

## **15. Applying the ABCs -- User Interface Construction**

### ***Overview***

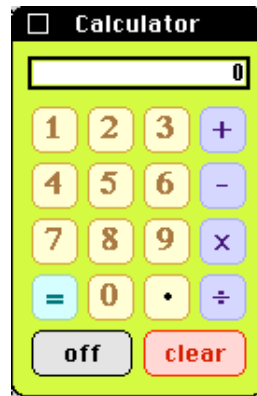
In Chapter 14, we discussed how the Application Builder Classes work, with special emphasis on how the Application Builder Editors simplify both the design of a user interface and the setting up of programmatic interconnections between the objects that implement a user interface. In this chapter, we'll demonstrate just how simple it is to build a user interface with Prograph, and how little code you'll have to write yourself, by presenting a complete calculator utility program. As in the address book example shown in Chapter 1, the majority of the program construction will involve user interface design rather than writing code.

We're going to take a slightly different approach in this chapter and the next chapter. Instead of showing you only one example program with a user interface, we're going to show you two different versions of the same program. In the first version, the application-specific functions of the user interface will be handled by subclasses of ABC controls. In the second, they will be handled by a separate class that is easier to reuse and extend in future programs. Neither program is truly "better" or "worse" than the other. They are presented to show you that there is sometimes more than one programming philosophy that can be applied to the same programming task, each with its own advantages and disadvantages.

### ***The Calculator Program -- Version 1***

Our exercise in constructing a user interface will be to write a calculator utility program. The program will present the user with a window that resembles a typical hand-held calculator (see Figure 15.1). The calculator keys for the digits, decimal point and operators will be mimicked by push buttons in the window, while a text box (not a text-editing box) will provide the display for the current number, whether it's an operand or the result of a calculation.

While the program behaves in many ways like a real calculator, it does not mimic every behavior of a calculator. For example, on a real calculator, the user can press the "+" key, then the "=" key repeatedly to add a number over and over again. This does not happen with our calculator program. In addition, no bounds checking is performed on the numbers to see if an entered number or the result of a calculation is too large or too small to display. Remember that the goal of this section of the book is not to teach you how to make a great calculator, but to show you how to take advantage of Prograph's user interface-handling classes and design tools.

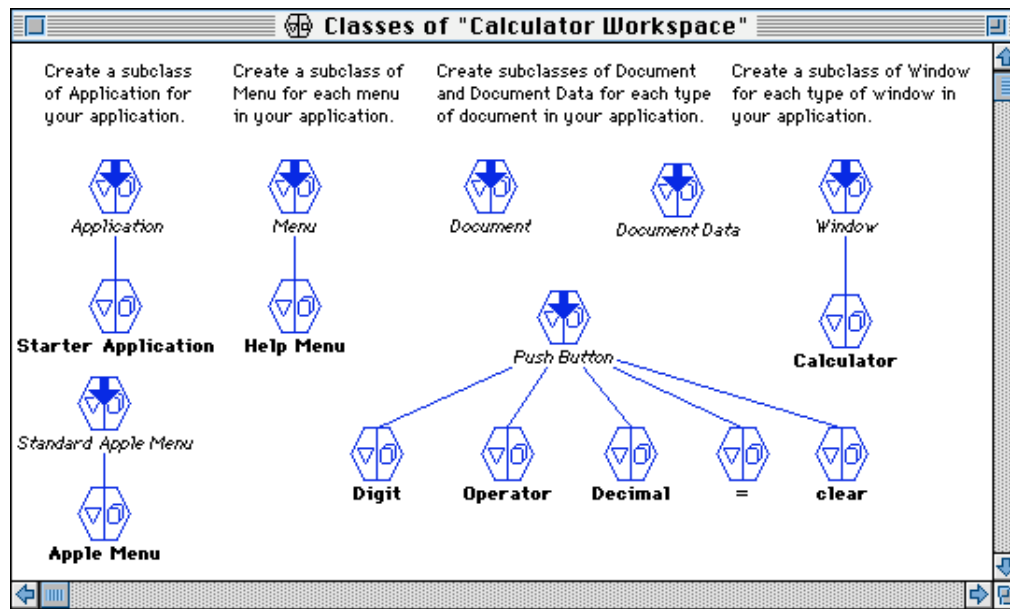


**Figure 15.1: The calculator display window of the Calculator program**

In this first version of the program, we will use the programming style used very often by those who use the ABCs -- we will *directly* subclass user interface classes to hold the application-specific code of our program. It's important to state right from the start that this is *not* a truly proper programming style from an OOP standpoint, but it is a common programming style for Prograph programmers. In addition, this style is good for very rapid program development and prototyping. Therefore, even though it is flawed, we feel that we should demonstrate and discuss this programming style here.

We will subclass the **Window** and **Push Button** classes and add all of the code for our calculator to these subclasses (see the Calculator Workspace section in Figure 15.2). The Workspace section typically is the only section of the ABCs themselves that is modified by the programmer. It's only when you define totally new specialized types of views or controls that you'll subclass the ABCs themselves and then modify them. In the present program, we will subclass the ABCs to create keys for our calculator.

The **Calculator** class is a subclass of **Window** that encompasses all of the GUI elements that make up the calculator. The **Digit**, **Operator**, **Decimal**, **=** and **clear** subclasses of **Push Button** encapsulate the workings of the calculator keys. All of the code we'll write for this program (and it won't be much) will be in these classes, which we'll discuss individually throughout this chapter. Our code will take advantage of the rest of the ABCs, stored in other sections, which will help us create a very versatile program while keeping our coding to a minimum.

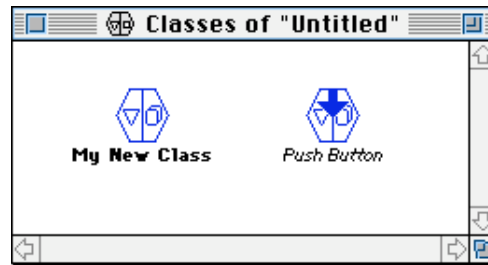


**Figure 15.2: Classes of the Calculator Workspace section**

Open the ABC Starter Application project. When you are prompted for a new name to save as our calculator program project, enter the name “Calculator Project”. While searching for sections of the Application Builder Classes to add to our project, you’ll be prompted for a name for the workspace section of the new program. Enter the name “Calculator Workspace”. When the Sections window opens, it will display all of the sections of the ABCs, now part of our program.

Before we continue with the rest of the program design, you should open the Calculator Workspace section and create the subclasses of Push Button (Digit, Operator, Decimal, = and clear) shown in Figure 15.2. But if you look at the default Workspace section that is supplied with the ABC Starter project, there is no Push Button class in the Workspace to subclass. How do we place the subclasses of Push Button in this section?

If all of the subclasses derived from a single parent class of the ABCs had to be present in a single section, the ABCs would probably all be crammed into just a few sections. It would be harder to make our own reusable subclasses since any new subclasses would also have to go into these sections. The sections would rapidly get huge and unmanageable. To prevent this, Prograph provides an *alias to a class* (see an example in a new section in Figure 15.3) that acts much like an alias to a file in the operating system. Just as a file alias is a marker for where the original file can be found, an alias to a class tells Prograph that we want to access a class in the section where the alias is, even though that class isn’t physically there. The alias can be subclassed and Prograph will treat it as if the original class itself (in another section) was subclassed. By keeping subclasses in their own sections, we can more efficiently manage our code when we reuse subclasses over and over again in new programs.

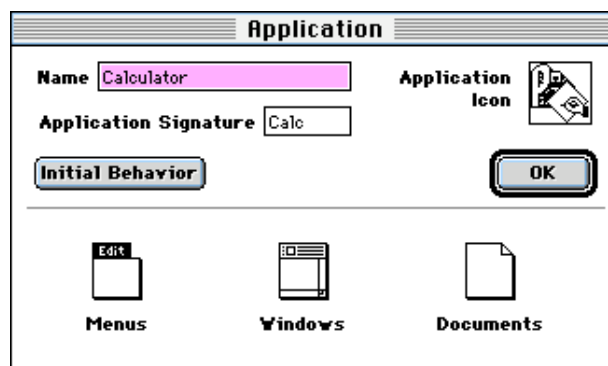


**Figure 15.3: A class and an alias to a class in another section**

Create a class in the Calculator Workspace section, then select the Convert To Alias... menu item. A list of available classes in the project will appear in a dialog box. Select the Push Button class. The icon will change to that of an alias to a class, with the name Push Button. Now subclass the Push Button class alias to create our new Digit, Operator, Decimal, = and clear subclasses.

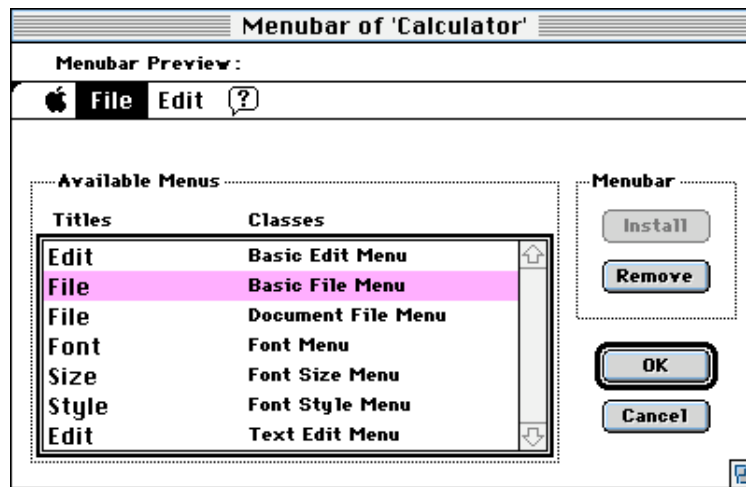
### *Designing the User Interface of the Calculator Program*

Let's get started by designing the user interface. Select the Edit Application menu item. At this point, the Application Editor will open (Figure 15.4). Enter "Calculator" for the name of the program and 'Calc' as the signature for a Macintosh application. The Macintosh uses signatures to associate document files with the application that created them, so you can double-click on a document icon to open the program. While the application signature is less important for programs without file access or documents, you should get into the habit of giving your program a unique signature anyway since they are also used for associating icons with programs.



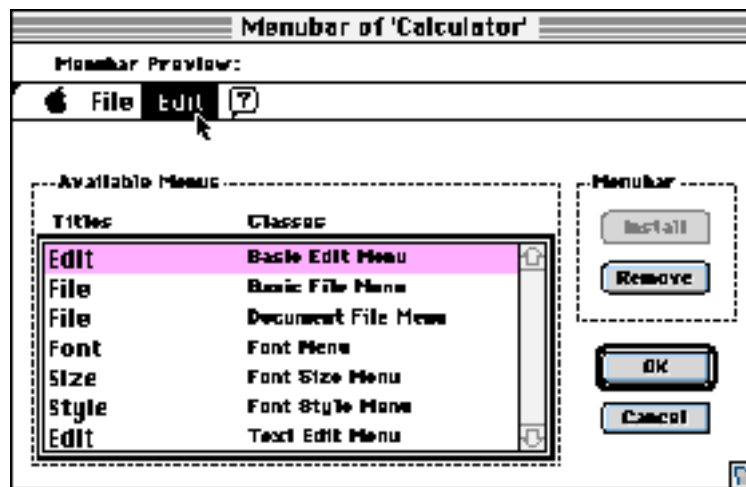
**Figure 15.4: Defining the Calculator Application**

Open the Menubar Editor by clicking the Menus icon in the Application Editor. The Starter Application installs two menus by default into a program user interface -- the Basic File menu and the Basic Edit menu. The Basic File menu provides a Quit menu item to end execution of a program, while the Basic Edit menu adds essential cut-and-paste capabilities. These two menus can be seen in the menu bar preview at the top of the Menubar Editor window (Figure 15.5).



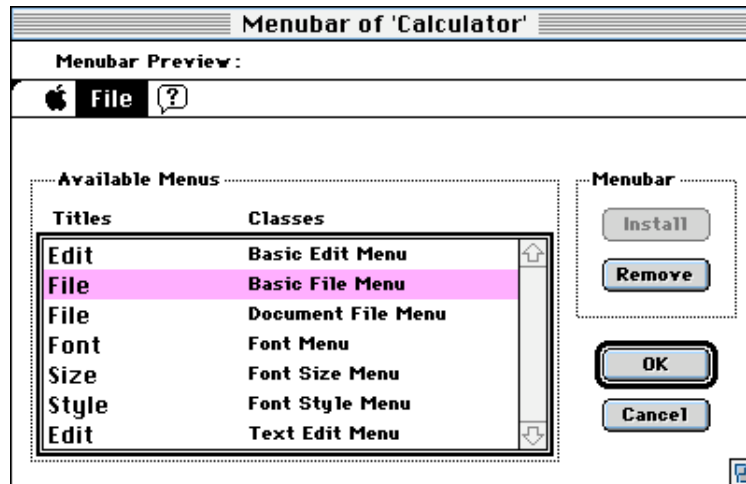
**Figure 15.5: Default menus of the Starter Application**

These two basic menus are all that are needed by simple applications or utility programs. However, for our calculator program we won't need cut-and-paste capabilities at all, since to simplify the program we will not be entering numbers into the numerical display of the calculator directly. Therefore, we'll remove the Basic Edit menu and leave our program with only a File menu. In a more complex calculator program, you might wish to leave this menu intact so you can cut and paste the results of a calculation into another program. Select the edit menu in the menu bar preview at the top of the Menubar Editor (see Figure 15.6), then click on the Remove button.



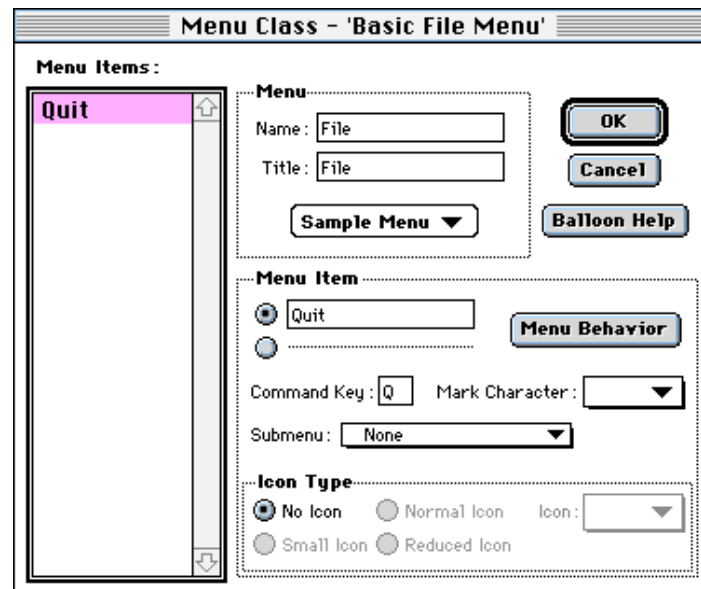
**Figure 15.6: Selecting a menu from the Menubar Preview**

The menu bar preview should now reflect the single File menu remaining in our program (Figure 15.7).



**Figure 15.7: The Menubar for the Calculator program**

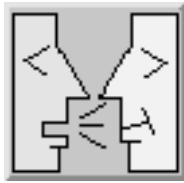
Double-click on the Basic File menu item in the list of available menus to open the Menu Editor (Figure 15.8). This File menu is pretty minimal. All it contains is a single menu item to quit the program. For our calculator program, this is all we need, so we'll use this menu definition "as is".



**Figure 15.8: The File menu of the Calculator program**

Select "Quit" in the list of available menu items to enable the menu item editing fields of the Menu Editor (they'll be grayed out until you select a menu item to work on). Now click on the Menu Behavior button for the Quit item to enter the Menu Behavior Editor, seen in Figure 15.9.

We should now specify what action the Quit item will take when selected by the user. Luckily, the Starter Application project already has defined this for us. When the Quit menu item is selected, a behavior named Quit is triggered. This calls a class method named Quit. The Quit method is found in the Application class, since the sole input for the method will be the instance of the Application class in the TheApplication persistent.



By The Way...

Remember that the first input to a class method is an instance of the class containing the method itself. This is an example of data-driven method calling for sending a request to a specific object (in this case, the Application class) to perform an action for us.

The screenshot shows a dialog box titled "Menu Behavior of 'Quit'". It has two main sections. The first section contains "Behavior Name" with the text "Quit" and a "Lookup" button, and "Method Name" with the text "/Quit" and a "Create" button. The second section is titled "Method Inputs" and contains a list box with the entry "Persistent: The Application". To the right of the list box are "Add" and "Remove" buttons. Below the list box is a "Selected Input" section with two empty text boxes and dropdown arrows. At the bottom of the dialog are "Clear", "Cancel", and "OK" buttons.

**Figure 15.9: Specifying the behavior for the Quit item of the File menu**

The menus for the calculator program are now fully defined and all of the code needed to implement them has been added already by classes in the ABCs and by default settings included in the ABC Starter project. Pretty simple, huh?

Where is all of this information that is set up by the ABEs? If you look in the Menus section, you'll find a subclass of the Menu class called Basic File Menu. In the Starter Application class, a subclass of Application that was added to the project when we opened the ABC Starter Application project file, is an attribute named Initial Menus that is a list that holds the names of the menu subclasses that will be used by our program. The two elements of that list are "Apple Menu" and "Basic File Menu".

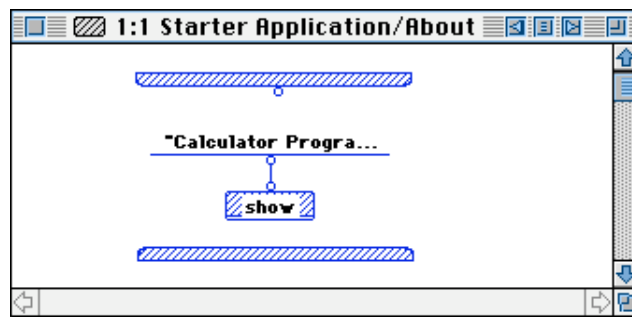
What about the *behaviors* of the items in the Basic File Menu? How are they implemented? In the Basic File Menu class is an attribute named Behaviors that holds instances of the Menu Behavior class, one for each item in the menu. In the case of our menu, the Menu Behavior object's name is "Quit" and its method is "/Quit". The

**Specifiers** attribute of the class contains an object of type **Persistent Specifier**, whose **Value** attribute holds the name of a persistent from the Calculator Workspace section that holds the object whose method will be executed. For the Quit menu item, this object is called “The Application”, which holds the instance of **Starter Application** that will coordinate our program.

This is all pretty complicated. Luckily, we really don’t need to know all of this to use the ABCs. The Application Builder Editors will set up all of these interactions and create these objects for us automatically. All we need do is define how we want our menus to look and act.

Even though we have reused the menus already defined by the ABCs, it is very easy to define menus or menu items of your own. To create a new menu, select the Create Menu menu item when you are in the Menubar Editor, then add the new menu to the menu bar. To add a new menu item, enter the Menu Editor and select the New Menu Item menu item. Highlight the new menu item in the Menu Editor’s list of menu items, and click on the Menu Behavior push button to use the Menu Behavior Editor to define a behavior for the menu item. That’s all there is to it!

Before we leave the discussion of menus entirely, let’s discuss one special menu item -- the About Application item. This item is automatically placed in the Apple Menu of a Macintosh program by the ABCs. Its purpose is to display a copyright notice, program version number, credits, or whatever other information you want to present to the user. When the About Application menu item is selected, the **About** method of the **Application** class (or its subclass if it exists) is called. The simplest way to provide this display is to include a **show** primitive in the **About** method, as shown in Figure 15.10, where we’ve overridden the **Application** class’ **About** method to show the message “Calculator program by Phil Cox”.



**Figure 15.10: About method of the Starter Application class**

Let’s continue by designing the main display window of the calculator program, shown in its final form in Figure 15.1. Close the menu editors and return to the Application Editor. Now click on the Windows icon to enter the Windows Editor.

When the Windows Editor opens, select the New Window menu item to define our new window to depict the calculator. A dialog box will appear to ask you for the

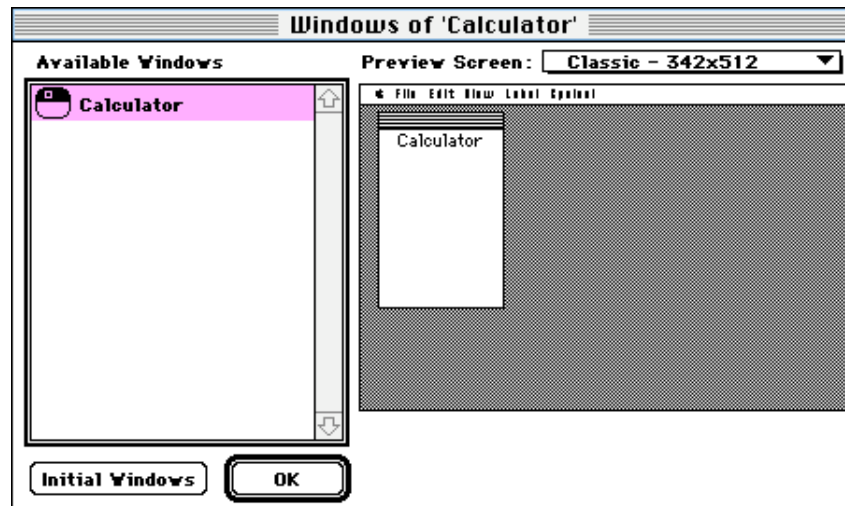


name of this window, which will also be the name of the class that implements the window (Figure 15.11).



**Figure 15.11: Entering the name for the Calculator Window class**

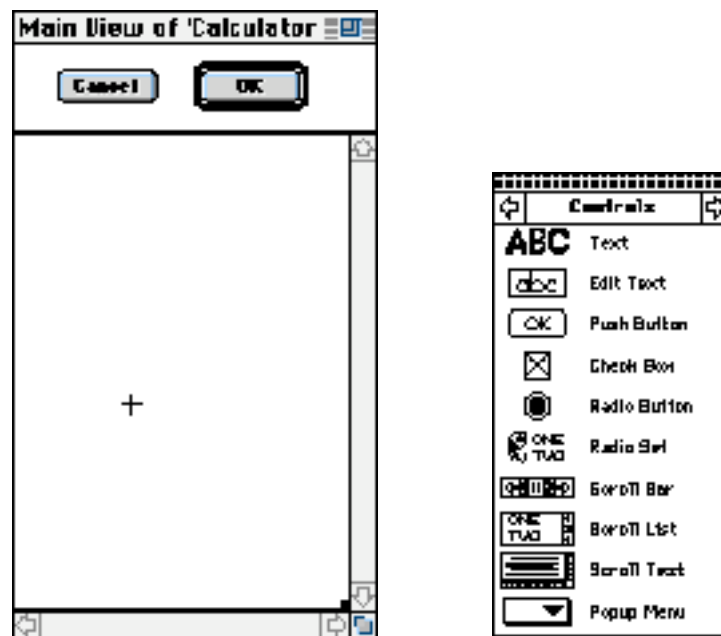
Enter the name Calculator, then select OK. When the dialog box disappears, the new Calculator window will now be added to the list of available windows for our program (see Figure 15.12), and a **Calculator** subclass of the **Window** class will be added to the Calculator Workspace.



**Figure 15.12: The new Calculator Window and its placement on the screen**

The display on the right-hand side of the Windows Editor lets you determine the placement of the window on the screen. The Preview Screen popup menu just above it lets you select between several possible screen sizes so that you can place the window properly for the monitor or monitors on which the program user interface might be presented. The Initial Windows push button lets you set a window to be opened automatically by your program when it starts up.

Move the Calculator window to the top left corner of the screen display (this is a pretty safe place for just about any monitor size). Next, select Calculator from the list of available windows and enter the View Editor for the window (Figure 15.13). The main view of the Calculator window is displayed in the editor. This view, now empty, will house all of the GUI elements we'll create for the calculator.



**Figure 15.13: Main View of the Calculator window with the Controls palette floating window**

The Calculator Window view is filled in by dragging one Text object (for the numerical display of the calculator) and 22 push buttons or Push Button subclasses (for the calculator keys) from the Controls palette onto the View Editor window. But wait! Although the palette has a Push Button item in it, there are no Digit, Operator, Decimal, = or clear items for the subclasses of a Push Button. If we drag a Push Button onto the Calculator window main view, it won't do any of the actions expected of a Digit button or a Decimal button. How do we get around this?

### ***Modifying Prograph's GUI Editors***

The programmers at Prograph International intended for Prograph to be flexible and extensible. This didn't just mean the ability to add your own pre-existing Prograph code (or code in other languages) to your programs. It also meant the ability to modify the Application Builder Editors themselves so that new subclasses of the ABCs could be used and added to new programs as easily as existing ABC classes. We will add our new Push Button subclasses to the Controls palette so we can drag them onto the Calculator window main view just as we would a regular Push Button. The difference is that these "special" Push Buttons will have extra functions that we'll code into the subclasses.

To add items to the palette, we must change the Preferences settings of the Editors themselves. The ABEs are written in Prograph themselves and you'd expect them to show up as sections of code in the Sections Window. However, when you program normally with Prograph, these sections and other sections used only by the Prograph

interpreter are hidden from your view. To get to them, we must first make them visible. Select the Preferences... menu item to open the Preferences dialog box (Figure 15.14).

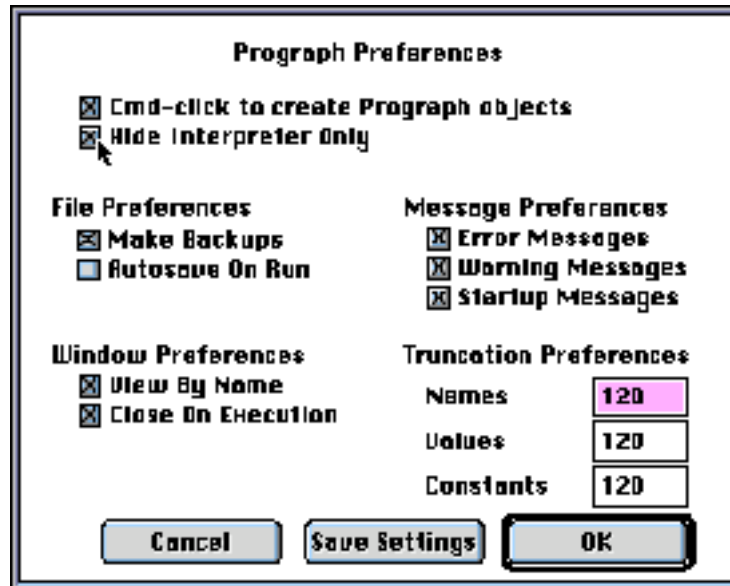


Figure 15.14: The Preferences dialog box

Uncheck the Hide Interpreter Only checkbox. The sections containing the Prograph code of the Application Builder Editors will now appear in the Sections window (see Figure 15.15).

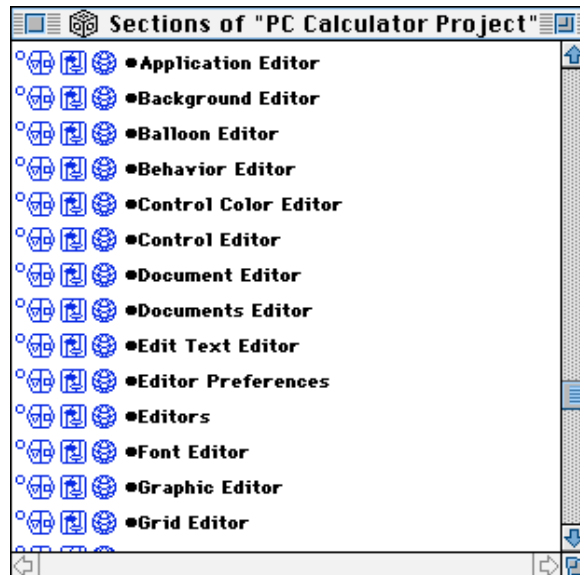
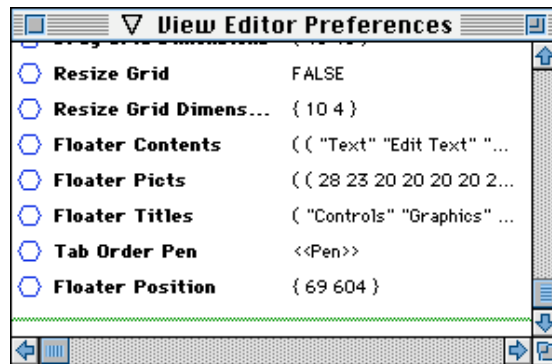


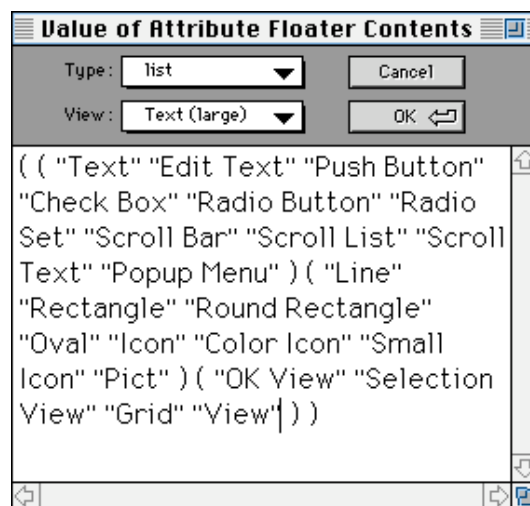
Figure 15.15: Sections window with Application Builder Editor sections visible

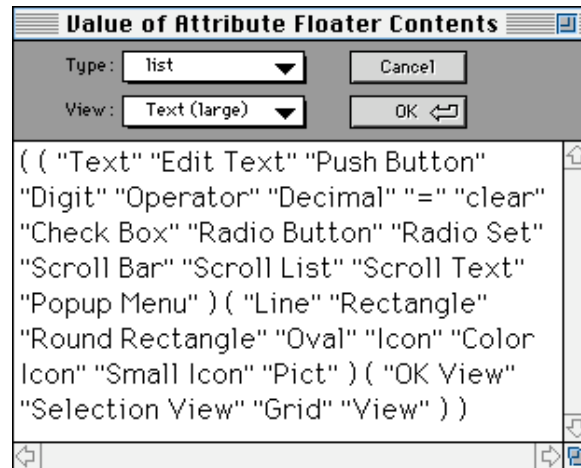
Open the section named “•Editor Preferences” to reveal the **View Editor Preferences** class. This class determines, among other things, how the View Editor’s palettes will appear and behave. It is the attributes of this class that we’re interested in now, so open its Attributes window (see Figure 15.16). Three of these attributes determine the contents of the floating palettes of the View Editor that present our choices of user interface elements to place in a window. The **Floater Titles** attribute sets the titles at the top of the palette -- “Views”, “Controls” and “Graphics”. **Floater Contents** contains the individual items in the palettes, such as the Text, Edit Text, Check Box, Scroll List and Popup menu of the Controls palette. The **Floater Picts** attribute contains resource identifier numbers for the icons displayed next to each palette item.



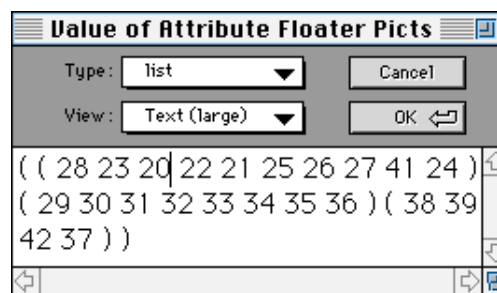
**Figure 15.16: Attributes of the View Editor Preferences class**

Edit the **Floater Contents** attribute’s first sublist (the one with “Text”, “Edit Text”, etc. in it), as shown in Figure 15.17, so that the Control palette will recognize our new control types -- *Digit*, *Operator*, *Decimal*, = and *clear*. The completed list should look like Figure 15.18, with our new **Push Button** subclasses placed directly after Push Button in its list.

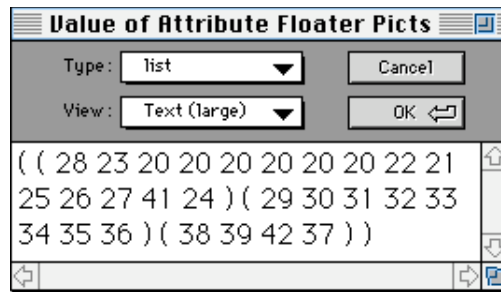


**Figure 15.17: Floater Contents attribute of the View Editor Preferences class****Figure 15.18: Edited Floater Contents attribute of the View Editor Preferences class with new Push Button subclasses added**

Now we must also tell the Control palette to display icons for these new subclasses that may be dragged onto the Calculator window's view. Open the **Floater Picts** attribute of the **View Editor Preferences** class (Figure 15.19). This list contains picture resource numbers for the icons in the Controls palette. Since we won't be adding new icons to the ABEs (this requires a special resource editor, which is beyond the scope of this book), we'll just reuse the Push Button's picture. The "Push Button" string is the third item of the Floater Content's first list (for the Controls palette), and its corresponding icon picture resource number is the third item of the first list in Floater Picts (the number 20).

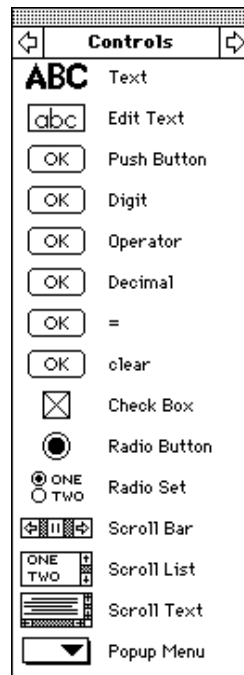
**Figure 15.19: Floater Picts attribute of the View Editor Preferences class**

Duplicate the picture resource number for the Push Button icon once for each new subclass so that the picture list looks like Figure 15.20. We can now add Push Button subclasses to the Calculator window and automatically enable them to have the actions of the subclasses.

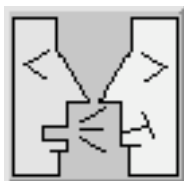


**Figure 15.20: Edited Floater Picts attribute of the View Editor Preferences class with icon numbers for Push Button subclasses added**

The Controls palette should now contain our new subclasses and resemble Figure 15.21.



**Figure 15.21: Controls palette with new Push Button subclasses added**



**By The Way...**

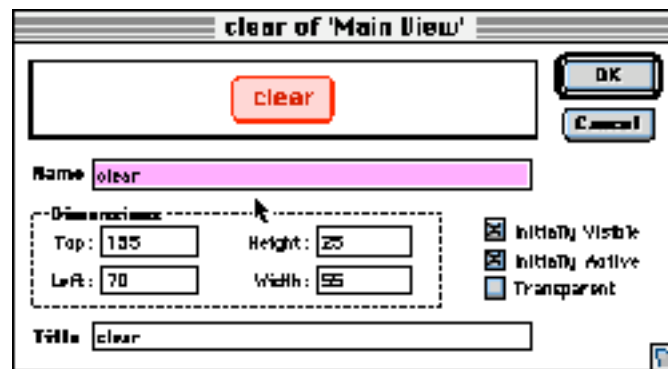
If you want to unclutter your Sections window, you can rehide all of the ABE and interpreter sections by going back to the Preferences dialog box and rechecking the Hide Interpreter Only check box.

The subclasses of **Push Button** can now be added to our main view just like any other GUI element -- by dragging it from the floating palette window to the View Editor's representation of the window being designed. These subclasses will be used as

follows: The calculator keys for the numbers 0 to 9 will be instances of the **Digit** subclass of **Push Button**, the mathematical operator keys (+, -, x, ÷) will be **Operator** subclass instances, the decimal point key will be a **Decimal**, the enter (=) key will be a **=** subclass, and the clear key will be a **clear**. A regular **Push Button** will suffice for the “off” button since all it will do is call the **Quit** class method of **Application**.

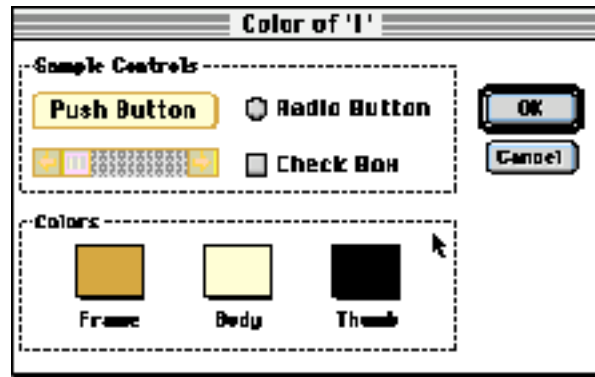
Now resize and position the push button, push button subclasses and text object of the Calculator window as shown in Figure 15.1.

Initially, the push buttons and its subclasses all have the title “Button” that appears in the graphics of the Push Button. The proper titles for each key are assigned by selecting each button in turn and entering its appropriate Push Button Editor (shown in Figure 15.22 with the settings for the “clear” button of the calculator). Each button is given a name as well for its identification. Keep in mind that the title bar of the editor dialog for each subclass of a **Push Button** will reflect the subclass; for example, the clear button is edited with a Push Button Editor whose dialog is named “clear of ‘Main View’” since it’s an instance of the **clear** subclass of a **Push Button** in the Main View of the calculator window. For each number-entry key of the calculator, use the appropriate number for each button’s title and name (0-9). Name and title the operation keys with the mathematical symbol for their operation (+, -, x, ÷, =), and name and title the button for quitting the program “off”.



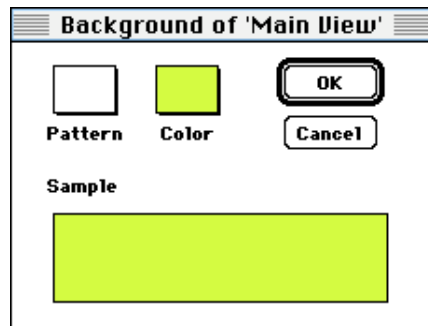
**Figure 15.22: Push Button Editor**

The appearance of the buttons is specified by selecting each button, then finding the **Color...** menu item to open the Color Editor (see Figure 15.23). We can choose colors for the inside and frame of the Push Button.



**Figure 15.23: Color Editor**

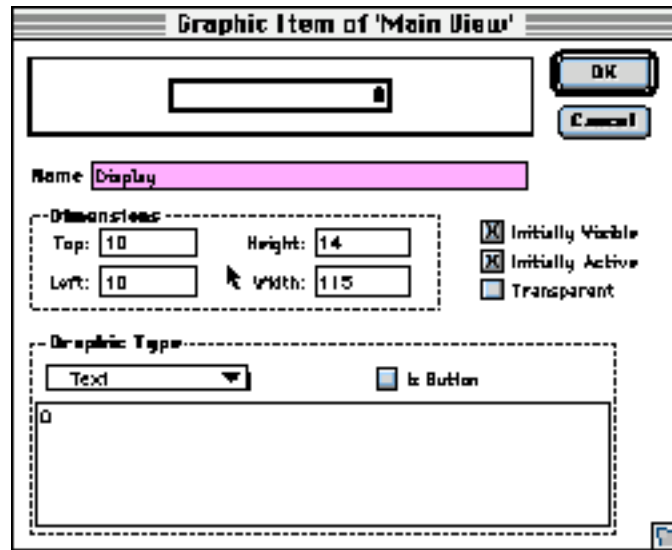
You can also assign a color to the background of the Calculator window by selecting the Background... menu item and entering the Background Editor (Figure 15.24). We have the choice of both a color and a patterned background for the window.



**Figure 15.24: Color Editor**

The numerical display at the top of the Calculator Window is made up of a Text object (see Figure 15.1). Selecting this Text object gets you into the Text Editor (see Figure 15.25). In this editor, we enter the name of the text object and a default value for its displayed text at the bottom of the editor.

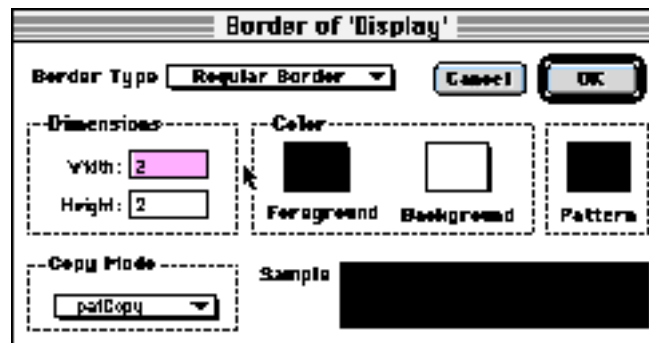




**Figure 15.25: Text Editor**

The text display is as flexible as you'd expect Prograph to make it. By selecting the text object, then viewing the Edit menu, you'll find a number of text-related options. You may select the text's font, size, style (bold, italic, outline, condensed, expanded) and justification (left, right, centered). For our calculator program, we want the numerical display to be right-justified in the numerical display box.

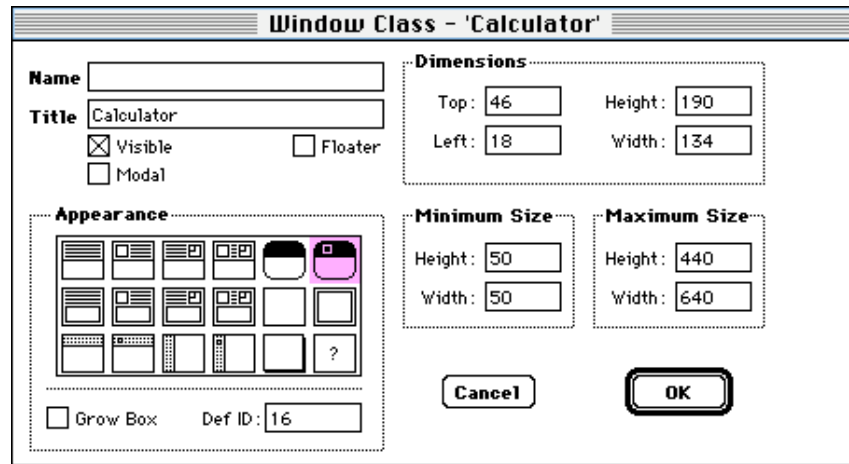
To make the numerical display really stand out, we'll create a dark border around it. With the text object still highlighted, select the Border... menu item to bring up the Border Editor, shown in Figure 15.26. With this editor, you have the choice of giving a GUI element a border or not, the thickness of the border (height, width), the color and pattern of the border and the type of border (plain box, drop shadow, etc.). We'll use a regular box border two pixels wide for our numerical display.



**Figure 15.26: Border Editor**

The final step in defining the appearance of the Calculator window is setting the window's type. Select the Window Specifications... menu item to bring up the Window

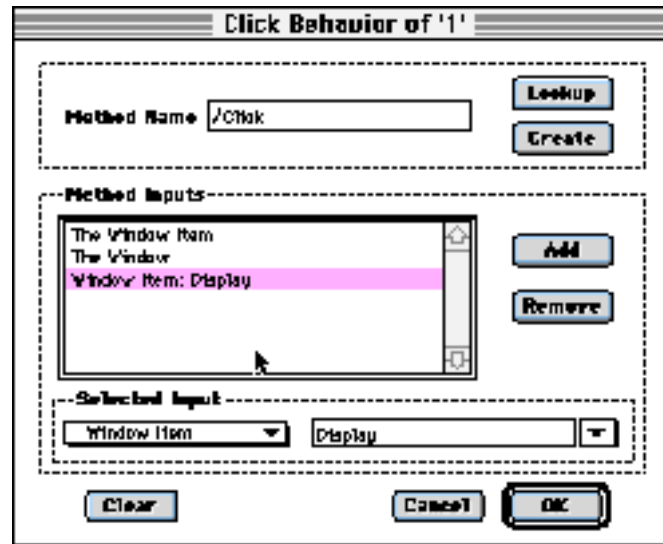
Editor, shown in Figure 15.27, and select a rounded rectangular window with a close box in its title bar. Now the finished Calculator Window should look like Figure 15.1.



**Figure 15.27: Background Editor**

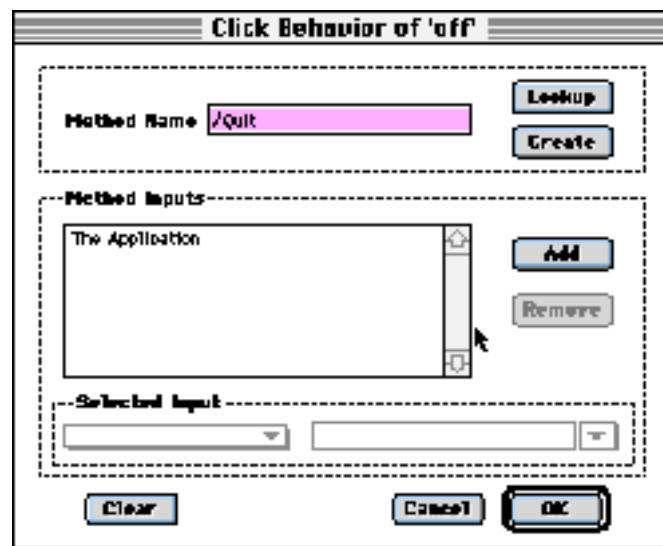
At this point, we're ready to add behaviors to our calculator keys. To define each key's behavior, select a push button in the Calculator window view, then enter the Behavior Editor via the Click Behavior... menu item. The entries for the behavior of the "1" calculator key are shown in Figure 15.28. The Method Name entry of the editor defines which class method will be called for this button. For the "1" key, we will call the Click method of the Digit subclass of Push Button. This method's inputs will be the The Window Item, which is the current Digit object (the "1" key in this case), The Window, which is the current window (the Calculator window), and Window Item named Display, which is the Text object in which the numerical display is shown (refer to Figure 15.25). As can be seen in Figure 15.27, when you select a Window Item from the leftward pop-up menu of Selected Input, a second pop-up menu appears on the right from which the particular window item is chosen. This pop-up name is not a good choice, since the item selected by the pop-up is really a method input. This pop-up probably should have been called "Input Type".

The remaining Digit buttons, the Operation buttons, the Decimal button, the = button and the clear button have the identical behavior defined as does the "1" key (Figure 15.27). That is, each of these subclasses of Push Button will each have their own class method named Click, which will be called as their Click Behavior.



**Figure 15.28: Click Behavior for the calculator's 1 button**

The only button whose behavior differs is the “off” key, which exits the application by calling the Quit method of the Application class (see Figure 15.29). Type in “/Quit” into the Method Name field of the Click Behavior Editor and add one input to the method, then select The Application from the pop-up menu of the Selected Input section of the editor.



**Figure 15.29: Click Behavior for the calculator's off button**

We should mention one last thing. In Figure 15.4, which showed the Application Editor, there was a button labelled “Initial Behaviors”. This lets you set a behavior for your program that is executed before any windows are opened. This is a good place to

place code for initializing your program, such as a method to read default program settings from a Preferences file.

That's all there is to the user interface! Without writing a single method of our own, we already have a program that emulates an electronic calculator's appearance. When the program opens, our calculator appears on the screen, ready to use. Its buttons will highlight when pressed, and its menu will allow you to quit the program. While many user interface design tools will allow you to design and test your user interface, only Prograph's GUI editors allow you to integrate your own code with that of the application framework so easily. Pretty good for so little effort. Get used to this. With Prograph's user interface tools, you can spend less time sitting at the computer staring at code and more time doing what you really want to -- sitting at the computer staring at a video game!

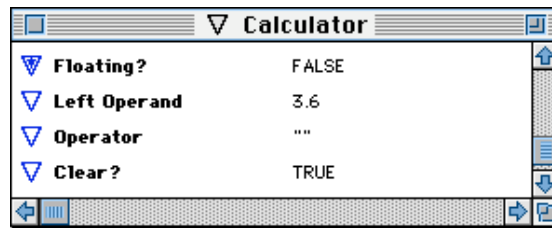
### *Writing the Calculator Program Code*

At this point, we need to write the code that will make the keys of the calculator do what they should -- place numbers in the numerical display or add or subtract. The majority of this work will be done by the Click class methods of the Digit, Operator, Decimal, = and clear subclasses of Push Button, which we'll write shortly. These methods will emulate the keys of the calculator. The Digit class' Click method will attach the digit that the key represents to the number currently displayed. The corresponding method for the Operator class will set up the mathematical function indicated by the key and the = class' method solves that mathematical function. The Click method of the Decimal class will add a decimal point to the currently displayed number if it doesn't already have one. Finally, the clear class' Click method reclears the numerical display to "0" and removes all temporarily-stored numbers and operations. We will also write a few miscellaneous methods to help these methods carry out their tasks.

The number being displayed by the calculator will be stored in string format rather than as a real number. There are two reasons for this: First, the numerical display will be formed by a Text user interface object. This object presents text in a window, so any number to be displayed would have to be converted to string format anyway. Second, it is much easier for the Digit and Decimal class' Click methods to add characters to a string for each digit or the decimal point rather than performing complicated calculations to add them to a real number. By accessing the number in string format, we've simplified our work. Even so, when we do need to use the number *as a number*, such as when we perform calculator operations like addition or subtraction on it, we can convert it from string format to number format and vice-versa with the `from-string` and `to-string` primitives.

Before we get into writing code for the Calculator program, let's add some attributes to the Calculator class that will help the calculator perform its functions. We place these attributes in the Calculator class (the class handling the window that displays the calculator) so that all of the Push Button subclasses can get access to them when

they need to. At the end of the list of **Calculator**'s inherited attributes, we'll add three attributes of our own (see Figure 15.30). The first is a real number named **Left Operand**, which holds the first number of a calculation (for example, the first of two numbers to be added, or the number from which a second number will be subtracted). The next attribute, named **Operator**, stores the mathematical calculation to be performed upon the numbers in string format. The final attribute, **Clear?**, is a Boolean flag that signals if the number to be worked on has been cleared.

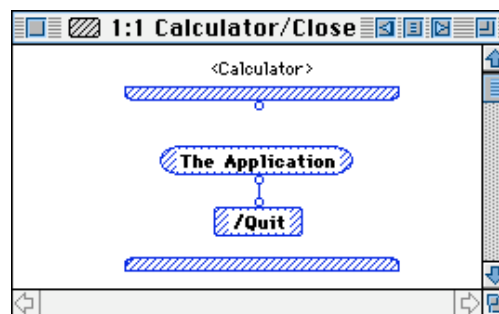


Calculator	
Floating?	FALSE
Left Operand	3.6
Operator	""
Clear?	TRUE

**Figure 15.30: Attributes of the Calculator class**

Now we can start writing code. We'll start with the **Calculator** class, which handles the **Calculator** window itself. We will override two class methods of its parent class, the **Window** class, so that they will perform new actions for our program's window. First, we will override the **Close** method so that closing the **Calculator** window not only closes the window but also exits the program. Create a new method named **Close** in the **Calculator** subclass and complete it as in Figure 15.31. This method will use the **Application** class' **Quit** method to end the program. The **Quit** method will close the window as part of its duties.

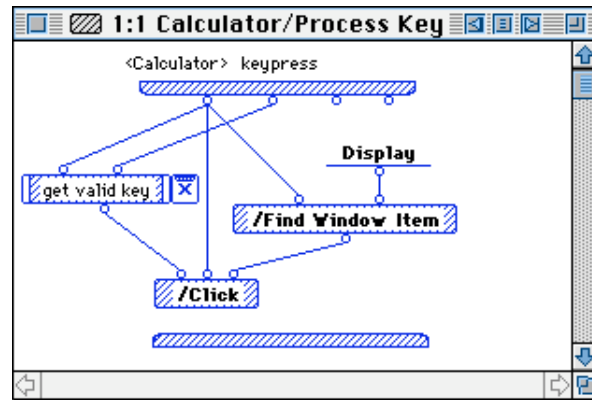
Remember that when overriding a superclass' method, you sometimes might have to execute the code of the superclass method before executing your new code. In these cases, the **Super** annotation of the **Controls** menu should be used as the first operation of the new overriding method.



**Figure 15.31: Overridden Close method of the Calculator class**

The **Process Key** method of the **Window** class is meant to handle special keypresses that have specific meaning for the application in question. This method is empty in the **Window** class -- it's up to us to override it in the **Calculator** class and add

code to this method to make it do anything at all. We will use it to provide keyboard equivalents of the calculator keys so that the user can, for example, type in a number instead of repeatedly clicking **Digit** buttons in the Calculator window. Copy the **Process Key** method from the **Window** class and paste it into the **Calculator** subclass (so the method will get the proper number of inputs), then complete it as in Figure 15.32.

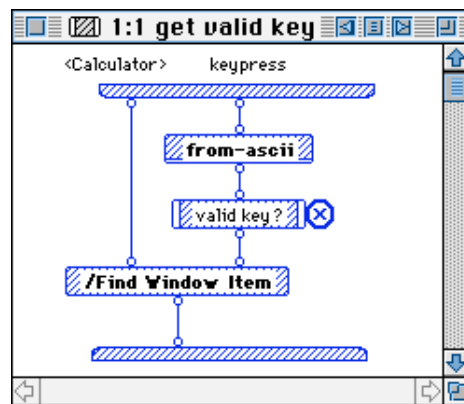


**Figure 15.32: Overridden Process Key method of the Calculator class**

This method receives as its first two inputs the **Calculator** window object and an identifier for the key that was pressed by the user. The **get key** local method finds which **Push Button** corresponds to the key pressed, and **Find Window Item** retrieves the numerical display text object. This information is all sent as input to a **Click** class method. Remember when we defined the behaviors of all of the calculator key **Push Buttons**? Each one called a class method called **Click**. For numerical keys, the **Click** method resides in the **Digit** class, for each operation key it resides in the **Operation** class, and so on. The sole purpose of naming the methods identically was to make the **Process Key** method code and other programmer-written code in the **Calculator** program much simpler. Once we have retrieved the actual **Push Button** object that the key press triggers, all we have to do is call its own **Click** method. If the key pressed was not a valid one, the **get key** local method terminates the execution of the **Process Key** method.

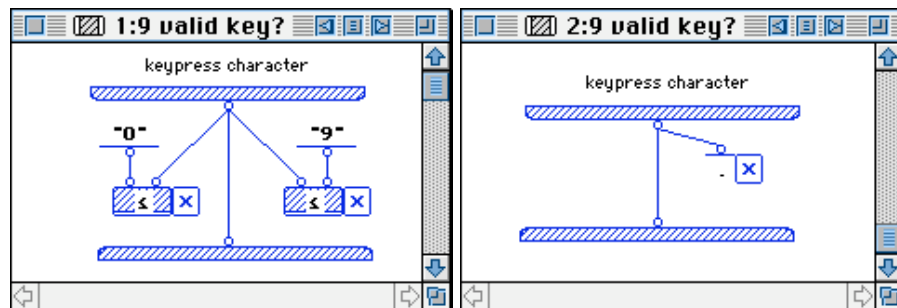
The **get valid key** local method is shown in Figure 15.33. It retrieves the key pressed, makes sure that it corresponds to one of the calculator keys, then sends the name of the appropriate calculator key **Push Button** to **Find Window Item**, which returns the **Push Button** itself from the method. The **keypress** is input into this method as an ASCII code -- a numerical code for the character that the key represents. The **from-ascii** primitive converts this code back into the appropriate character.

Note the “fail” control on the **valid key?** local method. We will be performing a match test within this local method that will either succeed or fail, depending upon the validity of the key press. This *fail* control *propagates* the results of its logical test to the enclosing **get key** method. In other words, if **valid key?** fails, this control will ensure that **get key** also fails.

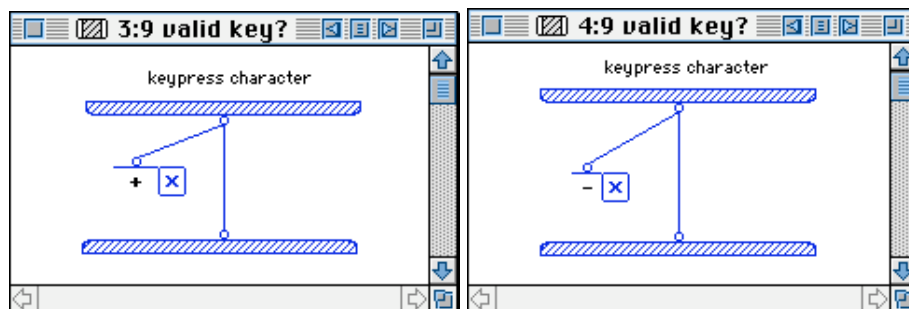


**Figure 15.33: The get valid key local method**

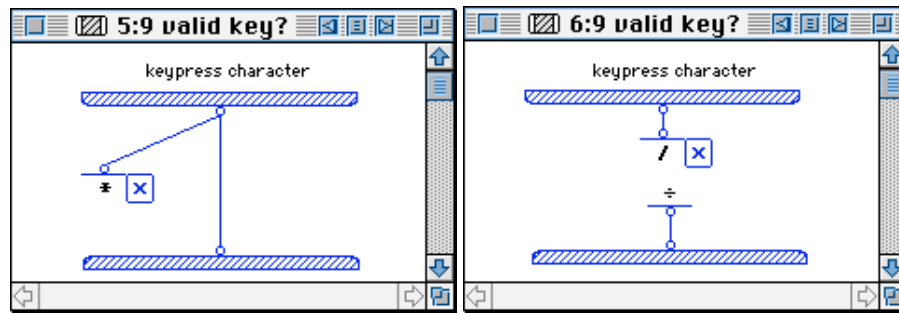
The valid key? local method is shown in Figures 15.34-15.38. The first case checks if a digit key was pressed and returns the digit. The second case confirms a press of the decimal point (or period) on the keyboard. Cases 3 through 6 detect the keys for mathematical operators, with the “/” key remapped into the division operator ( $\div$ ). The seventh case determines if the equal sign was pressed. The final two cases confirm the press of the “c” key (in either small case or capital case). If none of these cases succeed, a *fail* control transmits news of the failure to the valid key? local method’s *fail* control, which in turn propagates it to the get key local method’s attached control.



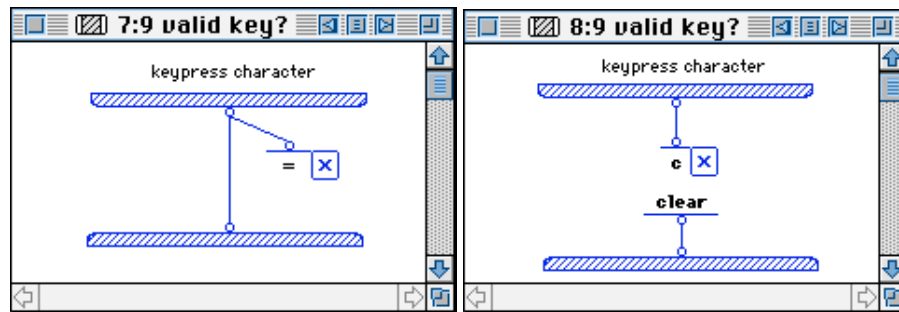
**Figure 15.34: The first and second cases of the valid key? local method**



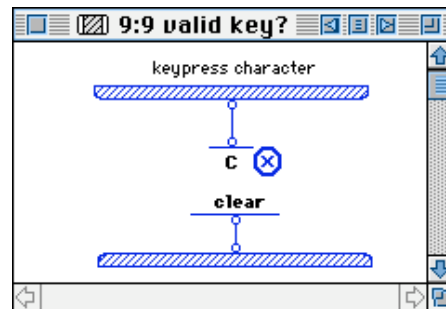
**Figure 15.35: The third and fourth cases of the valid key? local method**



**Figure 15.36: The fifth and sixth cases of the valid key? local method**



**Figure 15.37: The seventh and eighth cases of the valid key? local method**



**Figure 15.38: The ninth case of the valid key? local method**

The last method of the Calculator class is the Get Existing Value method (Figure 15.39), which returns the current number in the numerical display of the Calculator window. This method first checks if the currently-displayed number is to be cleared, and if so, returns a zero. If not, the number currently being displayed is returned.





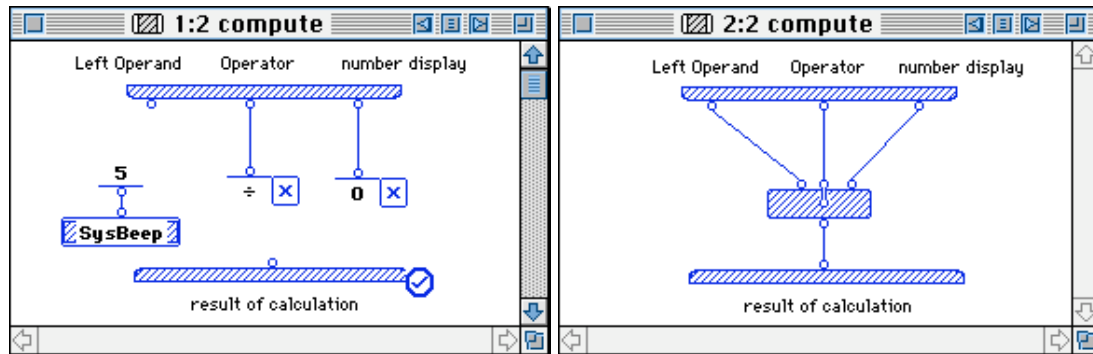
```

graph TD
    Title["<Calculator>"]
    Display["number display"]
    LO["Left Operand"]
    OP["Operator"]
    GV["/Get Value"]
    FS["from-string"]
    C1["compute"]
    C2["compute"]
    RW["/Reset Window"]
    RV1["reset value"]
    RV2["reset value"]

    Title --- Display
    Title --- LO
    Title --- OP
    Title --- GV
    LO --- C1
    OP --- C2
    GV --- FS
    C1 --- RW
    C2 --- RV1
    FS --- RV2
  
```

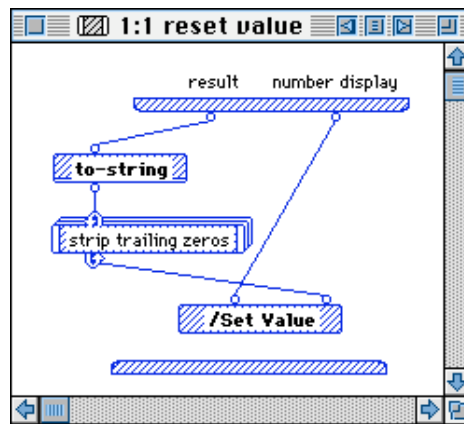
273

The compute local method (see Figure 15.41), for all it accomplishes, is actually a very simple method. All it does is call the appropriate mathematical primitive by means of an *inject* construct, using the currently displayed number and the stored left operand as the primitive's inputs. The method also contains a test for division by zero and prevents this from occurring, while warning the user with a beep.



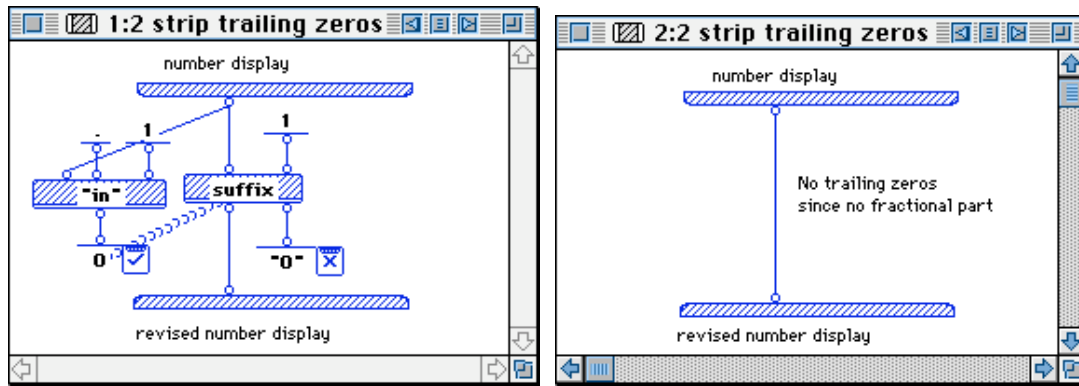
**Figure 15.41: The compute local method**

The reset value local method (Figure 15.42) simply encapsulates the steps involved in setting the new value of the number display.



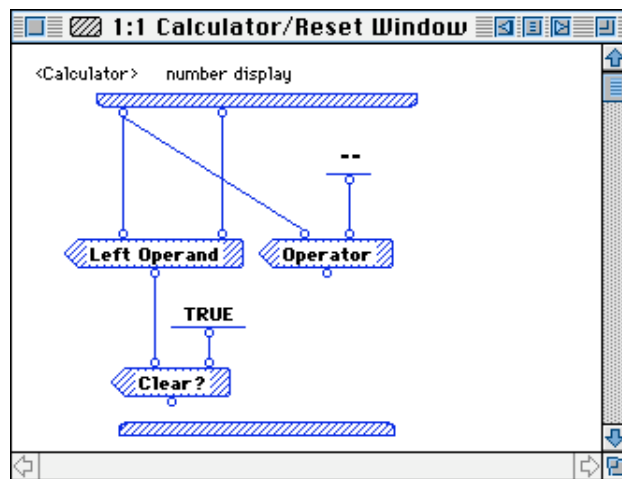
**Figure 15.42: The reset value local method**

The strip trailing zeros local method (Figure 15.43), as its name implies, removes zeros from the end of a number via the `suffix` primitive, which strips characters from the end of the string representing the number. Before doing so, it checks that trailing zeros (after a decimal point) are really present by searching the number display string for the character “.”. If none is present, the method does nothing.



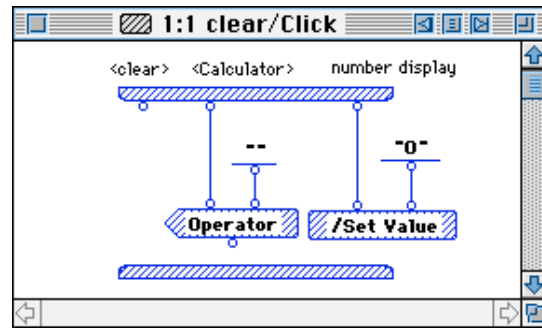
**Figure 15.43: The strip trailing zeros local method**

The final method of the Calculator class is Reset Window, shown in Figure 15.44. This method makes sure that the left operand hasn't been cleared and that the operator is current (that is, not empty).



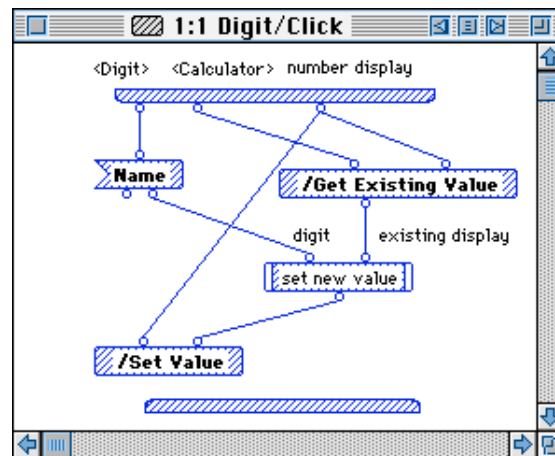
**Figure 15.44: The reset window local method**

Let's proceed with the Click methods for each calculator key, which define the behavior of these keys. The clear button's Click method is shown below in Figure 15.45. It resets the Calculator's Operator attribute with an empty string and displays a zero in the Calculator window's text object.



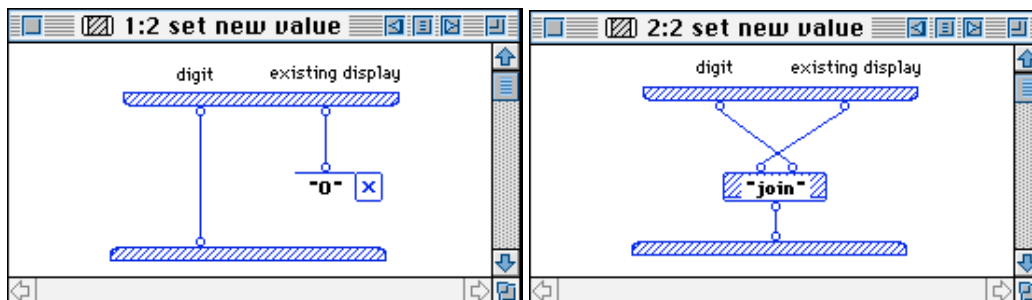
**Figure 15.45: The Click method of the clear class**

The Digit class' Click method (Figure 15.46) gets the currently-displayed number with the Calculator class' Get Existing Value method, and the name of the calculator key being clicked, then appends the new digit to the end of the currently-displayed number.



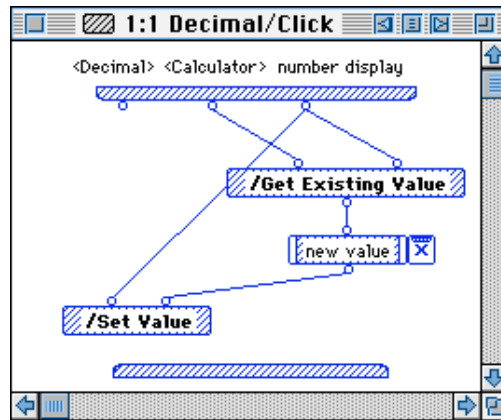
**Figure 15.46: The Click method of the Digit class**

The set new value local method (see Figure 15.47) checks if the displayed number is a zero. If not, it joins the digit corresponding to the keypress to the end of the currently-displayed number.

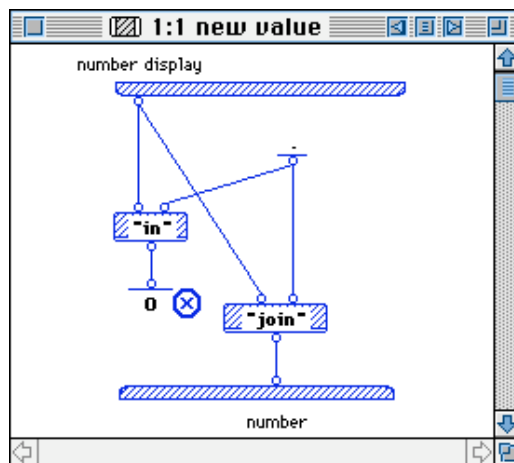


**Figure 15.47: The set new value local method**

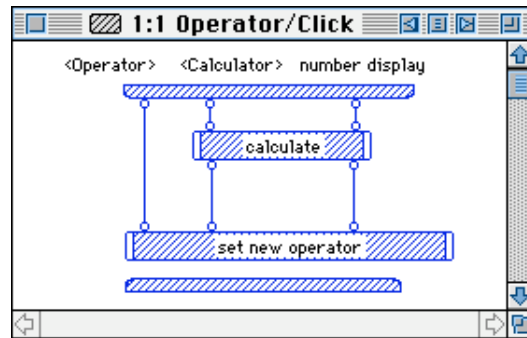
The **Decimal** class' **Click** method is called when the decimal point calculator key is pressed (see Figure 15.48). It gets the currently-displayed number with the **Get Existing Value** local method (Figure 15.39), then sets a new value for the number and displays it.

**Figure 15.48: The Click method of the Decimal class**

The **new value** local method (Figure 15.49) checks if a decimal point already exists in the number. If so, the method *fails*. If not, a decimal point is appended to the number.

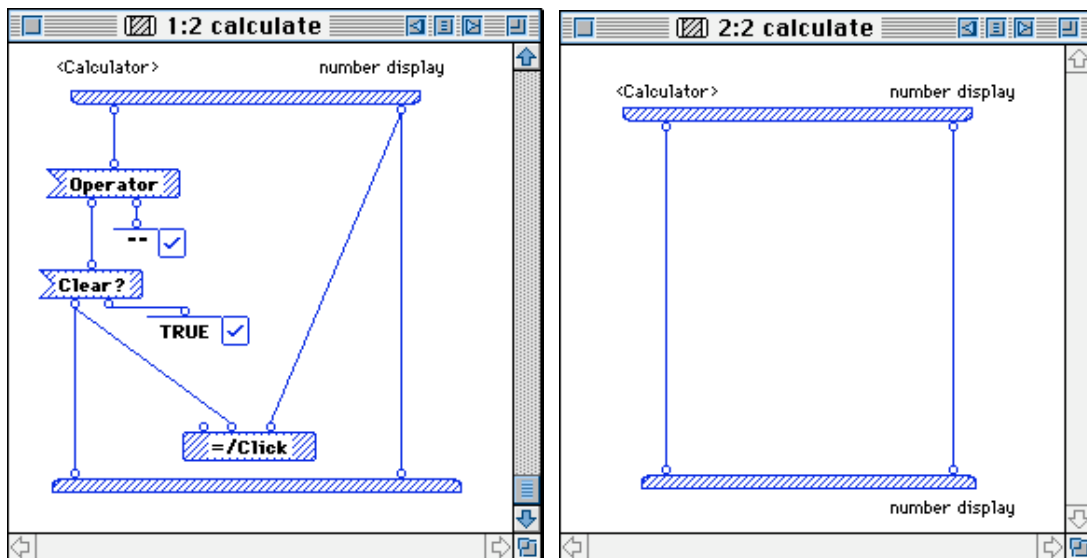
**Figure 15.49: The new value local method**

The **Operator** class has its own **Click** method (see Figure 15.50) that performs a calculation upon the numbers. It simply calls two local methods to do its work.



**Figure 15.50: The Click method of the Operator class**

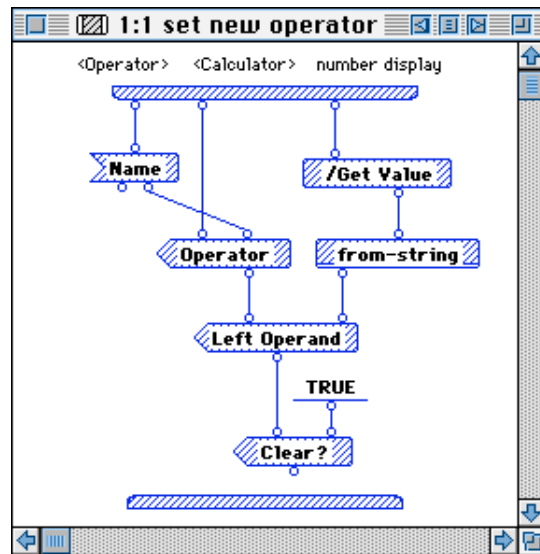
The first local method, *calculate* (Figure 15.51), makes sure that the number hasn't been cleared and that the operator is current (that is, not empty). It then calls the *Click* command of the *=* class, using an *explicit* method call reference. The *=* class' *Click* method "solves" the mathematical operation and displays its result. The direct calling of another class' method (via the explicit reference call of "ClassName/Method") should really be avoided if possible, since it decreases the reusability of the *Operator* class that is making the call. However, it is a simple solution to the problem of calling a method contained in an object not supplied as an input to the calling method. We employ it here simply to demonstrate this type of method call.



**Figure 15.51: The calculate local method**

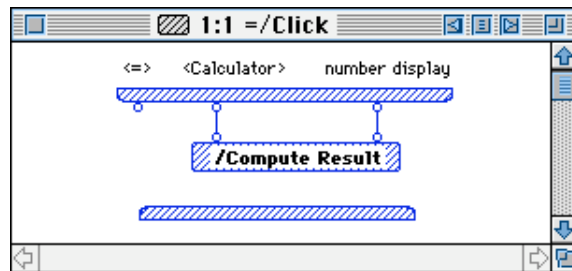
The *set new operator* local method, shown in Figure 15.52, prepares the program for the next calculation by setting the current *Operator* to the operator indicated by the calculator key that was pressed, putting the currently displayed number in *left operand* for it to be worked on, then setting the *Clear?* flag. The number stored in *left operand* is

read from the numerical display of the window (a Text object) with the Text class' Get Value method, which returns the string stored in the Text object.



**Figure 15.52: The Click method of the Operator class**

The final click method for the program is that of the = class, the push button for the “=” key. This method, shown in Figure 15.53, just calls the Calculator class' Compute Results class method.



**Figure 15.53: The Click method of the = class**

That's all there is to it! That wasn't so bad, was it? An entire calculator program in only *eleven programmer-written methods*. All the rest of the code is contained in the Application Builder Classes themselves. You might be tempted to take the compiled program and the handful of pages of printed source code to your friends who program in C, Pascal or C++ and *laugh your head off* as they struggle for days or weeks to code the same calculator! (Just make sure they don't own a gun). This is just a small utility program -- the real point here is that writing large full-fledged applications is also made easier with the ABCs and ABEs. The larger the program, the more time you save by using the Application Builder.

## *Summary*

Giving a program a rich user interface is simple with Prograph CPX. Instead of hard-coding a series of operating system calls or retooling our program to fit a rigid application framework, we can tailor Prograph CPX to meet our program's needs with a few simple clicks of a mouse and choices from pop-up menus. By the time you've finished designing your program's user interface, there's very little coding left to do.

We also introduced a few new concepts in the course of this chapter:

- Defining *custom user interface elements* is child's play in Prograph. All you need do is subclass an existing GUI element, then add the new one to the View Editor's palettes so that it may be added to future programs as well.
- The *inject* construct helps to reduce the amount of code needed in a program by merging similar code into a single function that can call one of several other methods by name.

This version of the Calculator program is short and was written quickly. It minimizes code by subclassing already-available user interface element classes in the ABCs. All of the digit inputs use the same code and all of the operations can be carried out with the same inject. This is certainly an efficient solution to the problem of writing a calculator program, but it has two drawbacks. First, the implementation of the calculator is very strongly tied to its representation in the user interface. This makes it harder to add new functions to the calculator in the future. Second, it creates subclasses of user interface classes that can't be reused and added to a wide range of new programs (such as a "print" button, which could be added to the Controls Palette and added to any program). So while this program was built quickly, it is essentially a dead end. However, this is not a bad approach for programs that will have a limited usefulness or a limited lifetime. Let's progress now to another way to approach the calculator program.