16

Building a One-Button Calculator

This chapter describes how to build a simple calculator using many of the techniques introduced in the previous chapter. In addition, it shows how to define a custom object, Calculator, for the application and connect the interface to this object. The calculator's abilities will grow over the course of this project, but its first task will be to convert Celsius temperatures to Fahrenheit. As a temperature converter, the application's calculator window looks like this:

Figure 16-1. The Universal Calculator

The user enters a Celsius temperature in the left text field, then presses Return or clicks the Calculate button, and the Fahrenheit equivalent appears in the right field. What happens internally is that when the user signals that the input is complete, the Calculator object takes the input value from the left TextField object, performs the calculation, and then sends a message to the right TextField object to set its value to the result.

Creating the Interface

As you did in the first project, create a new project by choosing New from Project Builder's Project menu. Name this new project ªCalculatorº and save it in your home directory. When the project window appears, double-click **Calculator.nib** to open the interface file. (**Calculator.nib** is listed under Interfaces in Project Builder's Files display.)

Interface Builder becomes active and opens the template file for this new project. If the interface file for the previous project is still open you'll notice that Calculator's File window opens and overlaps the File window for Simple. By allowing multiple nib files to be open at the same time, Interface Builder makes it easy to copy and paste objects from one to the other.

Clicking a File windowÐor any application window that has an icon in that File windowÐmakes the nib file associated with the File window the current nib file. In Interface Builder, commands such as Save or Close operate on the current nib file.

Since Simple's nib file is finished, close it by making it current and then choosing the Close command from the Document menu.

Next, drag the interface objects shown in Figure 16-1 above from the Basic Views palette to the application's standard window. (You'll find it easier to align the different objects if you first turn on the grid.) You'll be using only two types of objectsÐButtons and TextFieldsÐalthough the TextFields will be configured as both titles and editable text fields:

Title text field

Editable text field

Since the input and output fields are nearly identical, it's fastest to configure one first, then duplicate it and modify the copy to create the other field. Drag an editable text field into the window and stretch it a bit horizontally. Delete the word ªTextº by double-clicking it and pressing the Delete key. Now drag in two title text fields and place one above and the other to the left of the editable text field. Edit the titles to match those in Figure 16-1 by double-clicking them in turn. You can also change their fonts and sizes using the Font panel, which is accessible through Interface Builder's Format menu.

Resize the window so that it resembles the window in Figure 16-1 above. Now open the Window Inspector by dragging to Attributes in the Inspector panel's pop-up list. Change the window's title to ªUniversal Calculator.º

With the exception of the Return icon in the Calculate button, the Universal Calculator window in your application should look identical to the one in the figure above. To add the icon, open the Images display (shown in Figure 16-2) by clicking the Images button in the File window.

Figure 16-2. The Images Display of the File Window

This display shows the images that are used throughout the Application Kit and lets you add new images by dragging them in from the File Viewer. Once an image is displayed in this window, you can drag it onto Button objects in your application.

Drag an NXreturnSign image from the File window to the Calculate button in the Universal Calculator. When the cursor intersects some part of the button, it changes to the link cursor, indicating that releasing the mouse button will assign the image to the button. Release the mouse button and notice that the button resizes to accommodate the title and the icon.

The Button Inspector now lists the name of the button's icon. You can alter the position of the icon in relation to the button's text by using the buttons that are grouped in the Icon Position cluster. Try several different placements if you like. If you place the icon above or below the title, the Calculate button grows so that both the icon and the title are visible. It doesn't shrink, however, if the extra area is no longer needed. In that case, you have to resize it by hand or use the Size to Fit command.

Defining the Calculator Class

The Calculator object is this application's control center. The Calculate button in the interface sends a message to the Calculator object to perform the calculation; in other words, the Calculator object is the target of the Button's action method. The Calculator object must then send messages to the two TextField objects to ascertain the input value and set the output value. We'll use the Classes display of the File window to design the Calculator class to handle these tasks. Click the Classes icon at the top of the File window.

Figure 16-3. The Classes Display

This display shows a hierarchy of the classes available to your application. With the exception of the First Responder entry, class names are displayed in gray, indicating that these classes can't be edited. (First Responder, as mentioned previously, is not a particular class, but the class of an object that has first-responder status in a window.)

Notice that the Inspector panel that you opened previously now displays the Class Inspector. With this inspector, you can examine (and edit, for classes you build) the outlets and action methods of the class.

Returning to the File window, select the Application class entry. (A quick way to locate a class is to type its name in the Find field and press Return.) Its superclass, Responder, is displayed as the title of one browser column.

With the Application class selected in the Classes window, the Class Inspector displays an Application object's outlet (**delegate**) and action methods (such as **hide:** and **terminate:**). Again, these entries are displayed in gray since they aren't editable.

Using the File window and the Class Inspector, you can define the class name, superclass, action methods, and outlet instance variables of a custom object. You start defining the new class by selecting where it will go in the class hierarchy. Since a Calculator object has very limited functionality and won't be displayed, we'll make the Calculator class a subclass of Object.

Scroll the browser in the File window to the extreme left so that the Object class appears. Click Object, making sure that only this class is selected. The class you define will become a subclass of Object. Now, drag to the Subclass button in the pull-down list. When you release the mouse button, a new class called ªMyObjectº appears in the right column. The class name also appears in the text field in the Class inspector. Edit this field to read ªCalculatorº and press Return. Notice that the File window now displays the name of the new class in its proper position in the class hierarchy. The name is in black since this class is editable.

Warning: Always check that the intended superclass is selected before you add a subclass. It's easy to inherit from the wrong class.

The next step in defining the Calculator class is to add two outlets corresponding to the TextField objects a Calculator object sends messages to. Make sure the Outlets button in the Class Inspector is highlighted and then enter ªinputFieldº in the text field below the button. Click the Add Outlet button; the new outlet appears in the Outlets list. Again, it's in black, indicating that you can rename or remove this outlet. In the same way add an outlet named ªoutputFieldº.

A Calculator object also needs to respond to an action message from the Calculate button in the application. Let's specify this new action method. Click the Actions radio button so that it is highlighted and then enter ªcalculate:º in the text field at the bottom of the panel. Click Add Action to add this method name to those displayed in the Actions list. (Since all action methods take one argument, the **id** of the sender, the method name must end with a colon. If you forget to add a colon, Interface Builder will add one for you.)

This completes the definition of the Calculator class. The Class Inspector should look like this:

Figure 16-4. Calculator Class Interface

Interface Builder can now create Objective-C interface and implementation files for the Calculator class. These files will only be templates; we'll fill them in shortly.

In the File window, drag to the Unparse button in the pull-down list. A panel opens asking if you want to create **Calculator.[hm]** (a shorthand for **Calculator.h** and **Calculator.m**). Confirm that you do, and the template files are written into the project directory. Another panel opens asking if you want to add **Calculator.[hm]** to the project. Again, confirm that you do. Project Builder's window comes forward and shows you that the Calculator class has been added to the project.

In this project, you defined a class and had Interface Builder write template files for it. Interface Builder's File window can also be used to import the class declaration from class files that already exist. Although we won't try this here, you'd simply drag the icon for the class's interface file from

the File Viewer into the File window. Interface Builder then parses the file and adds the name of the class in the appropriate position in the class hierarchy. If the new class conflicts with an existing one, Interface Builder gives you the choice of replacing the existing one or canceling the operation. If you want to make this new class part of the project, you must also drag the class files into the Files display of Project Builder's project window.

Warning: Once you've edited a template file, don't use Unparse again for that class unless you want to overwrite the edited file with a new template file. Interface Builder will warn you before carrying out such an operation.

Now that the Calculator class is defined, you can create an instance of this classÐa Calculator object. Verify that the Calculator class is selected in Interface Builder's File window and then drag to the Instantiate button in the pull-down list. When you release the mouse button, the File window switches to the Objects display, and a new object appears. This icon, titled ªCalculator,º represents your application's Calculator object. In the next section, you'll use this icon to make connections between interface objects and the Calculator object.

Connecting the Objects

After gathering the interface objects and creating a Calculator object, you need to interconnect them. To gain an understanding of how objects are interconnected in Interface Builder, let's first look at one of the predefined connections.

When a user chooses the Hide command, an application removes all but its application icon from the screen. The MenuCell titled ªHideº sends the message and the Application object's **hide:** method performs the operation. To see this connection, click the Hide command in Calculator's main menu. Drag to Connections in the Inspector panel's pop-up list to reveal the Connections display for a MenuCell:

Figure 16-5. The Connections Display

The left column shows the MenuCell's sole outlet, **target**. The right column lists the action messages that the target objectÐin this case an Application objectÐrecognizes. Notice that the entry **hide:** is highlighted and is marked with a small dimple. The dimple indicates that a connection using this action message has been previously established. The list titled ªConnectionsº near the bottom of the panel summarizes the connections for the inspected object.

To see a graphic depiction of the connection, click the entry in the Connections list. The connection is displayed in the workspace by a black line drawn between the MenuCell that sends the action message and the File's Owner. Figure 16-6 shows this connection.

Figure 16-6. Displaying a Connection

Warning: A single click in the Connections displays shows the connection; a double-click removes the connection. Be careful not to remove a connection that you only want to display.

Now that you've seen how connections are indicated, let's create some in the calculator application. First, let's connect the Celsius TextField to the Calculate button so that when the user presses Return after entering a Celsius value, the button will act as if it had been clicked. Control-drag from the Celsius TextField toward the Calculate button. You'll notice that a black line trails from the cursor.

When the cursor overlaps the Calculate button, a box appears around the button. Release the mouse button. The source and destination of the connection are now identified, and the TextField Inspector lists the TextField's outlets and the action messages that a Button object responds to. Select the **target** outlet in the first column and the **performClick:** action method in the second column. Finally, click the Connect button to establish the connection. The new connection is listed in the lower part of the Inspector panel.

When the user clicks the Calculate button (or it receives a **performClick:** message), the Calculator object should receive a **calculate:** message. To identify the source and destination of this connection, Control-drag a connecting line from the Calculate button to the Calculator icon in the File window. In the Button Inspector, establish that the Button's target receives a **calculate:** action message.

Next, you have to connect the Calculator object's outlets to the appropriate TextField objects. Control-drag a line from the Calculator icon in the File window toward the Celsius TextField object.

The CustomObject Inspector shows a Calculator object's two outlets, **inputField** and **outputField**. Select **inputField** and click Connect. Notice that a dimple appears next to the **inputField** listing in the Inspector panel, indicating that the connection has been made.

Following the same steps, connect the **outputField** outlet to the Fahrenheit TextField.

If you want to review the target/action connections within your application, select a Control object and then, in the Connections display, click the action message that's marked with a dimple. Connection lines will appear on the screen to identify the object that will receive this message. To review outlet assignments, select the object whose outlets you want to review and click the outlet names in the Connections display. Again, lines will appear on the screen for each connection that's been established.

Save the nib file by choosing the Save command from the Document menu. You can now test the interface by choosing the Test Interface command in the Document menu. The controls should operate correctly (for example, pressing Return after you enter a number in the Celsius field highlights the Calculate button), but of course no calculation takes place. For that, we have to define the Calculator class and then compile the application.

Writing the Calculator Class Definition Files

Interface Builder has given you template files for the Calculator class; now you can add the code that converts from one temperature scale to the other. To open the files, return to Project Builder and double-click **Calculator.h** and **Calculator.m**, which you'll find in the Files display under Header and Classes. The Edit application opens these files. The Calculator class template files and the alterations you need to make to them are described in the next sections.

Calculator.h

The interface to the Calculator class is defined in **Calculator.h**:

```
#import <appkit/appkit.h>

@interface Calculator:Object
{
    id  inputField;
    id  outputField;
}

- calculate:sender;
```

```
@end
```

Calculator is a subclass of Object. As you specified in the class editor, a Calculator object has two instance variables that can be used to store the **id**s of the calculator window's input and output TextFields. Also, as listed in the class editor, a Calculator object declares the **calculate:** action method.

Calculator.m

Calculator.m will contain the implementation of the Calculator class:

```
#import "Calculator.h"

@implementation Calculator

- calculate:sender
{
    return self;
}

@end
```

The **calculate:** method must send a message to the object referred to by its **inputField** variable to retrieve the Celsius value, calculate the Fahrenheit equivalent, and then send a message to the object referred to by its **outputField** variable to set the value it displays. One implementation of this method looks like this:

```
- calculate:sender
{
    float degreesF;

    [inputField selectText:self];
    degreesF = ((9.0 * [inputField floatValue])/ 5.0) + 32.0;
    [outputField setFloatValue:degreesF];
    return self;
}
```

The first message in this method implementation selects the text in the input field. We select the text so that the user can immediately enter a new value after finishing a previous calculation. The function of the next two lines should be self-evident. (These lines could be combined into one message, eliminating the **degreesF** variable, but are broken out into two lines for clarity.)

Edit the **Calculator.m** file to include this method implementation. Finally, save the file. You're now ready to compile and test the application.

Testing the Application

To compile and run the calculator application, click Run in Project Builder's project window.

If any errors are detected while the application is being built, they will be listed in the summary view of the Project window. Click an entry and Edit opens the file to the appropriate line, making it convenient to correct the problem. (You may want to introduce an error, just to see how this works!)

Once the application has been successfully compiled and linked, it begins to run. Test its features to verify that they all work properly.

Modifying the Calculator

So far, the Universal Calculator can handle any calculationÐas long as it's converting degrees Celsius to Fahrenheit. The rest of this chapter describes how to add to the calculator's functionality and, in passing, introduces several features concerning menus and submenus. The final sections of this project demonstrate how to add icons and sounds to an application.

Adding a Submenu

Since the calculator has only one button, extending its functionality beyond temperature conversion means redefining the meaning of the button. (Of course, you could add buttons, but that would be too easyÐand wouldn't require a submenu!) The modified calculator application will allow the user to select the type of calculationÐeither temperature conversion or square root calculationÐfrom a submenu. The titles of the input and output fields will change to reflect the type of calculation selected.

Click the menu button at the top of the Palettes window to display the menu palette. Drag the menu item titled ªSubmenuº from the Palettes window to the main menu of your application and release the mouse button. The menu item inserts itself within the list of other menu items, and the menu resizes to accommodate the width of the new item. You can reposition a menu item by dragging it vertically within the menu. A submenu containing one menu item appears to the side of the main menu.

MenuCell selection is indicated by highlighting: black text on a white background. Selected MenuCells can be cut, copied, and pasted within a menu or between menus using the standard editing commands.

You can edit the text a MenuCell displays by double-clicking it. Similarly, you can edit the keyboard equivalent for the item by double-clicking the right part of the MenuCell. A square appears indicating that a keyboard equivalent can be added or edited.

Edit the text in the new main menu item so that it reads ªCalculationsº and press Return. The main menu resizes to accommodate the menu item's text, and the submenu's title changes to match the text. Now add another item to the submenu by dragging the MenuCell titled ªItemº from the Palettes window to your application's submenu.

Finally, edit the text of the submenu's two items to read ªTemperatureº and ªSquare Rootº. The finished menus should look like those shown in Figure 16-7.

Figure 16-7. The Menu and Submenu

Now, select the Calculator class in the Classes display of the File window. To edit the class definition, open the Class Inspector (by choosing the Inspector command from Interface Builder's Tools menu). The revised Calculator object must respond to action messages from the new submenu, so let's add **convertToTemp:** and **convertToSqRoot:** methods. It will also need to send messages to the TextFields that titles the input and output fields, so let's add **inputTitle** and **outputTitle** outlets. Figure 16-8 shows how the Class Inspector should look after you make these changes to the Calculator class interface.

Figure 16-8. Revising the Calculator Class Description

Next, establish the connections from the submenu items to the Calculator object. While holding down Control, drag the cursor from the Temperature submenu item to the Calculator object in the Objects display of the File window. Double-click the **convertToTemp:** entry in the Inspector panel to establish the connection. Likewise, specify that the Square Root submenu item sends a **convertToSqRoot:** message to the Calculator object.

Now, connect the Calculator's **inputTitle** and **outputTitle** outlets to the proper TextFields in the Calculator window. (The Calculator object will send messages to these objects to change their text from ªCelsiusº and ªFahrenheitº to ªxº and ªsqrt(x)º, as the user picks one or the other type of calculation.) Control-drag from the Calculator object in the File window to the TextField that reads ªCelsiusº. Double-click **inputTitle** in the Connections inspector to establish the connection. Follow the same process to connect the **outputTitle** outlet to the TextField currently titled ªFahrenheitº.

Finally, use the TextField inspector's alignment buttons to specify that the text in these fields is right aligned. In this way, although a title's text may change from ªCelsiusº to ªxº, it will stay visually associated with the input field it labels.

The revised interface is complete; the only changes that remain affect the Calculator class files. The next two sections describe the changes you need to make.

Modifying Calculator.h

The new calculator is designed either to convert temperatures or to calculate square roots; in other words, the calculator has two states. One way to keep track of the current state of the calculator is to add an instance variable that can have either of two values. We'll add the integer variable **calcType** for this purpose. For convenience, let's also define the constants TEMP and SQROOT to correspond to the two states. The **inputTitle** and **outputTitle** instance variables also need to be declared. These changes add eight lines to the **Calculator.h** file. The lines you need to add are shown in bold:

```
#import <appkit/appkit.h>

#define TEMP 1
#define SQROOT 2

@interface Calculator : Object
{
    id inputField;
    id outputField;
    id inputTitle;
    id outputTitle;
    int calcType;
}

- init;
- calculate:sender;
- convertToTemp:sender;
- convertToSqRoot:sender;

@end
```

Modifying Calculator.m

The implementation file must be modified in three ways. It needs an initialization method to establish the value of the **calcType** instance variable (and thus the calculator's initial state). The **init** method below handles this initialization. When the calculator first appears, it will be configured to perform temperature conversions. It also must be modified so that the **calculate:** method performs the proper calculation according to the calculator's current state. Finally, it needs to implement the **convertToTemp:** and **convertToSqRoot:** action methods. These methods set the value of **calcType** and change the titles of the input and output fields.

Make these changes to the **Calculator.m** file. As before, each line you need to add or alter is shown in bold.

```
#import "Calculator.h"

@implementation Calculator

- init
{
    [super init];
    calcType = TEMP;
    return self;
}

- calculate:sender
{
        [inputField selectText:self];
        if (calcType == TEMP) {
            float degreesF;
            degreesF = ((9.0 * [inputField floatValue])/5.0) + 32.0;
            [outputField setFloatValue:degreesF];
        } else if (calcType == SQROOT) {
            double sqRoot;
            sqRoot = sqrt([inputField doubleValue]);
            [outputField setDoubleValue:sqRoot];
        }
        return self;
}

- convertToTemp:sender
{
    calcType = TEMP;
    [inputTitle setStringValue:"Celsius:"];
    [outputTitle setStringValue:"Fahrenheit:"];
    [outputField setStringValue:""];
    [inputField selectText:self];
    return self;
}

- convertToSqRoot:sender
{
    calcType = SQROOT;
    [inputTitle setStringValue:"x:"];
    [outputTitle setStringValue:"sqrt(x):"];
    [outputField setStringValue:""];
    [inputField selectText:self];
    return self;
}

@end
```

After you edit and save these files, compile the application. Watch for error messages from the compiler. In most cases, they will signal typographical errors in the source code. Make the necessary corrections and recompile the application. Finally, run the application and test its new features.

Note: If the application fails at run time, the problem is probably caused by an inconsistency

between the method and instance variable names you declared in the Class inspector and those in the Calculator class definition files. Use Interface Builder to check the method and variable names in the Class Inspector panel against those in **Calculator.h** and **Calculator.m**.

Adding an Icon

With the Images display of the File window, you can access existing system images, as illustrated earlier in this project, or you can create images from data in either TIFF (Tag Image File Format) or EPS (Encapsulated PostScript) file format. Once you import the image, it can be assigned to Button objects in your application. Figure 16-9 shows some examples of buttons that display icons.

Figure 16-9. Icons and Buttons

To see how this works, click the Images suitcase in the File window to display a variety of icons used in the Application Kit. The titles under the icons are displayed in gray to indicate that these icons can't be deleted nor can their names be edited. However, you can copy and paste any icon that appears in this window.

Let's add an image to this window. Using the File Viewer, switch to **/NextLibrary/Documentation/NextDev/Examples/IBTutorial/Images**. You'll notice that this directory contains the TIFF file **willy.tiff**. Drag the file icon from the File Viewer to Interface Builder's File window. When you release the mouse button, Interface Builder displays a panel asking if you want to add **willy.tiff** to the project. Click Yes, and Project Builder's window comes forward to show you that the file has been added under Images in the Files browser.

(In general, it's best to add TIFF or EPS format files to a project rather than use them to create local images. By adding the image file to the project you make one copy of the image data available to all nib files in the project. If, on the other hand, you ask Interface Builder to create an image with the data, the image data is copied from the image file into the nib file. Thus, each nib file that requires the image would have to have a separate copy of the data.)

If the image that you add to the File window is no larger than 48 by 48 pixels, the Images display shows the actual image. Larger images (as in this case) are displayed by Interface Builder's Image Inspector.

The Image Inspector has two uses: It gives you the dimensions of the image in pixels, and it lets you see the actual icon image even for icons larger than 48 by 48 pixels. Figure 16-10 shows a detail of the Image Inspector.

Figure 16-10. The Image Inspector

To place the image on a button in your application, simply drag the icon from the File window to a Button object in your application's window. (The cursor must be over the button when you release the mouse button; otherwise, the image isn't transferred.)

Adding Sound

To manipulate the sounds in your application, Interface Builder provides two tools, the Sounds display of the File window and the Sound Inspector. The Sounds display is the repository for your application's sound resources. By dragging a sound icon from the Sounds display onto a Button object in your application, you can associate a sound with that object. The Sound Inspector lets you play sounds from sound files on disk and lets you record your own sounds. It also gives you a graphic display of the sound and allows you basic editing capability.

Open the Sounds display by clicking the Sounds suitcase in the File window. Each of the icons in the Sounds window represents a sound. The gray titles indicate that these sounds can't be edited since they are system sounds. You select a sound by clicking its icon. A selected sound can be copied, pasted, and (except for system sounds) deleted. In fact, it's common to create a new sound for editing by copying an existing sound.

Make a copy of the Basso sound in the Sound window. The new sound icon is labeled ªSound.º Now, open the Sound Inspector by double-clicking the new sound's icon.

The Sound Inspector shows a graphic representation of the selected sound's waveform. The graph plots the change of the sound's amplitude over time. You can play the entire sound by clicking the Play button, or you can select and play only a portion of the displayed sound. For a demonstration, drag horizontally across a portion of the graph and click Play. Notice that the sound meter below the waveform shows the instantaneous and peak volumes for the sound that's played.

Using your computer's microphone, you can replace the selection in the Sound Inspector with sound you record. Click the Record button to start recording. When you're through recording, click Stop to end the recording session and display the waveform. Clicking Pause halts the recording until the next time Pause is clicked.

You can add sounds to the Sounds window by dragging the sounds file icon from the File Viewer to the File window. Using the File Viewer, switch to the **/NextLibrary/Documentation/NextDev/Examples/IBTutorial/Sounds** directory. Within this directory there are three sound files: **drum1.snd**, **drum2.snd**, and **drum3.snd**. Drag **drum1.snd** into the Sounds window. A panel appears and asks if you want to insert the sound into the project. Click Create Local Sound, and the sound is inserted into the nib file.

Note: Inserting a sound into the nib file makes it directly editable within Interface Builder, as you'll see shortly. Often, however, it's better to insert the sound file into the project (rather than into the nib file), so that a single sound file can be accessed from multiple program modules.

The graph in the Sound Inspector shows the sound's waveform. Click Play to hear the sound.

Now, let's create a sound for the Calculate button in the Calculator application. Select a portion of the drum sound. For example, you might find that the decay portion of one of the louder drum beats, as shown in Figure 16-11, makes a satisfying button-click sound.

Figure 16-11. The Sound Inspector

Once you've found a portion of the waveform that you want for the Calculate button, select and then delete the portions that precede and follow it. Click OK to save the modified sound.

Let's associate the sound with the Calculate button. Drag the sound icon from the Sounds window to the Calculate button and release the mouse button. The button becomes selected to confirm that the sound has been assigned to the button. If you look at the button's attributes in the Button Inspector, you'll see that **drum1** is listed. By deleting this name, you can remove the association of the sound with the button. You can check the operation of the button by putting Interface Builder in test mode and then clicking Calculate.

This ends the Universal Calculator project. Save the project and then compile and run the

application to test its operation. You might try adding other features to the calculator to test your understanding of the concepts introduced so far.