

## **16. Applying the ABCs -- Part 2: Son of Calculator**

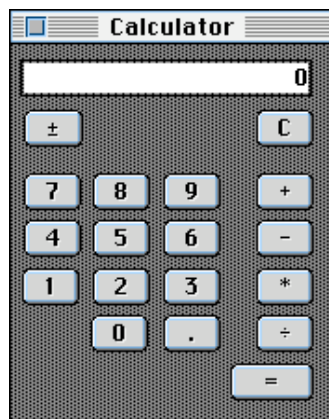
### ***Overview***

The Calculator program of the previous chapter was short and elegant. Its simplicity made it fast to program, but there was a price to be paid for its simplicity -- a tight coupling between its calculator functions and the representation of the calculator its user interface. Such tight coupling makes it harder to modify, extend, or reuse the calculator code, as well as modify the user interface.

In this chapter, we'll rewrite the Calculator program by splitting its core functions into an independent class not derived from the Application Builder Classes. While this will add a few more methods to the program, its benefits are to allow us to reuse the calculator code in other programs and to *subclass* the class that implements the calculator in the future to easily add more features. The small amount of extra programming we have to do now will therefore save us a ggreate amount of programming in the future. This is one of the real benefits of OOP.

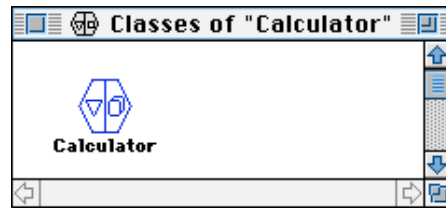
### ***The Calculator Program -- Version 2***

Our second Calculator program will appear like the window shown in Figure 16.1. You might notice that we've also added a new button to the calculator --  $\pm$  -- which will change the sign of the currently displayed number of the calculator.



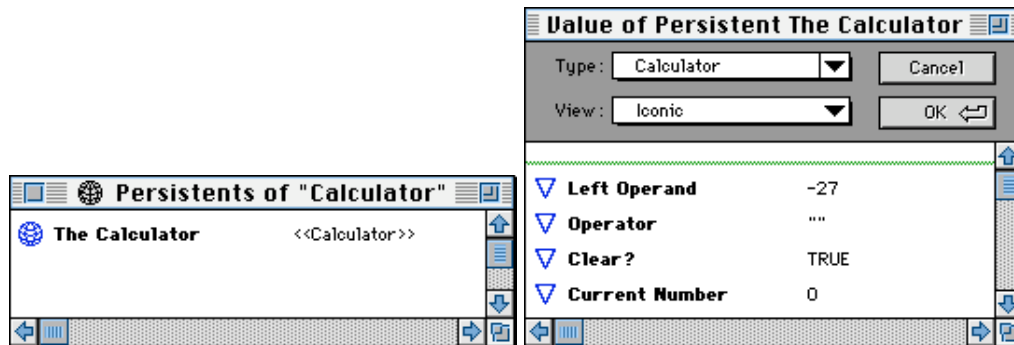
**Figure 16.1: The Main Window of the Calculator program**

The class that will make this program work is a class named `Calculator`, which we create and place in a section also called `Calculator` (Figure 16.2). Most of the code that we'll have to write for this program will be confined to this class alone. The remainder of the program code will call these methods to do the real work of the calculator.



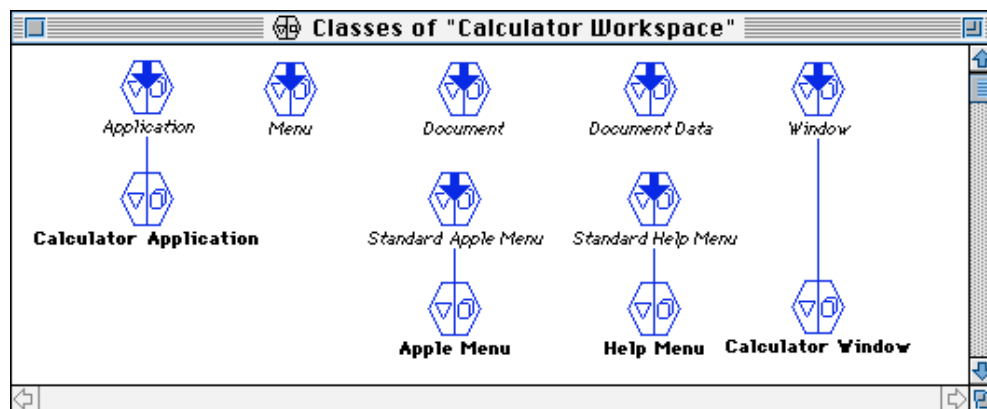
**Figure 16.2: Classes of the Calculator Workspace section**

To easily access a **Calculator** object at several places in our program, we will place an instance of the **Calculator** in a persistent called **The Calculator**, also in the **Calculator** section (see Figure 16.3).



**Figure 16.3: The 'The Calculator' persistent**

While building the user interface of the program, two new classes will be added to the **Calculator Workspace** section (see Figure 16.4). The first new class for our program will be **Calculator Window**, which is created by the Application Builder Editors for us when we define the window's view. The second is **Calculator Application**, which really is just a renamed version of the **Starter Application** subclass of **Application** that is supplied with the ABC Starter Application project. So, as you can see, we really won't have to write much code at all to handle the user interface.



**Figure 16.4: Classes of the Calculator Workspace section**

Even though some of the code of this program will be similar to that of Chapter 15, the design of the program and therefore much of its code is different enough that we might as well build this program from scratch by opening the ABC Starter Application project that comes with the Prograph development system. Save the new program project under the name “Calculator Project” and the new workspace section of the program as Calculator Workspace” (but keep these sections in a different folder than the identically-named sections of the previous program, so the Prograph environment can differentiate between them).

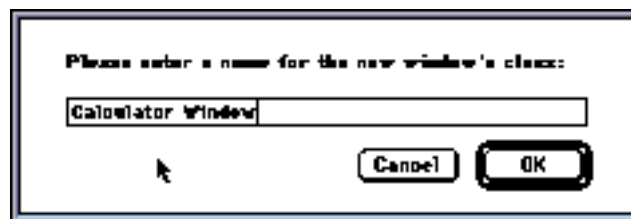
### ***Designing the User Interface of the Calculator Program***

It’s time to begin building the user interface. Select the Edit Application menu item and enter the Application Editor. Once again, enter “Calculator” for the name of the program and ‘Calc’ as the signature for a Macintosh application.

Now open the Menubar Editor. As we did in the first Calculator program, we will remove the Basic Edit menu and leave our program with only a File menu. The File menu’s single Quit menu item will once again be sufficient for our program. Remember that the default behavior of this menu item is to call the **Quit** method of the **Application** class. Our program now has a fully functional menu without any additional programming on our part.

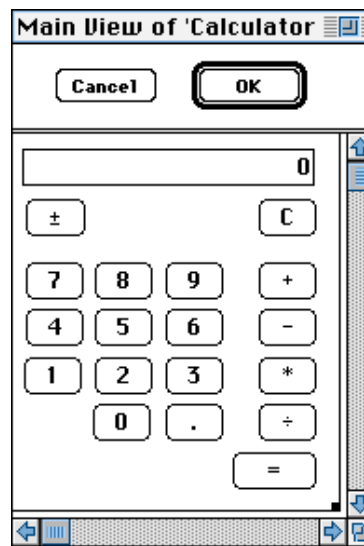
Let’s proceed with the main display window of the calculator program. Return to the Application Editor, then enter the Windows Editor.

When the Windows Editor opens, select the New Window menu item to define our new window. Enter the name Calculator Window when prompted for the name of the window (Figure 16.5). The new Calculator Window will now be added to the list of available windows for our program.



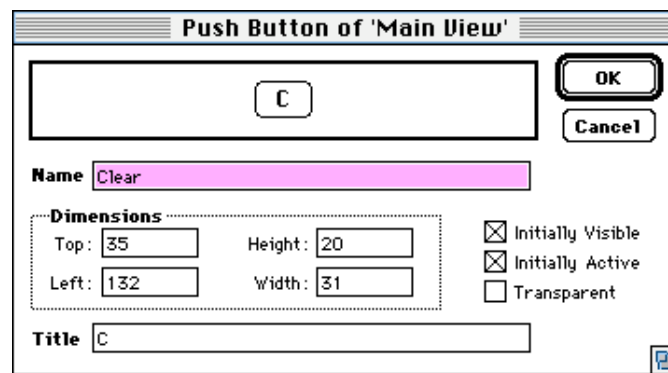
**Figure 16.5: Entering the name for the Calculator Window class**

Select the Calculator Window from the list of available windows and enter the View Editor for the window. Arrange the Calculator Window view by dragging one text-editing box (for the numerical display of the calculator) and 18 push buttons (for the calculator keys) from the Controls palette onto the View Editor window. Resize and position them as shown in Figure 16.6.



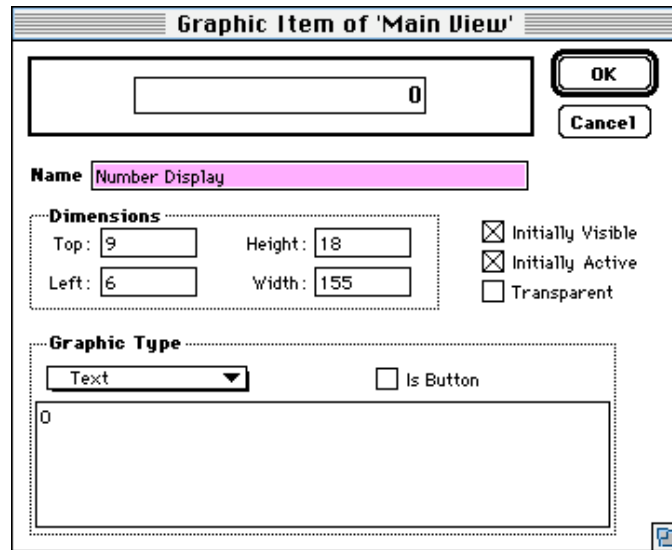
**Figure 16.6: Main View of the Calculator Window with added push buttons and text-editing box**

The push buttons will all need to be given an appropriate name and label by selecting each button in turn and entering its Push Button Editor (Figure 16.7 shows the data for the Clear button of the calculator). For each number-entry key of the calculator, use the appropriate number for each button's title (that is, "1" through "0") and the text equivalent of the number for each key's name (for example, "Four" for the "4" key). The operation keys are named for their operations (Add, Subtract, Multiply, Divide) and titled with the mathematical symbol for their operation, while the  $\pm$  button is named Negation.



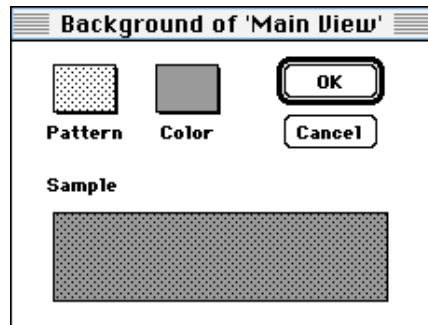
**Figure 16.7: Push Button Editor**

The Text object for the numerical display of the calculator should be given the name Number Display and an initial text display of 0 (see Figure 16.8).



**Figure 16.8: Text Editor**

Let's add a background to the Calculator window. Select the Background... menu item to bring up the Background Editor, shown in Figure 16.9, and select a lightly dotted background pattern and medium gray background color.



**Figure 16.9: Background Editor**

The final step in defining the appearance of the Calculator Window is defining the window characteristics. Select the Window Specification... menu item to bring up the Window Editor (Figure 16.10), and select a rectangular window with a title bar and a close box. Now the finished Calculator window should look like Figure 16.1.

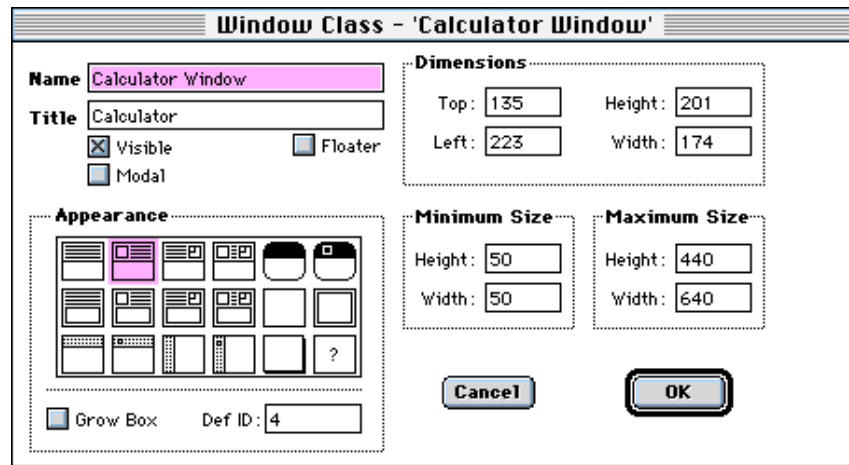


Figure 16.10: Window Editor

Now that the Calculator window looks like a calculator, let's make it *act* like one by adding behaviors to its keys via the Behavior Editor. The entries for the behavior of the Clear calculator Push Button are shown in Figure 16.11. This button will call the Clear method of the Calculator class. The method's sole input is The Window, in other words, an object of type Calculator Window. This makes the Clear method a class method of Calculator Window.

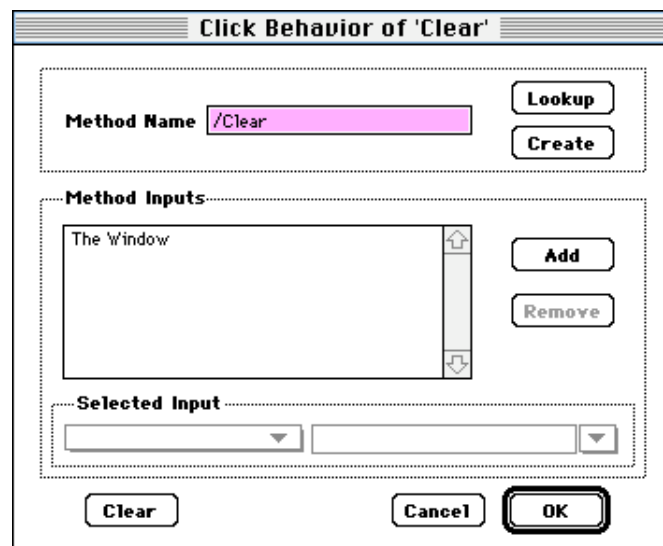
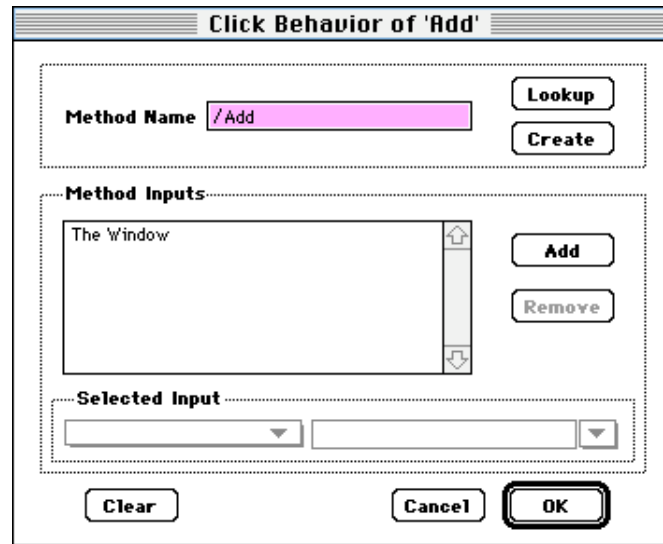


Figure 16.11: Click Behavior for the calculator's Clear button

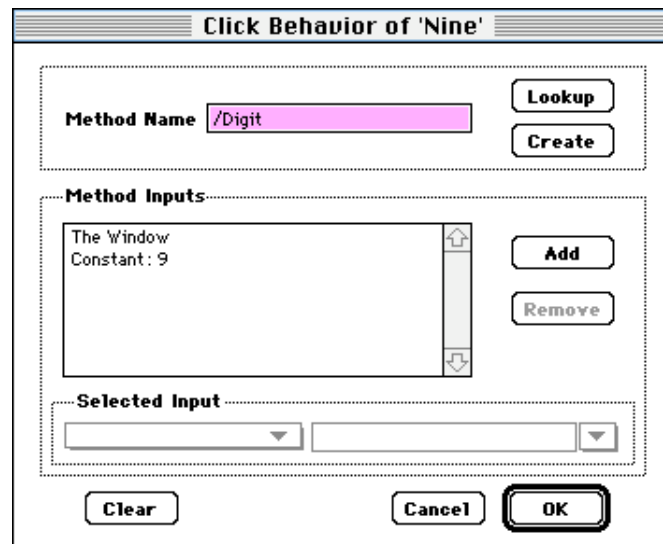
The behaviors of the “.”, “=” and “±” push buttons are defined with the same method input (The Window), but are given method names of Decimal, Equal and Change Sign, respectively. Each number key of the calculator is given a behavior in which the method name is the text form of the number that labels the key and which has only one input -- The Window. Similarly, the operator keys each have a behavior with a

method name that describes the mathematical operation that the key will carry out and a single input for the The Window (see Figure 16.12 for the “+” button).



**Figure 16.12: Click Behavior for the calculator's + button**

The behaviors of the digit push buttons are defined with the same method name (/Digit), but an extra method input -- a *constant* -- is given a different value for each digit key. The value of the constant is simply the number of the digit push button -- 1 for the “1” key, 2 for the “2” key, and so on. In this manner, all of the digit push buttons reuse the same behavior method code. Figure 16.13 shows the behavior definition for the “9” key of the calculator.



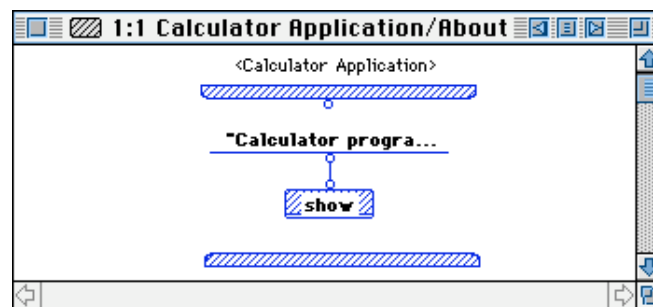
**Figure 16.13: Click Behavior for the calculator's '9' digit button**

That completes the design of the user interface for our second version of the Calculator program. So far, it's not very different from the first version of the program. In fact, there are really only one significant difference in the user interface. Instead of creating special subclasses of **Push Button** to handle digit input, mathematical operators, and so on, we've just used the generic **Push Button** included with the Application Builder Classes. The behaviors of the buttons will take advantage of the new **Calculator** class that will handle the tasks of the hand-held calculator. The real changes in our new calculator program will come now as we write the code for the Calculator program.

### ***Writing the Calculator Program Code***

For this version of the program, we'll change four methods included in the ABCs to modify their function by *overriding* them. After this, we'll design the **Calculator** class and write the class methods that perform the calculations for this program. Finally, we'll write the click methods for the **Push Button** objects in the Calculator window.

The **Application** class in the Calculator Workspace section contains a class method named **About**. This method is used to display an About Box on the screen, which presents to the user information such as the program version, author and copyright. Rather than modify code within the ABCs themselves, we create another **About** method in the **Application** class' subclass -- the **Calculator Application** class. This second copy of the **About** method will then *override* the original copy in the parent class. When the program runs, it will call the **About** method in the child class -- the one that we're about to change. The overriding **About** class method will just display the message "Calculator program by Dr. Scott B. Steinman", as shown in Figure 16.14.

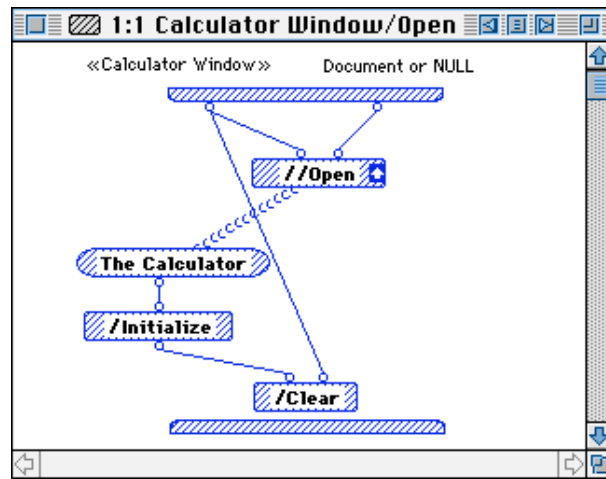


**Figure 16.14: The overridden About method in the Calculator Application subclass of Application**

We will also override some class methods of the **Window** class to make them perform specific actions for our Calculator window. The **Open** method of the **Window** class sets the position of a window on the screen, then opens the window and activates it. While we'd like our Calculator Window to do this as well, we'd also like it to do a little bit more work when it's opened. We'd like it to initialize the calculator before the user works with it. In the **Calculator Window** subclass created by the Application Builder Editors, we'll add a method named **Open** to override that of the **Window** class. Copy the



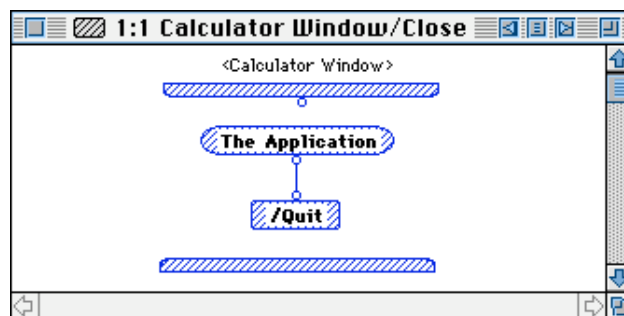
Window class' **Open** method into the **Calculator Window** subclass and modify its first case as shown in Figure 16.15. The second case, which brings an existing window to the forefront of the screen, will be left unchanged.



**Figure 16.15: The overridden Open class method of the Calculator Window class**

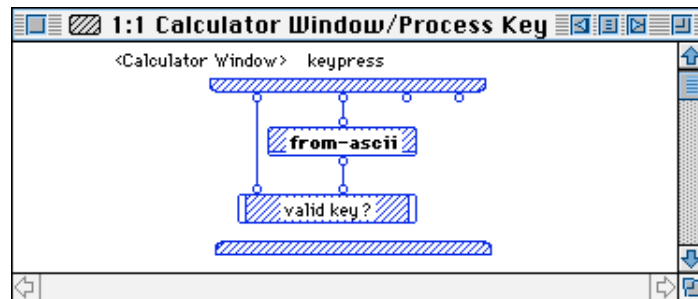
In the **Calculator Window** subclass' **Open** method, we first call its superclass' **Open** method, which will open the **Calculator Window** before we perform our own additional actions. Then we retrieve the **Calculator** object stored in the **The Calculator** persistent and call the **Calculator**'s initialization method and clear its numerical display's memory. By overriding the **Open** method, we get the **Calculator** object automatically initialized every time the program is started.

The last two methods to be overridden are the same ones we overrode in the first version of the **Calculator** program -- the **Window** class' **Close** and **Process Key** methods. Cut and paste these from the parent **Window** class into the **Calculator** subclass to give us a starting point for modifying them. The **Close** method, as before, will quit the application when the window is closed when the user clicks in its close box (see Figure 16.16).

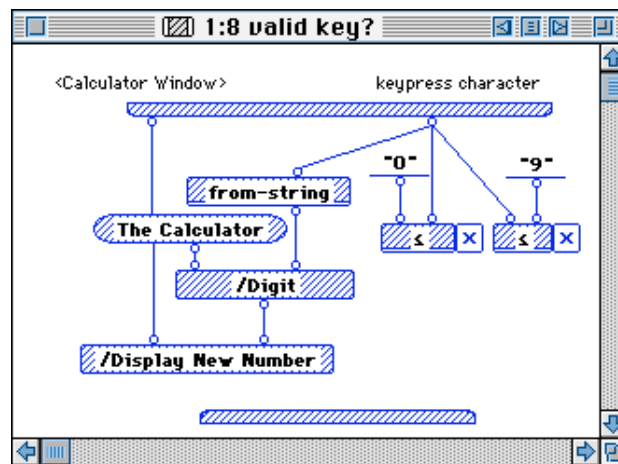


**Figure 16.16: The overridden Close class method of the Calculator Window class**

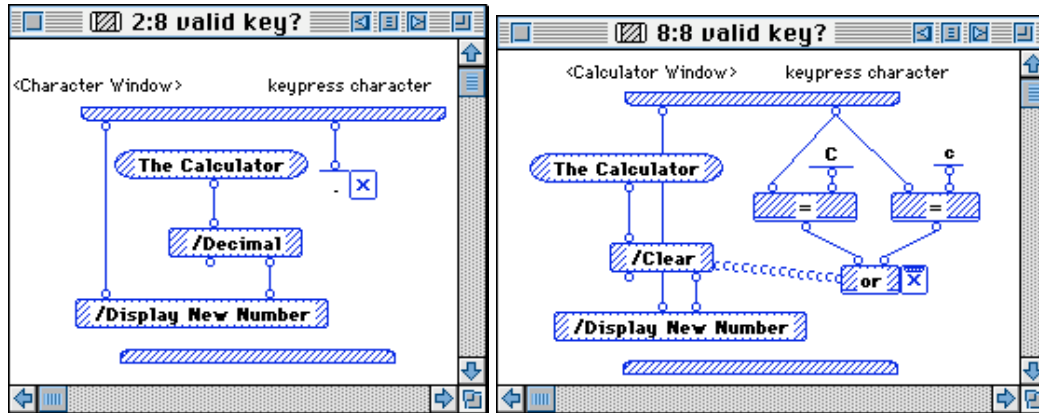
The Process Key method shown in Figure 16.17 differs from that of the previous version of our program. It retrieves the current keypress and sends it to a local method named valid key?. In the overridden Process Key method, we use only the first two nodes out of the four that the parent class' Process Key method had, but don't delete the other two nodes -- the code that calls the Process Key method still expects to find these inputs.

**Figure 16.17: The overridden Process Key class method of the Calculator Window class**

The first case of the valid key? local method, which handles digit input only, is shown in Figure 16.18. The valid key? method has two inputs -- an instance of the Calculator Window subclass of Window, and the character ("0"- "9") of the key that was pressed. This method converts the keypress' character to its corresponding digit, then calls the Digit method of the Calculator class. Finally, the Display New Number method, which will present the result of the Digit method in the Calculator window. We'll discuss these methods shortly.

**Figure 16.18: The first case of the valid key? local method**

The remaining cases of valid key? handle the decimal point key, the operators, the equal key and the clear key. Each key, other than the clear key, is serviced with a case similar to that shown on the left-hand side of Figure 16.19. For each of these cases, the character corresponding to the key's label is tested for, then the appropriate class method of **Calculator** called. The case for the clear key, shown on the right-hand side of the figure, tests for whether the user pressed *either* 'c' or 'C' on the keyboard. Here we remove the control normally placed on the = primitive, and add a node to extract its Boolean output, which indicates the success or failure of the logical test. The outputs of both = tests are sent to an **or** primitive. If this test fails, the local method is terminated.



**Figure 16.19: Additional cases of the valid key? local method**

Now let's work on the **Calculator** class itself. In this new version of the **Calculator** class, we want to separate the code of the **Calculator** class from any code dealing with the **Calculator** window or the window's parts. One new attribute and several redesigned class methods will reflect this new approach to the **Calculator** class.

The class has four attributes, shown in Figure 16.20, three of which correspond to the attributes of the **Calculator** class of the first version of this program -- **Left Operand**, **Operator** and **Clear?**. Notice that we've added one more attribute to the **Calculator** class -- **Current Number** -- which holds the value of the number currently being worked on by the calculator. In the previous version of the calculator program, we retrieved this value from the numerical display text-editing box of the **Calculator** window. In our new **Calculator** class, we store the current number *within* the **Calculator** class.

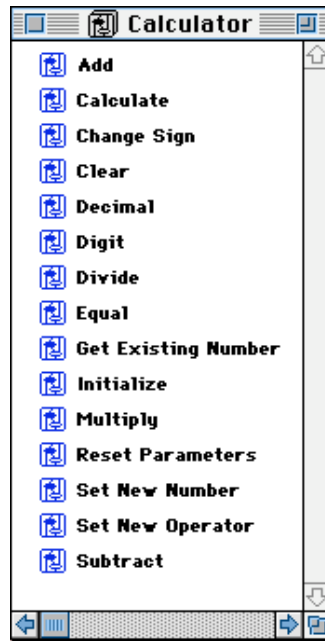
Calculator	
Left Operand	0.0
Operator	""
Clear?	TRUE
Current Number	0.0

**Figure 16.20: Attributes of the Calculator class**

The class methods for the **Calculator** class (see Figure 16.21) fall into several categories. First, a method is provided for number keys of the calculator to attach a digit to the right of the current number stored in the **Current Number** attribute (and which also will be displayed in the numerical display text-editing box of the **Calculator Window**). Secondly, there are methods for each of the operations of the calculator -- addition, subtraction, multiplication, division and changing the sign of a number. Third, there is a method to handle typing of a decimal point into a number to separate the integer and fractional portions of the number. Fourth, there is the **Calculate** method that determines the results of the calculation. Fifth, methods already exist for initializing the attributes of the **Calculator** class and clearing the **Current Number** and display to zero. Finally, methods are provided to store the operands of the calculations in the **Current Number** and the operation to be performed on them in the **Operator** attribute.

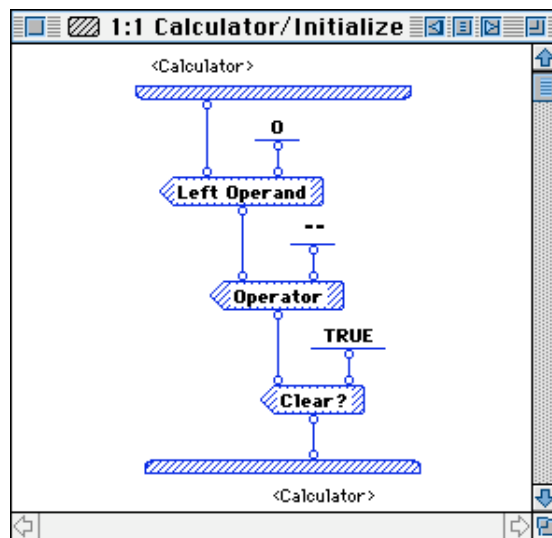
The first thing you'll notice in the new **Calculator** class is that its class methods, unlike those of the previous chapter's **Calculator** class, no longer take an object of type **Calculator Window** as an input. The second thing you'll see is that several of these methods return the value of the current number, so that it can be passed on to methods of the **Calculator Window** class, which will then display the number in the **Calculator** window by itself. The **Calculator** class no longer depends upon elements of the user interface to do some of its chores. The **Calculator** and **Calculator Window** classes now work independently. This independence makes it easier to modify or subclass each of these classes for future programs.

Although it is organized completely differently than the classes of the previous **Calculator** program, it doesn't use more methods than did its predecessor. Some extra methods will be added to the **Calculator Window** class, but they are quite short and simple. What this boils down to is that changing the program design to be more reusable doesn't mean a sacrifice by adding a lot of extra work programming.



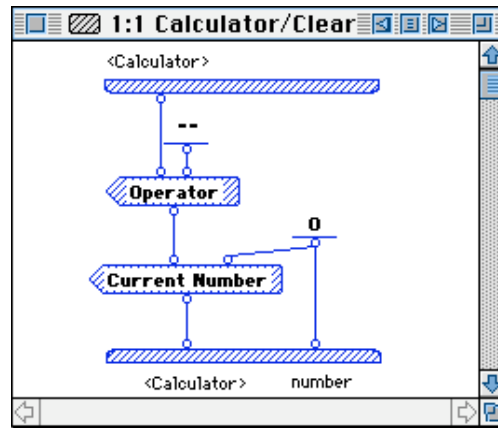
**Figure 16.21: Class methods of the Calculator class**

The `Initialize` method is called when the Calculator Window is first opened (see Figure 16.22). It simply sets the initial state of the Calculator class' attributes.



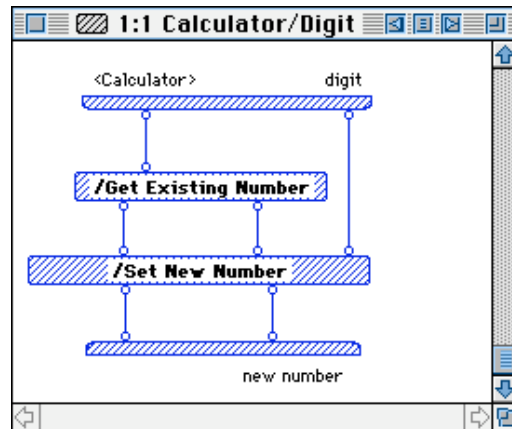
**Figure 16.22: The Initialize method of the Calculator class**

The `Clear` method (Figure 16.23) performs the same as that of the first version of the program, except that it places the value 0 in the `Current Number` attribute of Calculator rather than enter it into the text-edit box of the Calculator Window.



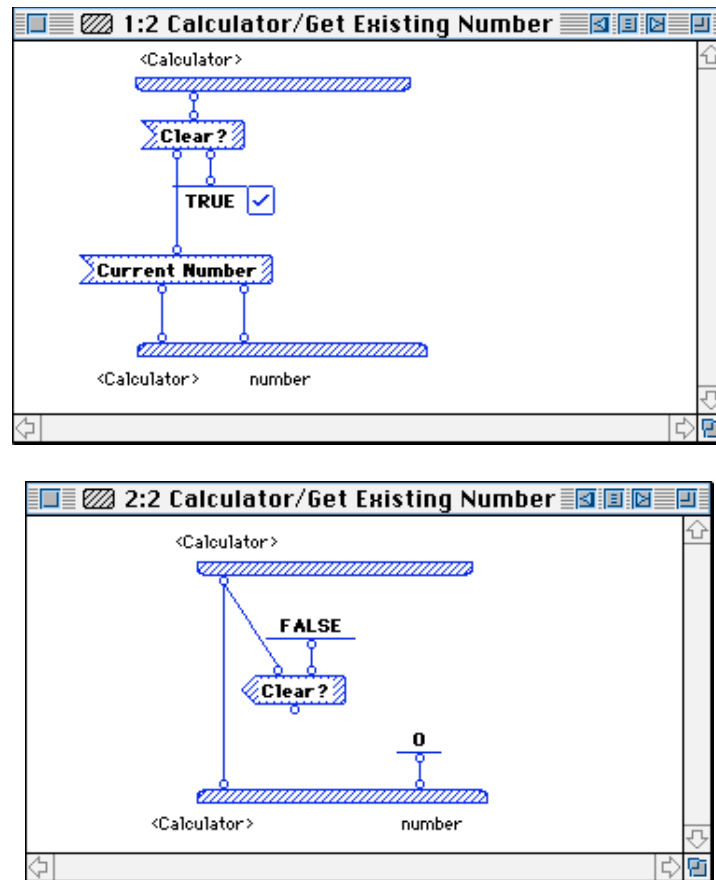
**Figure 16.23: The Clear method of the Calculator class**

The numerical keys of the calculator all perform essentially similar actions. Therefore, they all share the Digit method as their click behaviors. The Digit method (Figure 16.24) begins by getting the current number of the calculator via the Get Existing Number method, then appends the digit pressed (or typed) by the user with the Set New Number method.



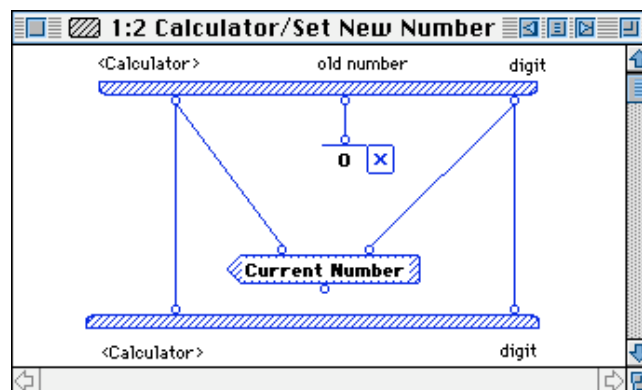
**Figure 16.24: The Digit method of the Calculator class**

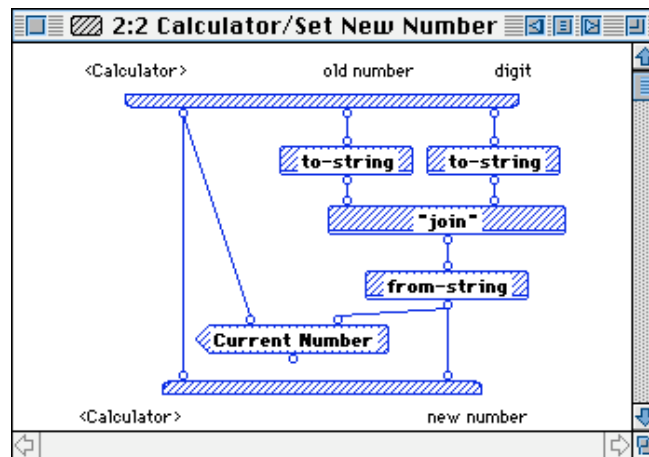
The Get Existing Number method (Figure 16.25) is the analogue of the get existing value local method of previous chapter. This method checks if the currently-displayed number should be cleared and returns a zero if it should. If not, the current number is returned.



**Figure 16.25: The Get Existing Number method of the Calculator class**

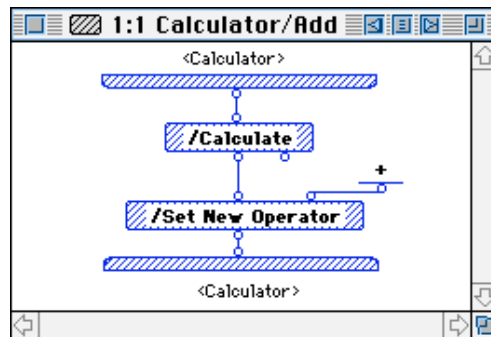
Set New Number (Figure 16.26), like the new value local method of the first version of the Calculator program, checks the number now being displayed by the calculator. If it's currently zero, it passes through the digit pressed by the user so that it will replace the zero in the Current Number attribute. If the Current Number is not zero, the digit pressed is added to the number. The current number and the digit are first converted to character form, then the digit is added to the end of the number's string representation with a "join" primitive. The new number string is then reconverted back into a real number, then stuffed back into the Current Number attribute.





**Figure 16.26: The Set New Number method of the Calculator class**

Just as the digit keys all share a single method to perform similar actions, the class methods that make the operator keys function are all very similar since they share common actions and therefore can share common code. In Figure 16.27, we see the Add method. This method calls two other methods -- Calculate and Set New Operator.

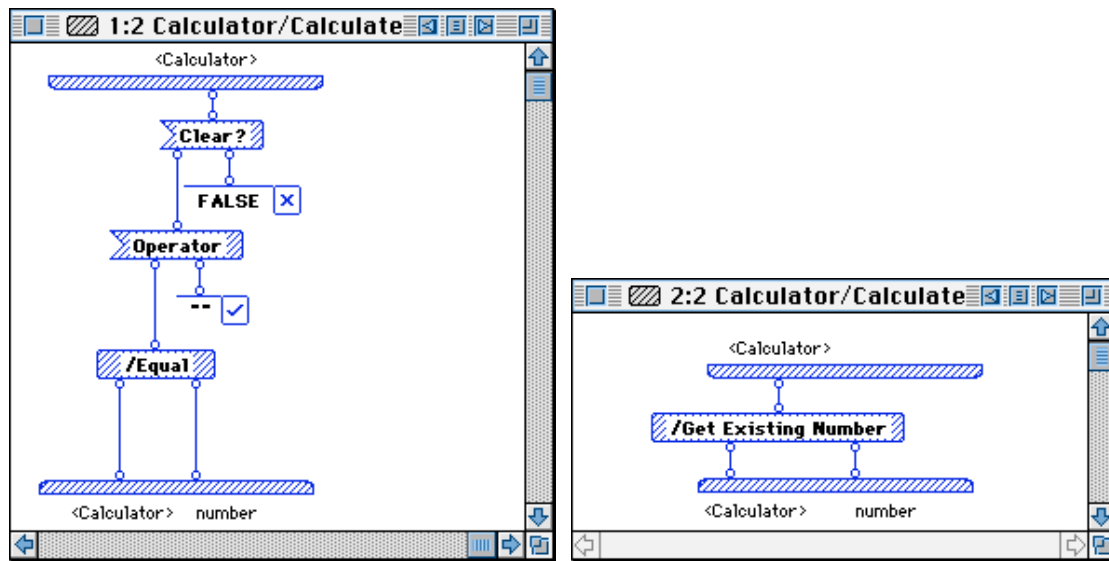


**Figure 16.27: The Add method of the Calculator class**

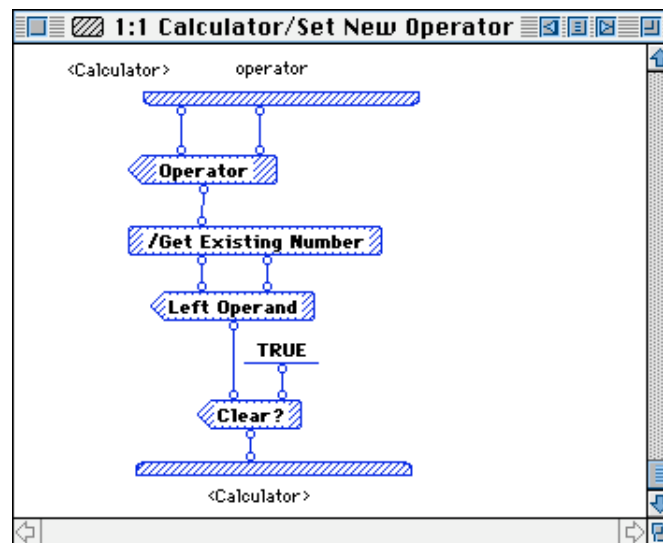
The new versions of the Calculate and Set New Operator methods are shown in Figures 16.28 and 16.29. The Calculate method checks if the Clear? flag is set. If so, the current number is returned. Otherwise, the Operator attribute is cleared and the Equal method, which performs the actual calculation, is called.

The Set New Operator method sets the value of the Operator attribute with the new operator, then places the Current Number into the Left Operand attribute in preparation for the number to be entered as the second operand for a calculation.





**Figure 16.28: The Calculate method of the Calculator class**



**Figure 16.29: The Set New Operator method of the Calculator class**

The Equal class method, shown in Figure 16.30, is the workhorse of the Calculator program, since it is responsible for the actual calculations of the program. It is quite similar to that of its predecessor program except for its inputs and outputs. The Equal method's compute and strip trailing zeros local methods, also similar to their predecessors, are depicted in Figures 16.31-16.32.

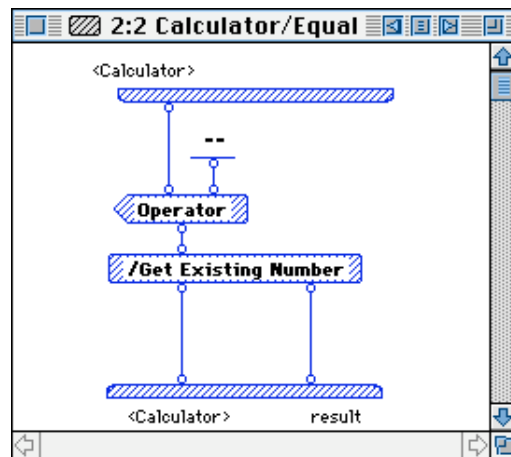
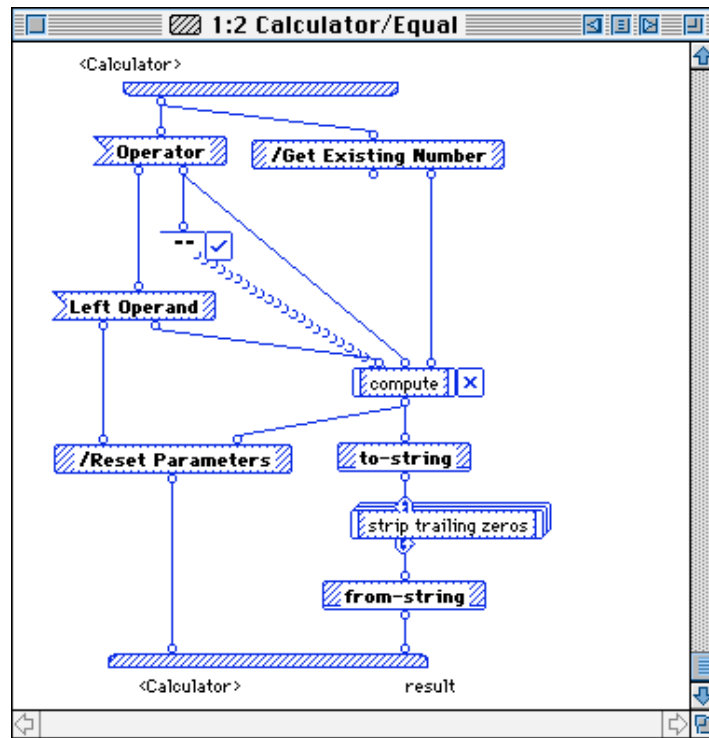
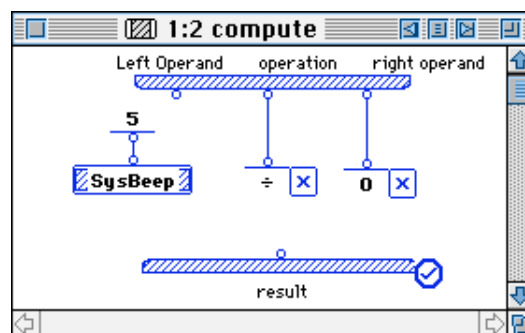
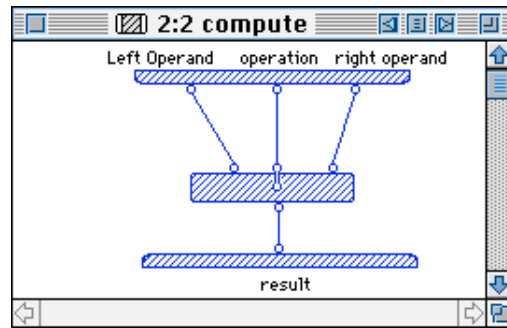
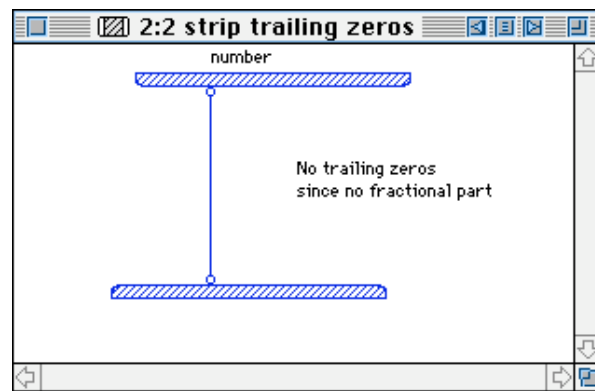
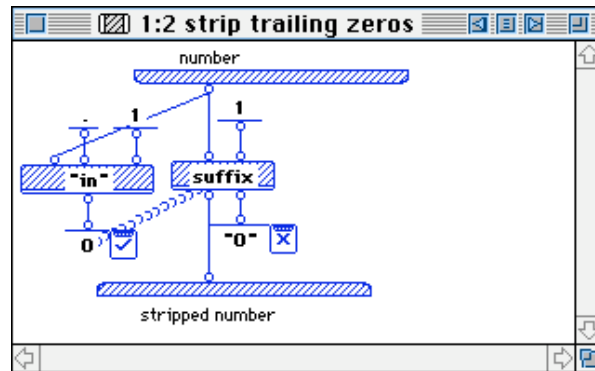


Figure 16.30: The Equal method of the Calculator class



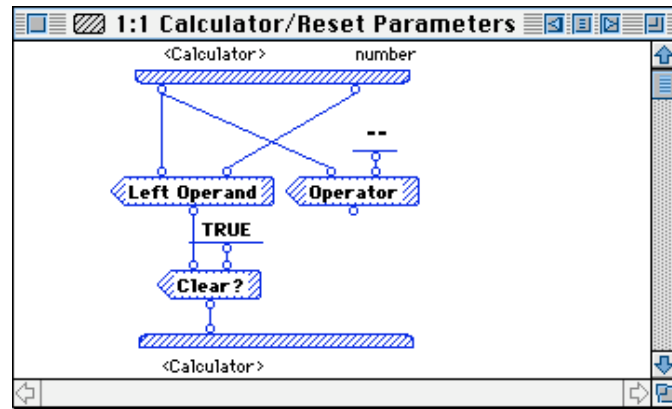


**Figure 16.31: The compute local method**



**Figure 16.32: The strip trailing zeros local method**

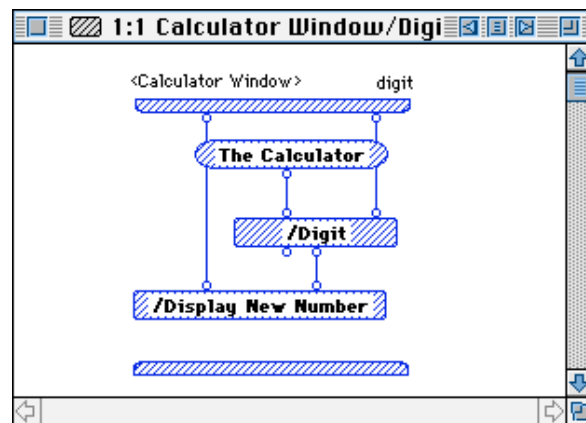
The last method of the Calculator class is the class method **Reset Parameters** (see Figure 16.33). Once again, it is identical to that of its predecessor except for its inputs.



**Figure 16.33: The Reset Parameters method of the Calculator class**

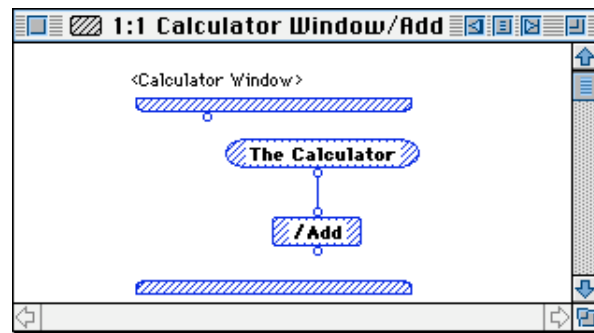
At this point, we can add the remainder of the new methods to the Calculator Window class. These new methods are called as click methods for the Push Buttons of the Calculator window. They act as coordinators of the actions of the calculator by requesting actions from the Calculator class, then displaying the results in the Calculator window. Due to the similarity of their actions, you'll find that these methods are all very similar in content.

The Digit method of the Calculator Window class (see Figure 16.34) simply requests that the Calculator class' Digit method add the pressed digit to the end of the current number, then it displays the new number in the Calculator window's text-editing box.



**Figure 16.34: The Digit method of the Calculator Window class**

The operator -handling methods (Add, Subtract, Multiply, Divide, Change Sign), as well as the Decimal, Equal and Clear methods, are all alike in their actions -- they request that the Calculator class' corresponding operator methods do their thing. Figure 16.35 shows the Add method of the Calculator Window class as an example of how the operator methods work.



**Figure 16.35: The Add method of the Calculator Window class**

Our new Calculator program is now complete. What have we gained by rewriting it? Let's say we wanted to reuse the code of the calculator program, or modify it, or add new calculator functions. In the previous version of the program, the workings of the calculator were intimately tied to how the calculator's Push Buttons worked. If you wanted to write another program that acted like a calculator, you'd have to add these Push Buttons, even if you didn't want to use Push Buttons to model the calculator. If you just wanted to add calculator-like functions to another program without also including the Calculator program's user interface, you'd be in trouble. Changing the inner workings of the calculator would also be difficult since you'd have to directly modify each individual Push Button subclass as well. Finally, if you wanted to add new functions to the calculator, it would require making new subclasses of Push Button, further increasing the ties between the calculator workings and the user interface.

In the new version of the Calculator, we've eliminated the dependencies between the workings of the calculator (in the Calculator class) and the representation of the calculator in the user interface (in the Calculator Window class). The tasks of the calculator are handled by the Calculator class alone, not user interface classes. This is desirable for three reasons: First, it makes it possible to experiment with how the calculator works if it's *encapsulated* into its own class. You can change the workings of its methods and not "break" a program which uses the class. Second, it enables you to *reuse* the calculator and add its function to another program by adding the Calculator class to that program. Third, it makes it easier to *extend* the usefulness of the calculator by *subclassing* the Calculator class and adding new functions to it (like trigonometric functions or financial functions) or changing how it works (like using scientific notation for numbers). Such flexibility is only possible with independent, uncoupled classes like we've constructed in this chapter.

## Summary

Although it's easy to add user interfaces to programs with the Application Builder Classes and Editors, the programmer has a fundamental choice to make when doing so. There are two possible ways to build the program:

- 1) Subclassing the user interface classes of the ABCs and adding class methods to them to perform the application-specific tasks.
- 2) Creating a separate application-specific class that performs these tasks and whose class methods are called as behaviors by the user interface elements.

Which approach should you use? This depends upon how important it is to reuse the code. If the program is a “one-shot” application -- used only for a single task that is unlikely to change in the future -- the first approach may make sense because it takes less time to get the program up and running. However, if code reuse is important for future programming projects, the flexibility of the second approach becomes more important. But don’t take our word for it -- try both approaches yourself and see which best suits your programming style.