Chapter 7
Inheritance and Subclassing

Objective

In this lesson, you will learn how and when to use inheritance to customize the behavior of objects in your application. You will see how CA-Visual Objects uses inheritance to deliver a powerful development environment and, consequently, you will gain an appreciation for the benefits of subclassing and code reuse.

Overview

Inheritance is one of the fundamental principles of object-oriented programming. It allows you to specify new classes in terms of existing ones. The new class inherits all of the attributes (*instance variables*) and behavior (*methods*) of the existing class and allows you to add any distinguishing features that are needed.

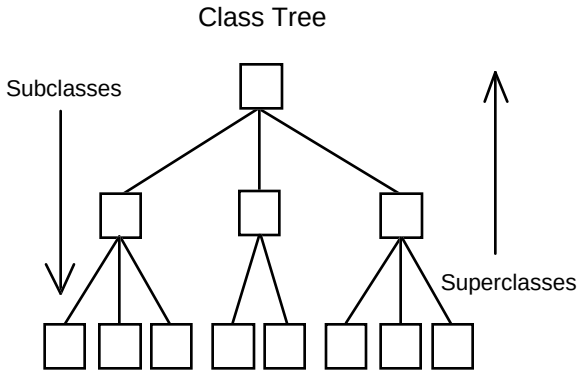
Subclass, Superclass, and Parent

When a class inherits from another class, it is said to be a *subclass* of that class. A *superclass* is any of the classes from which a class derives its behavior, and the *parent* is the immediate superclass.

Inheritance allows for incremental development by creating new classes on the stable foundation of existing classes. In this way, a great deal of code can be reused (by inheriting behavior and characteristics), and a significant level of robustness can be easily maintained.

Class Tree

CA-Visual Objects allows for each class to inherit from a single parent class. This is called single inheritance. On the other hand, each class can have as many subclasses as required. This gives rise to a class tree, a hierarchical representation of the relationships between classes:



It is easy to see that classes defined at higher levels in the class tree are more likely to be general, and have a higher level of abstraction, while classes defined at lower levels are more specialized.

In CA-Visual Objects, a subclass is created using the INHERIT keyword in the CLASS statement. For example, to create a subclass of DataWindow called EditItemWindow, you would use the following class declaration statement:

```
CLASS EditItemWindow INHERIT DataWindow
```

EditItemWindow has all of the properties and behaviors of the DataWindow class, plus a few of its own which you will code. One way to look at EditItemWindow is that it is a *kind of* data window.

When and How to Create a Subclass

Whenever you require special behavior, first look to see if you already have a class that provides what you need. If one exists, use it. However, if there is no class that does exactly what you want, but there is one which provides the same behavior at a more basic level, then this can be used as a parent to create a subclass with the desired characteristics. In this way, subclassing is like specialization.

During the development of an application, you may find that a class is too specific to be of general use. In this case, you must create a class at a greater level of abstraction, which is more general. By doing this, you actually remove specialized behavior and attributes from the class, making it more generic. It should now be possible to create various subclasses—each with its own area of specialization—to fill the required roles. As you can see, this provides you with a very powerful and flexible development environment.

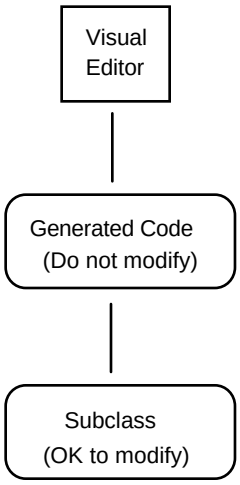
Subclassing with Generated Code

The visual editors in CA-Visual Objects, such as the Window Editor and the Menu Editor, generate source code for your application. If you examine the code created by these editors, you will notice that inheritance is used to define a new class for your purposes.

For example, the Window Editor generates a new window class definition in terms of an existing window class. This empowers the visual editors with a greater degree of flexibility, and ensures that your application will be consistent and robust.

Each time you edit a binary entity, by saving the current design in one of the visual editors, CA-Visual Objects regenerates any required class definitions and supporting entities. For this reason, you should *never* modify the generated code directly, since it will be overwritten by the regenerated code from the editor.

If you wish to customize the behavior of the generated class or the supporting entities, create a subclass based on the class created by CA-Visual Objects, as shown in the diagram below. Any specialized behavior can be added to this new class. As you review and analyze source code in the chapters to follow, you will see this technique employed over and over again in the South Seas Adventures application.



For a more complete discussion of object-oriented programming in CA-Visual Objects, please refer to the "Objects, Classes, and Methods" chapter in the *Programmer's Guide, Volume III.*

Exercise

Customizing Generated Code

1. Open the South Seas Adventures by double-clicking on its button in the Application Browser:

   The Module Browser appears.

2. Open the File:Forms module by clicking on its button in the Module Browser.

3. Examine the class definition created by the Window Editor by double-clicking on the _FileOpenDialog class entity:

```
CLASS _FileOpenDialog INHERIT DialogWindow

   PROTECT oDCtheGroupBox1 AS GroupBox
   PROTECT oCCAdventureRadioButton AS ;
          RadioButton
   PROTECT oCCCustomerRadioButton AS RadioButton
   PROTECT oCCPaymentRadioButton AS RadioButton
   PROTECT oCCEmployeeRadioButton AS RadioButton
   PROTECT oCCInvoiceRadioButton AS RadioButton
   PROTECT oCCItemRadioButton AS RadioButton
   PROTECT oCCOkButton AS PushButton
   PROTECT oCCCancelButton AS PushButton
```

   Notice that _FileOpenDialog inherits from the DialogWindow class. This predetermines a great deal of the window's behavior and characteristics.

4. Close the Source Code Editor by double-clicking on its system menu.

The Open File dialog box, shown below, allows the user to select which kind of file they wish to open. To make it work properly, you will have to add a FileType instance variable that can be inspected once the dialog box has been closed.

Recall that it is never good to modify generated code, because when CA-Visual Objects needs to regenerate code based on modifications to the underlying entity (such as the Window entity), your source code modifications will be lost.

The best way to handle a situation like this is to create a subclass of _FileOpenDialog in order to isolate and protect your changes. The standard used throughout the South Seas Adventures application is to remove the leading underscore (as in FileOpenDialog) to create the new subclass.

5. Double-click on the FileOpenDialog class entity to see how the subclass was created with the additional FileType instance variable:

```
CLASS FileOpenDialog INHERIT _FileOpenDialog
   EXPORT FileType AS USUAL
```

6. Close the Source Code Editor by double-clicking on its system menu.

7. Close the File:Forms Entity Browser by double-clicking on its system menu.

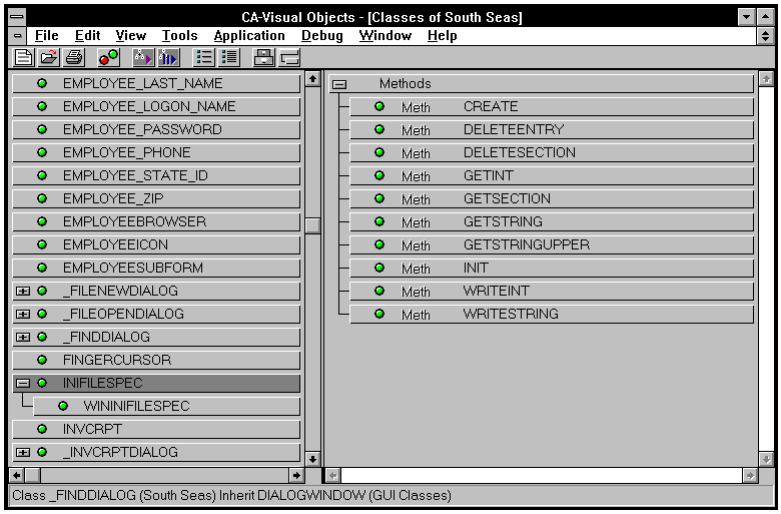
Using Inheritance in Your Programs

You will want to take advantage of the power of inheritance in your own programs, too, not just in modifying code generated by CA-Visual Objects.

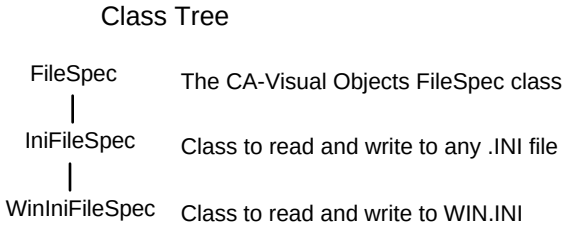
1. Launch the Class Browser by choosing the Tools Class Browser command.
2. Highlight the IniFileSpec class in the left panel of the Class Browser and click the Expand button to its left to view its subclasses.

   The WinIniFileSpec class appears below IniFileSpec in a tree-like structure:

Class Browser



The purpose of these classes is to read and write .INI files. A general class, FileSpec, is used as a starting point. The IniFileSpec class can read and write to any .INI file, while the WinIniFileSpec class can only read and write to the WIN.INI file. The following class tree results:



The source code for the classes is shown below. Notice that there are no additional instance variables defined for these classes—only new class methods.

```
CLASS IniFileSpec INHERIT FileSpec

CLASS WinIniFileSpec INHERIT IniFileSpec
```

Note: At any time, you can double-click on an item in either pane of the Class Browser to view it in the Source Code Editor.

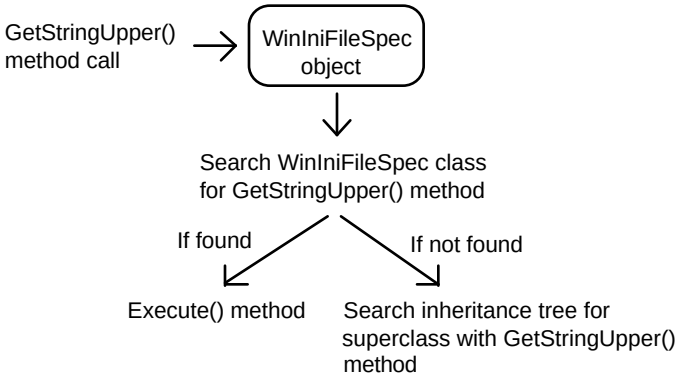
As is evident using the Class Browser, IniFileSpec contains ten new methods, while WinIniFileSpec contains only six new methods. This is because WinIniFileSpec is able to reuse some of the code already written.

For example, each class has its own GetString() method. This is because getting a string from the WIN.INI file requires a different function call than getting a string from another .INI file, so the GetString() method requires a different implementation in each class. From the perspective of using the method, however, the syntax is exactly the same for both classes.

However, only the IniFileSpec has a GetStringUpper() method. Does this mean that only IniFileSpec objects can perform the GetStringUpper() method? Certainly not!

Since WinIniFileSpec inherits from the IniFileSpec class, it also inherits all methods defined by IniFileSpec. Thus, WinIniFileSpec objects can respond to GetStringUpper() method calls. How does this work?

When a call is made to WinIniFileSpec:GetStringUpper(), a method of that name is executed if such a method exists in the current class. If no such method is found, a search is performed upwards through the inheritance tree for the closest superclass which contains a method with this name. In this case, it would be IniFileSpec, as shown in the following diagram:



However, if none of the superclasses contain this method, CA-Visual Objects generates a runtime error.

The reason the two classes can share the same GetStringUpper() method is because of its implementation. This method simply calls GetString(), which, although specialized for each class, maintains a consistent user interface. Double-click on the GetStringUpper() method to view its source code:

```
METHOD GetStringUpper(sSection, sEntry) ;
       CLASS IniFileSpec

       RETURN UPPER(SELF:GetString(sSection,
sEntry))
```

How does GetStringUpper() know which method to invoke?

In CA-Visual Objects, the keyword *SELF* always refers to the current object. In a method, SELF is not necessarily of the same class as the class to which the method belongs. This is because all subclasses of this class may also call this method. This code is shared. In this case, SELF may refer to either a WinIniFileSpec object or an IniFileSpec object, and the appropriate method is called in either case.

Summary

As you have seen, inheritance is a very important and powerful feature of object-oriented programming. CA-Visual Objects helps you make the most of this powerful concept and allows you to quickly and easily develop robust applications that meet your business needs.

You can now move on to the next chapter, which demonstrates how to create menus and toolbars, which you can then add to your windows.