available to other instances of the object's class. Per-instance methods do not affect any other objects in the application.

All predefined methods for a class are listed in the class's methods list. When a method is called, the SCL compiler searches the object class's methods list for the method. If no matching method is found there, the compiler searches the class's parent class, and so on. If an object has per-instance methods, the method lookup begins in the object's list of per-instance methods.

Per-instance methods are programmatically added, changed, and deleted with these Object Class methods:

- `_addMethod`
- `_changeMethod`
- `_deleteMethod`

For more information about these methods, see the Object class in the Component Reference section of the SAS/AF online Help.

Note: Although supported, per-instance methods represent a non-object-oriented approach to SAS/AF programming. A side effect of this approach is that the compiler cannot identify per-instance methods or attributes that have been added programmatically via SCL. Only those per-instance methods and attributes that have been added through the Properties window are recognized at compile time.

Method Signatures

A method's signature uniquely identifies it to the SCL compiler. A method's signature is comprised of the method's name, its arguments, and their types and order. Precise identification of a method based on these parameters is necessary when overloaded methods are used (see [“Overloading Methods” on page 80](#)).

For example, the following SCL METHOD statements show methods that have different signatures:

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

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.

Signatures in SAS/AF software are usually represented by a shorthand notation, called a *sigstring*. This sigstring is stored in the method metadata as SIGSTRING. For example, the four method statements above have the following sigstrings:

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

The parentheses group the arguments and indicate the type of each argument. The value outside the parentheses represents the return argument. The V character (for “void”) indicates that no return value is used. A notation that has a value other than V indicates that the signature has a return value of that particular type. For example, Method4, above, returns a numeric value.

Although the optional return variable is listed as part of the sigstring, it is listed only for convenience and should not be understood as part of the actual signature. In a sigstring, neither the presence of a return variable nor its type affects the method signature.

When a method is called, the SAS SCL compiler matches the arguments in the call to the arguments in the actual method. Thus, calling the method with the (CN)V signature actually executes Method1, below. Calling the method with the (NC)V arguments executes Method2:

```
/* Method1 */
/* Responds to calls using the (CN)V signature */
object.setColNum('colname', num);

/* Method2 */
/* Responds to calls using the (NC)V signature */
object.setColNum(num, 'colname');
```

If a method has a return value, you can execute that method and use the value that it returns in an assignment. For example, consider an object that has a method with a signature **()N**. The following are valid operations:

```
dcl num returnVal;
returnVal=object.getData();

or

if object.getData() > 1 then do ...

or

if ( object.getData() ) then do ...
```

Defining signatures for methods helps other developers understand the method syntax while also offering enhanced compile-time checking and run-time efficiency. The default signature for methods is `() V`, meaning that no arguments are passed into the method and no values are returned.

Although the SCL compiler uses the method name to help uniquely identify a method, the name is not formally part of the signature metadata. In other words, although a method's signature includes its name and arguments (and their types and order), the signature metadata itself consists of

- argument name
- argument type
- argument usage
- argument description

Just like return variables, the usage and description of an argument are not used to differentiate methods, and are not part of a method's signature, even though they are part of the signature metadata.

Once you define a signature for a method and deploy 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.

Overloading Methods

SAS Component Language (SCL) supports method overloading, which means that a class can have methods of the same name as long as they can be distinguished on the basis of their signatures (that is, as long as their arguments differ in number, order, and/or type). If you call an overloaded method, SCL checks the method arguments, scans the signatures for a match, and executes the appropriate code.

For example, if you had a `setColor` method on your class, you could define overloaded methods with the following signatures:

Table 10.1 Overloaded `setColor` Methods

Method Calling Statement	Method Signature
<code>object.setColor(color);</code>	(C)
<code>object.setColor(r, g, b);</code>	(NNN)

In the example above, `color` is a character argument, and `r`, `g`, and `b` are numeric arguments. Both methods change the object's color even though the first method takes a color name as input and the second method takes numerical RGB values (in the range of 0255). The advantage of overloaded methods is that they require programmers to remember only one method instead of several different methods that perform the same function with different data.

Methods in general, not just overloaded methods, can specify only one return argument. Each method in a set of overloaded methods can have a different return argument type, but the method parameters must be different for each method since the return type is not considered part of the signature. For example, you can have

```
mymethod (NN)V
mymethod (CC)N
```

but not

```
mymethod (NC)V
mymethod (NC)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;
```

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 (just as the return value is not), so you may not define two signatures that differ only by scope. (See the next section, “Method Scope”.)

In addition, a method that has no signature (that is, which has `(none)` as a signature) cannot be overloaded.

Method Scope

SAS Component Language (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`. Only `Public` methods appear in the Properties window at design time. 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.

Method Metadata

In addition to a method's name, implementation, signature, and scope, SAS/AF software stores other method metadata for maintaining and executing methods. You can query a class (or a method within a class) to view the method metadata, and you can create your own metadata list to add or change a method. For example, to list the metadata for a particular method, execute code similar to the following:

```
init:
    DCL num rc metadata;
    DCL object obj;
    obj=loadclass('sashelp.mycat.maxClass.class');
```

```

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

```

Here is the returned method metadata:

```

(NAME='getMaxNum'
SIGNATURE=(
  NUM1=(  TYPE='Numeric'
         INOUT='Input'
         DESCRIPTION='First number to compare'
        )
  NUM2=(  TYPE='Numeric'
         INOUT='Input'
         DESCRIPTION='Second number to compare'
        )
  RETURN=(  TYPE='Numeric'
           DESCRIPTION='Returns the greater of two numeric values.'
           INOUT='Return'
          )
)
)
SIGSTRING='(NN)N'
ENTRY='sasuser.mycat.maxClass.scl'
LABEL='getMaxNum'
CLASS=4317
METHODID=4331
STATE='N'
DESCRIPTION='Returns the greater of two numeric values.'
ENABLED='Yes'
SCOPE='Public'
)

```

The method metadata contains the following named items:

Note: Because many methods are defined at the C code level, some metadata values, such as Entry, may not provide information that you can use.

Name

is the name of the method. Method names can be up to 256 characters long.

State

specifies whether the method is new (N), inherited (I) from another class, overridden (O), or a system (S) method.

Entry

is the name of the SAS/AF catalog entry that contains the method implementation. Typically, this item is an SCL entry.

Example: **sasuser.app.methods.scl**

Label

is the name of the SCL labeled section where the method is implemented. This item is valid only if the value of the Entry metadata item is an SCL entry. A typical implementation of a method may look like the following:

Example: newMethod

```

newMethod: public method
  item:input:num
  return=num;

```

```

        /* ...more statements... */
    endmethod;

```

Signature

is a list of sublists that correspond to the arguments in the method signature.

If an empty list is passed for this value, then SAS/AF software registers a signature of ()V, which means that the signature contains no arguments. If you call a method whose signature is ()V and specify any input, output, or return arguments, the compiler reports an error.

If a missing value (.) or zero is passed for this value, then the method does not have a stored signature. There is no compile-time checking for methods that have no stored signature. Likewise, if a method does not have a signature (the signature is (none)), it cannot be overloaded (although it may still be overridden). See [“Overloading Methods” on page 80](#) for more information.

Note: By default, legacy classes do not have signatures.

For each argument in a method signature, the following named items are stored in a sublist that has the same name as the argument:

Type

is the argument type. Valid values are

- Character
- Numeric
- List
- Object (generic)
- Class Name (a four-level name of a specific object)
- an array of any valid type

Type is the argument type. Valid values are

INOUT determines how the argument will be used in the method. Valid values are I | O | U | R (corresponding to Input, Output, Update, Return).

Description is a text description of the argument.

Return_MethodID is the unique identifier of the method; it is assigned when the method is created and is returned in this named item.

Scope

specifies which classes have permission to execute the method. Valid values are Public | Protected | Private. The default scope is Public.

If Scope='Public', then any class can call the method. The method is displayed in the component's Properties window at build time.

If Scope='Protected', then only the class and any subclasses can call the method. The method does not appear in the component's Properties window at build time, nor can you access the method via frame SCL.

If Scope='Private', then only the class itself can call the method. The method does not appear in the component's Properties window at build time, nor can you access the method via frame SCL.

Enabled

determines whether a method can be executed. Valid values are 'Yes' | 'No.' The default value is 'Yes.'

Description
is a description of the method.

Attributes

Introduction

Attributes are the properties that specify the information associated with a component, such as its name, description, and color. Each attribute includes metadata information such as Type, Value, and Scope. You can define and modify attributes for a class with the Class Editor. You can define, modify, and create links between attributes of an instance with the Properties window.

Attributes are divided into categories such as the “Appearance” category, which contains attributes that control color and outline type. These categories make it easier to view and find related items.

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.

Attribute Metadata

SAS/AF software 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. You can also create your own metadata list to add or change an attribute. For example, to list the metadata for the `label` attribute, execute code similar to the following:

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

  obj=loadclass('sashelp.classes.pushbutton_c.class');

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

The following attribute metadata is returned:

```
(NAME='label'
 INITIALVALUE='Button'
 STATE='N'
 TYPE='Character'
 DESCRIPTION='Returns or sets the text displayed as the label'
 CATEGORY='Appearance'
 AUTOCREATE=''
 SCOPE='Public'
 EDITABLE='Yes'
 LINKABLE='Yes'
 SENDEVENT='Yes')
```

```

TEXTCOMPLETION= 'No'
HONORCASE= 'No'
GETCAM= ''
SETCAM= ''
EDITOR= ''
VALIDVALUES= ''
CLASS=4317
)

```

The attribute metadata list contains the following named items:

Name

is the name of the attribute. Attribute names can be up to 256 characters long.

State

specifies whether the attribute is new (N), inherited (I) from another class, overridden (O), or a system (S) attribute. System attributes are supplied by SAS/AF software. The only item in a system attribute's metadata list that you can modify is its Initial Value.

Type

specifies the type of data being stored:

- N = Numeric
- C = Character
- L = List
- O = Object (generic)
- Class Name = four-level name of a non-visual class (such as `sashelp.classes.colorlist_c.class`)
- An array of a specific type and the boundary for each dimension of the array. For example, `Type = sashelp.classes.colorlist_c.class (1,2,3)` represents a three-dimensional array of type `colorlist_c`, with the upper boundaries for each dimension being 1, 2, and 3, respectively.

AutoCreate

specifies whether the attribute is automatically created and deleted by SAS/AF software. This item applies only to attributes of type list (L) and specific objects (O:classname). By default, the value is 'Yes,' which indicates that SAS/AF will create the list or instantiate the object when the object that contains the attribute is instantiated. SAS/AF will also delete the list or terminate the object when the object that contains the attribute is terminated. If the value is 'No,' then it is the responsibility of the SCL developer to control the creation and deletion of the attribute.

InitialValue

(optional) specifies the initial value of the attribute; must be of the defined data type.

Scope

specifies which class methods have permission to get or set the value of the attribute. Valid values are Public | Protected | Private.

- If `Scope='Public'`, then any method of any class can get or set the attribute value. The attribute is displayed in the component's Properties window, and the attribute can be queried or set with dot notation in frame SCL.
- If `Scope='Protected'`, then only the class or its subclasses can get or set the attribute value in their methods. The attribute does not appear in the component's Properties window, nor can you access the attribute via frame SCL.

- If `Scope='Private'`, then only the class can get or set the attribute value in its methods. The attribute does not appear in the component's Properties window, nor can you access the attribute via frame SCL. Additionally, subclasses cannot access the attribute.

Editable

indicates whether an attribute can be modified or queried.

- If `Scope='Public'` and `Editable='Yes'`, then the attribute can be accessed (both queried and set) from any class method as well as from frame SCL program.
- If `Scope='Public'` and `Editable='No'`, 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 `Scope='Protected'` and `Editable='No'`, 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 `Scope='Private'` and `Editable='No'`, 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 metadata settings creates a private, pre-initialized, read-only constant.

Note: If `Editable='No'`, the Custom Set Method is not called (even if it was defined for the attribute). The default is 'Yes'.

Linkable

specifies whether an attribute can obtain its value from another attribute via attribute linking. Valid values are 'Yes' | 'No'. Only public attributes are linkable.

SendEvent

specifies whether an event should be sent when an attribute is modified. See “Events” on page 90 for details.

When `SendEvent='Yes'`, 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 conditionally execute code when the value of the attribute changes.

If you change the `SendEvent` value from 'Yes' to 'No', and if `Linkable='Yes'`, 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.

ValidValues

specifies the set of valid values for a character attribute. Use blanks to separate values, or, if the values themselves contain blanks, use a comma as the separator. For example:

```
ValidValues='North South East West'
ValidValues='North America,South America,Western Europe'
```

If `Type='C'` and a list of valid values has been defined, the values are displayed in a drop-down list for the attribute's Initial Value cell in the Class Editor and for the attributes Value cell in the Properties window.

ValidValues is also used as part of the validation process that occurs when the value is set programmatically using dot notation or the `_setAttributeValue` method. For more information on how the ValidValues list is used as part of the validation process, see [“Validating the Values of Character Attributes” on page 116](#).

You can also specify an SCL or SLIST entry to validate values by starting the ValidValues metadata with a backslash (\) character, followed by the four-level name of the entry. For example:

```
ValidValues='\sashelp.classes.ItemsValues.scl';
```

Note: If you use an SLIST entry for validation, all items in the SLIST must be character values. If you use an SCL entry for validation, you must ensure that the SCL entry returns only character items.

For more information on how to use an SCL entry to perform validation, see [“Validating the Values of Character Attributes” on page 116](#).

Editor

specifies a FRAME, PROGRAM, or SCL entry that enables a user to enter a value (or values) for the attribute. If supplied, the editor entry is launched by the Properties window when a user clicks the ellipsis button (...) in the Value cell or the Initial value cell in the Class Editor. The value that is returned from the editor sets the value of the attribute. For more information, see [“Assigning an Editor to an Attribute” on page 117](#).

TextCompletion

specifies whether user-supplied values for the attribute are matched against items in the ValidValues metadata for text completion. This item is valid only if the attribute is Type='C' and a ValidValues list exists. For example, if ValidValues='Yes No' and the user types 'Y', the value 'Y' becomes 'Yes' and appears in the input area.

HonorCase

specifies whether user-supplied values must match the case of items in the ValidValues list in order to constitute a valid input value. This item is valid only if the attribute is Type='C' and a ValidValues list is defined.

For example, if HonorCase='Yes', ValidValues='Yes No', and the user types 'yes', the value is not matched against the valid values.

However, if HonorCase='Yes', ValidValues='Yes No', TextCompletion='Yes', and the user enters 'Y', the value is found in the list of valid values and is expanded to 'Yes'.

GetCAM

specifies the custom access method for retrieving the attribute's value (GETCAM). For information on how to assign a CAM, see [“Assigning a Custom Access Method \(CAM\) to an Attribute” on page 121](#).

SetCAM

specifies the custom access method for setting the attribute's value (SETCAM). For information on how to assign a CAM, see [“Assigning a Custom Access Method \(CAM\) to an Attribute” on page 121](#).

Category

(optional) specifies a logical grouping for the attribute. This item is used for category subsetting in the Class Editor or for displaying related attributes in the Properties window. For example, the Properties window displays the following attribute categories for components that are supplied by SAS:

Appearance

Misc

Behavior	Region
Data	Model/View
Drag and drop	Size/Location
Help	System

Not all categories appear for each class.

You can also create your own category names simply by specifying a new category name in the Class Editor. The category defaults to 'Misc' if one is not supplied.

Automatic

is used only if an instance variable (IV) is linked to the attribute. (See the IV metadata item below.) This metadata item exists to support legacy class information only. When an IV is linked to an attribute, the attribute's value persists on the IV itself instead of on the attribute. If automatic is set to 'Yes,' then the IV is an automatic instance variable as defined in SAS/AF legacy classes.

IV

specifies the name of an instance variable on which the attribute value is stored. This metadata item exists to support legacy class information only. Components that are based on the SAS Component Object Model (SCOM) architecture do not use instance variables. It is recommended that you avoid using instance variables.

Description

is a short description for the attribute. This item appears as help information in the Class Editor and in the Properties window.

Attribute Values and Dot Notation

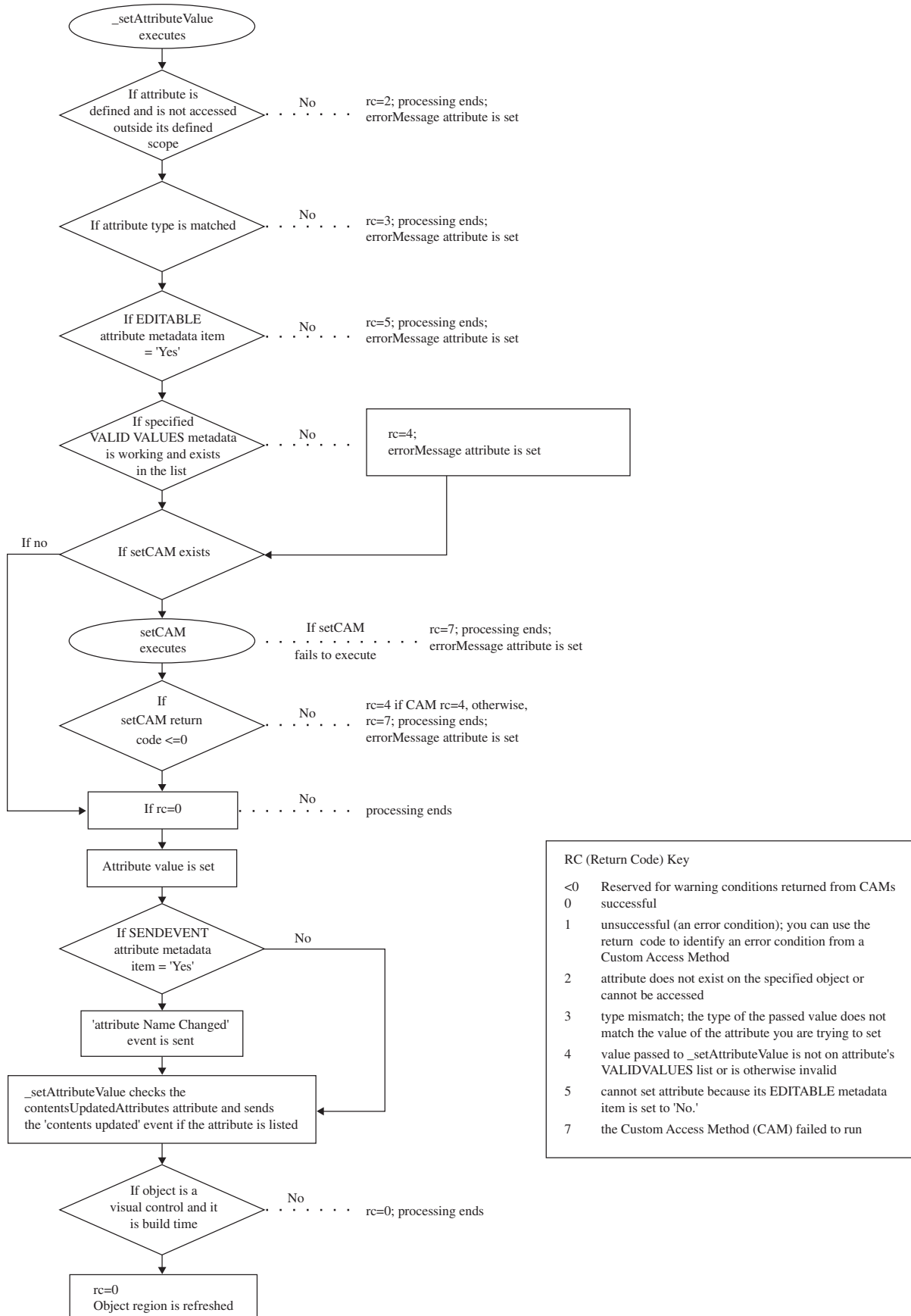
When you use dot notation in SCL to change or query an attribute value, SAS/AF software translates the statement to a `_setAttributeValue` method call (to change the value) or to a `_getAttributeValue` method call (to query the value). These methods are inherited from the Object class and provide the basis for much of the behavior of attributes.

For example, the `_setAttributeValue` method

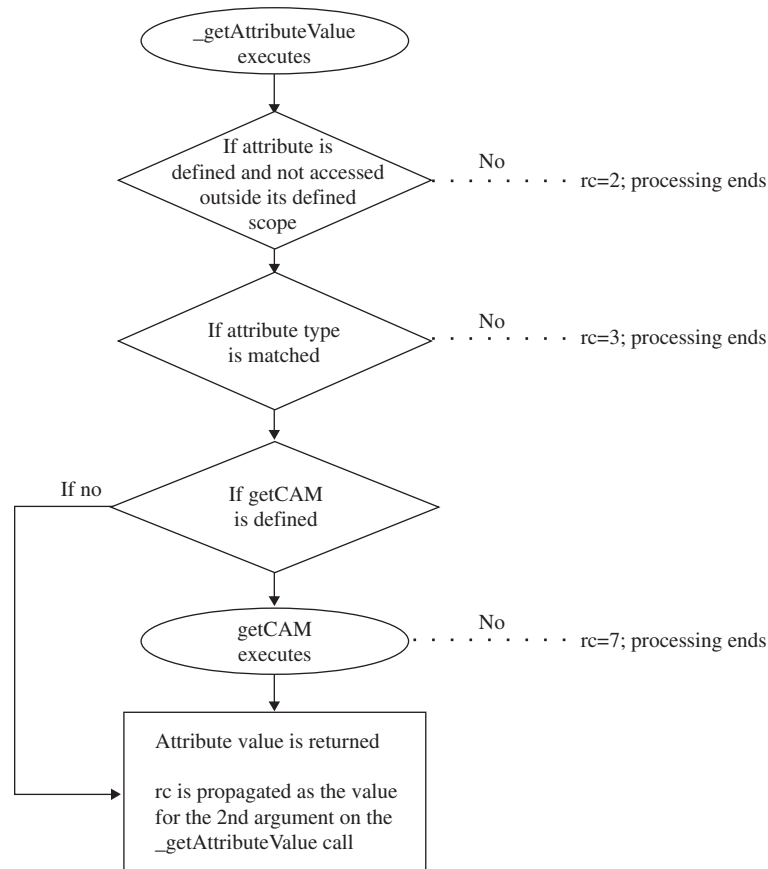
- verifies that the attribute exists
- verifies that the type of the attribute matches the type of the value that is being set
- validates the value against the ValidValues metadata item if it exists for the attribute
- invokes the custom set method (setCAM) if it exists
- stores the value in the attribute
- sends the “attributeName Changed” event if the attribute has a SendEvent='Yes' metadata item
- sends the “contents updated” event if the attribute is specified in the object's `contentsUpdatedAttributes` attribute to notify components in a model/view relationship that a key attribute has been changed

The following figures detail the flow of control for the `_setAttributeValue` and `_getAttributeValue` methods.

Figure 10.2 Flow of Control for _setAttributeValue



RC (Return Code) Key	
<0	Reserved for warning conditions returned from CAMs
0	successful
1	unsuccessful (an error condition); you can use the return code to identify an error condition from a Custom Access Method
2	attribute does not exist on the specified object or cannot be accessed
3	type mismatch; the type of the passed value does not match the value of the attribute you are trying to set
4	value passed to _setAttributeValue is not on attribute's VALIDVALUES list or is otherwise invalid
5	cannot set attribute because its EDITABLE metadata item is set to 'No.'
7	the Custom Access Method (CAM) failed to run

Figure 10.3 Flow of Control for `_getAttributeValue`

Events

Introduction

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. Event handlers then provide a response to the change.

Use the Class Editor to create events that are specific to an entire class. Use the Properties window to create events that are specific to a single object.

SAS/AF software supports system events, which can include user interface events (such as mouse clicks) as well as “attribute changed” events that occur when an attribute value is updated. For system events, the State metadata item is 'S'. In order for “attribute changed” events to be sent automatically, the component must have the SendEvent metadata item for the attribute set to 'Yes'. See [“Enabling Attribute Linking” on page 130](#) for details.

SAS/AF software also supports user-defined events, which can be registered to specific classes as needed and are inherited by subclasses.

Event Metadata

Events are implemented and maintained with metadata. You can query a class (or an event within a class) to view the event metadata, and you can create your own metadata list to add or change an event. For example, to list the metadata for the `click` event, execute code similar to the following:

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

  obj=loadclass('sashelp.classes.listbox_c.class');

  rc=obj._getEvent('click',metadata);
  call putlist(metadata,'',3);
return;
```

The following event metadata is returned:

```
(NAME='click'
 STATE='S'
 DESCRIPTION='Occurs on a click'
 CLASS=4415
 EXECUTE='System'
 METHOD=''
 ENABLED='Yes'
 CLASS=4415
 )
```

The event metadata list contains the following named items:

Name

is the name of the event. Event names may be up to 256 characters long.

State

is the current state of each event. Valid values are I | N | S (corresponding to Inherited, New, or System). This is a read-only metadata item.

Execute

describes how the event is sent. Events can be sent automatically either before or after a method. They can also be programmed manually with SCL. Valid values are: A | B | M (corresponding to After, Before, Manual). New events default to 'After'.

The Class Editor and the Properties window display Execute metadata in the Send column of the table.

Method

is the name of the method that triggers the event (if Execute=A | B). A method name must be specified for new events if they are to be sent automatically. The field remains blank if the event is sent manually (if Execute=M).

Enabled

indicates whether an event is enabled or disabled. Valid values are Y | N.

Description

is a descriptive summary of the event's purpose. For inherited and system events, the description is forwarded from the parent class. The maximum length is 256 characters.

Event Handlers

Introduction

An event handler is a property that determines which method to execute after an event occurs. Essentially, an event handler is a method that executes another method after an event is received. Use the Class Editor to create event handlers that are specific to an entire class. Use the Properties window to create event handlers that are specific to a single object.

Event Handler Metadata

SAS/AF software uses a set of event handler metadata to implement and maintain event handlers. This metadata exists as a list that is stored with the class. You can query a class (or an event handler within a class) to view the event handler metadata. You can also create your own metadata list to add or change an event handler. For example, to list the metadata for a particular event handler, execute code similar to the following:

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

  obj=loadclass('sashelp.classes.listbox_c.class');

  rc=obj._getEventHandler('_self_', 'click',
                        '_onClick', metadata);
  call putlist(metadata, ' ', 3);
return;
```

The following event handler metadata is returned:

```
(SENDER='_self_'
EVENT='visible changed'
DESCRIPTION='Refresh myself'
STATE='N'
METHOD='_refresh'
ENABLED='Yes'
)
```

The metadata contains the following named items:

State

is the current state of each event handler. Values include Inherited, New, or System, which are represented by I, N, and S, respectively.

Sender

is the name of the object that generates the event. The default value is `_self_`. Valid values are `_self_`, any object (*), or the name of a component on the frame.

Event

is the name of the event that is being handled.

Method

is the name of the method to execute when the event is sent.

Enabled

shows whether an event handler is enabled or disabled. Valid values are Y | N.

System events (State=S) are always enabled.

Description

is text that describes the event handler. The maximum length is 256 characters.

Interfaces

Introduction

Interfaces are collections of abstract method definitions that define how and whether model/view communication can take place. They enable you to redirect a method call from one component to a method call on a different component. The method definitions are just the information that is needed to define a method; they do not contain the actual method implementations. If two components share an interface, they can indirectly call each others' methods via that interface.

Interfaces are stored in SAS catalog entries of type INTRFACE. For example, the staticStringList interface is stored in the `sashelp.classes.staticStringList.intrface` catalog entry.

To retrieve the methods that an interface supports, use the `_getmethod` or `_getmethods` methods.

Interface Properties of a Class

A class can be defined to support or require one or more interfaces. 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. Interface data on a class consists of the following items:

State

specifies whether the interface is New or Inherited.

InterfaceName

specifies the four-level catalog entry name of the interface class (such as `sashelp.classes.staticStringList.intrface`).

Status

indicates whether the interface is Required or Supported.

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.

For more information on interfaces and their use, see [“Implementing Model/View Communication” on page 133](#).

Chapter 11

Developing Components

Creating Your Own Components	95
Introduction	95
Subclassing Methodology	95
Creating a Simple Subclass by Overriding an Attribute	96
Creating a Subclass by Overriding a Method	97
Extending a Class with New Attributes and Methods	99
Creating a Class with SCL	100
Introduction	100
Converting a Class to an SCL Entry	102
Tips for Creating Classes with SCL	102
Implementing Methods Using CLASS and USECLASS Statement Blocks	103
Using SCL to Instantiate Classes	105
Introduction	105
Using Constructors	105

Creating Your Own Components

Introduction

When an existing SAS/AF software class does not provide the behavior you desire, you can extend the functionality of an existing class by creating a subclass. Extending a class be as simple as creating a new frame class with a blue background or as complex as writing your own model and view classes. By virtue of the SCOM framework, you can take advantage of object-oriented functionality and add to or modify any of the existing properties of a class. Subclassing and class-wide modifications are performed in the Class Editor or with SCL class syntax.

Subclassing Methodology

When creating a subclass, you should follow these general steps:

1. Decide which class to subclass. Normally this is the class that contains the behavior closest to the functionality you want to implement.
2. Extend the existing properties by modifying or adding attributes, methods, events, or event handlers.

3. Test the subclass with either of these methods:
 - Add the new class to the Components window so that it appears in the list of available classes and can be tested on a frame.
 - Instantiate the subclass using the `SCL_NEW_` or `_NEO_` operators and test it programmatically.

Creating a Simple Subclass by Overriding an Attribute

Introduction

As an example of both the methodology and the techniques that are used to create a simple subclass, the following example demonstrates how to build a simple button class that closes a frame. Because you include a “Close” button of some sort on nearly every frame you build, modifying a subclass of the existing `pushbutton_c` class will save you from setting the same attributes each time you want such a button. After creating the class and adding it to the Components window, you simply drag your Close Button class onto the frame; no coding is required, and you do not have to set its properties.

You can add or override properties using either the Class Editor or the Properties window. Although they look alike and contain what appears to be the same information, it is important to understand the differences between these two tools:

- Changes that you make in the Class Editor affect all instances that are based on the class, whether they appear on different frames or not.
- Changes that you make in the Properties window affect only the specific instance of the class. The Properties window displays public properties for each instance on a frame.

Creating the Close Button Subclass

This example creates a subclass of the push button control, using the Class Editor to override the values of the `label` and `commandOnClick` attributes. For more information on specific tasks in this example, see the SAS/AF online Help.

To create a button that closes a frame:

1. Create a new catalog named `sasuser.buttons` to hold the new class.
2. Create a new class with a description of `Close Button`. The description is used to identify the class in the Components window. Unique descriptions are required (if you want to determine the difference between components with accuracy), but are not programmatically verified.
3. Select `sashelp.classes.pushbutton_c.class` as the parent class. Obviously, to know which class to subclass you must be familiar with the class hierarchy.
4. Override the `label` attribute. Specify `Close` as the initial value of the `label` attribute. The `label` attribute represents the text that appears on the button.
5. Override the `commandOnClick` attribute. Specify `End` as the initial value of the `commandOnClick` attribute. The `commandOnClick` attribute represents the command that is executed when the button is clicked.
6. Save the class. For the entry name, specify `CloseButton` (with no spaces).

You may also include a short description of what the class does, such as `Close frame button`. This description, if provided, overwrites the description provided in step 2 above.

Testing the Close Button Subclass

The Close Button class is now ready for use. Follow these steps to make the class available for use and testing:

1. Create a new frame.
2. Make the Close Button available for use in the Components window by right-clicking in the Components window and selecting **Add Classes**. Select or enter the `sasuser.buttons.closebutton.class`. The Close Button class appears in the Components window.
3. Drag a Close Button to the frame. The button label is `Close`.
4. Test the functionality of the button by selecting **Build** ⇌ **Test**. When the frame appears, click `Close` to close the frame.

The example demonstrates how you can create a very useful class without any programming.

Creating a Subclass by Overriding a Method

Introduction

When you create a new class, you must decide what unique actions you want to include in its definition. The actions are defined by their methods. Methods from the parent class are inherited by the subclass that you create. Although method inheritance is one of the primary benefits of object-oriented programming techniques, there are times when you will want to augment a class's functionality by adding new methods or overriding inherited methods:

- Create a new method for your subclass if you want to add behavior that is not currently available in the parent class.
- Override an existing method to modify the behavior of the parent class in your subclass. An overridden method is one that extends a method that has the same name and signature on a parent class. In effect, overridden methods add some kind of functionality to a class, but they also call the same method on the class's parent to ensure that the method's core functionality is preserved. When overriding a method, you provide a new method definition, but you cannot change the method's name, signature, or scope. You can also overload methods to complement existing behavior. For more information on overloading, see [“Overloading Methods” on page 80](#).

To demonstrate how you can add new behavior to a class, consider the Close Button class that was presented above. Although the Close Button class is useful, a user of your application might accidentally click a “Close” button. To help prevent such mistakes, you could add a confirmation dialog box to the Close Button class. The class could have the following behavior:

- When the Close button is clicked, a dialog box is displayed to confirm the action.
- If the user selects `Yes`, the frame closes. Otherwise, the frame remains open.

To implement this behavior in a new class, review the existing class:

- Does the parent class provide a behavior when the button is clicked on the frame?

After deciding to augment the behavior of a class, you must decide whether to override an existing method or add a new one. In the Close Button class, the `_onClick` method that serves as the event handler is called whenever the button is clicked. You can override this method and have the new implementation display the dialog box. If you override a method, you should execute the same method on the parent by calling `_super()` in your code.

See the SAS/AF online Help for assistance with specific tasks.

Overriding an Existing Method

To add new behavior to the Close Button class, override the `_onClick` method:

1. If you created the Close Button class that was presented earlier, you must re-inherit the `commandOnClick` attribute that was overridden so that the END command will not be executed when you click the button.
2. In the Class Editor, override the inherited `_onClick` method. The State metadata item changes to 'O'.
3. The `_onClick` Source Entry metadata item now contains the four-level name of the SCL entry that will contain the new code:
sasuser.buttons.closebutton.scl.
4. Display the Source for the method via the pop-up menu and enter the following code in the SCL entry:

```
useclass sasuser.buttons.closebutton.class;

onclick: method;
  _super(); /* call onClick method on parent */
  dcl num rc;
  dcl list messageList = {'Close this frame?'},
        char(1) sel;
  sel = messageBox(messageList, '?', 'YN');
  if sel = 'Y' then call execcmd('end;');
  rc=dellist(messageList);
endmethod;
enduseclass;
```

5. Compile and close the SCL entry.

Note: For more information on methods and SCL, see [“Implementing Methods with SCL” on page 109](#), and [“Adding SAS Component Language Programs to Frames” on page 35](#).

Testing the Overridden Method

After you have created and saved the class, you can test the new behavior. If you created the frame with the “Close” button from the last example, open that frame and test it to see the new behavior. The changes to the Close Button class are picked up automatically when the frame is opened and the Close Button object is instantiated.

If you do not have a frame with the Close Button object already on it, follow these steps to test the new behavior of the Close Button:

1. Create a new frame.

2. Right-click inside the Components window and select **Add Classes** from the pop-up menu to add the new class.
3. Drag a Close Button object from the Components window onto the frame.
4. Test the frame.

Extending a Class with New Attributes and Methods

Introduction

Although the Close Button class presented in the previous example is useful, you might not always want a confirmation dialog box to be displayed when a frame is closed. To implement this behavior, review the following question:

- Does the parent class provide data that could be used to specify whether to display a confirmation dialog box?

You can check attributes on the class to see whether you need to override an existing attribute or add a new attribute to maintain this information. Since the Close Button class does not have such an attribute, you have to add a new one to the class.

To add flexibility to the Close Button class, add an attribute to the class that enables developers who use the object in their frames to specify whether to display the confirmation dialog box.

To add an attribute to the Close Button class:

1. Close all frames that contain a version of the Close Button class.
2. Open the `sasuser.buttons.closebutton.class` class if it is not already open.
3. Create a new attribute called `displayExitDialog`. This will be the attribute that determines whether the confirmation dialog box should be displayed. Attributes keep data on a component where it can be easily accessed by other components via SCL or attribute linking.
4. Set the following metadata items for the new `displayExitDialog` attribute:
 - Type = **Character**
 - Initial Value = **Yes**
 - Valid Values = **Yes, No**
 - Category = **Behavior**

Editing the Existing Method to Query the New Attribute

Edit the existing overridden `_onClick` method on the Close Button class by adding four lines of code:

```
useclass
sasuser.buttons.closebutton.class;

onclick: method;
  _super();      /* calls the parent's _onClick method */

  if upcase(displayExitDialog) = 'NO' then
    call execcmd('end;');
  else do;
```

```

        dcl num rc;
        dcl list messageList = {'Close this frame?'},
            char(1) sel;
        sel = messageBox(messageList, '?', 'YN');
        if sel = 'Y' then call execcmd('end;');
        rc=dellist(messageList);
    end;
endmethod;
enduseclass;

```

Testing the New Attribute and Method

After saving the class, test the new attribute:

1. Test the frame and the Close Button class as they are when the frame opens to see the default Close Button behavior. The confirmation dialog box is displayed.
2. To see the alternate behavior of the Close Button, use the Properties window to set the Close Button object's `displayExitDialog` attribute to **No**.
3. Re-test the frame. No confirmation dialog box is displayed.

Creating a Class with SCL

Introduction

As an alternative to creating a class interactively with the Class Editor, you can create a SAS/AF class entirely in SCL with the CLASS/ENDCLASS statement block. You can define all property information through SCL. When you have finished defining a class with SCL, you must save it as a CLASS entry so that it can be used as a class.

Creating a class with SCL has several advantages:

- Lengthy or repetitive changes to class information (such as adding or deleting the signatures for several methods) are easier with a text editor than with the interactive, graphical approach of the Class Editor.
- Classes that are defined in an SCL entry can define and implement methods in one location.
- The CLASS block provides improved error detection at compile time, as well as improved run-time performance.

Metadata information, such as **description**, is added to the class or property by including a forward slash (/) delimiter and the appropriate metadata items enclosed in parentheses before the semicolon (;) that ends the statement. Use a comma to separate multiple metadata items (see **description**, **SCL**, and **Label** in the definition of the **add** method below).

Consider the following example, which defines the Combination class in SCL:

```

CLASS sasuser.myclasses.Combination.class
  extends sashelp.fsp.Object.class
  / (description='My Combination Class');

  /* define attributes */
  Public num total

```

```

/ (description='total attribute');
Public char catstr
/ (description='catstr attribute');

/* define methods */
add: public method
  n1: num
  n2: num
  return=num
  /(description='Adds two numbers',
    SCL='sasuser.myclasses.Combination.scl',
    Label='add');
concat: public method
  c1: char
  c2: char
  return=char
  /(description='Concatenates two strings',
    SCL='sasuser.myclasses.Combination.scl',
    Label='concat');
ENDCLASS;

```

To compile the SCL program and save it as a CLASS entry:

1. Save the SCL entry. You must save an SCL entry before using **Save as Class** or the SAVECLASS command.
2. From the Source window, select **File** ⇨ **Save As Class**.

Alternatively, you can enter the SAVECLASS command.

Saving an SCL program as a class is equivalent to saving a class that you created interactively with the Class Editor.

You can implement the methods directly in the same SCL as the class definition. The following code defines the Combination class and implements its methods:

```

CLASS
sasuser.myclasses.Combination.class
  extends sashelp.fsp.Object.class
  / (description='My Combination Class');

/* define attributes */
Public num total
/ (description='total attribute');
Public char catstr
/ (description='catstr attribute');

/* define methods */
add: public method
  n1:num
  n2:num
  return=num
  /(description='Adds two numbers');
  total=n1+n2;
  return (total);
concat: public method
  c1:char
  c2:char
  return=char

```

```

        ((description='Concatenates two strings'));
        catstr=c1 || c2;
        return(catstr);
    ENDCCLASS;

```

Additionally, you can create an abstract class by adding the optional reserved word **ABSTRACT** before the **CLASS** statement. For example:

```

ABSTRACT CLASS myClass
    EXTENDS sashelp.fsp.Object.class;
    /* ...insert additional SCL here... */
ENDCLASS;

```

For complete information about the **CLASS** statement, including all valid metadata that you can include with the class and properties definitions, see the *SAS Component Language: Reference*.

Converting a Class to an SCL Entry

You can convert any SAS/AF class to an SCL entry. This enables you to view and extend a class directly through its SCL. There are two ways to convert a class to SCL:

- You can use the Class Editor to save a class as an SCL entry by selecting **File** ⇨ **Save As** and setting the Entry Type to **SCL**. After saving the class as SCL, open the SCL entry to view or modify the class.
- You can programmatically convert classes to SCL using the **CREATESCL** function. The following code is an example of the **CREATESCL** syntax:

```

rc = CreateSCL ('lib.cat.yourClass.class',
               'lib.cat.yourSCL.scl', 'description');

```

In this example, **lib.cat.yourClass.class** is the class to convert, **lib.cat.yourSCL.scl** is the SCL entry to be created, and **description** contains the description of the class that is stored in the SCL entry.

Tips for Creating Classes with SCL

When you create a class using SCL, there are several recommended practices that might help your development efforts:

- For components that might be edited in the Class Editor, it is more appropriate to create one SCL entry for the class definition and another for the method implementations of that class. Method implementations are not stored with a **CLASS** entry. If you use the Class Editor to save a class as an SCL entry, and if the original **CLASS** entry was created from an SCL entry that contained method implementations, you may overwrite the method implementations for that class.
- It is recommended that you implement a standard naming convention for the catalog entries that are used to create a class. Since methods for a class are most often implemented in an SCL entry that has the same name as the **CLASS** entry, you might consider consistently appending “class” to the name of the SCL entry that defines the class. For example, consider a class named **Document** that was created using SCL and whose methods are implemented in a separate catalog entry. There would be a total of three entries:
 - the SCL entry that defines the class
(**sasuser.myclasses.Documentclass.scl**)

- the SCL entry that implements the methods of the class (`sasuser.myclasses.Document.scl`)
- the class entry itself (`sasuser.myclasses.Document.class`).
- Add values for the descriptive method-definition metadata, including **Description** and all argument descriptions (such as **ArgDesc1** or **ReturnDesc**). For example, consider the descriptive metadata for a method named `getAmount` that has a signature (C)N:

```
getAmount: public method
    account:input:char
    return=num
    /(Description='Returns the amount in the specified account',
       ArgDesc1='Account to retrieve',
       ReturnDesc='Amount in the account');
```

In this example, **Description** is the description of the method, **ArgDesc1** is the description of the `account` argument, and **ReturnDesc** is the description of the return argument.

For complete information, refer to the CLASS statement in the *SAS Component Language: Reference*.

Implementing Methods Using CLASS and USECLASS Statement Blocks

Introduction

The CLASS and USECLASS statements can be used to implement methods for a class in SCL and to bind them to the class at compile time, which improves code readability, code maintainability, run-time performance, and compile-time detection of many types of errors. SCL users can directly reference attributes and methods in a CLASS or USECLASS block without specifying the object ID or the system variable `_SELF_`. The CLASS statement is used both for defining and for implementing methods. The USECLASS statement is used only for implementing methods.

CLASS Statement

The CLASS statement constructs a class with SCL code, including the class definitions and, optionally, the method implementations. The ENDCLASS statement ends the CLASS statement block. In the following example, the method implementation is coded inside the CLASS statement block:

```
class sasuser.myclasses.one.class extends sashelp.fsp.object.class;
    /* define attribute */
    Public num sum
    / (description='sum attribute');

    _init: public Method
    / (State='O');
    _super();
    sum=0;
    endmethod;

    Sum: public method
    n:Num
    return=num;
```

```

        sum=sum+n;
        return (sum) ;
    endmethod;
endclass;

```

The approach above is appropriate for smaller projects where all class methods are maintained by a few developers. As projects increase in complexity and the number of developers involved grows, it is recommended that you use separate entries for class definitions and method implementations. It is the responsibility of each component developer to maintain an SCL entry that contains a USECLASS statement block for the method implementations.

USECLASS Statement

The USECLASS statement is similar to the CLASS statement, except that you cannot create class attributes or events in a USECLASS block. The ENDUSECLASS statement ends a USECLASS statement block.

In the following example, the methods are defined and stored in one SCL entry, whereas the method implementations are coded in a USECLASS statement block and are stored in a separate SCL entry (**sasuser.myclasses.oneCode.scl**). The following SCL code is stored in **sasuser.myclasses.one.scl**:

```

class sasuser.myclasses.one.class extends sashelp.fsp.object.class;
    _init: public method
        / (state='O',
           SCL='sasuser.myclasses.oneCode.scl');

        /* define attribute */
        Public num sum
        / (description='sum attribute');

        m1: method
            N:Num
            Return=Num
            / (SCL='sasuser.myclasses.oneCode.scl');
endclass;

```

The method implementations for the One class are stored in **sasuser.myclasses.oneCode.scl**:

```

useclass sasuser.myclasses.one.class;
    _init: public Method;
        _super();
        sum=0;
    endmethod;

    m1: method
        N:Num
        Return=Num;
        sum = sum + N;
        return (sum);
    endmethod;
enduseclass;

```

Note: The `_super()` routine is valid only inside CLASS or USECLASS blocks. The CALL SUPER routine can be used anywhere.

Using SCL to Instantiate Classes

Introduction

You can instantiate a SAS/AF class with the `_NEW_` operator, which combines the actions of the `LOADCLASS` function with the initialization of the object with its `_new` method. For example:

```
dcl sashelp.classes.librarylist_c.class libraries;
init:
  libraries = _new_ sashelp.classes.librarylist_c();
  call putlist(libraries.items, 'Libraries=', 1);
return;
```

You can use the `_NEW_` operator with the `IMPORT` statement so that you can refer to a class without having to specify its entire four-level catalog name. The `IMPORT` statement specifies a search path for `CLASS` entry references in an SCL program. You can also combine the `_NEW_` operator with the `DECLARE` (or `DCL`) statement for single-step declaration and instantiation:

```
import sashelp.classes.librarylist_c.class;
dcl librarylist_c document=_new_ librarylist_c();
```

Using Constructors

Introduction

The `_NEW_` operator also enables you to create an instance of a class and to run a class constructor. A *constructor* is a method that is automatically invoked whenever an object is instantiated. You can use a constructor to initialize the object to a valid starting state. The method that is used as the constructor of a class has the same name as the class. You can specify a signature with no arguments to override the default constructor (that is, its signature is `(V)`), or you can overload the constructor with signatures that use one or more arguments. You cannot, however, specify a return type.

Consider a class that is defined as follows:

```
class sasuser.test.Account
  extends sashelp.fsp.object.class;
  /* attributes */
  public num accountNumber;
  public num balance;

  /* constructor */
  Account: public method
    id:input:num;
    accountNumber=id;
    balance=0;
  endmethod;
endclass;
```

When an `Account` object is instantiated, the constructor assigns a specific number as the `accountNumber` and initializes the `balance` attribute to 0. For example:

```
import sasuser.test.Account.class;
dcl account newAccount = _new_ account(1234);
```

The `_NEW_` operator calls the account constructor method and passes a value 1234 to its numeric argument. This creates a new Account object that has 1234 as its `accountNumber` and a balance of 0.

You can also overload the constructor to accept a different number of arguments. Consider a subclass of the Data Set List Model class:

```
class sasuser.myclasses.ourData
  extends sashelp.classes.datasetlist_c.class;

  OurData: public method
    lib:char;
    /* library is an inherited attribute */
    if lib ne ''
      then library=lib; /* set library attribute */
  endmethod;

  OurData: public method
    lib:char
    level:num;
    /* library and levelCount are inherited attributes */
    if lib ne ''
      then library=lib; /* set library attribute */
    if level in (1,2)
      then levelCount=level; /* set levelCount attribute */
  endmethod;
endclass;
```

You can pass one argument to the `_NEW_` operator to call the constructor with one argument in its signature. In the following example, the constructor initializes the **library** attribute:

```
import sasuser.test.OurData.class;
dcl OurData table = _new_ OurData('sasuser');
```

You could also pass two arguments to the `_NEW_` operator to call the constructor with two arguments in its signature. In the following example, the constructor sets the **library** and **typeFilter** attributes:

```
import sasuser.test.OurData.class;
dcl OurData table = _new_ OurData('sasuser',1);
```

Because the **library** attribute of the table object is initialized in both cases, you can immediately query `table.items` to retrieve the list of SAS tables in the specified library. If **typeFilter** is 1, then the `items` list contains only the names of the tables and not the full two-level SAS name.

For complete information on the `_NEW_` operator and constructors, see the *SAS Component Language: Reference*.

Defining Constructors in the SCL for a Class

The Account class and OurData class examples above demonstrate how you can define and implement constructors for a class within a CLASS/ENDCLASS block. There are

several important items to remember when you define constructors in the SCL for a class:

- If you specify the Label metadata item in the method definition for the constructor, it must be the same as the name of the class.
- To override the default constructor for a class, add the **STATE='O'** metadata item to the method definition. For example, you could override the default constructor for a class named Document with the following code:

```
Document: public method
  /(State='O',
    Description='Override of the default constructor',
    SCL='sasuser.myclasses.Document.scl');
```

Note that the default constructor has a signature of ()V.

- If you want to use a method with the same name as the class and not have it function as a constructor, you must specify a **CONSTRUCTOR='N'** metadata item in the method definition. For example:

```
class sasuser.myclasses.Report
  extends sashelp.fsp.Object.class;
  /* constructor override with sigstring ()V */
  Report: public method
    /(State='O',
      Description='Constructor method',
      SCL='sasuser.myclasses.Report.scl');

  /* method with sigstring (C)V */
  Report: public method
    ch:char
    /(State='N',      /* optional */
      Description='Report method',
      SCL='sasuser.myclasses.OtherMethods.scl',
      Constructor='N');
endclass;
```

- By default, all constructors are stored in the CLASS entry as methods named `_initConstructor`. If the method definition specifies **CONSTRUCTOR='N'**, then the method is stored with the same name as the class. If you add constructors to a class that you define in SCL and save the SCL as a CLASS entry, the constructors appear in the Methods list of the Class Editor with the same name as the class. For example, if you opened the Report class defined above in the Class Editor, you would see two methods: an overridden method named Report and a new method named Report.

Defining Constructors Using the Class Editor

You can also use the Class Editor to override the default constructor or to add new constructors to a class, using the same processes that you would for any other method. However, there are several other items you must remember when working with constructors in the Class Editor:

- You can override the default constructor for a class that you are editing in the Class Editor by selecting the method with the same name as the class, then selecting **Override** from the pop-up menu.
- You can overload the class constructor by adding a new method with the same name and providing a signature other than ()V.

- When you edit the class constructor in the Class Editor, the value of the SCL Label field defaults to the name of the class. You cannot edit this value.
- Unlike defining a constructor in SCL, if you add a method with the same name as the class in the Class Editor, it is not stored on the class as a constructor, but rather as a standard method.
- If you save the class as an SCL entry from the Class Editor and view the SCL class definition code that is generated, the constructors appear as methods with the same name as the class. For example, if you edit a class named Document, override its Document method in the Class Editor, and save it as an SCL entry, the SCL entry includes the following code:

```
class sasuser.myclasses.Document
  extends sashelp.fsp.Object.class;

Document: public method
  /(State='O',
    Description='Override of the default constructor',
    SCL='sasuser.myclasses.Document.scl');

/* ...insert additional methods here... */
endclass;
```

See [“Defining Constructors in the SCL for a Class” on page 106](#) for more information.

Chapter 12

Managing Methods

Implementing Methods with SCL	109
Introduction	109
SCL and Overridden Methods	110
Using USECLASS Statement Blocks with Methods	111
Improving the Performance of SCL Code in Methods	112
Overriding Frame Methods to Augment SCL Labeled Sections	113

Implementing Methods with SCL

Introduction

You can implement methods for your components with SCL code. The SCL implementation for a method can be stored in three different places:

- the SCL entry that is identified as the Source Entry for the method in the Class Editor
- the SCL entry that is identified in the SCL method declaration statement of a CLASS block. In the following example, the SCL implementation for the m1 method is stored in the entry named **sasuser.myclasses.oneCode.scl**.

```
Class One extends sashelp.fsp.object.class;
  m1: public method
    n:num
    return=num
    / (SCL='sasuser.myclasses.oneCode.scl');
EndClass;
```

- the SCL class definition itself, where the method is both defined and implemented. For example:

```
Class One extends sashelp.fsp.object.class;
  m2: public method
    n:num
    return=num;
  dcl num total;
  total=n * 2;
  return total;
endmethod;
```

```

        /* ...insert additional methods here... */
    EndClass;

```

When writing methods, keep the following tips in mind:

- Method implementations always start with the METHOD statement and end with ENDMETHOD.
- Method names are not case-sensitive and can contain underscores, but not embedded blanks or other special characters. Methods that are supplied by SAS are named using a leading underscore, a lowercase first letter, and subsequent uppercasing of any joined word (such as `_setBackgroundColor`) to promote readability, but your methods do not have to conform to the SAS convention.

Note: Methods supplied by SAS include a leading underscore so that they can be differentiated from user methods. If you name a new method using a leading underscore, a warning message appears.

- Method calls must include all arguments that you have specified in the method signature.
- You must either declare any variable that you use within a METHOD block, or store the variable as a private attribute on the class. For example:

```

addToList: public method
    item:input:char
    aList:update:list;
    dcl num rc;          /* rc must be declared */
    rc=insertc(aList, item, -1);
    /* ...insert additional SCL statements here... */
endmethod;

```

You can use either the DECLARE statement or its abbreviated form DCL to declare local variables in a method.

SCL and Overridden Methods

You can write methods to override existing methods, but you should execute the method on the parent class with the `_super()` routine. The `_super()` routine determines the current method context and then executes the method of the same name on the parent class. The point at which you execute the parent method can significantly affect the behavior of the overriding method. For example, if the `_super()` call occurs before any other statements in the METHOD section, the inherited behavior is executed before the overridden code executes. In order to use `_super()`, you must be inside a CLASS or USECLASS block.

Some methods require you to invoke the inherited method in a particular order. Each of the following examples is assumed to exist inside an appropriate CLASS or USECLASS block:

- If you override an object's `_init` method, you must invoke the super `_init` method before any other processing:

```

init: public method;
    _super(); /* call super */
    /* ...insert additional code here... */
endmethod;

```

- If you override an object's `_bInit` method, you must invoke the super `_bInit` method before other processing:


```
bInit: public method;
    _super(); /* call super */
    /* ...insert additional code here... */
endmethod;
```

- If you override an object's `_term` method, you must invoke the `super _term` method as the last operation:

```
term: public method;
    /* ...insert additional code here... */
    _super(); /* call super */
endmethod;
```

- If you override an object's `_bTerm` method, you must invoke the `super _bTerm` method as the last operation:

```
bTerm: public method;
    /* ...insert additional code here... */
    _super(); /* call super */
endmethod;
```

Note: When overriding a method, you may not change its signature. If you require a different signature, you should overload the method.

Using USECLASS Statement Blocks with Methods

By using `USECLASS/ENDUSECLASS` statements around an SCL method block, you can use methods and attributes for the specified class without repeating the class identifier. Due to compile-time binding, the SCL compiler can distinguish between local variables and class attributes. Refer to the *SAS Component Language: Reference* for complete details on the `USECLASS` statement.

You can also use the `_super()` routine in the implementation of a method that is contained in a `USECLASS` statement block without having to specify the object identifier.

- Your method can execute the `_super()` routine without specifying a super or parent method. The SCL compiler assumes that you want to execute the method of the same name on the parent. For example, to override an `_init` method:

```
init: public method;
    _super(); /* run _init method on parent */
    /* ...insert additional code here... */
endmethod;
```

Note: So that any new properties you have added are saved with a class, be sure to save a `CLASS` entry before compiling the SCL entry that contains its method implementation(s).

If you override a method that has a signature of '(None)' and implement the method inside a `USECLASS/ENDUSECLASS` block, you must include a signature designation in the `METHOD` statement. For example, if you override the `_foo` method and `_foo` has a signature of '(None)', your SCL code could include

```
USECLASS mylib.mycat.myclass.class;
foo: method/(signature='N');
    /* ...insert additional code here... */
endmethod;
enduseclass;
```

Any method that is implemented within a USECLASS/ENDUSECLASS block must designate a signature.

Improving the Performance of SCL Code in Methods

Here are several tips to make the SCL code in your methods work more efficiently:

Be careful when you use recursion in methods.

You can use recursive calls such that a method can invoke itself. However, be sure to provide an exit case so that you do not recurse infinitely. Also be sure to invoke the inherited method with `_super()` and not with the form `object.method()`. Attempting to use a direct call to the method when you should use `_super()` results either in an infinite loop or in an out-of-memory condition.

Use the `_term()` method to delete objects.

If you programmatically create instances of objects in your SCL, you should always invoke the `_term()` method when your application no longer needs those objects. Objects that are no longer used are not automatically deleted while the application is running. Invoking the `_term()` method deletes the object and frees the memory that it occupies.

```
DCL sasuser.myclasses.myObj.class demoObj = _new_ myObj();
/* ...insert additional SCL statements here... */
demoObj._term();
```

Delete SCL lists that have been created by a method.

If any of your object's methods create new SCL lists, delete these lists with the `DELLIST` function. SCL lists take up memory and are not always automatically deleted. Leaving too many SCL lists open in memory can cause your application to run out of available memory.

Do not bypass SCL method-calling functions.

Always invoke the methods in the same manner, and do not bypass the method-calling functions that SCL provides. For example, consider a banking account class that has two methods, `deposit()` and `update()`. The `deposit()` method records a deposit, and the `update()` method updates a data set with the new account balance. You can implement these methods together in a single SCL entry, `ACCOUNT.SCL`:

```
/* ACCOUNT.SCL: methods for the ACCOUNT class */
update: public method;
/* ...SCL code to update a data set with the account information... */
endmethod;

deposit: public method
  amount:update:num;
/* ...code to process a deposit goes here... */
endmethod;
```

Since you want to perform an update after each deposit, you may need to invoke the update operation from the `DEPOSIT` code. You may be tempted to call this operation directly, either with a `LINK` statement or by using `CALL METHOD`:

```
/* ACCOUNT.SCL: methods for the ACCOUNT class */
deposit: method
  amount:update:num;
/* ...code to process a deposit goes here... */

/* this is the wrong way to do an update! */
```

```

link update;
endmethod;

or

/* ACCOUNT.SCL: methods for the ACCOUNT class */
deposit: method
    amount:update:num;
    /* ...code to process a deposit goes here... */

    /* this is also the wrong way to do an update! */
    call method('account.scl', 'update');
endmethod;

```

Both of these mechanisms may work when you first develop your application, but they violate basic principles of object-oriented programming. To understand why, consider what happens when someone decides to use your ACCOUNT class. They decide that your account class provides most of the functionality that they want, except that they want to record transactions in an audit trail. To do so, they override the update() method to also update the audit trail. If your deposit() method is implemented using either of the techniques presented above, then the new update() method is never called when a deposit is made.

The proper way to perform the update is to call the update() method:

```

/* NEW ACCOUNT.SCL: methods for the NEW ACCOUNT class */

USECLASS newaccount.class;
deposit: method
    amount 8;
    /* ...code to process a deposit goes here... */

    /* This is the correct way to do an update! */
    update();
endmethod;

```

Then, if someone overrides the update() method, your deposit() method automatically invokes that new update() method. Of course, the developers of the new update() method that adds an audit trail should also invoke your original update() code using `_super()`.

```

update: method;
    _super();
    /* ...SCL code to update a data set */
    /* with the account information... */
endmethod;
ENDUSECLASS;

```

Overriding Frame Methods to Augment SCL Labeled Sections

The `_initLabel`, `_mainLabel`, and `_termLabel` methods of the Frame class run the corresponding named section in the FRAME entry's SCL code. For example, the `_initLabel` method runs the INIT section of the FRAME entry. This means that you can override any of these methods and perform preprocessing and postprocessing with

respect to the corresponding label. For example, if you want to process some information both before and after the INIT section of a FRAME entry runs, you can override the `_initLabel` method of the Frame class and write your method similar to this:

```
USECLASS sasuser.myclasses.newclass.class
  INITLABEL: method;
    /* perform preprocessing */
    /* ...insert additional SCL statements here... */

    /* run the INIT section of the FRAME entry */
    _super();

    /* perform postprocessing */
    /* ...insert additional SCL statements here... */
  endmethod;
ENDUSECLASS;
```

The `_objectLabel` method of the Widget class is similar to the section label methods of the Frame class. The `_objectLabel` method runs an object's labeled section in the FRAME entry's SCL code. For example, if a FRAME entry contains two objects, `Textentry1` and `Textentry2`, the `_objectLabel` method for `Textentry1` runs the section named `Textentry1`. Overriding this method enables you to perform preprocessing and postprocessing with respect to the corresponding object's label.

When overriding a method, remember that you cannot change its name, scope, or signature.

Chapter 13

Managing Attributes

Specifying a Component's Default Attribute	115
Validating the Values of Character Attributes	116
Assigning an Editor to an Attribute	117
Introduction	117
Creating a Custom Attribute Editor	118
Adding a Custom Attributes Window to a Component	119
Assigning a Custom Access Method (CAM) to an Attribute	121
Introduction	121
CAM Naming Conventions	123
Avoiding Unnecessary CAM Execution	124
Avoiding CAM Recursion	124
Using the Refresh Attributes Event	125
Introduction	125
Example	126
Using List Attributes	127

Specifying a Component's Default Attribute

Every component has a `defaultAttribute` attribute that is inherited from the Object class. A component's `defaultAttribute` is set to the attribute whose value is most often needed from that component. For example, the `defaultAttribute` attribute for a combo box control is `selectedItem`, which is the item that is selected from the list that the combo box displays.

The `defaultAttribute` attribute also affects some behaviors of the Class Editor and the Properties window. Actions that prompt for an attribute are initially set to the value of a component's `defaultAttribute`. For example, when you specify an attribute link, the value of Link defaults to the value of the source component's `defaultAttribute` attribute. The `dragInfo` and `dropInfo` values that pass information between objects during drag and drop processes also use the value of the `defaultAttribute` attribute.

Many SAS classes, including most visual components, have values specified for `defaultAttribute`.

To specify or change the value of the `defaultAttribute` attribute for a particular class using the Class Editor:

1. Select the **Attributes** node, and then select the **defaultAttribute** attribute.
2. Right-click and select **Override** from the pop-up menu.
3. In the Initial Value field, select an attribute from the drop-down list. Note that the list contains only attributes that are currently defined on the class.
4. Save the class.

You can also specify the value of **defaultAttribute** at build time using the Properties window. Valid selections are limited to the attributes that are currently defined for the instance of the class.

Validating the Values of Character Attributes

Attributes in SAS/AF software enable you to perform data validation without having to add code to your frame SCL entries. The process behind attribute validation depends on the attribute type and on how you specify some of the attribute metadata.

You can assign a list of valid values to a character attribute. Valid values restrict the value of an attribute to a set of values that you specify for the attribute. The values are stored in the ValidValues metadata item for the attribute. The validation occurs when some action attempts to set the attribute value (that is, when the `_setAttributeValue` method runs). At build time, the validation is performed when you set the attribute value with the Properties window. At run time, the validation occurs when you assign a new value from SCL code. For example, specifying **Yes** and **No** as the valid values for an attribute ensures that only those two values are used at build time and run time.

To define valid values for a component's attribute using the Class Editor:

1. Select the **Attributes** node, and then select the character attribute for which you want to add valid values.

Note: If the attribute is inherited, you must override it by selecting **Override** from the pop-up menu.

2. In the Valid Values cell, enter the list of valid values. Use either a space or a comma to separate single-word items, and use a comma (,) to separate multiple-word items. For example,

Red Blue Green

or

Crimson Red, Midnight Blue, Forest Green

You can also use the Valid Values editor to add valid values. Click the ellipsis (...) button to open the Valid Values editor.

See the SAS/AF online Help for more information on adding valid values in the Class Editor. For more information on the ValidValues metadata item, see [“Attributes” on page 84](#).

Although the Class Editor displays the defined valid values in the combo box that is displayed when you select the Initial Values cell of an attribute, no validation is performed on the initial value that you select. It is the responsibility of the component developer to assign a valid value to the InitialValue metadata item.

You may want to use an SCL entry to perform programmatic validation or to match values against items in an SLIST entry. To assign an SCL or SLIST entry to perform validation for a character attribute, use the Class Editor to set Valid Values to the appropriate entry name. Note that you must preface the catalog entry name with a backslash (\) character.

For example, the `defaultAttribute` attribute that all components inherit from `sashelp.fsp.Object.class` has its ValidValues metadata item set to `sashelp.classes.defaultattributevalues.scl` to run the specified SCL entry when the value of `defaultAttribute` is set. The SCL entry contains the following code, which returns a list of all attributes that are defined on the object:

```
/* Include this ENTRY statement to process ValidValues */
entry list:list
    optional= objectId:object
    attributeName:char
    environment:char(2);
dcl num rc;
INIT:
    attributesList=makelist();
    objectID._getAttributes(attributesList, 'Y');
    do i=1 to listlen(attributesList);
        rc=insertc(list, nameitem(attributesList,i), -1);
    end;
    rc=dellist(attributeList, 'Y');
    rc=sortlist(list);
return;
```

The `defaultattributevalues.scl` entry executes when the `defaultAttribute` attribute of any object is set. For details on the ENTRY statement used in this example, see [“Creating a Custom Attribute Editor” on page 118](#).

You can also use a custom access method to process valid values for an attribute. For more information on custom access methods, see [“Assigning a Custom Access Method \(CAM\) to an Attribute” on page 121](#).

Assigning an Editor to an Attribute

Introduction

An editor is a SAS/AF catalog entry that provides assistance when a user tries to set an attribute's value. Typically, an editor is implemented as a FRAME entry that is defined as a dialog box. You can specify an editor to assign an attribute value, which can make build-time operations easier and can decrease the chance of input error. Editors are usually reserved for broad ranges, where valid values presented in a drop-down list would not accurately or completely represent valid input. For example, instead of making users remember and correctly input numeric RGB or hexadecimal color values, or even color names, you could assign the Color Editor (`sashelp.classes.colorEditor.frame`) to the attribute to simplify the selection of color values. An editor that is specified for an attribute is stored in the attribute's Editor metadata item.

To assign an editor for a component attribute using the Class Editor:

1. Select the **Attributes** node, and then select the attribute for which you want to assign an editor.

Note: If the attribute is inherited, you must first override it by selecting **Override** from the pop-up menu.

2. Select the Editor cell, and then click the ellipsis (...) button.
3. Specify the catalog entry that you want to use as the editor.

You can also use the Properties window to define an editor for a specific instance of a component. In the New Attributes dialog box that appears when you add an attribute in the Properties window, select the **Value/CAMs** tab, and then click the ellipsis (...) button next to the Editor field to add an editor.

SAS/AF software provides several ready-to-use editors that you can specify to modify attribute values in components that you develop. For a complete list of supplied editors, see the SAS/AF online Help.

Creating a Custom Attribute Editor

You can create your own frame or SCL entry to use as an attribute editor. The frame or SCL entry functions just like other frames or SCL programs, except that you must include an ENTRY statement with an OPTIONAL= option before the INIT label. The ENTRY statement can include the following arguments:

objectID

is the object identifier of the current object for entries that are invoked from the Properties window. For entries that are invoked from the Class Editor, the object identifier is 0.

classID

is the object identifier of the class that is used to create the current object for entries that are invoked from the Properties window. For entries that are invoked from the Class Editor, **classID** is the object identifier of the class that is currently loaded and displayed.

environment

is a two-character value that contains **CE** for entries invoked from the Class Editor and **PW** for entries invoked from the Properties window.

frameID

is the object identifier of the active frame that contains the object for entries that are invoked from the Properties window. For entries that are invoked from the Class Editor, **frameID** is 0.

attributeName

is a character variable (**char (32)**) that contains the name of the selected attribute.

attributeType

is a character variable (**char (83)**) that contains the type of the selected attribute, which can be **Character**, **List**, **Numeric**, **Object**, or the four-level name of a specific CLASS entry.

value

is either a character or numeric variable that can contain the value of the attribute.

mode

is a character variable (**char (1)**) that can be set to **E** if the attribute is editable.

For example, in the Font Editor that is provided in SAS/AF software, the complete SCL program for the editor is:


```

entry optional=
objectId:object classId:object environment:char(2)
frameId:object attributeName:char(32) attributeType:char(83)
value:char mode:char(1);

INIT:
  value=fontlist(", 'N', 1, 'Select a font.', 'N');
  if value ne ' ' then value = scan(value,1,' ');
return;

```

The program simply calls the SCL FONTLIST function, but by including the ENTRY statement, you can wrap that functionality inside an attribute editor.

You can specify just those arguments that you need to either process information in the editor or to return to the Class Editor or the Properties window. For example, you would include the **environment** parameter in the ENTRY statement if you needed to determine whether the entry was called when a user was modifying an attribute in the Class Editor or the Properties window.

Adding a Custom Attributes Window to a Component

An editor is used to set a single attribute value. By contrast, a custom attributes window (which is also known as a “customizer”) can be used to set attribute values for an entire component. You can add a custom attributes window to a class to provide an alternate way to set attribute information (that is, to replace the use of the Properties window for a specific class) or to provide another way to access specific attributes by appending the custom window to the Properties window.

You must first create a frame to serve as your custom attributes window. The SCL for the frame must contain an ENTRY statement. For example:

```

entry optional= objectId:object;
init:
  /* ...insert SCL for your frame... */
return;

```

In this example, the **objectId** argument that is passed represents the current object that is being edited. This object argument enables you to use dot notation in the SCL code that sets attribute values based on the selections a user makes in the custom attributes window.

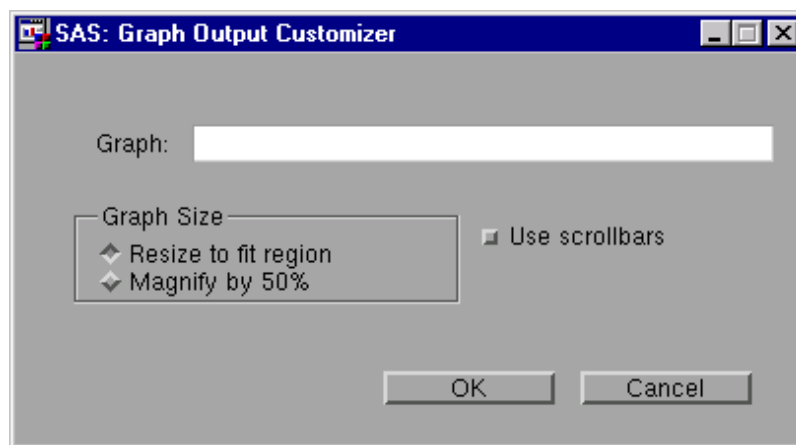
To specify a FRAME entry to use as a custom attributes window for a component:

1. In the Class Editor, select **View** ⇒ **Class Settings**.
2. Select **Use Custom Attributes Window**, and then enter the name of the frame you want to use as the custom window.
3. Select **Append** to include the custom attributes window with the Properties window, or select **Replace** to use the custom attributes window instead of the Properties window.
4. Click **OK**.

If you select **Append**, a user of your component can access your custom attributes window by selecting the row labeled (Custom) in the Properties window and clicking the ellipsis (...) button in the Value cell.

If you select **Replace**, a user of your component will see the custom attributes window instead of the Properties window. It is the responsibility of the component developer to account for all attribute values in the design of the frame and in the SCL code that processes the frame. The SCL for a custom attributes window that “replaces” the Properties window must still include the ENTRY OPTIONAL= statement described above.

Consider a component that is a subclass of `sashelp.classes.graphoutput_c.class`. You could create a frame named `sasuser.test.attrgrph.frame` that could provide a simple interface for setting values for the major attributes in the component. This frame could be appended to the Properties window.



The frame SCL must contain the following code:

```
/* sasuser.test.attrgrph.scl */
entry optional= objectID:object;
init:
  /* ...insert SCL that initializes the values... */
  /* ...of the controls in your frame... */
return;
  /* ...insert additional SCL code here... */

term:
  /* SCL must set the value of the attributes for objectID. */
  /* Assume that the following object names are the names */
  /* of the controls on the frame: */
  /* - graphName is the text entry control */
  /* - graphSize is the radio box control */
  /* - scrollbarsCheckBox is the check box control */

  /* set graph name */
  objectID.graph=graphName.text;
  /* set resizeMode or magnify */
  if graphSize.selectedIndex=1
    then objectID.resizeToFit='Yes';
    else if graphSize.selectedIndex=2
      then objectID.magnify=50;
  /* set the scrollbars option */
```

```

objectID.scrollbars=scrollbarsCheckBox.selected;
return;

```

To create the subclass of the graph output control that uses the custom attributes window:

1. Create a new class whose parent is `sashelp.classes.graphoutput_c.class` and whose description is **Graph Output Control with Customizer**.
2. Select **View** ⇒ **Class Settings** to view the Class Settings dialog box. Select **Use Custom Attributes Window**, and then enter `sasuser.test.attrgrph.frame` as the frame. Select **Append**, and then click **OK** to close the dialog box.
3. Save the class as `sasuser.test.CustomGraphOutput.class`, and then close the Class Editor.

If you add this class to a frame, you can then use either the custom attributes window or the Properties window to set the `graph`, `resizeToFit`, `magnify`, and/or `scrollbars` attributes. You can open the custom attributes window from the Properties window by selecting the row labeled (Custom) and clicking the ellipsis (...) button in the Value cell.

Assigning a Custom Access Method (CAM) to an Attribute

Introduction

A custom access method (CAM) is a SAS/AF software method that is associated with an attribute. The CAM is automatically executed to perform additional processing when the attribute's value is queried (with either dot notation or a direct `_getAttributeValue` call) or set (with either dot notation or a direct `_setAttributeValue` call). You can use the Class Editor to assign a CAM to a class attribute.

CAMs operate just like any other method, with a few special considerations:

- You should never call a CAM directly; instead, rely on the `_getAttributeValue` or `_setAttributeValue` methods to call it automatically.
- CAMs in SAS classes are protected methods to help enforce indirect execution via the `_getAttributeValue` and `_setAttributeValue` methods. Component developers are encouraged to create all CAMs as `Scope=Protected` methods to encourage the use of dot notation and to inhibit the direct calling of CAMs by developers who use the component.
- A CAM always has a single signature and should not 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 as a return code that indicates success or failure. For example, if a CAM is specified for a character variable, its signature is (C)N.
- The CAM must be defined for the object that contains the attribute that calls it. In other words, if `object.myAttribute` calls a CAM named `setcamMyAttribute`, then `setcamMyAttribute` must be a method on `object`.
- CAMs may be added only to New or Overridden attributes.

- Default names for custom access methods in SAS classes follow this format:
 - `_setcamAttributeName`
 - `_getcamAttributeName`

See “Attribute Values and Dot Notation” on page 88 for more information about CAMs and about the flow of control for the `_setAttributeValue` and `_getAttributeValue` methods.

For example, consider an object that has an attribute whose value is set to a SAS catalog. You can add a CAM to that attribute to determine whether the value is an existing SAS catalog as follows:

1. Create a new class whose parent is `sashelp.fsp.object.class` and whose description is **Document Object**.
2. Save the class as `sasuser.test.document.class`.
3. In the **Attributes** node of the Class Editor, select **New Attribute** from the pop-up menu and add a character attribute named `catalogToRead`.
4. In the Set CAM cell for the `catalogToRead` attribute, select the `setcamCatalogToRead` method from the drop-down list. When you are prompted to add the method, click **Yes**.
5. In the New Method dialog box, click **Source** and add the following code to the `sasuser.test.document.scl` entry:

```
USECLASS sasuser.test.document.class;
/* ...other methods can be defined here... */

setcamCatalogToRead: Protected method
  name:input:char(83)
  return=num;
  if name eq " " then return(0);

  if cexist(name) eq 0 then
  do;
    /* set the errorMessage attribute */
    errorMessage = 'ERROR: Catalog does not exist.';
    put errorMessage;
    return(1);
  end;
  else return(0);
endmethod;
enduseclass;
```

6. Compile and save the SCL, close the Source window and the New Method dialog box, and then close the Class Editor.

When the attribute is set via SCL (for example, `document.catalogToRead = 'sasuser.myclasses'`), the setCAM is called, verifying that the catalog is a valid SAS catalog. If the catalog that is specified is an invalid SAS catalog name or does not exist, an error message is generated and the setCAM program halts.

The same CAM could be expanded to do more than simple validity checking. For example, you could add processing to the CAM to read information from the selected catalog and to store that information in a list attribute named `contents` when the `catalogToRead` attribute is set:

```

USECLASS sasuser.test.document.class;
/* ...other methods can be defined here... */

setcamCatalogToRead: Protected method
    name=input:char(83)
    return=num;

dcl num rc;
if name eq " " then return(0);

if cexist(name) eq 0 then
do;
    /* set the errorMessage attribute */
    errorMessage = 'ERROR: Catalog does not exist.';
    put errorMessage;
    return(1);
end;
else do;
    /* use the catalog entry list model to read the catalog */
    /* and return the four-level name of each entry in it */
    dcl sashelp.classes.catalogentrylist_c.class catobj;
    catobj = _new_ sashelp.classes.catalogentrylist_c();
    catobj.catalog=name;
    rc=clearlist(contents);
    contents = copylist(catobj.items);
    catobj._term();
    return(0);
end;
endmethod;
enduseclass;

```

CAM Naming Conventions

CAM naming follows method-naming rules and conventions in several ways:

- CAM names can be up to 256 characters long. However, keep in mind that the USECLASS command can only accommodate method names up to 32 characters long. For easy maintenance, the method name is used as the SCL label in the METHOD statement.
- CAM names may contain only alphanumeric characters and the underscore character. The Class Editor does not support modifications to methods whose names contain special characters. Be sure to remove special characters from method names when converting legacy classes.
- In SAS classes, CAMs and other methods are named with a leading underscore. Users should not use leading underscores in names of methods or CAMs.

This format makes it easy to differentiate between CAMs and regular methods. For example, the method `_setBorderStyle` is easily distinguished from the associated CAM `_setcamBorderStyle`. Developers should follow the same format, without the leading underscore:

- `setcamAttributeName`
- `getcamAttributeName`

When you add a CAM to an attribute, the Class Editor automatically provides the CAM name in the format of `setcamAttributeName` or `getcamAttributeName`.

You should adhere to these naming formats to eliminate confusion between CAMs and regular methods. Doing so will ensure that CAMs are not accessed directly.

Avoiding Unnecessary CAM Execution

The ease with which dot notation enables you to get and set attribute values may lead you to write code similar to the following:

```
obj.color = obj2.text;
/* Assuming obj2.text contains a valid color value. */
```

This is perfectly valid code. However, `setCAMs` and `getCAMs` may be executing in the background each time code like this is run. Of particular concern is a `getCAM` that is associated with the right-hand side of an assignment.

Use caution when repeatedly evaluating a right-hand value, thus repeatedly running `_getAttributeValue` and any associated `getCAM`. The following code illustrates the potential inefficiency of repeatedly evaluating the right-hand side of an assignment:

```
do x = 1 to
listlen(someList);
  /* Assigns the values and runs obj2._getcamAttribute2 each time. */
  obj.attribute[x] = obj2.attribute2;
end;
```

Even if no `getCAM` is executing each time the value of `attribute2` is queried, a better way to implement such code would be to assign the value to a third variable and to use the variable in the loop, avoiding any `getCAM` code that runs for each iteration of the loop:

```
/* Assigns the value and runs _getcamAttribute2 once. */
dcl newVariable = obj2.attribute2;
do x = 1 to listlen(someList);
  /* Assigns the values without running _getcamAttribute2. */
  obj.attribute[x] = newVariable;
end;
```

Avoiding CAM Recursion

Do not set other attribute values from within `getCAM` code. It is possible to inadvertently create an infinite loop when setting the value of another attribute while inside the execution of a `getCAM` method. For example:

1. The `getCAMX` method is called when the value of the `X` attribute is queried.
2. In the course of its code, `getCAMX` sets the value of the `Y` attribute.
3. The `Y` attribute has a `setCAM` that queries the value of the `X` attribute, which starts the process again at step 1.

Using the Refresh Attributes Event

Introduction

When you develop a component, it may be designed such that its attributes are dependent on each other. In other words, a change to the value of one attribute can cause the value of one or more attributes to change as well. For example, the list box control has several attributes that are dependent on each other, two of which are `selectedIndex` and `selectedItem`. When a user sets the value for either attribute using the Properties window, the value of the other attribute is updated to keep them synchronized. The behavior that updates other attributes is typically implemented in the attribute's `setCAM`.

Consider what happens when a user sets the value of `selectedIndex` to 5. The `setCAM` for the `selectedIndex` attribute executes and sets the new value for the `selectedItem` attribute. If a user were to update `selectedItem`, its `setCAM` would update `selectedIndex`.

If any of these attributes are changed at build time while the Properties window is open, the “refresh Attributes” event can be sent to notify the Properties window that it should refresh its display to reflect the new attribute values. The Properties window is designed to listen for the “refresh Attributes” event.

It is the responsibility of the component developer to send the event. Typically, you can send this event in the `setCAM` following any code that changes the other attribute values. For example:

```
if frameID.buildTime='Yes' then
    _sendEvent('refresh Attributes', attributeList);
```

where *attributeList* is a list that contains information that the Properties window uses to update its displayed attribute values. For performance reasons, the event should be sent only if the frame is displayed at build time. There is no reason to update the Properties window at run time.

The format of the list represented by *attributeList* can be one of the following:

- a list with no named items that is passed with the event. All items in the list are processed for the currently selected objects. The items in the list correspond to the attributes whose values must be refreshed.

If a user selects multiple objects on a frame and updates a common attribute in the Properties window, the “refresh Attributes” event is sent to all selected objects.

Sample list: (`'borderColor'`, `'backgroundColor'`, `'borderStyle'`)

- a list with named items, where the named items represent the object identifiers that should be refreshed. A named item can point to an attribute or to a list of attributes that should be refreshed for the object that is represented by the identifier. If the named item is set to 0, a period (missing value), or (), then all of the attributes are refreshed for the object that is represented by the identifier.

When you use a list of object identifiers, the event can be sent to all valid objects on the frame. An object does not have to be selected.

For example, a list can include

```
(2324= 0, 3345=('selectedItem', 'items', 'comboBoxStyle'),
 7272='text')
```

All attributes would be refreshed for the object whose identifier is 2324; **selectedItem**, **items**, and **comboBoxStyle** would be refreshed for the object whose identifier is 3345; and **text** would be refreshed for the object whose identifier is 7272. Invalid attribute names would be ignored.

If no list parameter is passed with the event, then the entire list of attributes is refreshed for the selected object.

Note: It is the responsibility of the component developer to use the “refresh Attributes” event appropriately. For example, you should not refresh the attribute that the setCAM is updating, because the `_setAttributeValue` method call will not be completed during the CAM operation.

Example

As an example of using the “refresh Attribute” event due to a change in another attribute, consider a subclass of `sashelp.fsp.object.class` that has a character attribute named **testMode** and a numeric attribute named **newValue**. If the **testMode** attribute is set to **Test**, then the attribute value of **newValue** should be set to **999** and its editable status should be changed to **No**. If the value of **testMode** is set to anything other than **Test**, then **newValue** should be Editable=**Yes** and its value should remain unchanged.

To create the setCAM that sends the “refresh Attributes” event to implement the class:

1. Create a new class whose parent is `sashelp.fsp.object.class` and whose description is *Testing Object*.
2. Save the class as `sasuser.test.testing.class`.
3. In the **Attributes** node of the Class Editor, select **New Attribute** from the pop-up menu and add a character attribute named **testMode**. Add a numeric attribute named **newValue**.
4. In the Set CAM cell, select the `setcamTestMode` method from the drop-down list. When you are prompted to add the method, click **Yes**.
5. In the New Method dialog box, click **Source** and add the following code to the `sasuser.test.testing.scl` entry:

```
USECLASS sasuser.test.testing.class;
setcamTestMode: method
  attrval:char return=num;
  dcl list changeAttrList = makelist();
  dcl list refreshList = makelist();
  dcl num rc;
  /* set value and change editable status if value='Test' */
  if (upcase(attrval)='TEST') then do;
    rc = insertc(changeAttrList,'No',-1,'editable');
    _setAttributeValue('newValue',999);
  end;
  else
    rc = insertc(changeAttrList,'Yes',-1,'editable');

  _changeAttribute('newValue',changeAttrList);

  /* refresh datalist displayed in Property Window */
```



```

rc = insertc(refreshList, 'newValue', -1);
_sendEvent('refresh Attributes', refreshList);
rc = dellist(refreshList);
return (0);
endmethod;
enduseclass;

```

6. Compile and save the code, close the Source window, and then close the New Method dialog box.
7. Save the class as **sasuser.test.testing.class** and close the Class Editor.

You could then use the Testing class in a frame:

1. In the SAS Explorer, select a catalog, and then select **File** ⇒ **New** and specify a **FRAME** entry.
2. In the Components window, right-click and select **Add Classes** from the pop-up menu to add **sasuser.test.testing.class**. If the Components window is not displayed when the new frame appears in the Build window, then select **View** ⇒ **Components Window** to display the Components window.
3. Drag an instance of the Testing object and drop it onto the frame.
4. Select **View** ⇒ **Properties Window** and select the **testing1** component in the tree.
5. Test the CAM:
 - Change the value of **testMode** to **Test**. The value of **newValue** changes to **999** and the attribute is not editable.
 - Change the value of **testMode** to any value other than **Test**. The value of **newValue** is editable.

Using List Attributes

SCL list functions such as INSERTC and GETITEMN do not perform explicit “set” and “get” assignments. That is, when you use one of these SCL functions to manipulate a list attribute, the `_setAttributeValue` method is not invoked for the attribute. The value of the attribute might be changed, but the other actions that occur when an attribute value is changed in a `_setAttributeValue` call do not execute. The value cannot be validated, the “attributeName changed” event is not sent, and custom access methods do not run. Your SCL code must use explicit assignment operations, such as **object.attribute=value**, in order to invoke the expected attribute value setting behavior.

For example, consider a frame that contains a list box control named `listbox1`. If you want to write SCL that adds items to the list box, it might seem logical to write:

```

/* frame SCL */
dcl num rc;
init:
rc=insertc(listbox1.items, 'Red', -1);
rc=insertc(listbox1.items, 'Blue', -1);
rc=insertc(listbox1.items, 'Green', -1);
return;

```

However, because the `INSERTC` function does not invoke the `_setAttributeValue` method for `items`, the “items changed” event is never sent, and the list box is not updated on the frame.

Instead, you can use a list variable and set the `items` attribute to the value of the list. For example:

```
dcl list localList=makelist();
init:
    localList=listbox1.items;
    rc=insertc(localList, 'Red', -1);
    rc=insertc(localList, 'Blue', -1);
    rc=insertc(localList, 'Green', -1);
    listbox1.items=localList;
return;
```

The `_setAttributeValue` method runs for the `items` attribute when the dot notation call sets the value of `items`. The list box in the frame displays the updated list.

Chapter 14

Adding Communication Capabilities to Components

Introduction	129
Enabling Attribute Linking	130
Introduction	130
What Happens When Attributes Are Linked	130
Implementing Model/View Communication	133
Introduction	133
What Happens during Model/View Communication	133
Creating Your Own Models and Viewers	134
Enabling Drag and Drop Functionality	139
Introduction	139
What Happens during a Drag and Drop Action	140
Adding Drag and Drop Functionality to Your Components	142
Drag and Drop Example	145
Modifying or Adding Event Handling	148
Introduction	148
What Happens during Event Handling	149
Adding Events and Event Handlers to Your Components	149

Introduction

When you create components using the SAS Component Object Model (SCOM) framework, you can take advantage of four distinct processes that enable components to communicate with each other.

Attribute linking

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

Model/view communication

enabling a viewer (typically a control) to communicate with a model, based on a set of common methods that are defined in an interface.

Drag and drop operations

enabling information to be transferred from one component to another by defining “drag” attributes on one component and “drop” attributes on the other.

Event handling

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

This section describes how you can add these communication processes to your components.

Enabling Attribute Linking

Introduction

In SAS/AF software, attributes can facilitate communication between components. You can enable one component to automatically update the value of one of its attributes when the value of another component attribute is changed. In the SAS Component Object Model, this interaction is called “attribute linking,” and you can implement it without any programming.

As a component developer, you can decide whether the components you design support attribute linking. You can use the Class Editor to add attribute linking capabilities.

- To enable an attribute to receive its value via an attribute link, simply set the attribute's Linkable metadata item to “Yes.”
- To enable an attribute to function as a source attribute (that is, the attribute to which you link), you must set the attribute's SendEvent metadata item to “Yes.”

Most attributes on SAS classes are linkable.

What Happens When Attributes Are Linked

Introduction

During build time, you can use the Properties window to add attribute links. When you add an attribute link, the component's `_addLink` method runs. The `_addLink` method specifies the attribute to which you want to link, the object you are linking from, and the source attribute for the link.

Similarly, when an attribute link is removed, the `_deleteLink` method runs. The component's `_deleteLink` method specifies the name of the attribute whose link you want to delete, the source object, and the source attribute for the link.

Note: You could also write SCL code that uses `_addLink` or `_deleteLink` to programmatically add or delete an attribute link at run time.

The definitions of your component's attribute metadata determine what happens during attribute linking.

1. When a source attribute with `SendEvent=“Yes”` is modified, it sends an “attributeName Changed” event. The event passes an instance of `sashelp.classes.AttributeChangedEvent.class` that contains information about the event.
2. The target component has an event handler method with a single argument whose type is object. The argument receives the `AttributeChangedEvent` instance from the event.
3. The `_onAttributeChange` method queries the `AttributeChangedEvent` object for the name and the value of the attribute that has been changed, as well as for the object identifier of the source object. The `_onAttributeChange` method then invokes the `_setAttributeValue` method on the target attribute, which updates the value of the target attribute.

4. At build time, the `_onAttributeChange` method sends the “refresh Attributes” event. See “Using the Refresh Attributes Event” on page 125 for more information.

For details on the flow of control for the `_setAttributeValue` method, see “Attribute Values and Dot Notation” on page 88.

If you enable attribute linking, developers who use your components do not have to write any code to establish communication between components.

How the Attribute Changed Event Component Works

The Attribute Changed Event component

(`sashelp.classes.AttributeChangedEvent.class`) provides an object container for passing the information that is used in attribute linking. The information is stored in the following attributes of the Attribute Changed Event component:

`attributeName`

is the name of the changed attribute.

`value`

is the value of the changed attribute. This attribute is a complex attribute whose type is `sashelp.classes.type.class`, which contains the following attributes:

`type` represents the type of the attribute value (Character, Numeric, List, Object). Based on the value of `type`, you can query the appropriate “value” attribute.

`characterValue` stores the character value if `type` is “Character”.

`listValue` stores the SCL list if `type` is “List”.

`numericValue` stores the numeric value if `type` is “Numeric”.

`objectValue` stores the object identifier if `type` is “Object”.

`objectID`

is the identifier of the source object whose attribute was changed.

For complete information on the `AttributeChangedEvent` component, see the “SAS/AF Component Reference” in the SAS/AF online Help.

Example

Sometimes setting an attribute link between two attributes does not provide the complete behavior that your component or application needs. In these situations, you might want to define your own event handler to listen for an “attributeName Changed” event and to perform the desired behavior in the event handler method.

For example, consider a frame with a list box that contains the names of columns in a SAS table. A text entry control is also on the frame. As a user of the frame selects one or more columns in the list box, the text entry control displays a string that contains the list of selected columns, with blank characters separating the column names. The standard attribute linking functionality cannot handle this requirement because of the type mismatch of the attributes that are involved. (The list box stores selected columns as list items in its `selectedItems` attribute. The text entry control displays the text as a character string in its `text` attribute.) You must write code to convert the list of items in `selectedItems` to a concatenated character string that can be displayed in the text entry control.

The following example demonstrates how you can create a subclass of the text entry control to provide such functionality:

- Using the Class Editor, create a new class whose parent is `sashelp.classes.textentry_c.class` and whose description is **Smart Text Entry**.
- Save the class as `sasuser.myclasses.SmartTextEntry.class`.
- Right-click on the **Event Handlers** node and select **New Event Handler**. Add an event handler whose Event Generator is “Any Object (*)”. Set the Event Name to `selectedItems Changed` and the Method Name to `onSelectedItemsChanged`.
- When you are prompted to add the new method, click **Yes**. In the New Methods dialog box, click the ellipsis (...) button to modify the Signature field. In the Signature dialog box, click the **Add** button to add a new argument whose type is a specific class name, and then enter `sashelp.classes.attributechangedevent`. Click **OK**, and then click **OK** again to close the New Method dialog box and return to the Class Editor.
- Select the **Methods** node, and then select the new `onSelectedItemsChanged` method. Right-click and select **Source** from the pop-up menu, and then add the following SCL code:

```
useclass sasuser.myclasses.SmartTextEntry.class;
onSelectedItemsChanged: public method
    eventObj:sashelp.classes.attributeChangedEvent.class;
    dcl char(500) textstr,
        num i,
        list localList;
    localList = eventObj.value.listValue;
    do i = 1 to listlen(localList);
        textstr = textstr||' '||getitemc(localList,i);
    end;
    text=textstr;    /* set the text attribute */
endmethod;
enduseclass;
```

Compile and save the SCL entry.

- Save the class and close the Class Editor.

You can then use the new Smart Text Entry component in conjunction with a list box control. To create a frame that hosts the Smart Text Entry component:

- Create a new frame.
- Add the text entry subclass to the Components window by selecting **Add Classes** from its pop-up menu. Select or enter `sasuser.myclasses.SmartTextEntry.class`.

The class named Smart Text Entry appears in the Components window.

- Drag a list box control onto the frame. Open the Properties window and set the list box control's `selectionMode` attribute to “Multiple Selections”.
- Drag and drop a Variable List Model component onto the list box to establish a model/view relationship. In the Properties window, set the model's `dataSet` attribute to a valid SAS table such as `sashelp.prdsale`.

The list box should immediately be populated with the column names in the table.

- Drag and drop a Smart Text Entry component onto the frame. Resize the component as necessary.

6. Select **Build** ⇒ **Test** to test the frame.

As you make multiple selections from the list box, the text entry control is automatically updated to display a string of selected items. Because the list box control's `selectedItems` attribute was defined in the class with `SendEvent=Yes`, the `selectedItems Changed` event is sent each time the attribute value is changed. The event also includes an instance of the Attribute Changed Event component that contains information about the attribute and its value. The event handler on the Smart Text Entry component retrieves the new value of the Attribute Changed Event component that is passed to the method. The method then loops through the items in the list to create a string of column names that are displayed in the text entry.

Implementing Model/View Communication

Introduction

When you perform the object-oriented analysis for your application, you can separate the problem domain from the design of the application's user interface. The problem domain provides a perspective on what the application should do, including all business rules. Objects that represent the problem domain serve as models and are typically non-visual components. Viewers are components that display data to the user or provide access to the information in the model. To enable communication between the model components and viewer components, you can implement model/view communication.

Although models and viewers are typically used together, each is nevertheless an independent component. Their independence allows for customization, design flexibility, and efficient programming. The use of a model/view architecture also

- enables you to develop stable models that do not change much over time
- enables you to customize viewers based on the preferences of different users who might need to work with the same problem domains presented in the models
- simplifies application maintenance because the user interface is separated from the problem domain

As a component developer, implementing model/view communication can possibly require you to add a significant amount of SCL code. However, users of the models and viewers that you create do not have to do any programming to perform component communication.

Interfaces are designed to help implement model/view communication by providing a kind of relationship between the model and the viewer. An interface essentially defines the rules by which two components can communicate with each other. The interface that is used between a model and a viewer ensures that the viewer knows how and when to communicate with the model to access the data it needs.

What Happens during Model/View Communication

The following items describe the flow of control for model/view communication:

1. When the `model` attribute of a viewer is set, an event handler is established on the viewer to listen for the “contents updated” event to be sent from the model. The event handler is the `_onContentsUpdated` method. In addition, the `_setcamModel` method executes when the viewer's `model` attribute is set, both at build time and at

run time. The `_setcamModel` method includes a call to a method that is both implemented on the model and defined in the interface.

2. The “contents updated” event is sent by the model when one of the attributes in its `contentsUpdatedAttributes` attribute changes or when the model specifically sends the event.

The model's `contentsUpdatedAttributes` attribute contains the name of one or more other attributes. These attributes have been identified as critical components on the component. They affect the contents of the model, and the viewer must be notified when their values change. The “contents Updated” event passes the name of the changed attribute as an input argument to the `_onContentsUpdated` method.

3. The viewer's event handler, which is the `_onContentsUpdated` method, calls back to the model to retrieve updated information. The viewer is able to communicate with the model due to the methods defined in the interface.

For example, consider a frame that has a list box control that you want to use to present a list of standard color names (such as Red, Green, and Blue). The Color List Model component provides this list of names based on values that are supported by SAS software. You can establish a model/view relationship between a list box and an instance of the Color List Model component by dragging the model from the Component window and dropping it onto the list box in the frame.

At the point where the drop occurs on the list box, the SAS/AF classes have been designed to verify whether the two objects know how to communicate with each other via model/view communication using a common interface. If the model has a supported interface that matches a required interface on the viewer, the model is “attached” to the viewer. In this case, the color list model supports the `sashelp.classes.staticStringList` interface, and the list box requires the same interface, so a model/view relationship exists. Two types of processing occur once the model/view relationship is established:

1. The `model` attribute on the viewer is set to the name of the model, which executes the `setCAM` for the attribute (`_setcamModel`). In the list box/color list model example, the implementation of the `_setcamModel` method for the list box contains code that queries the model and retrieves a list of items using a `_getItems` call, which is a supported method in the interface. The CAM then sets the value of the viewer's `items` attribute to the list that is returned by `_getItems`.
2. When the “contents Updated” event is sent by the model, the viewer's `_onContentsUpdated` method executes. This method's implementation is similar to the `_setcamModel` in that it queries the model using methods supplied in the interface, and it retrieves the model's information to update the viewer. In the list box example, a `_getItems` call is used to retrieve the list of colors each time the model is updated.

Creating Your Own Models and Viewers

Introduction

To implement model/view communication, you need to examine the design of the model, the viewer, and the interface to set the appropriate actions. Consider the following steps when designing components for model/view:

1. Decide whether an existing interface provides the design you need.

If it does not, you can create a new interface.

2. Add the interface as a supported interface for your model.
3. Create or identify the key attributes on the model. Modify the model's `contentsUpdatedAttributes` attribute to include the key attributes.
4. Implement the methods for the model specified by the interface. Ensure that the method or methods query or update the key attributes that you identified.
5. Add the interface as required for the viewer.
6. Override the viewer's `_onContentsChanged` and `_setcamModel` methods. Typically, the implementation invokes methods that have been defined in the interface and implemented in the model to retrieve new information from the model. This new information is used to update attributes or to perform an action on the viewer. The `_onContentsChanged` method handles the “contents Updated” event, and the `_setcamModel` method retrieves the initial values from the model when the viewer is instantiated.

Defining a Model Based on the StaticStringList Interface

The most simple example of implementing model/view involves providing support for an existing interface. Using the steps outlined above, you can:

- Provide support for the methods defined in `sashelp.classes.staticStringList.interface` so that you do not have to create a new interface.
- Add the `_getItems` method and implement code that populates an `items` attribute on the model.
- Work with an existing viewer that requires the `staticStringList` interface, such as a list box.

To create the model:

1. Using the Class Editor, create a new class whose parent is `sashelp.fsp.Object.class` and whose description is **My Model**.
2. Save the class as `sasuser.myclasses.myModel.class`.
3. On the **Interfaces** node in the tree, right-click and select **New Interface** from the pop-up menu. Specify `sashelp.classes.staticStringList.interface` for the interface and set **Status** to **Supports**.

The Class Editor prompts you to add the methods that are defined in the interface. Click **Yes**.

Note: Although this example may not provide implementation for each method that is defined in the interface, it is a recommended practice to implement in your class all methods that are specified in the interface. You do not necessarily know which methods the viewer might invoke.

4. Right-click on the **Attributes** node and select **New Attribute**. Add an attribute named `items` and assign it as a List type. Save the class.
5. Select the **Methods** node. Select the `_init` method with the signature of `()V`, and then right-click and select **Override** from the pop-up menu. Select the `_getItems` method, and then right-click and select **Source** from the pop-up menu. Add the following SCL code to implement both methods in the entry named `sasuser.myclasses.myModel.scl` that is created for you:

```
useclass sasuser.myclasses.myModel.class;
/* Override of _init method */
init: public method (state="0");
```

```

dcl num rc;
dcl list temp=makelist();
_super();

rc=clearlist(items);

temp=items;
rc=insertc(temp, 'One', -1);
rc=insertc(temp, 'Two', -1);
rc=insertc(temp, 'Three', -1);
items=temp;
endmethod;

getItems: public method return=list;
return(items);
endmethod;
enduseclass;

```

Compile the code, and then close the Source window and return to the Class Editor.

6. Close the Class Editor.

Once the model is complete, you can create a frame and test the model with a control that requires the same interface. For example, you could use a list box control since it requires the `staticStringList` interface:

1. Create a new frame.
2. Add your new model to the Components window by selecting **Add Classes** from its pop-up menu. Select or enter `sasuser.myclasses.myModel.class`.

The class named My Model appears in the Components window.

3. Drag a list box control onto the frame, and then drag and drop a My Model component onto the list box to establish a model/view relationship. Regardless of the version of SAS that you are using, test the frame using the TESTAF command to see that the list box populate correctly.

The list box should immediately be populated with the values that you specified in the model's `_getItems` method.

Creating a Model/View Relationship Based on a New Interface

If existing interfaces do not provide the necessary relationship for a model and a viewer to communicate, you can create a new interface. This example demonstrates how you can:

- create a new interface.
- implement the methods defined in that interface for a new model class that supports the interface.
- provide support in a viewer for the required interface by overriding the viewer's `_onContentsUpdated` and `_setcamModel` methods.

To create the interface:

1. Use the Interface Editor to create a new interface whose description is **My Interface**.
2. Right-click and select **New Method** from the pop-up menu, and then add a method named `getColumnData` with a signature `()L`.

3. Close the Interface Editor and save the new interface as `sasuser.myclasses.MyInterface.intrface`.

Alternatively, you can use SCL to create an INTRFACE entry:

```
interface sasuser.myclasses.MyInterface;
  getColumnData: public method return=list;
endinterface;
```

You can use the SAVECLASS command to save the SCL code as an interface. For more information, see SAS Component Language Reference.

To create the model:

1. Using the Class Editor, create a new class whose parent is `sashelp.fsp.Object.class` and whose description is **Column Data Model**.
2. Save the class as `sasuser.myclasses.ColumnDataModel.class`.
3. In the **Interfaces** node of the Class Editor, right-click and select **New Interface** from the pop-up menu. Specify the model that you created above (`sasuser.myclasses.myInterface.intrface`) for the interface and set **Status** to **Supports**.

The Class Editor prompts you to add the method that you defined in the interface. Click **Yes**.

4. In the **Attributes** node of the Class Editor, add an attribute named `columnData` and assign it as a List type. Add an attribute named `table` and assign it as a Character type. Add a third attribute named `columnName` and assign it as a Character type.
5. Select the attribute named `contentsUpdatedAttributes`, and then select **Override** from the pop-up menu. Select the Initial Value cell, and then click the ellipsis button (...) to edit the values. In the dialog box, select `columnName` and `table`, and then click **OK**.
6. Save the class.
7. In the **Methods** node of the Class Editor, select the `getColumnData` method, and then right-click and select **Source** from the pop-up menu. Add the following SCL code to the entry named `sasuser.myclasses.ColumnDataModel.scl` that is created for you:

```
useclass sasuser.myclasses.ColumnDataModel.class;
getColumnData: public method
  return=list;
  dcl num rc dsid levels;

  /* reset the existing items attribute */
  rc=clearlist(columnData);

  /* open the SAS table specified in the table attribute */

  dsid = open (table);
  if dsid ne 0 then do; /* process if table exists */
    if varnum (dsid, columnName) > 0 then do;
      levels=0;
      rc=lvarlevel (dsid, columnName, levels, columnData);
      rc=revlist (columnDataModel);
    end;
  end;
end;
```

```

        rc=close(dsid);
        return(columnData);

    endmethod;
enduseclass;

```

Compile and save the SCL code, and then close the Source window and return to the Class Editor.

8. Close the Class Editor.

Next, you can create a subclass of the list box control that requires the MyInterface interface that you created:

1. Use the Class Editor to create a subclass of the List Box control (`sashelp.classes.listbox_c.class`). Name your class **My List Box**.
2. Select the **Interfaces** node in the Class Editor tree and add a new interface. Specify the `sasuser.myclasses.myInterface.interface` interface.
3. Save the class as `sasuser.myclasses.myListBox.class`.
4. In the **Methods** node of the Class Editor, select the `_onContentsUpdated` method, and then right-click and select **Override** from the pop-up menu. Select the `_setcamModel` method, and then right-click and select **Override** from the pop-up menu. Right-click and select **New Method** from the pop-up menu, and then add a method named `getModelData`. Right-click and select **Source** from the pop-up menu to add the following SCL code, which implements all three methods:

```

useclass sasuser.myclasses.myListBox.class;
getModelData: public method;
    /* This method retrieves the data from the model. */
    /* modelID is an attribute inherited from Object */
    /* that contains the identifier of the model when */
    /* the viewer's model attribute is set.          */
    items=modelID.getColumnData();
endmethod;

onContentsUpdated: public method
    colItems:char;
    getModelData();
endmethod;

setcamModel: protected method
    attributeValue:update:char
    return=num;
    _super(attributeValue);
    getModelData();
endmethod;
enduseclass;

```

Compile the SCL and save the entry.

5. Close the Class Editor.

Once the model is complete, you can create a frame and test the model with a control that requires the same interface:

1. Create a new frame.

2. Add your new model and the list box subclass to the Components window by selecting **Add Classes** from its pop-up menu. Select or enter `sasuser.myclasses.ColumnDataModel.class` and `sasuser.myclasses.myListBox.class`.

The Column Data Model and My List Box classes appear in the Components window.

3. Drag a My List Box onto the frame, and then drag and drop a Column Data Model component onto the list box to establish a model/view relationship.

Note that the list box is not populated until you set the `table` and `columnName` attributes on the model. Use the Properties window to set `columnDataModel`'s `table` attribute to a valid SAS table such as `sashelp.prdsale`. Set `columnDataModel`'s `columnName` attribute to a valid column name in the table. For example, `Region` is a column in the `sashelp.prdsale` table.

Enabling Drag and Drop Functionality

Introduction

You can add drag and drop functionality to your components. Drag and drop is a user interface action that involves dragging components or objects using a pointing device and dropping them over other objects on the frame, which results in some action. For example, a text entry control that contains the name of a SAS table could be dragged onto a data table component and then dropped to display the contents of the SAS table in the data table. Drag and drop also works between objects on different frames.

Components that can be dragged are called drag sites, and components that can receive a dropped object are called drop sites. When an object is dropped, data is passed to the drop site and an action occurs. The action is determined by:

- the form in which the data is passed, called the data representation
- the type of drop operation

The data representation is the form of data that a drag site is capable of transferring and a drop site is capable receiving. The data can be as simple as a string of text or as complex as an SCL list. SAS/AF components that are based on the SAS Component Object Model (SCOM) use

`characterData`

a generic representation to indicate when a character string is passed as the data

`numericData`

a generic representation to indicate when numeric values are passed as the data

Legacy objects support three default data representations, all of which provide data in a character string:

- `_DND_TEXT` for text
- `_DND_FILE` for an external file
- `_DND_DATASET` for a SAS table

Additional data representations enable you to drag items from the SAS Explorer and to drop them onto a frame. These representations include

- `EXPLORER_DATASET_NAME` for SAS tables. The data is provided in a character string that contains the two-level SAS table name, such as `sasuser.fitness`.
- `EXPLORER_CATALOG_NAME` for SAS catalogs. The data is provided in a character string that contains the two-level SAS catalog name, such as `sashelp.classes`.
- `EXPLORER_ENTRY_NAME` for a specific catalog entry. When the selection includes a single entry, the data is provided in a character string that contains the four-level entry name, such as `work.a.a.scl`. When the selection includes two or more entries, the data is provided in a list in which each item is a character string that contains the four-level entry name.
- `EXPLORER_MEMBER_NAME` for multiple selections from a SAS library. When the selection includes a single entry, the data is provided in a character string that contains a two-level member name for catalogs or a three-level member name for SAS tables (including those of type VIEW). When the selection includes two or more members, the data is provided in a list in which each item is a character string. Note that multiple selections of SAS tables or SAS catalogs are sent in an `EXPLORER_MEMBER_NAME` data representation.

If you create a component based on SCOM architecture, you can define your own data representations if `characterData` and `numericData` do not meet your needs.

A data representation is essentially a “verbal contract” between two components. As a component developer, you must name the data representation. For example, the `characterData` representation states that whatever component uses this as its drag representation, the drag site will send a valid character value as the data. The drop site then expects to receive a character value and can react accordingly.

It is recommended that component developers fully document their data representations so that other developers can know how to use them. See [“How the Drag and Drop Component Works” on page 144](#) for more information about the data representation.

Drop operations define actions that are performed on the data representation:

Copy

Data is provided with no post-processing.

Link

Some mechanism synchronizes the source and destination.

Move

Data is provided and the source is removed.

Copy is the default.

Drag and drop operations in SAS/AF software have the following limitations:

- You cannot drag objects outside the SAS environment.
- Slider and scroll bar components do not support drag and drop operations.
- The behavior of drag and drop operations may vary according to the host environment.

What Happens during a Drag and Drop Action

SAS/AF software performs several operations to enable drag and drop functionality between two components. When the object is instantiated, SAS/AF checks to see whether the object's `dragEnabled` or `dropEnabled` attributes are set to “Yes.” If either of these attribute values is “Yes”, then:

1. the object is queried for its respective **dragInfo** or **dropInfo** attribute values.
 - For drag sites, SAS/AF reads the value or values of the **dragInfo** attribute's **dataRepresentation** and invokes the **_addDragRep** method to register each drag data representation that is supported by the object.
 - For drop sites, SAS/AF reads the value or values of the **dropInfo** attribute's **dataRepresentation** and invokes the **_addDropRep** method to register each drop data representation that is supported by the object.
2. the object is queried for its **dragOperations** and/or **dropOperations** settings.
 - For drag sites, the **_addDragOp** method is invoked to register the supported drag operations for the object.
 - For drop sites, the **_addDropOp** method is invoked to register the supported drop operations for the object.

SAS/AF permits a drop action to occur between two objects only when the two objects have a shared data representation and a shared operation. For example, consider an object named **listbox1** that is enabled as a drag site using the **characterData** representation. A second object named **listbox2** is enabled as a drop site using the **characterData** representation. A user can successfully complete a drag and drop action between the two objects if they share a common operation such as “Copy.” Objects that do not have a matching data representation and a matching operation will not permit a drop action to occur.

When a valid drag and drop action occurs between two objects, SAS/AF takes the value of an attribute on the drag site and sets it as the value of another attribute on the drop site. Several methods are invoked automatically, passing an instance of **sashelp.classes.DragAndDrop.class**:

1. The **_startDrag** method is invoked on the drag site. An instance of the Drag and Drop component is created.
2. The **_respondToDragOnto** method is invoked on a drop site as the cursor passes over a valid drop site.
3. The **_respondToDragOff** method is invoked on a drop site as the cursor moves off of a valid drop site.
4. The **_getDragData** method is invoked on the drag site after the user has released the drag item on a valid drop site. The implementation of this method prepares the data that is passed between the drag site and the drop site. It also sets the values of the appropriate attributes (including **attributeName** and **attributeValue**) on an instance of the Drag and Drop component before passing the object. The default implementation of **_getDragData**
 - determines whether an **attributeName** is defined for the associated **dragInfo** data representation that was selected for the drop
 - retrieves the current value of the named attribute and sets its name on the drag and drop object's **attributeName** attribute
 - retrieves the current value for that attribute and sets the value of the drag and drop object's **attributeValue** attribute

If no **attributeName** is specified as part of the **dragInfo** data representation, then you can override the **_getDragData** method to set the value of the drag and drop object's **attributeValue** attribute.

5. The **_validateDropData** method is invoked on the drop site to perform any validation that may be required.

6. The `_completeDrag` method is invoked on the drag site to complete any processing.
7. The `_drop` method is invoked on the drop site. The default implementation of the `_drop` method
 - retrieves the `attributeName` from the drop site's `dropInfo` attribute based on the selected data representation
 - sets the value of the attribute identified in `attributeName` using the `attributeValue` that was passed in the drag and drop object

If no `attributeName` exists on the drop site, you can override the `_drop` method and implement the desired drop behavior.

Adding Drag and Drop Functionality to Your Components

Defining Drag and Drop Properties

For most of your applications, you will likely find it sufficient to implement drag and drop functionality simply by setting attribute values.

You can define drag and drop properties for any component that is a subclass of the `Widget` class or the `Frame` class. To implement drag and drop, you define the drag properties of the component that you designate as a drag site, as well as the drop properties of another component that you want to behave as a drop site. A component can act as both a drag site and a drop site.

The following attributes must be defined for drag sites. You might want to review these attribute settings if you encounter problems with a drag site, or if you want more information on a drag site's behavior.

dragEnabled

sets the state that determines whether the object can be dragged when selected. In many cases, setting this attribute to “Yes” is all that is needed to implement a drag site that uses default values for `dragInfo` and `dragOperations`.

dragInfo

sets the information that defines what information is transferred from the object. The `dragInfo` attribute is a list that includes a named item for each drag representation:

- the `dataRepresentation`, which is a name that describes the type of data that a drag site is capable of transferring. The data can be as simple as a string of text or as complex as an SCL list.
- the `attributeName`, which is the name of the attribute whose value is passed to the drop site. (The drop site value is specified in the `dropInfo` attribute.)

A component becomes a drag site when its data representation is defined. More than one data representation can be defined for the drag site. When a component is dropped onto another component, the system checks through the list of representations for each component and chooses the first matching representation. This matching representation enables the drag site to format the data in the required representation. The drop site is then given its data, which it processes accordingly.

dragOperations

enables the object to indicate the action that occurs following the drag: Copy, Move, and/or Link. By default, an object allows the COPY action.

The following attributes must be defined for drop sites. You might want to review these attribute settings if you encounter problems with a drop site, or want more information on a drop site's behavior.

dropEnabled

sets the state that determines whether the control can serve as a drop site. In many cases, setting this attribute to **Yes** is all that is needed to implement a drop site using default values for **dropInfo** and **dropOperations**.

dropInfo

sets the information that defines what information is transferred to the object. The **dropInfo** attribute is a list that includes a named item for each drop representation:

- the **attributeName**, which is the name of the attribute whose value is set by the value of **attributeName** in the **dragInfo** attribute. (The `_setAttributeValue` method is called on the drop site's attribute, using the value of the attribute specified in the **dragInfo** attribute.)
- the **dataRepresentation**, which is a name that describes the type of data that a drop site is capable of receiving. The data can be as simple as a string of text or as complex as an SCL list. It is compared to the drag site's **dataRepresentation**. If a match exists, the drop action occurs.

More than one data representation can be defined. The last representation defined is given the highest priority.

dropOperations

defines the drag actions that the drop site can handle: Copy, Move, and/or Link. By default, an object allows the Copy action. In addition to the drag actions, you can specify the associated method that runs when the drop occurs, as well as the pop-up menu text that is displayed for non-default drag and drop processing.

Overriding Drag and Drop Methods

SAS/AF software implements the drag and drop process by automatically executing several methods. Although you can override these methods to add functionality, you cannot directly call them from an SCL program.

Table 14.1 Drag and Drop Methods to Override

Override this method...	To...
<code>_startDrag</code>	change the appearance or other state of the drag site when the drag begins. The code that you add runs when the user begins dragging from the drag site.
<code>_respondToDragOnto</code>	change the appearance or other state of the drop site as the dragged item passes over it. For example, you could change the drop site's color to indicate that a drop action is valid.
<code>_respondToDragOff</code>	change the appearance or other state of the drop site as the dragged item moves off of the drop site. For example, you could change the drop site back to its normal appearance if it was changed by <code>_respondToDragOnto</code> .
<code>_getDragData</code>	prepare the data that you want to pass to the drop site by setting attributes of the drag and drop object that is passed between all drag and drop methods.

Override this method...	To...
<code>_validateDropData</code>	perform any validation that you require before the drop action occurs. For example, you can verify the range of data that is passed and then set the <code>completeDrag</code> attribute of the drag and drop object to either “Yes” or “No” to indicate whether the drop can occur. If <code>completeDrag</code> is set to “No,” then the drop is cancelled, the <code>_drop</code> method does not run, and no attribute values are set on the drop site.
<code>_completeDrag</code>	complete processing on the drag site. For example, consider two list boxes on a frame that have the same data representation and a “Move” operation. When the item is dragged from the first list box and dropped onto the second, the “Move” operation indicates that the item should be removed from the first list box. You can override <code>_completeDrag</code> to implement the “Move” operation and delete the item from the drag site. Note: The <code>_completeDrag</code> method runs even if the drag and drop object's <code>completeDrag</code> attribute is set to “No.” If necessary, check the value of this attribute before executing the code.
<code>_drop</code>	process the drop action. This method executes only if the <code>completeDrag</code> attribute of the drag and drop object is set to “Yes.” Typically, you override this method to perform some action other than setting the appropriate attribute value on the drop site. For example, you may want to display a message box that confirms the success of the drag and drop action.

How the Drag and Drop Component Works

All drag and drop methods contain signatures that accept an object as their only argument. This object is an instance of the Drag and Drop Component (that is, `sashelp.classes.DragAndDrop.class`). It provides a container for the standard drag and drop information that is passed between a drag site and a drop site. The information is stored in the following attributes of the Drag and Drop component:

`attributeName`

is the name of the attribute that is passed from the drag site to the drop site.

`attributeValue`

is the value of the attribute that is passed from the drag site. This attribute is a complex attribute whose type is `sashelp.classes.type.class`, which contains the following attributes:

`type` represents the type of the attribute value (Character, Numeric, List, Object). Based on the value of `type`, you can query the appropriate “value” attribute.

`characterValue` stores the character value if `type` is “Character.”

`listValue` stores the SCL list if `type` is “List.”

`numericValue` stores the numeric value if `type` is “Numeric.”

`objectValue` stores the object identifier if `type` is “Object.”

For example, your method override could contain code as follows:

```
drop: public method
    dndObj: input:object;
```

```

if dndObj.attributeValue.type = 'Character' then
  do;
    /* Retrieve dndObj.attributeValue.characterValue, */
    /* then add any other code. */
  end;
else if dndObj.attributeValue.type = 'Numeric' then
  do;
    /* Retrieve dndObj.attributeValue.numericValue, */
    /* then add any other code. */
  end;
/* and so forth... */
endmethod;

```

completeDrag

is a character attribute that you can optionally set to indicate whether a drop can successfully occur. For example, you can perform validation in an override of the `_validateDropData` method, and you can cancel the drop action by setting this attribute to “No”. You can also query this value in an override of the `_completeDrag` method to verify whether the drop will actually occur before performing some action on the drag site.

dataOperation

is the selected drag operation; corresponds to one of the operations specified in the **dragOperations** attribute that is set on the drag site.

dataRepresentation

is the form of the data that is passed between sites; corresponds to the particular list item in the **dragInfo** attribute that is set on the drag site

dragSiteID

is the object identifier of the drag site

dragSiteLocation

indicates whether the drag started inside or outside of the current frame

XLocation

specifies the x location of the drag site or drop site

YLocation

specifies the y location of the drag site or drop site

For components that are based on the SCOM architecture, the data specified in the data representation (such as `characterData` or `EXPLORER_MEMBER_NAME`) is stored in the **attributeValue** complex attribute of the Drag and Drop component. If one item is dragged, then the data is stored in the **characterValue** attribute of the `attributeValue` object. If two or more items are dragged, then the data is stored in the **listValue** attribute as separate character items.

For complete details on the Drag and Drop component, see “SAS/AF Component Reference” in the SAS/AF online Help.

Drag and Drop Example

To demonstrate how drag and drop functionality works, consider an example that defines two subclasses of the list box control. The first subclass defines an object that is a drag site. The second defines an object that is a drop site. (Of course, it is entirely possible to have a single object that is both a drag site and a drop site.)

To create the first list box subclass:

1. Using the Class Editor, create a new class whose parent is `sashelp.classes.ListBox_c.class` and whose description is **Source List Box**.
2. Select the **Attributes** node, and then set the following values for attributes:
 - Select the **dragEnabled** attribute, and then right-click and select **Override** from the pop-up menu. Set the Initial Value to **Yes**.
 - Select the **dragOperations** attribute, and then right-click and select **Override** from the pop-up menu. Click the ellipsis (...) button in the Initial Value field. In the dragOperations dialog box, deselect the **Enabled** check box for **Default Copy Operation** and then select the **Enabled** check box for **Default Move Operation**. Click **OK** to return to the Class Editor.
3. Save the class as `sasuser.test.SourceListBox.class`.
4. Select the **Methods** node, and then select the second `_completeDrag` method in the list (with the `O:SASHELP.CLASSES.DRAGANDDROP.CLASS` signature). Right-click and select **Override** from the pop-up menu, and then select **Source** from the pop-up menu. Add the following SCL code to `sasuser.test.SourceListBox.scl`:

```
USECLASS sasuser.test.SourceListBox.class;

/* Override of the _completeDrag method */
completeDrag: method dndobj:sashelp.classes.draganddrop.class;
  dcl num rc;
  /* If the rep is one that is not understood, call super. */
  if ( upcase(dndobj.dataRepresentation) = 'CHARACTERDATA' ) then
  do;
    /* Check the status of the completeDrag attribute. */
    if ( ( upcase(dndobj.completeDrag) = 'YES' ) AND
        ( upcase(dndobj.dataOperation) = 'MOVE' ) ) then
    do;
      /* Remove the selected item from the items list. */
      if (selectedIndex ^= 0 ) then
      do;
        /* Delete the item from the items attribute. */
        rc = delitem(items, selectedIndex );

        /* Set the items attribute equal to itself to */
        /* update this list box. */
        items = items;
      end;
    end;
  end;
else
  _super( dndobj );
endmethod;
enduseclass;
```

5. Compile and save the code, and then close the Source window.
6. Close the Class Editor.

To create the second list box subclass:

1. Using the Class Editor, create a new class whose parent is `sashelp.classes.ListBox_c.class` and whose description is **Target List Box**.
2. Select the **Attributes** node, and then set the following values for attributes:
 - Select the **dropEnabled** attribute, and then right-click and select **Override** from the pop-up menu. Set the Initial Value to **Yes**.
 - Select the **dropInfo** attribute, and then right-click and select **Override** from the pop-up menu. Click the ellipsis (...) button in the Value field. In the Drop Info dialog box, click **Add**, and then set the Drop Attribute to a blank value and the Drop Representation to **CHARACTERDATA**.
 - Select the **dropOperations** attribute, and then right-click and select **Override** from the pop-up menu. Click the ellipsis (...) button in the Value field. In the dropOperations dialog box, deselect the **Enabled** check box for Default Copy Operation and then select the **Enabled** check box for **Default Move Operation**.
3. Save the class as `sasuser.test.TargetListBox.class`.
4. Select the **Methods** node, and then select the second `_drop` method in the list (with the `O:SASHELP.CLASSES.DRAGANDDROP.CLASS` signature). Right-click and select **Override** from the pop-up menu. Select the second `_validateDropData` method in the list (with the `O:SASHELP.CLASSES.DRAGANDDROP.CLASS` signature). Right-click and select **Override** from the pop-up menu, and then select **Source** from the pop-up menu. Add the following SCL code to implement both methods in `sasuser.test.TargetListBox.scl`:

```
USECLASS sasuser.test.TargetListBox.class;

/* Override of the _validateDropData method */
validateDropData: method dndobj:sashelp.classes.draganddrop.class;
  /* If the rep is one that is not understood, call super. */
  if ( upcase(dndobj.dataRepresentation) = 'CHARACTERDATA' ) then
    do;
      /* Ensure that the type is 'Character' and that the */
      /* value is not blank. If either of these checks */
      /* fail, then do not let the drop happen. */
      if ( ( upcase(dndobj.attributeValue.type) ^= 'CHARACTER' ) OR
          ^length( dndobj.attributeValue.characterValue ) ) then
        dndobj.completeDrag = 'No';
    end;
  else _super( dndobj );
endmethod;

/* Override of the _drop method */
drop: method dndobj:sashelp.classes.draganddrop.class;
  dcl num rc;
  /* If the rep is one that is not understood, call super. */
  if ( upcase(dndobj.dataRepresentation) = 'CHARACTERDATA' ) then
    do;
      /* Ensure that the attributeValue is the correct type. */
      /* If so, then insert it at the end of the items list. */
      if ( upcase(dndobj.attributeValue.type) = 'CHARACTER' ) then
        do;
          dcl num rc;
          rc = insertc( items,dndobj.attributeValue.characterValue, -1 );
```

```

        /* Set the items attribute equal to itself in order to */
        /* update this list box.                               */
        items = items;
    end;
end;
else _super( dndobj );
endmethod;
enduseclass;

```

5. Compile and save the code, and then close the Source window.
6. Close the Class Editor.

You could then use two classes in a frame:

1. In the SAS Explorer, select **File** ⇒ **New** then select a FRAME entry.
2. In the Components window, right-click and select **Add Classes** to add **sasuser.test.SourceListBox.class** and **sasuser.test.TargetListBox.class**. If the Components window is not displayed when the new frame appears in the Build window, then select **View** ⇒ **Components Window** to display it.
3. Drag an instance of the Source List Box object and drop it onto the frame, and then drag an instance of the Target List Box object and drop it onto the frame.
4. Select **Build** ⇒ **Test** to test the frame. Drag an item from the source list box and drop it onto the target list box.

In the Target List Box class, the `_validateDropData` method verifies that the data representation is `CHARACTERDATA`. Its `_drop` method queries the `attributeValue` attribute that is passed in the drag and drop object and adds the value to its `items` attribute. Finally, the `_completeDrag` method on the Source List Box verifies that a drop has successfully occurred by querying the `completeDrag` attribute on the drag and drop object. It then removes the item from the list of items displayed in the list box to complete the `MOVE` action.

Modifying or Adding Event Handling

Introduction

Events provide flexible, loosely coupled communication between objects. The SAS/AF event model is implemented through global event registration, which means that different components can listen to and provide handlers for events that are generated by other objects in the current application, objects in any other SAS/AF application that is also running, or the SAS session itself.

Events notify applications when a resource or state changes. Event handlers manage the corresponding response to any changes. 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.

SAS/AF software supports system events, which can include user interface events (such as mouse clicks) as well as “attributeName Changed” events that occur when an attribute value is updated. For system events, the value of the State metadata item is “S.” For “attributeName Changed” events, the component must have the `SendEvent` metadata

item for the attribute set to “Yes.” See [“Enabling Attribute Linking” on page 130](#) for details.

SAS/AF software includes a SAS Session component that provides access to changes made within the SAS environment. For example, SAS sends a system event when a new libref has been assigned or when a new catalog entry has been added. You can create an instance of the SAS Session component and implement handler methods for the events you want to process. For complete information on the SAS Session component, refer to the SAS Session component topic in the SAS/AF online Help.

What Happens during Event Handling

The SAS/AF event model is straightforward: objects “listen” for events that are sent by the system, by other objects, or by the object itself. If the object has an event handler that is defined to perform some action, then that action occurs when the object receives notification of the event.

Adding Events and Event Handlers to Your Components

SAS/AF software also supports user-defined events, which can be registered to specific classes as needed and can be inherited by subclasses. You can use the Class Editor to add user-defined events and event handlers for a class, or you can use the Properties window to add events and event handlers for an instance of a class. For details on how to work with events and event handlers, see [“Adding an event,”](#) [“Adding an event handler,”](#) and [“Working with events and event handlers in the Class Editor”](#) in the SAS/AF online Help.

You can also add events and event handlers programmatically. To do so, your SCL code should use the `_addEventHandler` method to indicate that an object should be notified when a certain event occurs and to specify which method should be invoked as the event handler. You would also invoke the `_sendEvent` method on the appropriate object to send the event and to run the event handling method that you specify in the `_addEventHandler` call.

The methods that support event communication are part of the Object class. In addition to `_sendEvent` and `_addEventHandler`, you can use `_deleteEventHandler` to indicate that an object should no longer be notified of a particular event.

For more information about these methods, refer to the Object class in the SAS/AF online Help.

Chapter 15

Deploying Components

Introduction	151
Managing Classes with Resources	152
Introduction	152
Synchronizing Resources	153
Analyzing Resources	153
Merging Resources	154
Renaming Class Catalogs and Libraries in a Resource	154
Making a New Class Available for Use	154
Introduction	154
Associating Resources with FRAME Entries	155
Modifying the Components Key of the SAS Registry	156
Generating Class Documentation with GenDoc	156
Introduction	156
Tips for Writing Meaningful Metadata Descriptions	157
What the GenDoc Utility Creates	158
Making Class Documentation Available	159

Introduction

After you finish designing and implementing your components, you need to consider how applications developers will use them. In many cases, it is helpful to create specific class libraries for your organization or for a specific project. SAS/AF software uses RESOURCE entries to organize and manage classes.

In general, you can follow these steps to make your components available to other developers:

1. Create a RESOURCE entry to group your components together, based on how developers will use them. See [“Managing Classes with Resources” on page 152](#).
2. Move classes from your component development and testing location to the library and catalog that are being used for applications development. Update the SCL source as necessary. See [“Renaming Class Catalogs and Libraries in a Resource” on page 154](#).
3. Instruct applications developers to add the resource that contains your components to the Components Window. See [“Making a New Class Available for Use” on page 154](#).

4. (Optional) Use the experimental GenDoc utility to generate documentation for your components. Copy the documentation files to the appropriate directory, and instruct your component users to update their HELPLOC system option to include a path to this directory. See [“Generating Class Documentation with GenDoc”](#) on page 156.

Managing Classes with Resources

Introduction

A RESOURCE entry, or simply resource, stores information about a set of classes. This information controls the classes that can be instantiated by a frame when that frame is initialized. Therefore, when you browse, edit, or execute a frame, it must be able to access the RESOURCE entry that was used when the frame was created.

SAS/AF software also uses resources to load classes efficiently. When a resource is loaded with a frame, SAS performs a single catalog I/O operation to load all appropriate class information into memory. If you were to instantiate each class at run time using separate LOADCLASS or `_NEW_` functions, SAS would perform a catalog I/O operation for every class.

The SAS/AF classes are stored in `sashelp.fsp.AFComponents.resource`, and the SAS/AF legacy classes are stored in `sashelp.fsp.build.resource`.

Organizing and manipulating resources and the classes that they contain is an important part of project management. You can create custom RESOURCE entries to

- make your components available for use in a specific project or application
- make your components available to specific groups of developers
- arrange related classes in a manner that makes sense for your needs
- reduce the number of classes that a frame is required to load by removing classes that you know are not needed

Multiple resources help you maintain and organize class libraries for development in SAS/AF software. For example, you could use the RESOURCE entry containing the standard classes provided by SAS (`sashelp.fsp.AFComponents.resource`), the resource containing SAS legacy classes (`sashelp.fsp.build.resource`), a resource entry containing classes that were developed for a particular project, and a resource containing classes that you are developing. When you are ready to deploy the components, however, you could create a single resource that contains only those classes used by the application.

You can use the Resource Editor to create, organize, and manipulate resources, as well as the classes that the resources contain. The Resource Editor is invoked when you open an existing RESOURCE entry or when you create a new RESOURCE entry.

You can also use the Resource Editor to specify which classes in a resource you want to display when the resource is displayed in the Components window. You can select a class and toggle its display status by selecting the **Toggle Display Status** check box. Although the resource should contain all classes that the application will use, only those components that can be dropped onto a frame should be set to display. For more information about resources and the Components window, see [“Associating Resources with FRAME Entries”](#) on page 155.

For complete information about working with the Resource Editor, see the SAS/AF online Help.

Synchronizing Resources

Each RESOURCE entry contains a complete, static copy of its classes, which means that a resource does not reread the underlying class information as it loads them. To update information in a resource when you change one of its classes, you must synchronize the RESOURCE entry. Synchronizing a resource requires the class metadata of the original classes, updating the version of the class that is stored within the resource. You must synchronize a resource whenever you change a class name, location, or description, or whenever you change property metadata information that is stored in a CLASS entry, such as an attribute's valid values or the location of SCL method code. You do not have to synchronize a resource if you modify the SCL code of methods on a class.

For example, suppose you create a resource that includes a class named `mylib.myclasses.Demo.class` and then save that resource and close the Resource Editor. You then edit the class to add a method. After you save the class, you need to synchronize the resource so that the changes you made to the class are included.

To synchronize a RESOURCE entry from within the Resource Editor, select **Tools** ⇒ **Synchronize**.

You can also edit a class directly from the Resource Editor, which enables you to automatically synchronize the class and the resource after you commit the changes to the CLASS entry. That is, if you edit a class from within the Resource Editor, you do not have to synchronize the resource. To edit a class from within the Resource Editor, select **Edit** ⇒ **Edit Class** or click the Edit toolbar button.

Analyzing Resources

The Resource Editor enables you to analyze a resource by displaying all of the classes that the resource contains, their ancestors, and the search path that is needed to find the classes. To analyze a resource in the Resource Editor, select **Tools** ⇒ **Analyze**.

Here is a sample analysis of a simple resource as it appears in the SAS log:

```
Analysis of resource entry:
SASUSER.TESTCATALOG.TESTRES.RESOURCE

SASHELP.CLASSES.CHECKBOX_C.CLASS
  Parent = SASHELP.CLASSES.AFCONTROL.CLASS
  Parent = SASHELP.FSP.WIDGET.CLASS
  Parent = SASHELP.FSP.OBJECT.CLASS

SASUSER.TESTCATALOG.NBUTTON.CLASS
  Parent = SASHELP.CLASSES.PUSHBUTTON_C.CLASS
  Parent = SASHELP.CLASSES.AFCONTROL.CLASS
  Parent = SASHELP.FSP.WIDGET.CLASS
  Parent = SASHELP.FSP.OBJECT.CLASS

Current search path is:
  SASHELP.CLASSES
  SASUSER.TESTCATALOG
```

Merging Resources

The Resource Editor enables you to merge classes that are referenced by another resource into the currently open resource. Merging several smaller resources into a single, larger resource can help improve performance by reducing the catalog I/O that SAS/AF performs. Note that the actual class information is merged, not the copy of the class information in the merged resource.

To merge resources from within the Resource Editor, select **Tools** ⇒ **Merge Resource** and then select the resource that you want to merge.

Classes with identical aliases are not merged. An alias is a reference that is used by a RESOURCE entry. It is created by concatenating the class name and its closest abstract parent class. As a result, when resources reside in different libraries and/or catalogs and two classes have the same name and type, the resulting aliases will be identical. If you try to merge classes that have identical aliases, the class in the imported resource will not be merged.

Renaming Class Catalogs and Libraries in a Resource

A resource must accurately reflect all CLASS entry information, as well as the library and catalog in which the class is stored. If you move or copy the classes that are referenced in a resource to another catalog or library, you will need to change the class information in the resource to the new location. To rename the library and catalog references for a resource using the Resource Editor, select **Rename Libraries** or **Rename Catalogs** from the pop-up menu in the Resource Editor window.

You can use the Resource Editor not only to rename the library and catalog references for its classes, but also to update the library and catalog references in all affected CLASS entries. To apply the rename action to all classes in the resource, check the **Apply changes to class entries** check box in the Rename Libraries or Rename Catalogs dialog box.

Note: Applying renamed library and catalog references to classes via the Resource Editor does not update any hardcoded object references or other library and catalog names in the method SCL for a class. If you use SCL to create classes with the CLASS/ENDCLASS syntax, the library and catalog names in the SCL also are not affected by changes in the Resource Editor.

You can also use the CATNAME statement to logically combine one or more catalogs by associating them with a catref (shortcut name). See [“Step 2: Set Up the Development Environment” on page 6](#) for more information.

Making a New Class Available for Use

Introduction

After a class has been created and is ready to be used on a frame for testing or production, you can make it available for use from the Components window. The Components window loads resources and individual classes that can be added to a frame at design time. All classes in a resource that are set to “Display” (via the **Toggle Display Status** check box in the Resource Editor) are shown in the Components window, including

- all visual controls and subclasses of visual controls
- non-visual components that work in conjunction with visuals, such as models
- legacy objects that are subclasses of `sashelp.fsp.Widget.class`

Components that are designed to be instantiated at run time should be included in a resource, but you do not have to consider their use in the Components window. Developers can use the `_NEW_` operator in their SCL code to add these components to their applications.

Associating Resources with FRAME Entries

The applications developers who use your components to create frames must have access to those components. You can make the components available for drag and drop operations from the Components window, or developers can explicitly associate a resource with a frame.

You can control the contents of the Components window that appears in the build environment in any of the following ways:

- Add individual classes to the Components window to test those classes. To add an individual class to the Components window, select **Add Classes** from the pop-up menu inside the Components window.
- Add classes to a resource, and then add that resource to the Components window to organize several classes. This process is recommended for deploying most components. To add a resource to the Components window, select **Add Resources** from the pop-up menu inside the Components window.
- Add classes or resources to the Components key of the SAS Registry to change the default contents of the Components window. See [“Modifying the Components Key of the SAS Registry” on page 156](#) for details.

The Components window displays any classes and resources that are defined in the Components key of the SAS Registry. Resources that are used by any open FRAME entries are also temporarily added to the Components window while those frames are open. The default resource settings include `sashelp.fsp.AFComponents.resource` and `sashelp.fsp.build.resource`.

When you open a frame, the Components window displays the resource that was used to create the frame. Classes that are used by the frame and are not contained in the resource are loaded individually by the frame but do not necessarily appear in the Components window.

The resource that was used when the frame was created is stored with the FRAME entry. By default, a new frame uses the resource specified in the Resource value of the `Products\AF\Design Time\Fram` key. You can explicitly change the active resource for the current frame from the default frame resource in either of the following ways:

- Enter the `RESOURCE resource-name` command.
- Include a `RESOURCE=resource-name` option as part of the BUILD command. For example:

```
build work.a.a.frame resource=sashelp.fsp.afcomponents.resource
```

To see which resource is used by the current frame, you can enter the `RESOURCE` command without a specified resource. The active resource is displayed on the SAS status bar.

Modifying the Components Key of the SAS Registry

The SAS Registry contains a key that stores the settings for the Components window. The values defined in this key specify the resources and/or classes that are displayed when the Components window is opened. You can add values to this key or modify existing values on the key to change the default setting.

To change the Components key:

1. Enter the REGEDIT command to open the SAS Registry.
2. Expand the registry to view **Products\AF\Design Time\Component Window\Components**.
3. Select **New String** from the pop-up menu to add a new value, or select the name of the value you want to modify and then select **Modify** from the pop-up menu.
4. Enter the changes for Value Data as required. (The value of Value Name is not significant.) Click **OK**.
5. Exit from the SAS Registry.

For more information on the SAS Registry, see [“Working with SAS/AF Keys in the SAS Registry” on page 197](#).

Generating Class Documentation with GenDoc

Introduction

SAS/AF software includes GenDoc, an experimental documentation utility that enables you to generate HTML files that document class, interface, resource, and frame entries. The generated HTML files can be viewed with most Web browsers. You can use the documentation produced by the GenDoc utility to provide assistance to developers who use the components that you develop.

You can start the GenDoc utility in any of the following ways:

- In the Class Editor, select **File** ⇒ **Save As HTML**.
- In the Interface Editor, select **File** ⇒ **Save As HTML**.
- In the Resource Editor, select **File** ⇒ **Save As HTML**.
- In the Build window while editing a frame, select **File** ⇒ **Save As HTML**.
- At the command prompt, enter

```
afa c=sashelp.aftools.scl2html.frame
```

You can use the interface to select the entry and entry type you want to document.

- Use the experimental SCL function CreateDoc:

```
CreateDoc('catalog-entry-name', 'output-directory');
```

where *catalog-entry-name* is the name of the class, interface, frame, resource, or catalog you want to document, and *output-directory* is the local directory to which you want to direct the HTML output.

- Use the SCL Static Analyzer and add the HTMLDIR= option to the CROSSREF statement. In the following example, *output-directory* is the local directory to which you want to direct the HTML output:

```
proc build batch;
    crossref proj=lib.cat HTMLDIR='output-directory';
run;
```

Tips for Writing Meaningful Metadata Descriptions

Because GenDoc reads the metadata that is associated with a class or interface, the quality and usefulness of the generated documentation depends on how much information you provided in your class or interface definitions. Complete and accurate descriptions of metadata items also make class maintenance and debugging easier, and they aid other developers who use your classes.

Before you use the GenDoc utility to generate documentation, be sure to include values for the Description metadata items in the following properties and/or property elements:

Attributes

In your description, include a clause that begins “Returns or sets...”. Since the elements describe things rather than actions or behaviors, it is also appropriate to omit a subject and verb and to simply use a noun phrase. For example, instead of “This attribute is a button label”, you could use “A button label”.

Methods

Since methods implement an operation, they usually use a verb phrase. While this may make methods “self-documenting,” you should try to extend this action with additional detail. Method descriptions typically begin with a verb such as “Deletes,” “Updates,” “Sets,” or “Returns.”

Method Arguments

Provide a description that begins with “Specifies” (when INOUT is “Input” or “Update”) or “Returns” (when INOUT is “Output” or “Return”), depending on how the argument is used in the method.

Method Return Argument

Provide a description that starts with “Returns.” Descriptive text that you supply for method return arguments appears only in documents that are generated by the GenDoc utility.

Events

Provide a description that indicates when or how the event occurs, such as “Occurs when ...”. “Attribute changed” events are automatically included for each attribute you add. These events are assigned a description in the form “Occurs when the ___ attribute is changed.”

Event Handlers

Provide a description that indicates how the event handler performs. Since event handlers respond to events, most event handlers use the form “Executes when the ___ event occurs.”

You may also consider these additional style guidelines:

- Use phrases instead of complete sentences for property descriptions.
- Use third person declarative rather than second person imperative. For example, a description should read “Gets the label” instead of “Get the label.”

What the GenDoc Utility Creates

By default, the GenDoc utility stores the HTML files that it creates in the directory that includes a **classdoc** subdirectory in the HELPLOC option. The HELPLOC option is a SAS system option that is defined or set in the SAS configuration file or in an AUTOEXEC file. You can also edit the HELPLOC option for the current SAS session by selecting **Tools** ⇒ **Options** ⇒ **System**. You can then expand the **Environment control** and **Help** nodes to modify the HELPLOC option.

The HELPLOC option should include at least two directory paths if you want to provide component documentation. One directory path identifies the online Help files that are shipped with SAS software. For example:

```
!sasroot\core\help
```

The second path must end with a directory named **classdoc**. For example:

```
d:\My SAS Files\classdoc
```

SAS must be able to write to this directory in order for GenDoc to create HTML files.

Since you cannot have two classes with the same name within a catalog, the GenDoc utility creates a subdirectory for the library and catalog under the **classdoc** directory that is listed in the HELPLOC option. The first subdirectory specifies the library. A second directory is created under the library directory and is named for the catalog containing the element that you want documented. For example, if you generated documentation for **sasuser.myclasses.SalesObject.class**, GenDoc creates the directory **/sasuser/myclasses** under **classdoc**.

The following list describes the types of information generated by GenDoc and the filenames of the resulting HTML documents:

Table 15.1 GenDoc File Output

Documentation for...	Contains...
Class Entry	parent or ancestor information and all properties (attributes, methods, events, event handlers, and supported or required interfaces) defined for the class. <i>HTML file:</i> classname.htm
Interface Entry	parent or ancestor information and all methods defined for the interface. <i>HTML file:</i> interfacename-interface.htm
Resource Entry	all class entries and interface entries included in the resource. <i>HTML files:</i> resourcename-resource.htm , plus an HTML file for each class and interface in the resource
Frame Entry	frame information and general attributes, the associated resource, and instance information for all visual and nonvisual components on the frame. <i>HTML file:</i> framename-frame.htm

Documentation for...	Contains...
Catalog	all class, interface, and frame entries stored in the catalog. <i>HTML files:</i> catalogname-index.htm , plus an HTML file for each class, interface, and frame entry stored in the catalog

For example, if HELPLOC is set to `d:\My SAS Files\classdoc` and you generate documentation for `sasuser.myclasses.SalesObject.class`, the following HTML file is created:

```
d:\My SAS Files\classdoc\sasuser\myclasses\SalesObject.htm
```

You can edit the HTML files to add other information, such as a complete description of a component.

GenDoc creates an index file that contains links to other files when you generate documentation for a resource or catalog. However, you can create your own HTML file to use an index if you want a more customized collection. To document a project or application, you can also create an HTML file that contains links to all of the necessary documents, or to the documentation for the catalogs and/or resources that are used by the project.

Making Class Documentation Available

After you have used GenDoc to create documentation for your components, you can make that documentation available to users of your components. Instruct your component users to include a path in their HELPLOC system option to the `classdoc` directory where you generated the class documentation. Depending on your system environment, you may have to copy the files to a new location that is accessible to all appropriate users.

After they set their HELPLOC options, they can access the generated documentation through the Class Editor, Properties window, Components window, and Class Browser in the same manner that help is displayed for SAS classes. For example, if you make documentation available for `myorg.classes.SalesObject.class` and the class has been included in a RESOURCE entry that is displayed in the Components window, a user can select the class in the Components window and then select **Help on Class** from the pop-up menu to display the documentation in a Web browser.

Of course, you can simply copy the generated HTML files to an accessible location on a Web server to make the class documentation available to any developer at any time, regardless of whether SAS is running. If you copy the HTML files, be sure to preserve the directory structure that GenDoc creates for the library-name and catalog-name that are associated with the class, interface, or resource. Unless this location is defined in a developer's HELPLOC option, the interactive help features will not work.

Part 4

Appendixes

<i>Appendix 1</i>	
Flow of Control	163
<i>Appendix 2</i>	
Adding Attachments to Frame Controls	179
<i>Appendix 3</i>	
Working with SAS/AF Keys in the SAS Registry	197

Appendix 1

Flow of Control

How SCL Programs Execute for FRAME Entries	163
Introduction	163
Processing the INIT Section	163
Processing Labeled Sections for Components	164
Processing the MAIN Section	165
Processing the TERM Section	166
Automatic Termination of SCL Objects When an Application Ends	167
Flow of Control for Frame SCL Entries	169
Introduction	169
FRAME Entries and Automatic Methods at Build Time	169
Build-Time Order of Program Execution	170
FRAME Entries and Automatic Methods at Run Time	172
Run-Time Order of Program Execution	173
Changing Frame Components and Frame SCL Programs	177

How SCL Programs Execute for FRAME Entries

Introduction

SCL programs for FRAME entries use code sections that are named with reserved labels to process the phases of program initialization (INIT), main processing (MAIN), and termination (TERM).

Processing the INIT Section

Statements in the INIT section typically perform actions such as

- initializing variables
- importing values through macro variables
- displaying initial messages on the window's message line
- opening SAS tables and external files that are used by the entry
- processing parameter values that are passed to the FRAME entry

By default, INIT executes only once, before the entry's window opens, for each invocation of a FRAME.

Processing Labeled Sections for Components

Introduction

Sections of frame SCL programs that are named with object labels execute when an action is performed on the component, such as selecting the component or changing its value. The CONTROL LABEL statement in SCL controls the execution of labeled sections. CONTROL LABEL is on by default for FRAME entries, making it easier to create labeled statement sections to execute when you perform an action on a component. For details on the CONTROL LABEL statement, see the *SAS Component Language: Reference* and the `_setControl` method of the Frame class in the SAS/AF online Help.

Order of Processing for Multiple Window Components

When a frame needs to process more than one component, a predefined order (known as the window order) controls the processing order. The window order starts from the top row of the window and moves to the bottom row. Additionally, window order moves from left to right in each row.

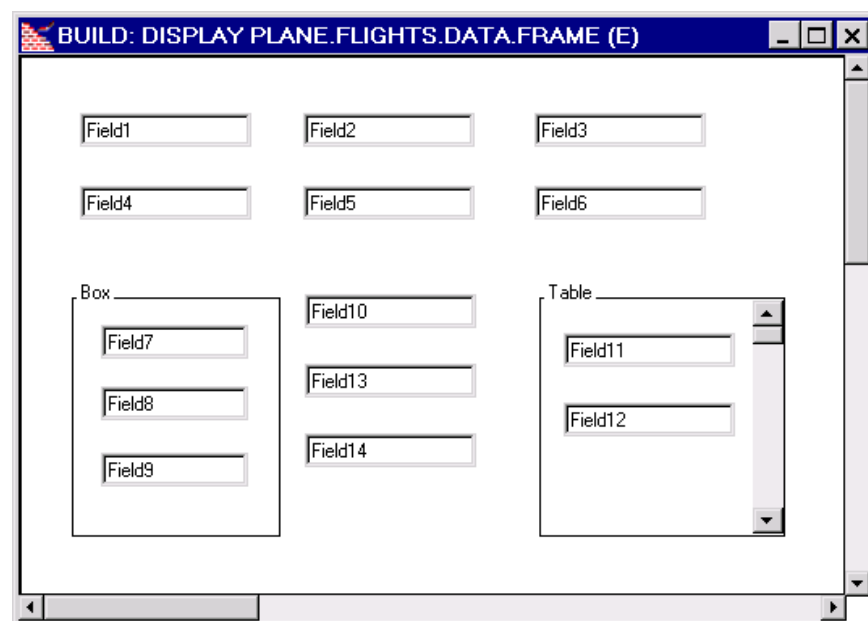
However, if one component (such as a container box) contains other components, then all the contained components are processed as a group before other components on the right are processed.

Some hosts and display devices also honor the window order when you use the TAB key to move the cursor between fields. See “Setting the Tab Order” in the SAS/AF online Help for more information.

Note: There is an exception to the window order. When the `_preTerm`, `_postInit`, and `_refresh` methods are run, all extended tables in the frame are processed before any other components.

Example: Order of Processing for Multiple Window Components

Consider the following FRAME entry:



The order of processing for this window is FIELD1, FIELD2, FIELD3, FIELD4, FIELD5, FIELD6, BOX, FIELD7, FIELD8, FIELD9, FIELD10, TABLE, FIELD11, FIELD12, FIELD13, and FIELD14.

Note carefully the order of processing components in the container box labeled **Box** and the extended table labeled **Table**, as well as the fields within them. For example, FIELD10 is not processed until after FIELD7, FIELD8, and FIELD9 are processed because they are contained in **Box**, which appears before FIELD10. Also, FIELD11 and FIELD12 are processed before FIELD13 because they are contained in **Table**, which starts in the row above FIELD13.

Processing the MAIN Section

Introduction

Statements in the MAIN section typically perform actions such as:

- validating field values
- calculating values for computed variables that are based on user input
- displaying selection lists that are developed through SCL functions
- submitting Base SAS software code
- invoking secondary windows
- querying and executing commands that are issued by users
- retrieving values from SAS tables or from external files

By default, MAIN executes each time the follows occurs:

- a user presses ENTER or RETURN (or any function key) after modifying one or more fields or text entry controls, provided that the modified fields contain valid values. After you modify fields and press ENTER, all modified fields are checked to verify that their values satisfy their attribute specifications. (However, the **required** attribute is not checked until you attempt to end from the entry.)
- you activate a control by modifying or selecting it.

Each of these actions first runs the labeled section that is associated with the object and then runs MAIN. The new values of the window variables are then displayed in the corresponding fields.

To override the default behavior of when MAIN executes, use the CONTROL statement in SCL. (For details, see the *SAS Component Language: Reference*.)

Forcing MAIN to Execute

Although the default behavior is for MAIN to execute only when all field or component values are valid, you can force MAIN to execute even when some fields or components contain invalid values or when an application-specific command is issued. SCL provides CONTROL ENTER, CONTROL ERROR, CONTROL ALLCMDS, and CONTROL ALWAYS statements that cause the MAIN section to execute whenever you press ENTER or RETURN, even if no fields are modified or if modified fields do not contain valid values. If the FRAME entry contains no selectable or modifiable objects, then MAIN cannot execute unless a CONTROL ENTER or CONTROL ALWAYS statement is in effect.

To enable MAIN to execute even if a field is in error, you must specify CONTROL ERROR or CONTROL ALWAYS. The `_setControl` method of the Frame class also modifies how the FRAME entry processes errors and input events.

See the *SAS Component Language: Reference* for information about CONTROL statement options.

Handling Invalid Values in MAIN

If a modification to a text entry control introduces an attribute error, the labeled section for that control does not execute. However, the labeled sections for other controls that are also modified will execute.

If an attribute error is detected, MAIN does not execute. Instead, fields or controls that contain invalid values are highlighted and an error message is displayed. The error attributes that you specified determine what is flagged as an error. You can enable users to correct an attribute error in the window. If the program contains CONTROL ERROR and CONTROL LABEL, or if the `_setControl` method has put these statements into effect, you can include statements in the labeled section that make a correction, as shown in the following example:

```
INIT:
    control error;
return;

TEXTENTRY1:
    if error(textentry1) then do;
        textentry1.text=.;
        erroroff textentry1;
        _msg_='Value was invalid. It has been reset.';
    end;
return;
```

Processing the TERM Section

Introduction

Statements in the TERM section typically perform actions such as

- updating and closing SAS tables and external files that were opened by the entry
- exporting values of SCL variables to macro variables for later use
- branching to another entry in the application by using the DISPLAY routine
- submitting Base SAS software code for execution
- deleting SCL lists or SCL objects that are created in the FRAME entry

TERM executes when you issue either an END command or a CANCEL command.

Processing an END Command

If you issue the END command,

- The SCL system variable `_STATUS_` is set to **E**.
- Modified fields are checked for valid values.
- Statements in the MAIN section execute if fields are modified and their values are valid.
- Required fields are checked to verify that they are not blank.
- Any SCL statements in the TERM section execute.
- The `_term` method runs for the frame and for all components.

- The window closes and control returns to the parent or calling entry.

Note: If any fields contain invalid values when MAIN executes, MAIN processing highlights the errors and returns control to the application. In that case, TERM does not execute.

If any empty fields have the **required** attribute, TERM does not execute. Instead, the empty required fields are highlighted and an error message is displayed. You must provide values for all required fields and issue the END command again before the TERM section will execute.

After all these conditions are satisfied, the SCL statements in TERM execute. After the TERM section executes, the window closes and the entry terminates. Control returns to the SAS process that invoked the entry.

Processing a CANCEL Command

If you issue the CANCEL command,

- The SCL system variable `_STATUS_` is set to `C`.
- No field validation is performed.
- Any SCL statements in the TERM section execute.
- The `_term` method runs for the frame and for all components.
- The window closes and control returns to the parent or calling entry.

Processing an ENDSAS, BYE, or System Closure Command

If you issue the ENDSAS or BYE command, or if you use the system closure menu when the FRAME entry is running without the SAS Application Work Space (AWS), then:

- The SCL system variable `_STATUS_` is set to ' ' (blank).
- Statements in the TERM section do not run.
- The `_term` method runs for the frame and for all components.
- The window closes.

Note: The SAS AWS is the container window for SAS software.

Automatic Termination of SCL Objects When an Application Ends

Introduction

When an application ends, the software scans for any remaining SCL objects that have not yet been deleted and sends a `_term` method to them. In most cases, this is completely benign. In a few cases, it might indicate a defect in your application; that is, perhaps you did not delete something that you should have, or perhaps your `_term` method omitted a `SUPER(_self, '_term')`;

Sometimes an application cannot delete an object because it does not know whether the object is still in use; the AUTOTERM feature deletes such objects by running the `_term` method.

Using the AUTOTERM= Option to Control the AUTOTERM Feature

If the AUTOTERM feature causes problems in your application, you can disable it by adding the AUTOTERM= option to your AF, AFA, or AFSYS command:

```
AF C=lib.cat.member.name AUTOTERM=term-value
AFA C=lib.cat.member.name AUTOTERM=term-value
AFSYS AUTOTERM term-value
```

(You can use the AFSYS command once an application is running.) Values for *term-value* are

ON

enables the AUTOTERM feature. AUTOTERM is on by default. Use this value to enable the feature if it is turned off.

OFF

disables the AUTOTERM feature, which means that the software does not invoke the `_term` method on objects when a task is terminated. This value makes the AUTOTERM feature compatible with earlier releases of SAS/AF software.

VERBOSE

prints a note and dumps the object list of each object that still exists when a task is terminated. This value works even if AUTOTERM is OFF; it serves as a debugging aid to identify which objects still exist but whose `_term` method has not run.

NOVERBOSE

disables the VERBOSE option. NOVERBOSE is the default.

You cannot combine options in one string; use separate AUTOTERM= options on the command line, or use separate AFSYS commands.

Using the VERBOSE Value to Help Debug Your Application

Use the VERBOSE value when you invoke your application or while the application is running. With VERBOSE, the software displays a note about any object that still exists at task termination, and it dumps the object list to the log before sending the `_term` method. Without VERBOSE, `_term` is invoked and no note is displayed in the log. Using VERBOSE as you develop your applications is helpful because it can highlight objects that your application fails to delete.

In a few rare cases, you might begin to see program halts as your application completes. These halts are often caused when one object is deleted and another object that references it does not know that it has been deleted. For example, consider a class named Manager that maintains a list of instances of Message File objects. The Manager class has a `_term` method that unconditionally sends `_term` to each item in its list of Message File objects:

```
term: private method;
  dcl num i;
  do i=1 to listlen(msgObjs);
    dcl object msgObj;
    msgObj=getitemn(msgObjs,i);
    msgObj._term();
  end;
  _self_._super();
endmethod;
```

However, when a task ends, AUTOTERM may send `_term` to a Message File object *before* running the Manager's `_term`, so the Manager's list becomes stale (that is, it contains object identifiers of deleted Message File objects). The proper fix in this and similar situations is to verify other dependent objects in the `_term` before sending methods to them. (In this case, the Message File object is not aware of the Manager and therefore cannot ask the Manager to remove the object identifier from the Manager's list of Message File objects.)

To determine whether a number is a valid list identifier, use

```
LISTLEN(number) > -1
```

Then, to determine whether a list identifier is an object identifier, use

```
HASATTR(listid, 'O')
```

(The 'O' is a letter O as in Object.) For example, the Manager's `_term` method is more correctly implemented as follows:

```
term: private method;
  dcl num i;
  do i=1 to listlen(msgObjs);
    dcl object msgObj;
    msgObj=getitemn(msgObjs,i);
    if listlen(msgObj) > -1 and hasattr(msgObj,'O') then
      msgObj._term();
  end;
  _self_._super();
endmethod;
```

Note: As an alternative to the above solution, when `_term` is run on a Message File object, the Message File object could send an event. The manager would then have a handler for the event and could remove the object from its list.

Flow of Control for Frame SCL Entries

Introduction

There is a specific flow of control for frame SCL. In general, the INIT section runs before the window opens, the MAIN section executes after a field is modified or a component is selected, and the TERM section runs when the window closes. Because the FRAME entry allows additional statements to run for components as well as for multiple extended tables via SCL methods, there are additional points of interest in the flow of control in FRAME entry programs.

After each SCL section runs, if the FRAME entry's status has been set to **H** (for Halt), execution halts. You can set the status to **H** either by assigning the `_STATUS_` variable in the SCL or by calling the FRAME entry's `_setStatus` method. Likewise, a STOP statement in the frame SCL program can halt the frame application, interrupting the normal flow of control. However, in both of these cases, all `_term` methods execute.

FRAME Entries and Automatic Methods at Build Time

When you build a frame, an instance of a frame is created. When a frame or frame SCL is used at build time, specific methods are automatically called. You can override these automatic methods at build time, if necessary.

Although frames and frame SCL entries are typically executed at run time, there are some cases where you might actually use an existing frame and its frame SCL entry at build time. For example, you might have created a custom editor for use during the development of other frames.

The following table provides information about automatic build time methods. For more information about these methods, refer to the SAS/AF online Help.

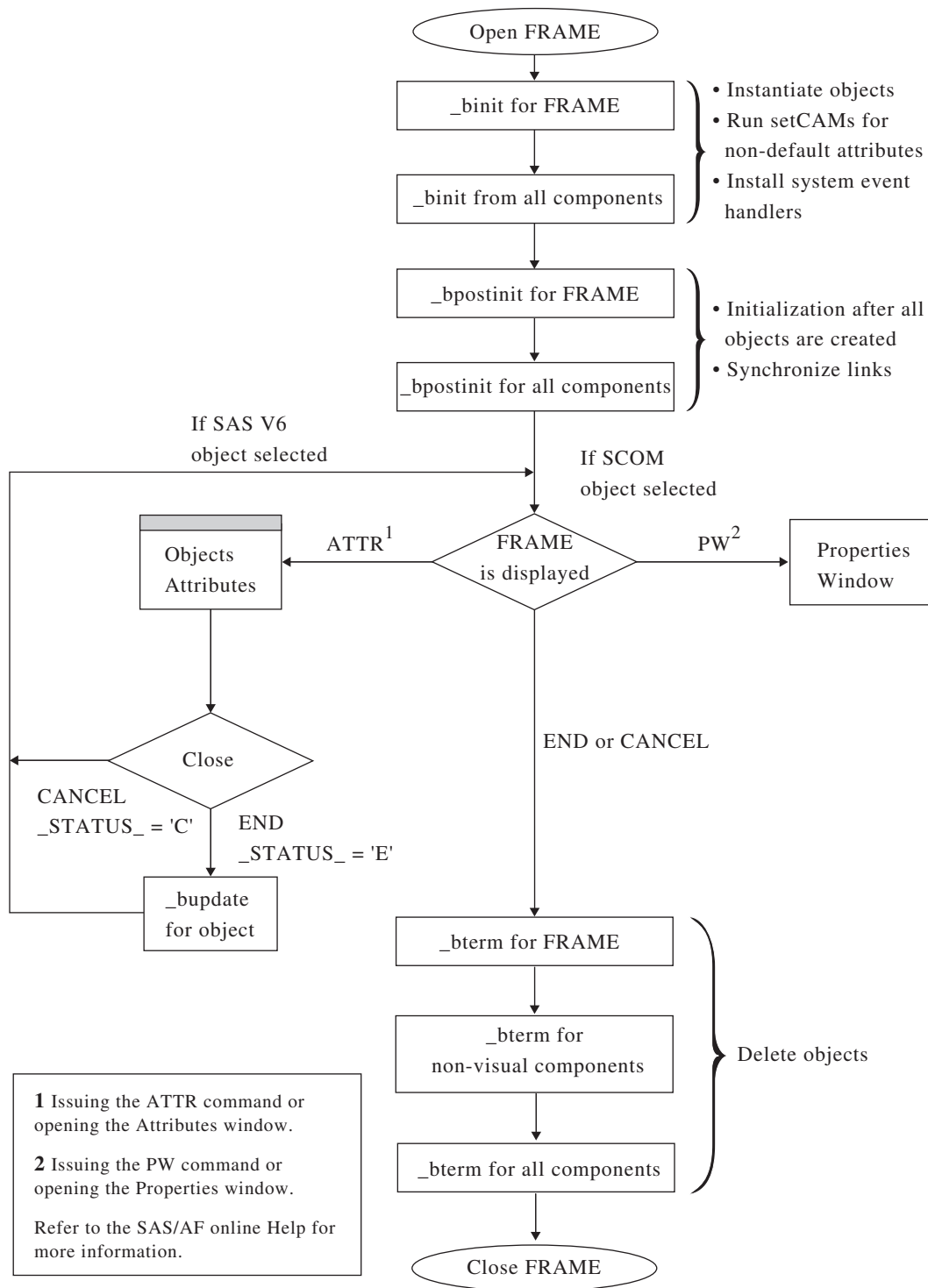
Table A1.1 Automatic Build-time Methods

Method	Class	Description
<code>_bInit</code>	Frame	a <code>_bInit</code> method that runs at build time when you open the FRAME entry, first for the frame, and then for each component. The Frame object and each component must successfully execute this method (and its parent's definition) to be usable.
<code>_bInit</code>	Object	initializes each object in the FRAME entry.
<code>_bPostInit</code>	Frame	performs postprocessing after the <code>_bInit</code> method for the Frame object. Also runs the <code>_bPostInit</code> for all the components; <code>_bPostInit</code> for non-visual components runs first, followed by <code>_bPostInit</code> for visual components.
<code>_bPostInit</code>	Object	performs postprocessing on objects after the <code>_bInit</code> method. You should override this method rather than <code>_bInit</code> if your initialization code refers to any objects besides <code>_SELF_</code> .
<code>_bUpdate</code>	Frame, Widget	updates the Frame object or widget with any values that you might have changed in the Attributes window. For your own GUI subclasses, you might need to override this method to execute code after the user closes the Attributes window. This method is used only by legacy classes. Any object that uses the Properties window or a custom Properties window does not invoke this method.
<code>_bTerm</code>	Object	terminates the components in the FRAME entry at build time. You only need to override this method for cleanup purposes. For example, you might need to delete SCL lists or non-visual components that you create during <code>_bInit</code> or <code>_bPostInit</code> .
<code>_bTerm</code>	Frame	terminates the Frame object at build time. You only need to override this Frame class method for cleanup purposes. For example, you might need to delete SCL lists or non-visual components that you create during <code>_bInit</code> or <code>_bPostInit</code> .

Build-Time Order of Program Execution

The following figure illustrates the order in which the automatic methods are invoked at build time and the circumstances under which they are invoked.

Figure A1.1 Automatic Methods and Build-Time Order of Program Execution



Note: Events are not sent during `_bInit` and `_init`, including `attributeName changed` events sent by `_setAttrValue` as well as events sent by `_sendEvent`. The reasoning behind this behavior is that it is impossible to know which objects exist when your `_init` is running. Therefore, not all objects that are listening for your event will be notified. You can not be sure that all objects exist until your `_postInit` runs.

FRAME Entries and Automatic Methods at Run Time

When a frame or frame SCL executes at run time, specific methods are automatically called. The following table provides information about automatic run-time methods. For more information about these methods, refer to the SAS/AF online Help.

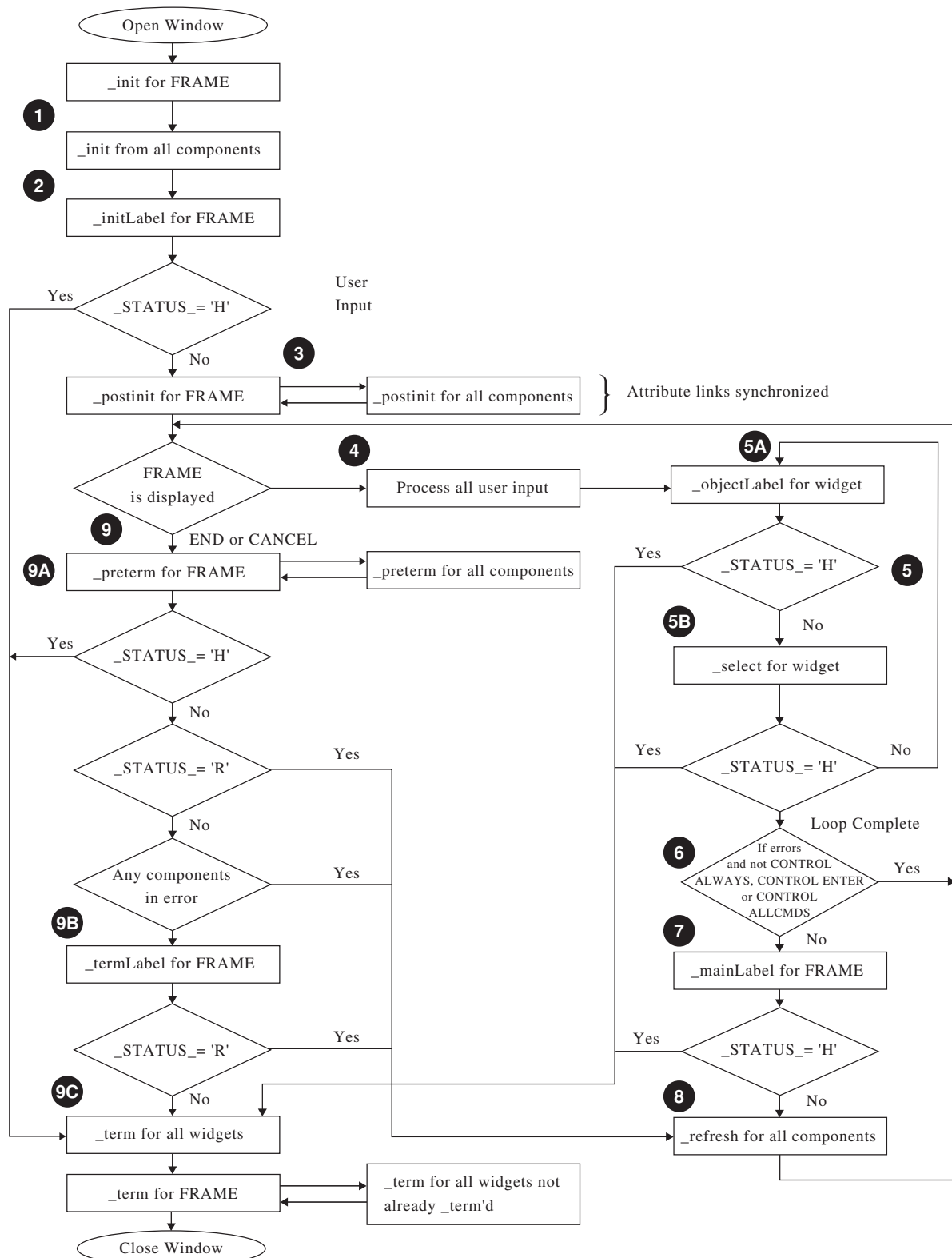
Table A1.2 Automatic Run-time Methods

Method	Class	Description
_init	Frame	initializes all components for the frame.
_init	Object	initializes the Frame object and all components.
_initLabel	Frame	runs the INIT section of the FRAME entry's SCL program.
_postInit	Frame	performs additional processing for the Frame object after _initLabel has run. Also runs the _postInit for all components; non-visual components are run first, followed by visual components.
_postInit	Object	performs additional processing on all widgets after the INIT section has run.
_objectLabel	Widget	runs the object label section in the FRAME entry's SCL program.
_select	Widget	responds to a user selection or modification.
_mainLabel	Frame	runs the MAIN section of the FRAME entry's SCL program.
_refresh	Widget	redraws a widget without updating its data.
_preTerm	Frame	performs additional processing before _termLabel is run.
_preTerm	Widget	performs additional processing before the TERM section runs.
_termLabel	Frame	runs the TERM section of the FRAME entry's SCL program.
_term	Frame	deletes the Frame object. Also runs the _term for non-visual components. The _term for visual components should have already run by this point. If this method runs the _term for visual components, then a problem exists with the visual components and a warning message appears.
_term	Object	deletes an object and the SCL list that is used to represent the object.

Run-Time Order of Program Execution

The following figure illustrates the order in which the automatic methods are invoked at run time and the circumstances under which they are invoked.

Figure A1.2 Automatic Methods and Run-Time Order of Program Execution



- 1 The `_init` methods for the FRAME entry and its components execute. The `_init` for non-visual objects is run first, followed by the `_init` for visual objects. The `_init` for visual objects follows a specific window order. See “[Example: Order of Processing for Multiple Window Components](#)” on page 164 for more information.

Event handlers are installed and drag and drop sites are also set up at this point.

- 2 The statements in the INIT section of the SCL program execute via the `_initLabel` method.
- 3 The `_postInit` methods for the FRAME entry and its components execute. Any attribute links are also synchronized at this point. That is, `_getAttribute` is invoked on the source attribute (to access the attribute value), and then `_setAttrValue` is called on the target attribute.
 - For extended tables that are being used as selection lists, the `_setSelection` method executes for all components in the row.
 - The `_getRowLabel` method and the `_getRow` method run for each row of each extended table until the table fills (or until the `_endTable` method is called, for dynamic tables).

Note: Because an object cannot be initialized twice, the `_init` and `_postInit` methods are not permitted after the object is initialized. After the first `_init` method is sent to an object, any subsequent `_init` methods result in an SCL program halt. After the first `_postInit` method is sent to an object, the FRAME entry will not send additional `_postInit` methods, but instead sends `_refresh` methods.

- 4 When one of these conditions is met:
 - a field is modified
 - a component is selected
 - a command is issued and CONTROL ALWAYS, CONTROL ALLCMDS, or CONTROL ENTER is in effect.

The following steps occur (these steps are not illustrated in the figure):

- If the modified component is in an extended table that is being used as a selection list, the `_selectRow` method executes.
 - If a field is modified, the field's informat is applied. If the value does not match the informat, the field is flagged as being in error. If a field within an extended table is modified, the corresponding row of the extended table is also marked as modified.
 - If the modified field matches the informat, the field's `_validate` method is queued to run. If the validation fails, the field is flagged as being in error.
 - The `_objectLabel` method for the object that is modified or selected is queued to run, unless CONTROL NOLABEL is in effect. CONTROL LABEL is the default for FRAME entries.
 - If the component that is modified or selected is in an extended table, the `_putRowLabel` method is queued to run immediately after the `_objectLabel` methods for all modified components in that row. The `_putRow` method for the row that is modified or selected is queued to run after the `_putRowLabel` method. The `_objectLabel` method of the extended table is queued to run immediately after all `_putRowLabel` and `_putRow` methods.
- 5 All methods that were queued in step 4 execute. (Only 5A and 5B are illustrated in the figure.)
 - 5A) The `_objectLabel` method for each modified component executes. For example, if you modify the PRICE field in one or more rows of an extended table, the PRICE label executes once for each modified field, and the `_CURREW` system variable is updated with the number of the updated row each time.

- 5B) If a modified field is not in error or if CONTROL ERROR is in effect, the `_select` method is invoked for the selected or modified component. The default `_select` method submits the SAS command that is assigned to that component.
 - 5C) Any `_putRowLabel` methods that are queued execute in order after all labels for components within the row execute. If the component is in an extended table, the `_CURROW_` system variable is updated to reflect the row number.
 - 5D) The `_putRow` method for the extended table containing modified components executes after all other labels and methods for that row execute.
 - 5E) The `_objectLabel` method for the extended table executes, and then the extended table receives a `_select` method after all `_putRowLabel` and `_putRow` methods run.
- 6 If any component is marked as being in error (after the labels or the `_select`, `_putRowLabel`, or `_putRow` methods execute) and CONTROL ALWAYS, CONTROL ALLCMDS, or CONTROL ENTER is not in effect, the FRAME entry returns control to the user and awaits the next modification or command.
- 7 Unless one of the following conditions is met, the statements in the `_mainLabel` method execute:
- one or more fields are in error
 - a CANCEL, ENDSAS, BYE, or RETURN command is issued
 - neither CONTROL ALLCMDS, CONTROL ALWAYS, CONTROL ENTER, or CONTROL ERROR is in effect
- 8 If the FRAME entry is not ending, each FRAME entry component that has been modified or marked as needing refreshment is refreshed by invoking its `_refresh` method. An object cannot receive a `_refresh` method unless it is completely initialized (that is, the object's `_init` and `_postInit` methods must have run). If a `_refresh` method is sent to an object before its `_postInit` method runs, the `_refresh` method is converted to a `_postInit` method. When you use a REFRESH statement or send a `_refresh` method to a FRAME entry before the FRAME entry receives a `_postInit` method, a `_postInit` method is sent to the FRAME entry, and then the refreshment proceeds.
- Extended tables receive the `_refresh` method first, and then the other objects receive a `_refresh` method. Extended tables are refreshed by executing their `_getRowLabel` and `_getRow` methods and refreshing all components within the rows. This action is performed either for all rows or only for rows that have been modified, depending on the settings of the table's **Putrow Options** attribute. The extended table updates the `_CURROW_` system variable to reflect the logical row number for each row that it processes. `_CURROW_` is added as an automatic instance variable of both the extended table and the objects within the table.
- For extended tables with the **selectionList** attribute, the `_setSelection` method executes for all components on the row.
 - When an extended table is refreshed, it executes the `_getRowLabel` method.
 - The `_getRow` method for the extended table executes.
 - The table executes the `_update` method for the objects in the current row.
 - If the end of a table is reached, the `_hide` method is called for the remaining objects in the rows that are still visible.
- 9 If you issue the END, CANCEL, ENDSAS, BYE, or RETURN command, program termination begins.

- 9A) The `_preTerm` methods for the Frame object and all components are executed unless the command is `BYE` or `ENDSAS`. This enables objects to set the entry's `_STATUS_` system variable to `R` (to resume rather than terminate) under certain circumstances. For example, empty required fields are not allowed, and they prevent normal program termination except after a `CANCEL`, `ENDSAS`, `BYE`, or `RETURN` command.
- 9B) If no `_preTerm` method sets the `_STATUS_` variable to `R` or `H` via the `_setStatus` method of the Frame class, the `_termLabel` method is executed. If a statement in the `TERM` section sets `_STATUS_` to `R`, the program resumes instead of terminating, unless the command issued is `ENDSAS`, `BYE`, `CANCEL`, or `RETURN`. The `TERM` label does not execute if an `ENDSAS`, `BYE`, or `ENDAWS` command is issued.
- 9C) If the program termination is not stopped by an `R` status, the `FRAME` entry window closes and the `_term` method executes for each object in the entry and for the `FRAME` entry itself.

Note: Once the `_term` methods begin, the components are deleted. `_term` methods should not attempt to apply methods to other components in the `FRAME` entry. Also, any value assigned to `_STATUS_` is ignored once `_term` methods begin since the `FRAME` window is closed.

Changing Frame Components and Frame SCL Programs

You can change the attributes, location, and appearance of components in a `FRAME` entry without recompiling the SCL source program. Only the following changes require you to recompile:

- changing the name of a component
- changing the type of a component from character to numeric and vice versa
- changing the length of a component
- changing the `SUBMIT` replacement string from the Attributes window
- adding a new component
- deleting an existing component
- changing the SCL source entry
- modifying the SCL source

When you are in the build environment, this message appears when you need to recompile:

`NOTE: Intermediate code has been removed.`

Appendix 2

Adding Attachments to Frame Controls

Introduction	179
Selecting the Attachment Mode	180
Initiating Define Attachment Mode	181
Selecting the Direction and Type for the Attachment	182
Making the Attachments	186
Defining Attachments to Sibling Components	188
Defining Attachments to Components That Have Borders	189
Moving Multiple Components That Include Attachments	190
Example	190
Restricting Component Size	190
Changing and/or Deleting an Attachment	192
Displaying Attachments	192
Situations in Which an Attachment Is Ignored	193
Errors and Error Handling	193
Tips for Using Attachments	194

Introduction

Attachments are connections between two or more components on a frame. These connections can control the placement and size of components. Component placement and size may be affected by attachments when a component is resized or moved, or when the component's frame is resized.

Attachments remain valid even if automatic component resizing becomes necessary. Automatic component resizing occurs when you resize either components themselves or the frame on which your components exist.

Selecting the Attachment Mode

Before you begin defining attachments, you need to decide how you want to define them. You can focus on defining attachments for one particular component (such as a push button) or on defining attachments for a parent component and its child components (such as a container box that includes check boxes). The following figures illustrate the difference between child component attachment and current component attachment modes.

Figure A2.1 Child Attachment Mode

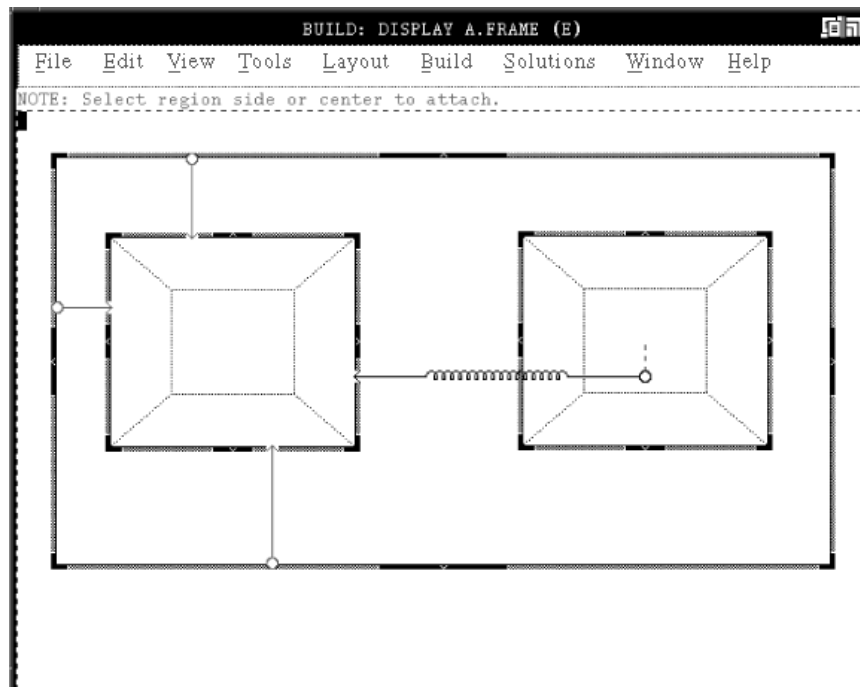
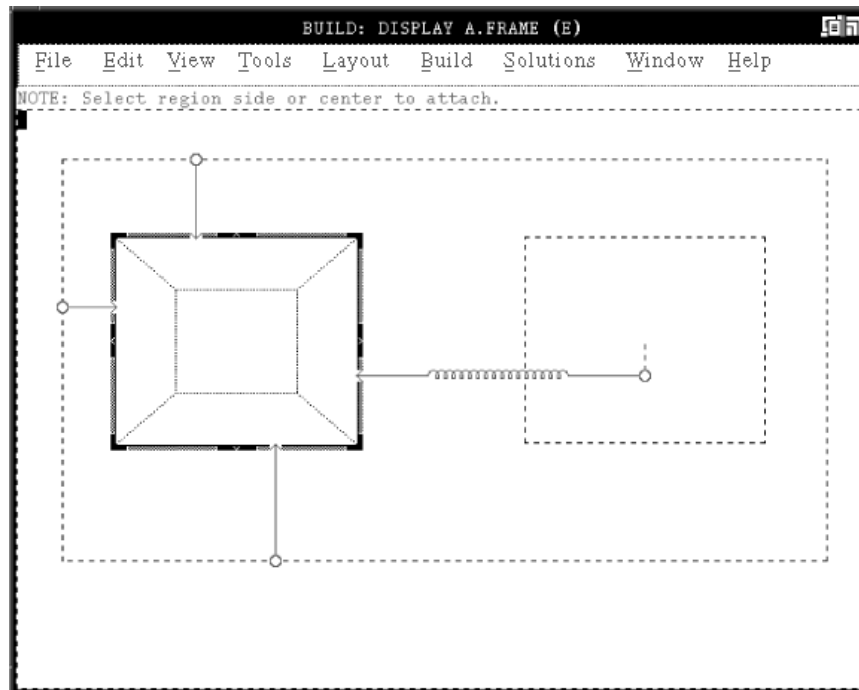


Figure A2.2 Current Attachment Mode



In either mode, attachments can be defined for the components that appear with sections drawn within them. In child attachment mode, all child components of the selected parent component are divided into sections, and their attachments are displayed. For current attachment mode, only the selected component is divided into sections, and only its attachments are displayed. In addition, other components on the frame may be hidden while you are defining attachments for a specific component.

Child attachment mode is the default, and it enables you to attach child components of a specific parent component to each other (sibling attachment) or to the parent component.

To initiate child attachment mode, select **Layout** ⇒ **Attach** ⇒ **Attach child region**.

Note: This technique is equivalent to the RM ATTACH KIDS command.

Before initiating the next step (Initiating Define Attachment Mode), select the parent component that contains the child components for which you want to define attachments.

Current attachment mode enables you to attach the currently selected component to its parent component or to its sibling components.

To initiate current region attachment mode, select **Layout** ⇒ **Attach** ⇒ **Attach current region**.

Note: This technique is equivalent to the RM ATTACH SELF command.

Before initiating the next step (Initiating Define Attachment Mode), select the component for which you want to define attachments.

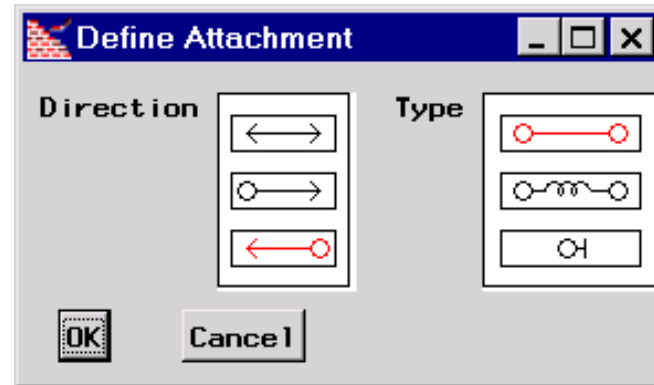
Initiating Define Attachment Mode

After you specify the attachment mode by designating whether you want to work in child component or current region attachment mode, you can turn on define attachment

mode. To initiate define attachment mode, select **Layout** ⇒ **Attach** ⇒ **Define Attachment**.

Note: This technique is equivalent to the RM ATTACH command.

The Define Attachments window opens, showing the default attachment of single-direction and absolute type:



The cursor shape changes to a focus icon, which looks like a magnifying glass in some operating environments.

Selecting the Direction and Type for the Attachment

The direction and type for attachments define how you want the connection between two components to be interpreted when one of the components moves or is resized. There are two basic types of attachments, absolute and relative:

Table A2.1 Attachment Types

The type ...	Is represented by ...	And maintains an attachment value of ...
absolute		a fixed number of pixels between the connection points of the attached components
relative		a percentage of the parent component's space between the connection points of the attached components. If the size of the parent component changes, the space between the components remains proportional.

Unless the size of the parent component changes, absolute attachments and relative attachments produce the same effect.

While you are working in define attachment mode, you can move or resize any component from which you are defining attachments. When you do this, the values of any affected attachments are modified. This enables you to move components closer together or farther apart without having to temporarily remove any existing attachments.

To demonstrate the difference between absolute and relative attachments, this example shows two components attached to their common parent.

Figure A2.3 The top item uses an absolute attachment and the bottom uses a relative attachment.

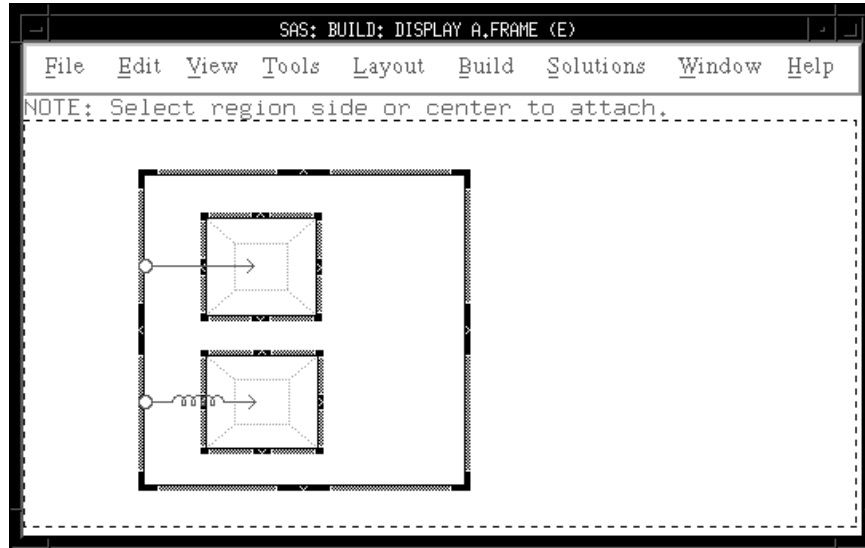
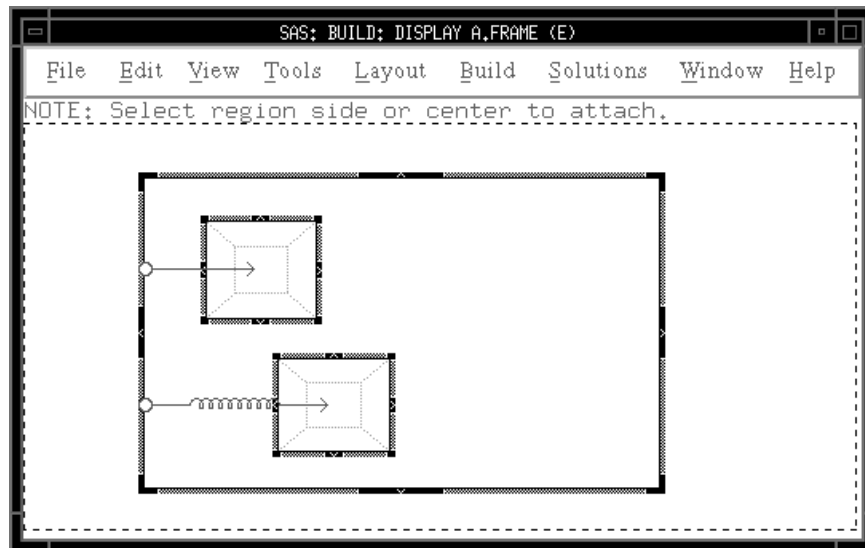




Figure A2.4 If the size of the common parent grows from either the right or left side, the absolute attachment preserves the actual distance between the top item and the left edge of the parent, and the relative attachment preserves the percentage distance between the bottom item and the left edge of the parent.



The attachment direction controls which component is affected when the attached component is resized or moved. Attachments can be single directional or bidirectional:

Table A2.2 Attachments

Attachment Type	Direction	Effect
bidirectional	↔	Both components respond to resizing or moving either component.
single directional		The component that the arrow points to responds to resizing or moving the component that is anchored by the end node.  is the default.

Resizing or moving a component is considered to be propagated by an attachment if resizing or moving a component, combined with the attachment, results in another component being resized or moved. An attachment is considered to be honored if resizing or moving a component is propagated because of the attachment.

In the following window, components a, b and c have a common parent component and are all attached to the right side of their parent with absolute attachments.

Figure A2.5 a's attachment is bidirectional; b's attachment is single directional going out of b; and c's attachment is single directional going into c.

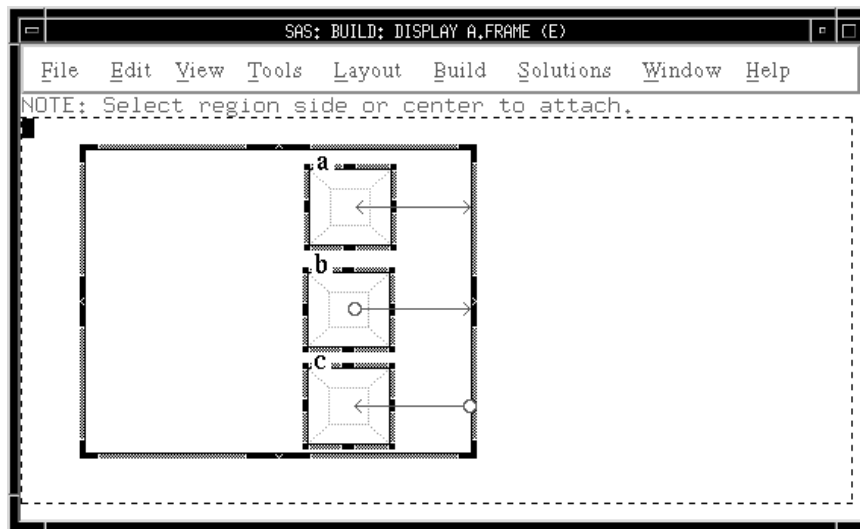


Figure A2.6 If the right side of the common parent is moved further to the right, only a and c move because the parent generated the initial resize event and the direction of a's and c's

attachments allowed the attachment to be honored. The attachment associated with *b*, however, was given a new value.

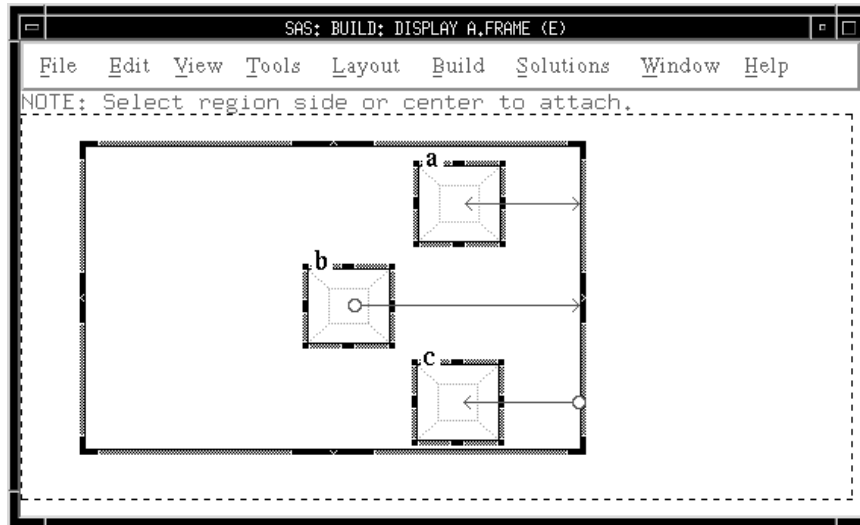


Figure A2.7 Notice that if *b* moves to the left, the right hand side of the parent region is pulled to the left since *b*'s attachment is honored only from *b*. Resizing the parent results in honoring the attachments to *a* and *c*. If *a* is moved, the parent will also move because *a*'s attachment is bidirectional.

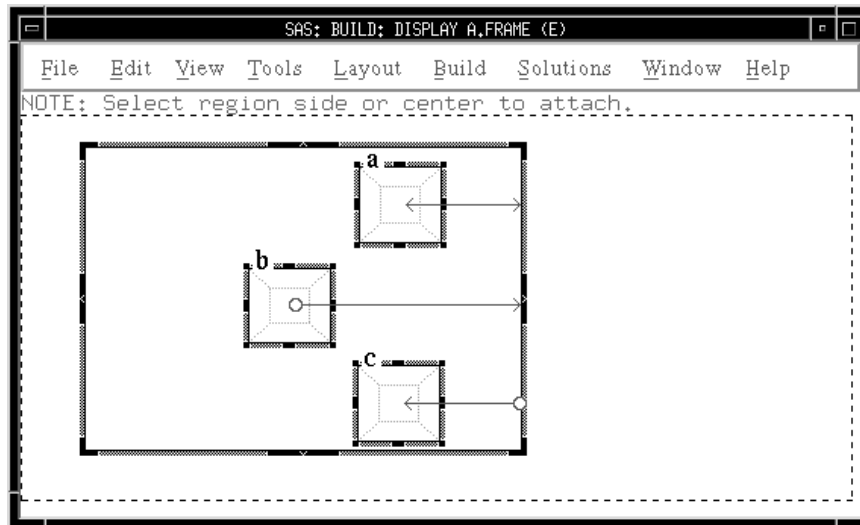
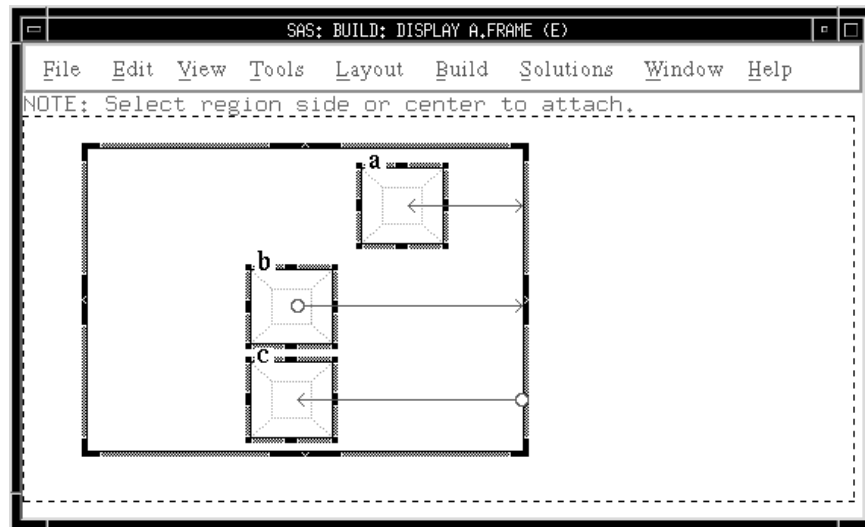


Figure A2.8 If *c* is moved, only *c* moves. Because the attachment associated with *c* is honored only coming into *c*, resizing *c* is not propagated (that is, resizing *c* does not move

or resize other regions). Note that the attachment from the parent region to *c* is given a new value in this case.



Making the Attachments

After you initiate define attachment mode, you actually define the attachments by clicking the mouse button in one component and dragging it to another. Each component can be attached at one of five different points: each side of a component and its center.

You can define only one attachment from each of these points. However, you can create many attachments to a single component section by defining the attachments at an attachment point from another component.

The section that owns the attachment is the section from which you initiate the drag. This section is referred to as the defining section. Knowing which section defines that attachment is important when you perform certain actions on the attachment. For example, to delete an attachment, you select {pseudo} in the Define Attachments window and then click in the defining section for that attachment.

Attaching the components from the side and from the center causes different results:

Table A2.3 Attachment Points

To...	Attach the regions from the...
resize the child component when the parent component is resized	side
move the child component when the parent component is resized	center

You can use side attachments to control component size, as explained in the following example:

Figure A2.9 Both a and b have their right sides attached to their common parent's right side, and b also has its bottom attached to its parent's bottom.

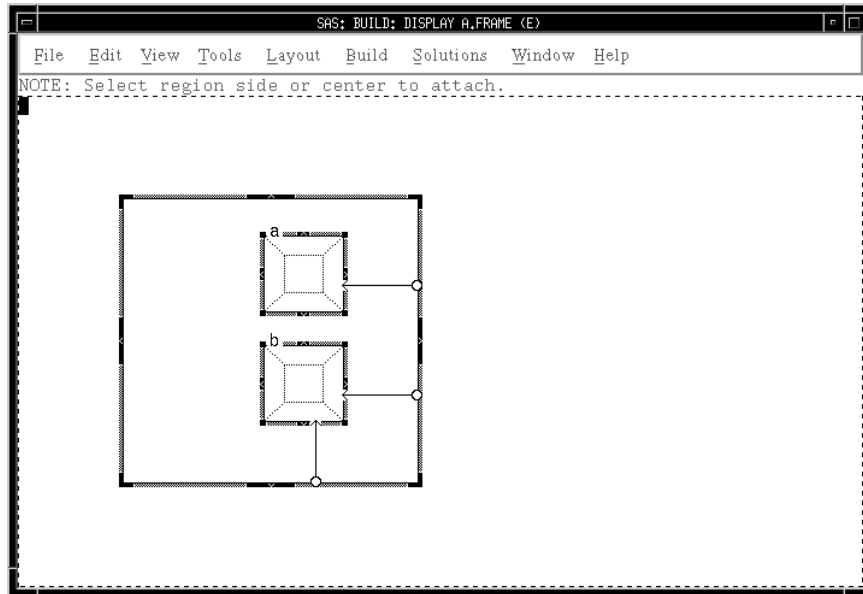


Figure A2.10 If the parent grows from the right side, the right side of both a and b move along with the parent's right side, but the left sides of a and b do not.

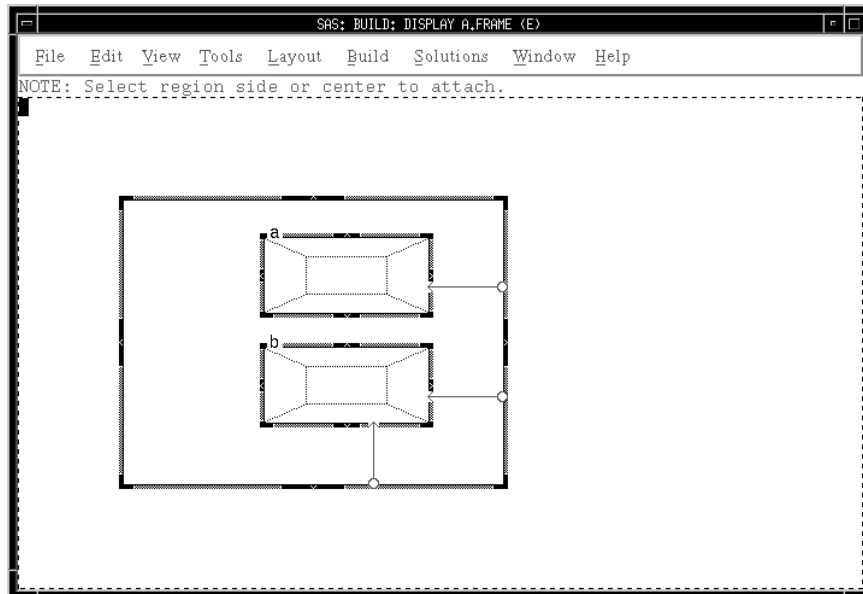
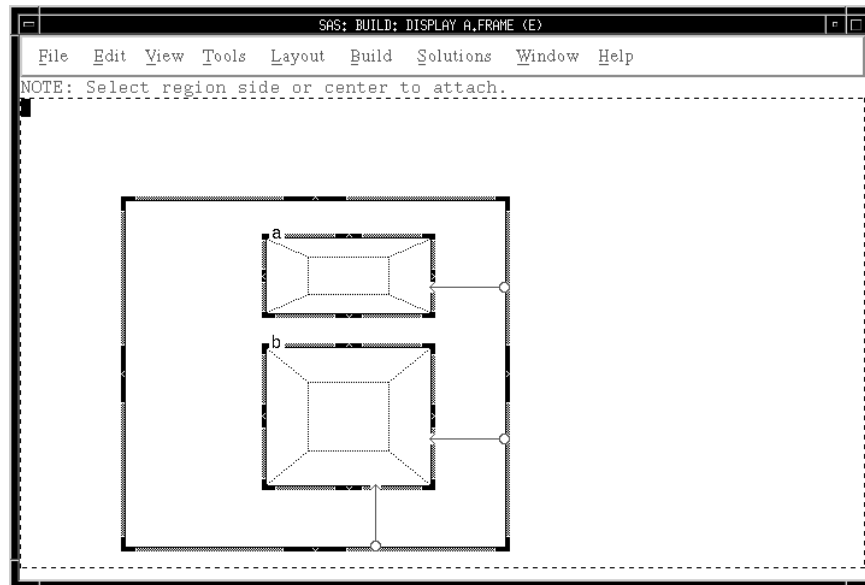


Figure A2.11 If the parent grows by moving the bottom down, the bottom of *b* moves down as well.



Defining Attachments to Sibling Components

Attachments can be further classified by whether they attach the defining component to its parent or to a sibling component. Sibling attachments connect a component to one of its siblings. All of the previous examples have shown parent attachments.

Sibling attachments can often simplify a set of attachments by making it visually easier to interpret the attachment logic. In addition, this type of attachment sometimes results in fewer attachments.

Figure A2.12 *a, b, and c should all follow the right side of their common parent. This can be accomplished by attaching all regions to the parent directly, but using sibling attachments is easier to understand.*

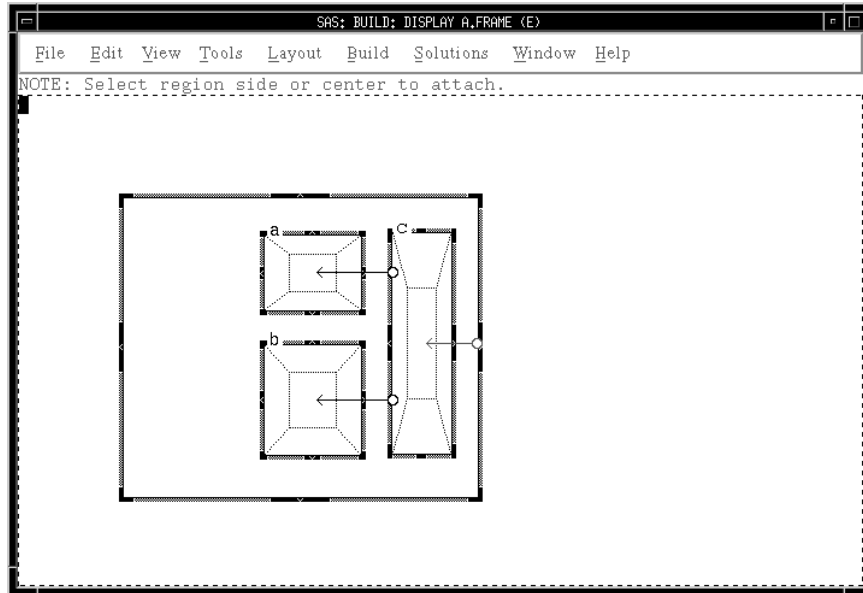
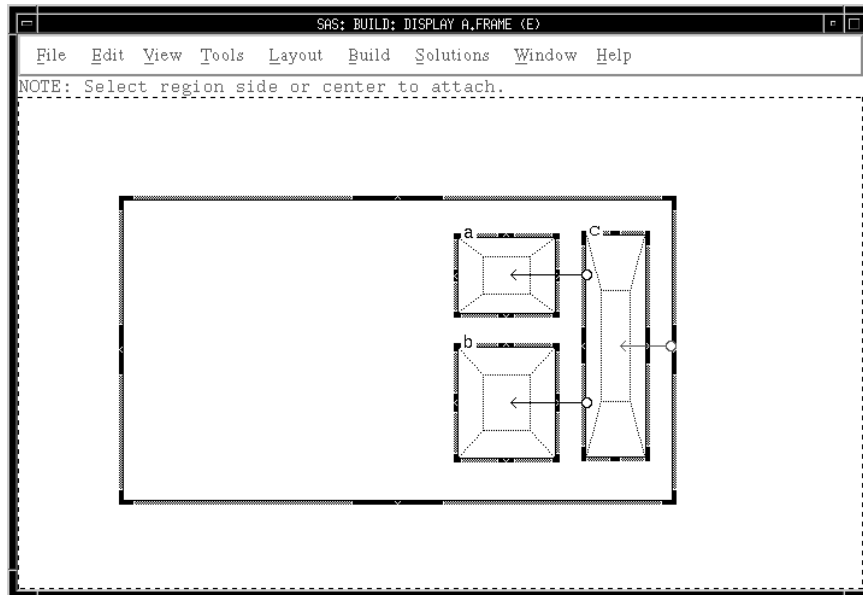


Figure A2.13 *If the right side of the parent grows to the right, all of the sibling regions move to the right as well.*



Defining Attachments to Components That Have Borders

Some components may have special outlines or scroll bars. By default, when an attachment is attached to the side of a component, the actual attachment point is

considered to be the edge of the component itself. This point is along the inside edge of any special outline that the component may have. It is also inside any attached scroll bars that may be associated with the component.

External attachments enable you to create attachments with attachment points on the outer edge of the component dressing. This type of attachment is often necessary when the component dressing is a fixed number of pixels, as it is with component outlines and component attached scroll bars. To create an external attachment, you should initiate rubber-band line mode with an extended drag operation instead of a normal drag.

External attachments look different from other attachments. They use a thicker line, and the terminating point of the directed arrow is the component border instead of the component itself.

Moving Multiple Components That Include Attachments

When more than one component is selected and moved at the same time, the attachments between these components, or between these components and other components, are not honored. Instead, attachments are temporarily ignored while the components are moved to their new locations. Attachment values are then recalculated and reset so that they are correct for the new location.

Example

Three push buttons are placed horizontally in a row. Push button 1 is bidirectionally attached on its right side to the left side of push button 2. Push button 2 is bidirectionally attached on its right side to the left side of push button 3. Additionally, there are two pixels between each push button.

If you select push button 1 and push button 2 and move them to the left by two pixels, push button 3 will not grow to the left by two pixels to keep an appropriate distance between itself and the selected push buttons. Instead, push button 1 and 2 are moved to the left and the distance between them is recalculated. Push button 3 now has four pixels between itself and push button 2 instead of two pixels. Now, when either push button 2 or push button 3 is moved by itself, the attachment maintains a four-pixel distance instead of a two-pixel distance.

Restricting Component Size

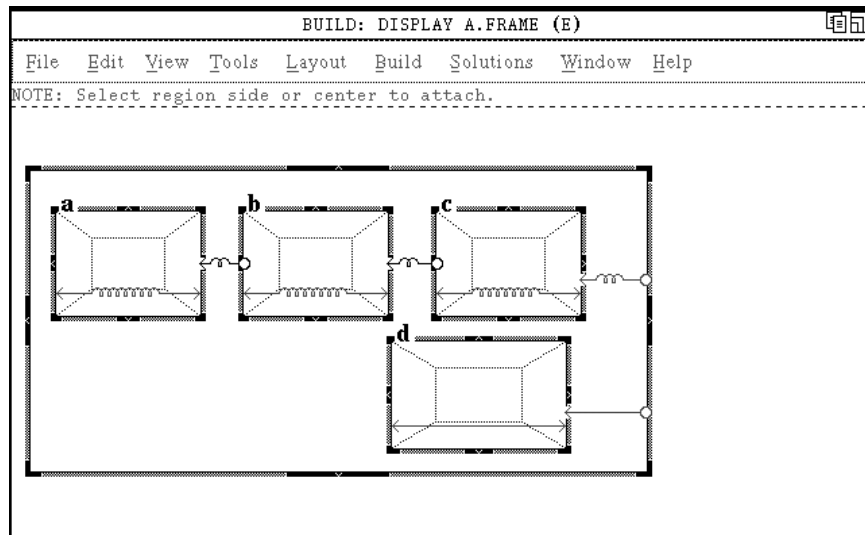
You can restrict component resizing by displaying a pop-up menu for a component while you are in define attachment mode. You can make the component size fixed absolute or fixed relative, or you can reset the fixed status of the component (remove any restrictions on resizing). There are separate horizontal and vertical settings, so it is possible to set a size restriction for a component in only one direction. By default, the size of components is not restricted.

If a component is fixed absolute, then its size may not change in the direction (either horizontal or vertical) that is fixed. If the component is fixed relative, then its size will be maintained as a certain percentage of its parent component's size in the appropriate direction.

The resize restrictions absolute and relative look similar to absolute and relative attachments.

The following figures illustrate a common use of fixed absolute and fixed relative component size restrictions.

Figure A2.14 *d* is fixed absolute and side attached absolutely to the right side of its parent. This attachment causes *d* to move with the right side of its parent and is equivalent to attaching to *d*'s center. Using this method instead of center attachment is a matter of personal preference. *a*, *b*, and *c* share the expansion of the parent equally.



Notice in this case that the fixed relative restrictions are required in order to achieve the desired result. If they are left off, then the right side of *c* will be resized, and the distance between *c* and *b* will be set correctly, but the actual size of *c* will be undefined.

Figure A2.15 *If the right side of the parent is grown, a, b, and c grow in proportion to the parent, and their respective positions are relatively preserved. d moves along with the right side of the parent.*

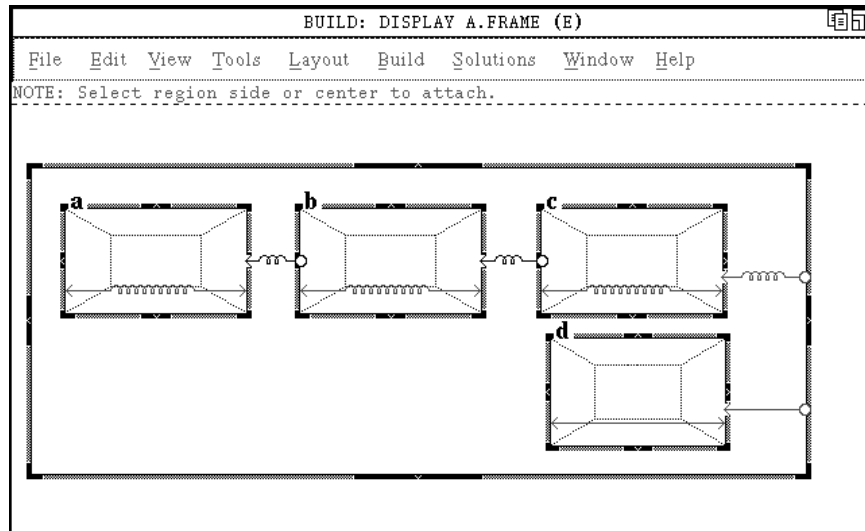
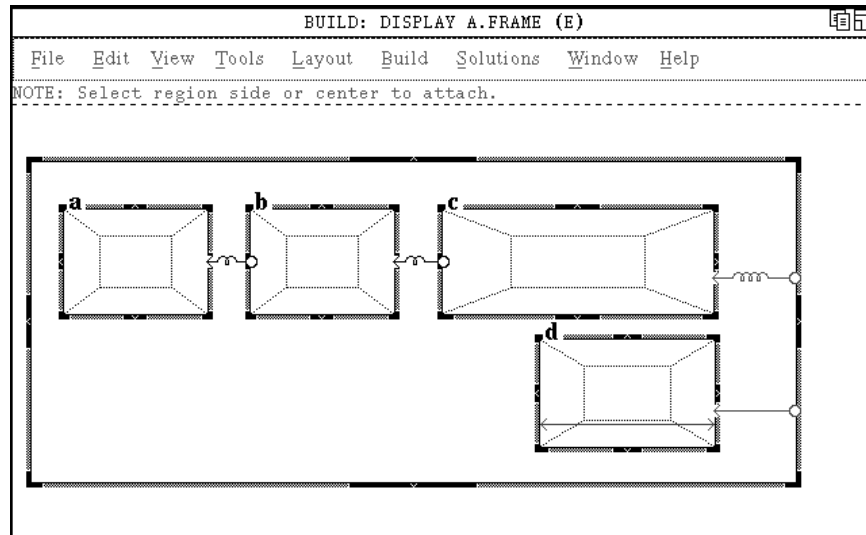


Figure A2.16 By contrast, without any relative size restrictions on *a*, *b*, and *c*, resizing the parent causes the right sides of both *c* and *d* to move to the right, preserving the relative and absolute distances respectively.



However, since there are no size restrictions, the left side of each component is not relocated. The distance between *b* and *c* is adjusted, but the right side of *b* moves since the direction of the attachment is into *b*. The same effect occurs for *a*. The distances between the components are maintained as required due to the attachments, but the resultant component sizes are not clearly defined.

Changing and/or Deleting an Attachment

You can correct your attachment definition by deleting the current attachment and defining a new attachment in its place.

When you want to delete an attachment, open the Define Attachment window and select the delete attachment symbol (O—).

Then click in the section that defines the attachment.

Displaying Attachments

If you click within a section while in define attachment mode and an attachment is defined from that section, a message reports exactly what kind of attachment it is. This information is useful when the connection points are so close together that it is difficult to determine the attachment type and direction from the visual indicator. For example, if you click on the right side of the *d* component in the previous window, the following message is displayed:

NOTE: Relative attachment (in) of right to right of parent.

You can also view the attachments for a component or a component's child components without entering define attachment mode. To do this, issue the following command:

```
rm showatt [all|kids]
```

If you issue just the `RM SHOWATT` command, the attachments for the current component are shown. An arrow shows the attachments, and the message line displays the following message:

NOTE: Press any key to return.

If you issue `RM SHOWATT KIDS`, the attachments for all of the active component's child components are shown. If you specify `RM SHOWATT ALL`, all attachments for all components that are descended from the current component are shown.

After you use `RM SHOWATT`, the visual attachment indicators remain displayed until you press a mouse button or a keyboard key.

Situations in Which an Attachment Is Ignored

These situations cause an attachment to be ignored:

- The component is resized only horizontally, and the attachment is vertical in nature, or vice versa.
- The attachment is attached to the side of a sibling, and the location of that side has not changed.
- The component that is currently being resized is attached to the component that was originally moved or resized to propagate the event.
- The attachment is single directional coming into the component instead of going out of the component.
- The actual distance in pixels between the two attached components is within one-half character of the desired distance. This situation applies to both absolute and relative attachments. This shortcut is taken when a component is resized only if the component to be adjusted must be aligned on a character cell boundary or if the attachment has been previously synchronized.
- The attachment is between one component in a group of multiple-selected components and another component, and the group of multiple-selected components is moved.
- The attachment is associated with a component that uses the `conformSize` attribute. Attachments have the potential to conflict with the `conformSize` attribute.

Errors and Error Handling

Whenever the region manager detects an error during the resizing of a component, the components are all restored to the location and size that they had before the component was resized. Here are the most common reasons that resizing a component fails:

- When the component is resized, the component becomes too small for the object that it contains.
- An attachment cannot be honored. Check all your attachments for conflicts.
- Honoring an attachment requires resizing the master component. The master component can only initiate a resize event and cannot be resized as a result of resizing another component. Check all your attachments to the master component.

- Resizing the component results in a component being placed outside of the boundaries of its parent (unless its parent is a viewport component, as used by the work area object). This typically implies that a required attachment is missing.
- Resizing the component results in a component being smaller than allowed by the region manager (eight pixels). This situation does not necessarily imply that the attachments are in error. You may simply be attempting to shrink the original component too much.
- Resizing the component causes components that contain text objects to overlap, which usually implies an error in the way the attachments are defined.

If, after examining the attachments and the results of attempted resizes, you do not understand the attachment problem, use the region manager to trace the handling of the resizing by issuing the command `RM GROW DEBUG`. The next time you try to resize the component, diagnostic messages will be sent to the Log window.

Tips for Using Attachments

By far the most important aspect of defining attachments for a FRAME entry is to carefully think about how you want various components and contained components to react to the appropriate resize event.

Carefully examine the components and determine which components should absorb the gain or loss of space. Examine the horizontal and vertical dimensions separately; it is fairly common for each dimension to handle resizing differently.

- Use only the attachments.

When a component is resized, all attachments are checked; therefore, unnecessary attachments may affect performance. Also, attachment information is stored with the FRAME entry, so unnecessary attachments increase the size of the stored entry.

The same is not true for component size restrictions. Virtually no overhead is paid for these settings. Sometimes, you may want to turn on a fixed size restriction, even if only for documentation purposes; that is, if the attachments are viewed by another person at a later date, they will be more understandable.
- Remember that the master component always resizes from the lower right.

This is true even if the master component is grown from the upper left of the window. Therefore, absolute attachments to the top or the left of the master component are unnecessary. Note that this is not true of components in general, only of the master component.
- Take advantage of Move Only situations.

If you know that a container or group component will only be moved and not resized due to attachments to its siblings and parent, then you do not need to define any attachments to the child components of the container or group.
- Keep the attachment logic simple.

If the attachments are getting too complicated, consider creating a container box just to hold a set of related components. Attach the container to its parent and siblings and set up attachments (if necessary) to the container's children.
- Avoid mixing absolute and relative attachments in the same direction.

It is easy to create a situation that cannot be satisfied when mixing absolute and relative attachments in the same direction. Sometimes such situations are

appropriate, but you should take a second look to make sure this is really what you need.

- Avoid relative attachments with text-based objects that must be aligned on a row/column boundary.

Usually the desired placement of a component after honoring a relative attachment does not exactly align on a row/column boundary, which necessitates a shift of the component to attain the required alignment. This situation can create odd visual effects, and it significantly increases the chance that textual objects will overlap. If you feel you must use relative attachments with text-based objects, be sure to leave space around each component to allow for shifting without creating an overlap situation.

A related problem occurs when a container contains character-aligned objects (widgets) and the container is relatively attached. The container is not character-cell aligned, and when the master component is resized, the widgets often shift within the container.

You can avoid this shifting by creating a dummy character-cell-aligned component (for example, a one-character text field that is protected and nondisplayable). Attach this dummy component to the master component relatively, and then attach the container to the dummy component absolutely. Make sure the absolute attachment is short so that the resulting location of the container is as close to relatively correct as possible.

When the master component is resized, the dummy component will be positioned relatively, and then it will align to a character-cell boundary. Then, the absolute attachment between it and the container will be honored. No shifting will occur within the container either since the net movement of the container is guaranteed to be an integral number of character cells. This guarantee is possible because the master component is always an integral number of character cells in size and the dummy component is always character-cell aligned.

- Remember that you can resize in define attachment mode.

Remember that you can still resize any component without attachments being honored by entering define attachment mode and performing the desired move or resize. This action causes any associated attachments to be redefined instead of honored.

- Create small and allow for growth.

In general, results are usually better when honoring the growth rather than the shrinkage of the master component. *You are better off defining your window to be as close to the smallest allowable size and ensuring that you handle size increases gracefully through attachment.* The primary reason for this is that if you create a window too large and then try to shrink it, the master component may become too small, and the containing component may also become too small to hold other components.

- Be careful when using very short attachments.

Avoid very short attachments to a component that contains a component that must be character-cell aligned. These attachments are most likely to be ignored because of the one-half character rule mentioned earlier. Short attachments are acceptable if placement or size variations within $\frac{1}{2}$ character are insignificant and if propagation is not an issue. Such attachments are a concern only when you are attaching to a component that contains an object that must be character-cell aligned.

- Test your attachments.

You can test all attachments while in build mode. Position the components, create the attachments, and save the entry so that the desired placement is not lost (in case the attachments do not work as you expect them). If the attachments do not work, then cancel them, re-edit the frame, and change the attachments appropriately.

- Use single-directional attachments in most cases.

Single-directional attachments are usually most appropriate, especially for attachments from parent to child. Sibling attachments can also be made single directional, but be sure to analyze how resizing the master component will affect subcomponents so that you define sibling attachments in the correct direction.

- Use relative component size restrictions to reduce the number of attachments required.

If you want a component to maintain a relative size, attach one side to the parent relatively and set the component-size restriction to fixed relative, instead of attaching both sides relatively. This technique results in significantly fewer attachments if you have many components that are being sized relatively in this manner.

- Check the kind of attachment for a component if you are not sure what kind is defined.

To determine the kind of attachment that is defined for a section, click in the section while you are in define attachment mode. A message will be displayed indicating what kind of attachment is defined there and what it is attached to.

Appendix 3

Working with SAS/AF Keys in the SAS Registry

About the SAS Registry	197
SAS Registry Editor	197
Introduction	197
Keys and Key Values	198
Modifying the Registry Settings for SAS/AF Software	198

About the SAS Registry

The SAS Registry stores configuration data about the SAS session and about specific applications, such as SAS/AF software.

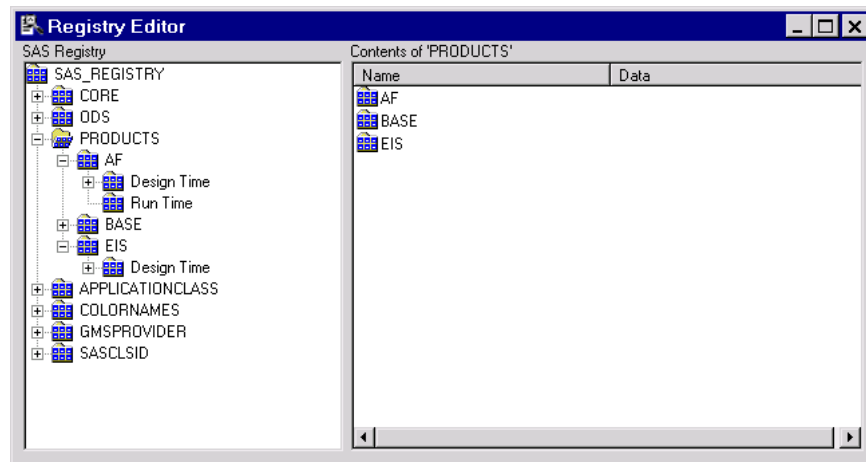
You can use the SAS Registry to customize your SAS session as well as SAS applications. Customization involves modifying configuration data, key values, or both. For example, you can use the SAS Registry to customize whether the Components window appears by default when you open a frame in SAS/AF software.

You can access the SAS Registry with the SAS Registry Editor or with PROC REGISTRY. The information in this appendix is based on using the SAS Registry Editor. For more information on PROC REGISTRY, see the Base SAS Procedures Guide.

SAS Registry Editor

Introduction

The SAS Registry Editor enables you to view and modify the SAS Registry. It provides a graphical alternative to PROC REGISTRY. To open the SAS Registry Editor, select **Solutions** ⇒ **Accessories** ⇒ **Registry Editor** or issue the REGEDIT command.



You can use the SAS Registry Editor to complete a number of tasks, including

- viewing the contents of the registry, which shows keys and key values
- adding, modifying, and deleting keys and key values.

Keys and Key Values

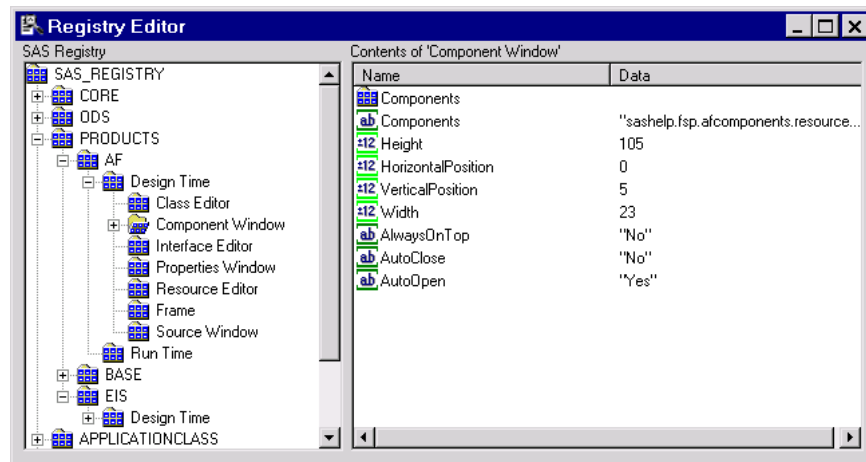
Values in the registry are stored in keys. Keys look like folders in the SAS Registry Editor, and appear on the left hand side (the tree view) of the window. The values stored in the currently selected key are displayed on the right hand side (the list view) of the window. If a key contains only subkeys, it has a plus sign (+) next to its folder.

You can expand or collapse a key to show or hide its subkeys by selecting the key's expansion icon (either a + or - sign), which is located to the left of the key's folder.

Note: In some operating environments, you can select an expansion icon by positioning the cursor on it and pressing the ENTER key.

Modifying the Registry Settings for SAS/AF Software

You can use the SAS Registry to modify settings that control the SAS/AF software design-time and run-time environments.



For example, you may want to change the resource that is associated with the Components window by default, or you may want the Properties window to appear without its navigation tree.

To set SAS/AF registry settings:

1. Open the SAS Registry Editor.
2. Expand the **Products\AF\Design Time** or **Products\AF\Run Time** key.
3. Select a key in the left pane of the window.
4. Select a key value in the right pane of the window, and then right-click to open the pop-up menu.
5. Select the appropriate command from the pop-up menu. Some commands invoke dialog boxes that enable you to edit the key value.

Glossary

application workspace

a window that contains other windows or from which other windows can be invoked, but which is not contained within any parent window that is part of the same software application. Short form: AWS.

attribute

a property of a SAS/AF component that specifies the data that is associated with a component, such as its color, size, description, or other information that is stored with the component. Although similar to instance variables, attributes can contain additional data.

attribute linking

a feature of the SAS/AF graphical user interface that enables you to designate values to be shared by multiple components without writing any SCL code. You can define attribute links between attributes of the same component or between different components within the same frame.

automatic instance variable

an instance variable whose value is automatically copied into the corresponding SCL variable when an SCL method executes, and is copied back into the object when the method returns. Assigning the automatic status to an instance variable makes writing new methods for a class easier because the SCL method can access the automatic instance variable directly instead of accessing it indirectly with SCL list functions. See also instance variable.

AWS

See application workspace.

CAM

See custom access method.

catalog entry

See SAS catalog entry.

catalog reference

a name that is temporarily associated with a catalog or with concatenated catalogs. You use a CATNAME statement to assign a catref. Short form: catref.

catref

See catalog reference.

class

a template for an object. A class includes data that describes the object's characteristics (such as attributes or instance variables), as well as the operations (methods) that the object can perform. See also subclassing and object.

complex attribute

an attribute whose type is either Object (O) or a fully qualified SAS/AF class name such as `sashelp.classes.draganddrop.class`. Using an object as an attribute enables you to use a single attribute to store a set of information. In SCL, you can access the attributes of a complex attribute object by 'stringing' together the dot notation that queries or sets the attribute. The objects that are used as complex attributes can have their own methods. A complex attribute must be a non-visual component (that is, one that is not based on `sashelp.classes.widget.class`).

component

a self-contained class that has specific properties, which can include attributes, methods, events, event handlers, and interfaces.

control

another term for visual component. See also visual component.

custom access method

a method that is automatically executed when an attribute is accessed. Short form: CAM. See also `getCAM` and `setCAM`.

data representation

the type of data that a drag site is capable of transferring or receiving. The data can be as simple as a string of text or as complex as an SCL list. See also drag site, drag and drop (DND), and drop operation.

dimension

a group of closely related hierarchies. Different hierarchies within a single dimension typically represent different measurements of a single concept. For example, a Time dimension might consist of two hierarchies: (1) Year, Month, Date, and (2) Year, Week, Day.

DND

See drag and drop.

drag and drop

a user interface action that involves dragging objects with a mouse (or other pointing device) and dropping them over other objects at run time, and having some action occur. On systems that do not use a mouse or other pointing device, you can drag an object by using the `WDRAG` command and cursor-key input to position the object and complete the drag action. Short form: DND.

drag site

a component that can be dragged. See also drag and drop.

drop operation

any of the following actions that can be performed on a data representation: (1) copy, when data is provided with no post-processing; (2) link, when some mechanism synchronizes the source and destination; (3) move, when data is provided and the source is removed.

drop site

a component that can receive a dragged object. See also drag and drop.

entry type

a characteristic of a SAS catalog entry that identifies the catalog entry's structure and attributes to SAS. When you create a SAS catalog entry, SAS automatically assigns the entry type as part of the name. See also SAS catalog entry.

event

a property of a SAS/AF component that notifies components or applications when a resource or state changes. An event occurs when a user action (such as a mouse click) occurs, when an attribute value is changed, or when a user-defined condition occurs. An event handler then provides a response to the change.

event handler

a property of a SAS/AF component that determines the response to an event. Essentially, an event handler is a method that executes after an event is received.

field

a window area in which users can view, enter, or modify a value.

field validation

the process of checking user-entered values either against attributes that have been specified for a field or against conditions that have been specified in an SCL program.

getCAM

a method that is associated with an attribute and that is automatically executed when the attribute's value is queried. See also custom access method and setCAM.

global command

a command that is valid in all of a particular SAS software product's windows.

graphics object

a graphical window element that displays a variety of graphics output created dynamically from specified data.

inheritance

in object-oriented methodology, the structural characteristic of class definitions by which methods and attributes of a class are automatically defined in its subclasses.

instance

another term for object. See also object.

instance variable

a characteristic that is associated with an object, such as its description, its color or label, or other information about the object. All objects that are created from the same class automatically contain the instance variables that have been defined for that class, but the values of those variables can change from one object to another. In addition, objects can contain local instance variables (that is, variables that are local to a particular instance of a class). Beginning in Version 8, the use of attributes reduced the need for instance variables. See also automatic instance variable and attribute.

interface

a type of class that specifies the rules that are used for model/view communication. Interfaces are collections of abstract method definitions that enable you to redirect a method call from one component to a method call on a different component. The method definitions are just the information that is needed to define a method; they do not contain the actual method implementations. If two components share an interface, they can indirectly call each others' methods via that interface.

KEYS entry

a type of SAS catalog entry that contains function key settings for interactive windowing procedures.

legacy class

any SAS/AF class that does not use the SAS Component Object Model (SCOM), including classes that were supplied in Version 6 of SAS/AF Software.

library reference

See libref.

libref

a name that is temporarily associated with a SAS library. The complete name of a SAS file consists of two words, separated by a period. The libref, which is the first word, indicates the library. The second word is the name of the specific SAS file. For example, in VLIB.NEWBDAY, the libref VLIB tells SAS which library contains the file NEWBDAY. You assign a libref with a LIBNAME statement or with an operating system command.

local environment list

a list that contains data that is available only to SCL entries that are invoked in the same SCL application. This list is deleted when the application ends.

member

a SAS file in a SAS library.

member name

a name that is assigned to a SAS file in a SAS library.

metadata

a description or definition of data or information.

method

a property of a SAS/AF component that defines an operation that the component can execute.

model

a type of non-visual component that provides attributes and methods for querying and modifying underlying data abstractions. For example, a Data Set List model contains methods for reading and manipulating SAS tables.

model/view

an abstract relationship between a model, the internal logic of applications, and a viewer. You can set model/view component communication during application build time (that is, within the Build window) either by dragging a model onto a viewer or by setting the 'model' attribute of a viewer in the Properties window. See also model, view, and viewer.

non-visual component

any component that is not visible in an application's graphical user interface.

object

an instantiation or specific representation of a class. For example, a list box on a frame is an instantiation of `sashelp.classes.ListBox_c.class`. The terms instance and object are synonyms.

overload

to define multiple methods that have the same name but different signatures within the same class. If you call an overloaded method, SCL checks the method arguments, scans the signatures for a match, and executes the appropriate code. See also signature.

override

to replace the definition of an inherited method or the default value of an instance variable in a subclass.

parameter

(1) in SAS/AF and SAS/FSP applications, a window characteristic that can be controlled by the user. (2) in SCL, a value that is passed from one entry in an application to another. For example, in SAS/AF applications, parameters are passed between entries by using the CALL DISPLAY and ENTRY statements. (3) a unit of command syntax other than the keyword. For example, NAME=, TYPE=, and COLOR= are typical command parameters that can be either optional or required.

parent class

the class from which another class is derived. The parent class is also known as a super class.

property

any of the characteristics of a component that collectively determine the component's appearance and behavior. Attributes, methods, events, and event handlers are all types of properties.

region manager

the portion of SAS that allows windows to consist of regions. The region manager controls the operations performed on regions.

resource

another term for RESOURCE entry. See also RESOURCE entry.

RESOURCE entry

a type of SAS catalog entry that stores information about a set of classes. This information determines which classes can be instantiated by a frame when that frame is initialized. Therefore, when you browse, edit, or execute a frame, the frame must be able to access the RESOURCE entry that was used when the frame was created.

resource list

the list of classes and sets of attributes that can be used to build objects and execute FRAME entries. Resource lists are stored in RESOURCE entries.

SAS catalog

a SAS file that stores many different kinds of information in smaller units called catalog entries. A single SAS catalog can contain different types of catalog entries. See also SAS catalog entry.

SAS catalog entry

a separate storage unit within a SAS catalog. Each entry has an entry type that identifies its purpose to SAS. Some catalog entries contain system information such as key definitions. Other catalog entries contain application information such as window definitions, Help windows, SAS formats and informats, macros, or graphics output. See also entry type.

SAS Component Language

a programming language that is provided with SAS/AF and SAS/FSP software. You can use SCL for developing interactive applications that manipulate SAS data sets and external files; for displaying tables, menus, and selection lists; for generating SAS source code and submitting it to SAS for execution; and for generating code for execution by the host command processor. Short form: SCL.

SAS Component Object Model

a framework for developing SAS/AF applications that provides developers with model/view communication, drag and drop communication, and attribute linking. Short form: SCOM. See also model/view, drag and drop, and attribute linking.

SAS data set

a file whose contents are in one of the native SAS file formats. There are two types of SAS data sets: SAS data files and SAS data views. SAS data files contain data values in addition to descriptor information that is associated with the data. SAS data views contain only the descriptor information plus other information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors' file formats.

SCL

See SAS Component Language.

SCOM

See SAS Component Object Model.

scope

a value that indicates whether a property can be called by all classes, by only the parent class, or only by subclasses. Scope is set to Public, Private, or Protected. Scope also determines whether properties are displayed at build time. For example, only public attributes are displayed in the Properties window. By default, the scope of a property is Public.

selection list

a list of items in a window, from which users can make one or more selections. Sources for selection lists are LIST entries, special SCL functions, and extended tables.

setCAM

a method that is associated with an attribute and that is automatically executed when the attribute's value is set. See also custom access method and getCAM.

signature

the name, order, and type of arguments for a method.

sigstring

a short notation that indicates a method signature.

source code

programming code that must be compiled before it can be executed.

source entry

a SAS catalog entry that contains the source code for a method. Source entries typically have an entry type of SCL.

state

a value that indicates whether a property is new, inherited, or overridden, or whether it is a system property.

subclassing

the process of deriving a new class from an existing class. A new class inherits the characteristics (attributes or instance variables) and operations (methods) of its parent. It can also possess custom attributes (or instance variables) and methods. See also class, attribute, instance variable, and method.

super method

a method that is replaced by an overridden method. The super method can be invoked from the overridden method with a CALL SUPER statement.

type

the data type of a variable, an attribute, or a method argument.

view

a particular way of looking at a model's data.

viewer

a component that provides a visual representation of a model's data.

visual component

a component such as an icon, a push button, a check box, or a frame that forms part of the graphical user interface of an application. Visual components are also referred to as controls.

widget

an element of a graphical user interface that displays information or that accepts user input. For example, a text entry field is a widget that is used for displaying and entering text, and a chart is a widget that is used for displaying information.

Index

Special Characters

- [_completeDrag method](#) 143
- [_drop method](#) 143
- [_getAttributeValue method](#)
 - definition 88
 - flow of control 88
- [_getDragData method](#) 143
- [_initLabel method](#) 113
- [_mainLabel method](#) 113
- [_objectLabel method](#) 114
- [_respondToDragOff method](#) 143
- [_respondToDragOnto method](#) 143
- [_setAttributeValue method](#)
 - definition 88
 - flow of control 88
- [_startDrag method](#) 143
- [_super\(\) routine](#) 110
- [_termLabel method](#) 113
- [_validateDropData method](#) 143

A

- [absolute attachments](#) 182
- [abstract classes](#) 76
- [application window features, enabling/
disabling](#) 62
- [application workspace \(AWS\)](#) 62
- [attachment mode, selecting](#) 180
- [attachments](#) 25
 - changing 192
 - defining 179
 - deleting 192
 - displaying 192
 - error handling 193
 - ignoring 193
 - tips for using 194
- [attachments, adding to frame controls](#) 179
 - absolute attachments 182
 - attachment mode, selecting 180
 - bidirectional attachments 183
 - child attachment mode 180

- [components, moving multiple](#) 190
- [components, restricting size of](#) 190
 - creating 186
- [current attachment mode](#) 180
- [define attachment mode, initiating](#) 181
- [defining sections](#) 186
- [direction, selecting](#) 182
 - for components with borders 189
 - for sibling components 188
- [ownership](#) 186
- [propagation](#) 183
- [relative attachments](#) 182
 - single directional attachments 183
 - type, selecting 182
- [Attribute Changed Event component](#) 131
- [attribute editors](#) 117, 118
 - assigning 117
 - creating your own 118
- [attribute links](#) 28
 - [Attribute Changed Event component](#) 131
 - definition 28
 - enabling 130
 - establishing 28
 - example 29, 131
 - uses for 28
- [attribute metadata items](#)
 - [AutoCreate](#) 85
 - [Automatic](#) 88
 - [Category](#) 87
 - [definition](#) 84
 - [Description](#) 88
 - [Editable](#) 86
 - [Editor](#) 87
 - [GetCAM](#) 87
 - [HonorCase](#) 87
 - [InitialValue](#) 85
 - [IV \(instance variable\)](#) 88
 - [Linkable](#) 86
 - [Name](#) 85
 - [Scope](#) 85

- SendEvent 86
 - SetCAM 87
 - State 85
 - TextCompletion 87
 - Type 85
 - ValidValues 86
 - attributes
 - adding 99
 - attributeName 144
 - CBTFrameName, context sensitive help in frames 54
 - CBTFrameName, online Help in frames 54
 - completeDrag 145
 - custom access methods (CAMs), assigning 121
 - dataOperation 145
 - dataRepresentation 145
 - dataSiteID 145
 - defaultAttribute, specifying 115
 - definition 84
 - dependencies 125
 - dot notation 88
 - dragEnabled 142
 - dragInfo 142
 - dragOperations 142
 - dropEnabled 143
 - dropInfo 143
 - dropOperations 143
 - getting 88
 - help, context-sensitive help in frames 54
 - help, online Help in frames 54
 - helpText 54
 - list attributes 127
 - metadata 84
 - refresh Attributes event, example 126
 - refresh Attributes event, using 125
 - showContextHelp 54
 - toolTipText 54
 - validating character values 116
 - values, getting and setting 88
 - XLocation 145
 - YLocation 145
 - attributes, setting
 - for entire component 119
 - singly 117, 118
 - with attribute editors 117, 118
 - with custom attributes windows 119
 - with dot notation 88
 - AutoCreate attribute metadata item 85
 - Automatic attribute metadata item 88
 - AUTOTERM feature 167
 - AUTOTERM= option 167
 - AWS (application workspace) 62
 - See also* application workspace (AWS)
 - AWSCONTROL system option 62
 - AWSMENUMERGE system option 62
 - AWSTITLE system option 62
- B**
- bidirectional attachments 183
 - BUILD procedure, deploying SAS/AF applications 58
 - Build window 15
 - BYE command 167
- C**
- CAMs (custom access methods) 121
 - See also* custom access methods (CAMs)
 - CANCEL command 167
 - catalogs, creating SAS SAS/AF applications 6
 - Category attribute metadata item 87
 - child attachment mode 180
 - child classes 74
 - CIMPORT procedure, deploying SAS/AF applications 59
 - Class Browser 70
 - class catalogs, renaming 154
 - class constructors
 - adding 107
 - defining with Class Editor 107
 - defining with SCL 106
 - overriding 107
 - overview 105
 - class documentation
 - generating 156
 - making available 159
 - Class Editor 67
 - defining constructors 107
 - CLASS statement 103
 - classes 73
 - abstract 76
 - ancestry 74
 - child 74
 - composite objects 75
 - composition 75
 - converting to SCL entries 102
 - creating with SCL 100
 - definition 73
 - delegation 75
 - descendants 74
 - documenting 156
 - editing 67
 - extending 99
 - families 74
 - grouping into resources 68
 - inheritance 74

- instantiating with SCL 105
- instantiation 74
- making available 154
- merging 154
- metaclasses 76
- models 76
- naming conventions 102
- parents 74
- relationships among SAS/AF classes 74
- siblings 74
- types of 76
- uses relationships 75
- views 76
- classes, from older SAS releases 72
 - See also [legacy classes](#)
- Close buttons, creating 96
- communication 27
 - See also [component communication](#)
 - See also [component-frame communication](#)
 - See also [drag and drop communication](#)
- compiling
 - frames 22
 - SAS/AF applications 8
 - SCL programs 46
- completeDrag attribute 145
- component communication 27
 - See also [attribute links](#)
 - See also [drag and drop communication](#)
 - adding to components 129
 - for data access 29
 - for data display 29
 - models, definition 29
 - models, example 31
 - models, uses for 30
 - SAS Component Object Model (SCOM) 27
 - viewers, definition 29
 - viewers, example 31
 - viewers, uses for 30
- component development
 - attributes, adding 99
 - Class Browser 70
 - class constructors, adding 107
 - class constructors, defining with Class Editor 107
 - class constructors, defining with SCL 106
 - class constructors, overriding 107
 - class constructors, overview 105
 - Class Editor 67
 - CLASS statement 103
 - classes, converting to SCL entries 102
 - classes, creating with SCL 100
 - classes, instantiating with SCL 105
 - classes, recommended practices 102
 - Close buttons, creating 96
 - creating your own components 95
 - extending classes 99
 - GenDoc Utility, deploying components 156
 - Interface Editor 68
 - methods, adding 99
 - methods, implementing 103
 - Resource Editor 68
 - SAS/AF applications 7
 - SCL analysis tools 70
 - SCL programs, creating 69
 - Source Control Manager 70
 - Source window 69
 - subclassing, by overriding attributes 96
 - subclassing, by overriding methods 97, 98
 - subclassing, methodology 95
 - testing components 97
 - testing new attributes 100
 - testing new methods 100
 - testing overridden methods 98
 - text editor 69
 - tools and utilities for 70
 - tools for 67
 - USECLASS statement 104
- component properties, setting 17
- component-frame communication 8
- components
 - adding 22
 - combining 24
 - composite 24
 - default attributes 115
 - dropping in frames 15
 - functions, table of 23
 - provided by SAS 7
 - re-using 24
 - SAS/AF software 16
 - selecting 15, 23
 - subclassing 25
 - supporting drag and drop communication 32
 - viewing 15
- components, deploying 151
 - See also [deploying components](#)
- components, positioning 22
 - See also [attachments](#)
- components, sizing 25
 - See also [attachments](#)
- Components window 15
- composite, definition 24
- composite objects 75
- composition 75
- CONFIG system option 61
- configuration files
 - definition 60

- naming 61
 - specifying 61
 - constructors 105
 - defining with Class Editor 107
 - defining with SCL 106
 - controls, adding to windows 95
 - See also subclassing*
 - COPY procedure, deploying SAS/AF applications 58
 - Coverage Analyzer tool 48
 - CPORT procedure, deploying SAS/AF applications 59
 - current attachment mode 180
 - custom access methods (CAMs) 121
 - assigning to attributes 121
 - limiting execution of 124
 - naming conventions 123
 - recursion, avoiding 124
 - custom attributes windows 119
 - custom commands, in SCL programs 42
 - custom configuration files 61
 - custom menus, creating in SAS/AF 51
 - customizers 119
- D**
- dataOperation attribute 145
 - dataRepresentation attribute 145
 - debugger commands 47
 - debugging applications 168
 - debugging SCL programs 47
 - defaultAttribute, specifying 115
 - define attachment mode, initiating 181
 - defining sections 186
 - delegation 75
 - deploying components 151
 - class documentation, generating 156
 - class documentation, making available 159
 - classes, making available 154
 - metadata descriptions 157
 - resources, analyzing 153
 - resources, associating with FRAME entries 155
 - resources, merging 154
 - resources, overview 152
 - resources, renaming class catalogs 154
 - resources, renaming libraries 154
 - resources, synchronizing 153
 - SAS Registry Components key, modifying 156
 - deploying SAS/AF applications 57
 - See also SAS sessions, configuring*
 - BUILD procedure 58
 - CIMPORT procedure 59
 - COPY procedure 58
 - CPORT procedure 59
 - deployment issues 57
 - MERGE statement 58
 - migrating from testing to production 58
 - to different operating environments 59
 - descendents 74
 - Description
 - attribute metadata item 88
 - event handler metadata item 93
 - event metadata item 91
 - dialog boxes, calling 45
 - dialog frames 20
 - dot notation 40, 88
 - drag and drop communication 32
 - See also component communication*
 - adding to components 139
 - components supporting 32
 - data representation 140
 - defining properties for 142
 - definition 32
 - drag sites, defining 32, 33
 - drag sites, definition 32
 - dragInfo 32
 - drop sites, defining 32, 33
 - drop sites, definition 32
 - dropInfo 32
 - example 33, 145
 - flow of control 140, 144
 - uses for 32
 - drag sites
 - defining 32, 33
 - definition 32
 - dragEnabled attribute 142
 - dragInfo 32
 - dragInfo attribute 142
 - dragOperations attribute 142
 - drop sites
 - defining 32, 33
 - definition 32
 - dropEnabled attribute 143
 - dropInfo 32
 - dropInfo attribute 143
 - dropOperations attribute 143
- E**
- Editable attribute metadata item 86
 - Editor attribute metadata item 87
 - END command 166
 - ENDSAS command 167
 - ENTRY statement 118
 - environment argument 118
 - environment variables, creating 62
 - event handler metadata items
 - definition 92
 - Description 93

- Enabled 93
 - Event 92
 - Method 92
 - Sender 92
 - State 92
 - event handlers 72, 92
 - event handling
 - adding to components 148, 149
 - flow of control 149
 - modifying 148
 - event metadata items
 - definition 91
 - Description 91
 - Enabled 91
 - Execute 91
 - Method 91
 - Name 91
 - State 91
 - events 90
 - adding to components 149
 - definition 90
 - Execute event metadata item 91
- F**
- families of classes 74
 - flow of control
 - _getAttributeValue method 88
 - _setAttributeValue method 88
 - AUTOTERM feature 167
 - AUTOTERM= option 167
 - BYE command 167
 - CANCEL command 167
 - debugging applications 168
 - drag and drop communication 140, 144
 - END command 166
 - ENDSAS command 167
 - event handling 149
 - example of order processing 164
 - frame components, changing 177
 - FRAME entries and automatic methods, build time 169
 - FRAME entries and automatic methods, run time 172
 - frame SCL components, changing 177
 - INIT section 163
 - labeled sections 164
 - MAIN section 165
 - MAIN section, forcing execution of 165
 - MAIN section, handling invalid values 166
 - MAIN section, overview 165
 - model/view communication 133
 - multiple window components 164
 - order of execution, build-time 170
 - order of execution, run-time 173
 - SCL objects, terminating on application end 167
 - system closure command 167
 - TERM section 166
 - VERBOSE value 168
 - frame components, changing 177
 - FRAME entries
 - and automatic methods, at build time 169
 - and automatic methods, at run time 172
 - compiling automatically 47
 - compiling in batch 46
 - how SCL programs execute for 163
 - overriding frame methods 113
 - frame SCL 36
 - changing components 177
 - entries, flow of control 169
 - frame-component communication 8
 - frameID argument 118
 - frames 19
 - See also* SCL (SAS Component Language)
 - adding menus 22
 - and SCL programs 36
 - attachments 25
 - compiling 22
 - components, adding 22
 - components, automatic resizing 25
 - components, combining 24
 - components, composite 24
 - components, positioning 22
 - components, re-using 24
 - components, selecting 23
 - components, subclassing 25
 - dialog 20
 - documenting 156
 - dropping components in 15
 - help, context sensitive 54
 - help, online 54
 - methods, writing 25
 - SCL entries, specifying 22
 - SCL programs for 37
 - SCL requirements 37
 - setting properties 22
 - source code 22
 - standard 20
 - storing 22
 - tips for using 22
 - types of 20
 - frames, opening from
 - command prompt 20
 - Explorer window 20
 - Program Editor 20
 - SAS/AF software development environment 20

G

GenDoc Utility
 definition 70
 deploying components 156
 output of 158
 GetCAM attribute metadata item 87

H

help, calling from applications
 HELP command 55
 HELPLLOC:// protocol 55
 HELPMODE command 55
 overview 55
 WBROWSE command 55
 help, context-sensitive
 attaching to frames 54
 tooltip help 54
 help, online
 adding to applications 53
 adding to frames 54
 help attribute
 context sensitive help in frames 54
 online Help in frames 54
 HELP command 55
 help files, specifying location of 62
 HELPLLOC:// protocol 55
 HELPLLOC system option 62
 HELPMODE command 55
 helpText attribute 54
 HonorCase attribute metadata item 87

I

inheritance 74
 INIT section 163
 INITCMD system option 62
 InitialValue attribute metadata item 85
 instance variables 72
 instantiation of SAS/AF classes 74
 Interface Editor 68
 interface properties of classes
 definition 93
 InterfaceName item 93
 State item 93
 Status item 93
 InterfaceName item 93
 interfaces, definition 93
 IV (instance variable) attribute metadata
 item 88

L

labeled sections 164
 legacy classes 72
 Linkable attribute metadata item 86

list attributes 127
 List Diagnostic Utility 48

M

MAIN section
 forcing execution of 165
 handling invalid values 166
 overview 165
 menus
 adding to frames 22
 creating 51
 MERGE statement
 deploying SAS/AF applications 58
 metaclasses 76
 metadata descriptions 157
 Method
 event handler metadata item 92
 event metadata item 91
 method metadata 81
 method names, case sensitivity 110
 method signatures 78
 methods 77
 _getAttributeValue, definition 88
 _getAttributeValue, flow of control 88
 _setAttributeValue, definition 88
 _setAttributeValue, flow of control 88
 adding 99
 automatic execution 77
 definition 77
 having same name 80
 naming conventions 77, 110
 overloading 80
 overriding 110
 per-instance 78
 recursion 112
 scope 81
 sigstrings 79
 storing 109
 subclassing 78
 virtual 78
 writing 25
 methods, implementing
 with CLASS statement 103
 with SCL 109
 with USECLASS statement 103, 111
 model components
 definition 29
 example 31
 uses for 30
 model/view communication 29
 creating a new interface 136
 determining when to use 30
 examples 31
 flow of control 133
 implementing 133

- models, basing on StaticStringList interface 135
- models, creating 134
- models, defining 135
- problem domains 133
- setting by dragging and dropping components 31
- viewers, creating 134
- models 76
- multiple window components 164

N

- Name attribute metadata item 85
- Name event metadata item 91
- naming conventions
 - configuration files 61
 - custom access methods (CAMs) 123
 - methods 77, 110
- NOAWSMENUMERGE system option 62
- NOSPLASH system option 62

O

- object-oriented programming (OOP) 71
 - See also* attributes
 - See also* classes
 - See also* events
 - See also* methods
 - and SAS Component Object Model (SCOM) 73
 - application design 72
 - attributes 72
 - definition 71
 - instance variables 72
 - legacy classes 72
 - nonvisual objects 72
 - objects 72
- OOP (object-oriented programming) 71
 - See also* object-oriented programming (OOP)
- optimizing SCL programs 48
- overloading methods 80
- overriding
 - attributes, to create subclasses 96
 - class constructors 107
 - drag and drop methods 143
 - frame methods 113
- overriding methods
 - and scope 114
 - SAS Component Language (SCL) 110
 - testing the overrides 98
 - to create subclasses 97, 98

P

- parent classes 74
- per-instance methods 78
- Performance Analyzer 48
- PMENU 51
- PROC BUILD, deploying SAS/AF applications 58
- PROC CIMPORT, deploying SAS/AF applications 59
- PROC COPY, deploying SAS/AF applications 58
- PROC CPORT, deploying SAS/AF applications 59
- PROC PMENU, creating menus 51
- procedures
 - BUILD 58
 - CIMPORT 59
 - COPY 58
 - CPORT 59
 - PMENU 51
- properties
 - component, setting 17
 - drag and drop communication 142
 - frame, setting 22
- properties of classes
 - definition 93
 - InterfaceName item 93
 - State item 93
 - Status item 93
- Properties window 17

R

- recursion 112
 - avoiding with CAMs 124
- refresh Attributes event
 - example 126
 - using 125
- relative attachments 182
- Resource Editor 68, 153
- RESOURCE entries 152
 - See also* resources
- resources 152
 - analyzing 153
 - associating with FRAME entries 155
 - merging 154
 - overview 152
 - renaming class catalogs 154
 - renaming libraries 154
 - synchronizing 153

S

- SAS catalogs, creating
 - SAS/AF applications 6
- SAS Component Language (SCL)

- See [SCL \(SAS Component Language\)](#)
- SAS Component Object Model (SCOM)
 - See [SCOM \(SAS Component Object Model\)](#)
- SAS language statements, supported by
 - SCL 38
- SAS libraries, creating
 - SAS/AF applications 6
- SAS libraries, renaming 154
- SAS Registry
 - key values 198
 - keys 198
 - overview 197
 - SAS/AF settings 198
- SAS Registry Components key,
 - modifying 156
- SAS Registry Editor 197
- SAS sessions, configuring 61
 - application window features, enabling/disabling 62
 - application workspace (AWS) 62
 - configuration files, definition 60
 - configuration files, naming 61
 - configuration files, specifying 61
 - environment variables, creating 62
 - help files, specifying location of 62
 - invoking applications 62
 - overview 60
 - SAS software splash screen, displaying at startup 62
 - SAS system options for 61
 - splash screens, specifying location of 62
- SAS software splash screen, displaying at startup 62
- SAS system options
 - AWSCONTROL 62
 - AWSMENUMERGE 62
 - AWSTITLE 62
 - CONFIG 61
 - HELPLLOC 62
 - INITCMD 62
 - NOAWSMENUMERGE 62
 - NOSPLASH 62
 - SASCONTROL 62
 - SET 62
 - SPLASH 62
 - SPLASHLOC 62
- SAS/AF applications 3
 - compiling 8
 - component development 7
 - component-frame communication 8
 - configuration file, specifying 61
 - definition 3
 - deploying 9, 57
 - designing for reuse 8
 - development environment, setting up 6
 - frame-component communication 8
 - invoking with INITCMD system option 62
 - launching 63
 - licensing SAS software products 7
 - mainframe support issues 3
 - migrating from testing to production 58
 - planning 5
 - porting to different operating environments 59
 - SAS catalogs, creating 6
 - SAS libraries, creating 6
 - SAS sessions, configuring 60
 - testing 8
 - usability testing 9
 - user interface development 8
 - uses for 5
 - using existing SAS components 7
- SAS/AF classes 73
 - abstract classes 76
 - attribute metadata 84
 - attribute values 88
 - attributes 72, 84
 - automatic methods 77
 - composition 75
 - constructors 105
 - converting to an SCL Entry 102
 - creating with SCL 100
 - creating with SCL, recommended practices 102
 - delegation 75
 - dot notation 88
 - event handler metadata 92
 - event handlers 72, 92
 - event metadata 91
 - events 72, 90
 - extending 99
 - implementing methods using the CLASS statement 103
 - implementing methods using the USECLASS statement 103
 - inheritance 74
 - instantiation 74
 - interface properties 93
 - interfaces 72, 93
 - metaclasses 76
 - method metadata 81
 - method scope 81
 - method signatures 78
 - methods 72
 - methods, definition 77
 - methods, flow of control 88
 - methods, implementing with SCL 109
 - methods, overriding 110
 - model/view components 76

- models 76
- overloading methods 80
- per-instance methods 78
- relationships among 74
- SCOM (SAS Component Object Model) 73
- subclass, creating by overriding a method 97
- subclass, creating by overriding an attribute 96
- subclassing methodology 95
- types of 76
- uses relationships 75
- viewers 76
- virtual methods 78
- SAS/AF software 13
 - attribute editors, creating 117
 - attribute linking 130
 - attribute validation 116
 - Build window 15
 - class browser 70
 - Class Editor 67, 107
 - component development 95
 - components 16
 - Components window 15
 - custom access methods (CAMs) 121
 - custom attribute editors, creating 118
 - custom attributes window 119
 - custom menus, creating 51
 - development environment 13
 - drag and drop, enabling 139
 - GenDoc Utility 70
 - Interface Editor 68
 - model/view communication 29, 133
 - models, creating 134
 - Properties window 17
 - Resource Editor 68
 - SAS Registry 198
 - SCL analysis tools 70
 - Source Control Manager (SCM) 70
 - Source window 17, 69
 - viewers, creating 134
- SASCONTROL system option 62
- SCL (SAS Component Language)
 - _NEW_ operator 105
 - analysis tools 70
 - automatic methods, order of execution
 - at build time 170
 - automatic methods, order of execution
 - at run time 172
 - calling dialog boxes 45
 - calling other entries 44
 - calling SAS catalog entries 44
 - class constructors, defining 106
 - CLASS statement 103
 - compiling 46
 - compiling FRAME entries
 - automatically 47
 - compiling FRAME entries in batch 46
 - constructing a program 37
 - constructors 105
 - controlling execution of 41
 - converting a class to an SCL Entry 102
 - creating 17, 69
 - creating a class 100
 - creating classes, recommended practices 102
 - custom commands 42
 - data types supported 39
 - debugger commands 47
 - debugging 47
 - definition 3
 - dot notation 40, 88
 - ENDUSECLASS statement 111
 - entries, specifying 22
 - flow of control 163
 - FRAME entries and automatic methods 169
 - functions 39
 - IMPORT statement 105
 - INIT section processing 163
 - instantiating classes 105
 - labeled sections 38, 113
 - list functions 127
 - MAIN section processing 165
 - method blocks 111
 - method scope 81
 - method signatures 78
 - METHOD statements 78
 - methods, implementing 109
 - objects, terminating on application end 167
 - opening windows 44
 - optimizing 48
 - order of execution, build time 170
 - order of execution, run time 173
 - overloading methods 80
 - overriding frame methods 113
 - overriding methods 110
 - performance 112
 - programs 35
 - requirements for programs 37
 - routines 39
 - SAS language statements supported 38
 - saving programs 49
 - Source window 17, 69
 - statements 38
 - storing programs 49
 - TERM section processing 166
 - testing 46, 47
 - USECLASS statement 104, 111
 - variables 39

- working with frames 36
 - SCM (Source Control Manager)
 - See [Source Control Manager \(SCM\)](#)
 - SCOM (SAS Component Object Model)
 - 27, 73
 - attribute linking 28, 129
 - attribute linking, enabling 130
 - drag and drop, enabling 139
 - drag and drop communication 32
 - drag and drop operations 129
 - event handling 129
 - model/view communication 29, 129
 - models 76
 - OO programming model 73
 - processes 129
 - viewers 76
 - Scope attribute metadata item 85
 - scope of methods
 - definition 81
 - in overridden methods 114
 - Sender event handler metadata item 92
 - SendEvent attribute metadata item 86
 - SET system option 62
 - SetCAM attribute metadata item 87
 - showContextHelp attribute 54
 - sibling classes 74
 - sigstrings 79
 - single directional attachments 183
 - Source Control Manager (SCM) 70
 - Source window 17, 69
 - splash screens, specifying location of 62
 - SPLASH system option 62
 - SPLASHLOC system option 62
 - standard frames 20
 - State
 - attribute metadata item 85
 - event handler metadata item 92
 - event metadata item 91
 - Static Analyzer 48
 - Status item 93
 - subclassing 95
 - by overriding attributes 96
 - by overriding methods 97, 98
 - components 25
 - methodology 95
 - methods 78
 - system closure command 167
- T**
- TERM section 166
- TESTAF command 47
 - testing
 - components 97
 - new attributes 100
 - new methods 100
 - overridden methods 98
 - SAS/AF applications 8
 - SCL programs 46, 47
 - text editor 69
 - TextCompletion attribute metadata item 87
 - tooltips 54
 - toolTipText attribute 54
 - Type attribute metadata item 85
- U**
- underscore (`_`), in method names 77
 - usability testing, SAS/AF applications 9
 - USECLASS statement 104
 - user interface development
 - Interface Editor 68
 - SAS/AF applications 8, 15
 - uses relationships 75
- V**
- validating character values 116
 - ValidValues 86
 - value argument 118
 - VERBOSE value 168
 - viewer components 76
 - definition 29
 - example 31
 - uses for 30
 - virtual methods 78
- W**
- WBROWSE command 55
 - windows, opening 44
- X**
- XLocation attribute 145
- Y**
- YLocation attribute 145