Chapter 2

Objects and Graphical Interfaces

CHAPTER 2 OBJECTS AND GRAPHICAL INTERFACES

In learning to develop graphical user interfaces, one of the most difficult things that programmers have to get used to is not having control. Most programmers are used to writing a program like it's a recipe: Do this, then that, and finally do this. When you put a graphical user interface on a program, the user is in control. The user decides what happens and when.

Goal

To explore the way user interfaces are built using the Application Kit and InterfaceBuilder.

Prerequisites

Familiarity with at least one graphical user interface and a basic understanding of objects and instance variables.

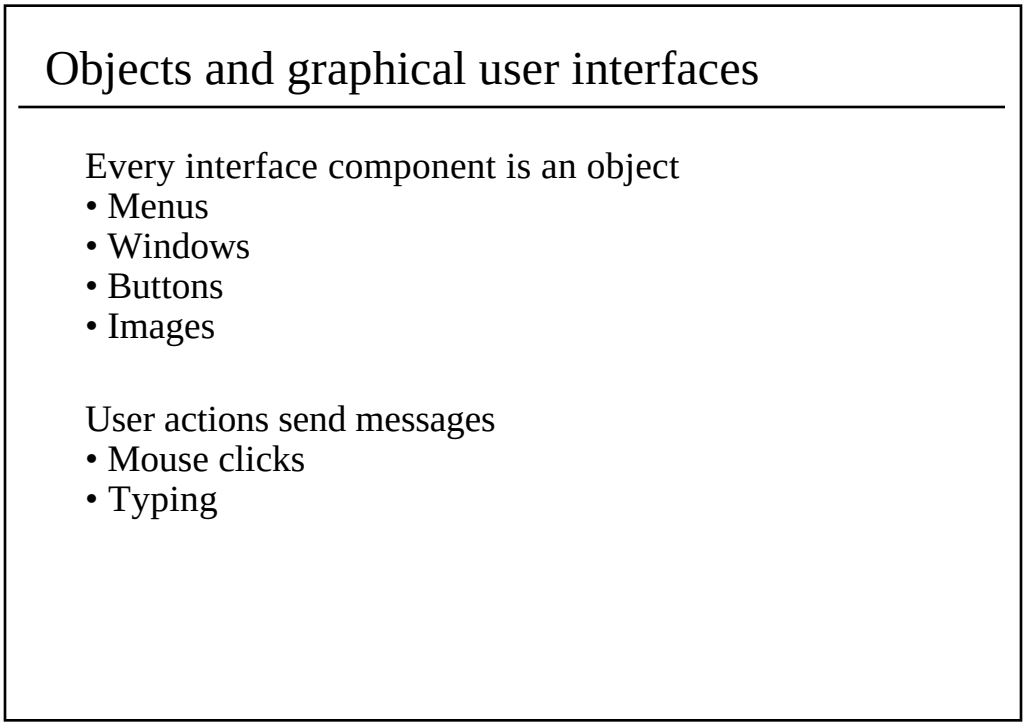
Objectives

After completing this chapter, you'll be able to:

- Describe the life of an application
- Define target and action
- Explain the purpose of InterfaceBuilder and the nib file
- Program in an event-driven environment

Reading

You can find more information about creating applications with graphical user interfaces in
/System/Documentation/Developer/TasksAndConcepts/DevEnvGuide/Chapters/(Composing the Interface)

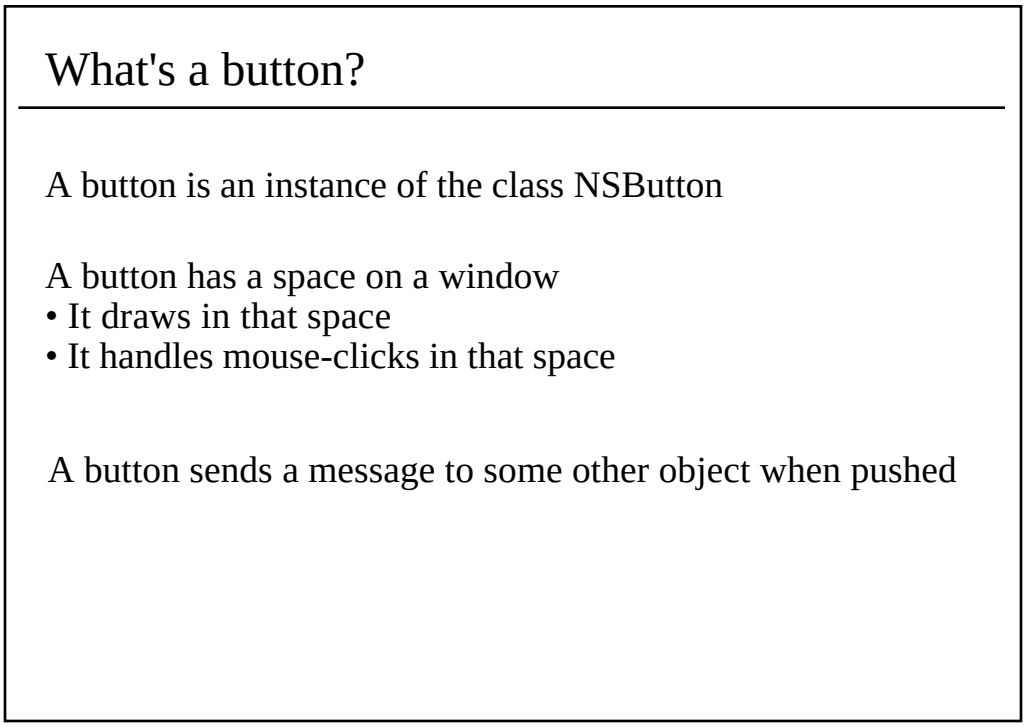


Objects and graphical user interfaces

Graphical user interfaces are event-driven—the user of the application is in control. When the user pushes a button, something happens. When the user chooses a command from a menu, something happens.

This type of event-driven program maps very well to an object-oriented system. All user interface elements are objects. Windows, buttons, menus, and images are all objects. These objects perform actions in response to messages. For example, a text field might receive a message telling it to display its contents. In response, the text field prints out its contents on the screen.

Because everything in a user interface is an object, it's easy to understand how user events are handled. User actions send messages to specific objects. For example, when a user pushes a button, that action sends a message to some other object. The receiving object could be any object in the application—another user interface object, or a custom controller object you write. The receiving object then does something in response to the user's action.

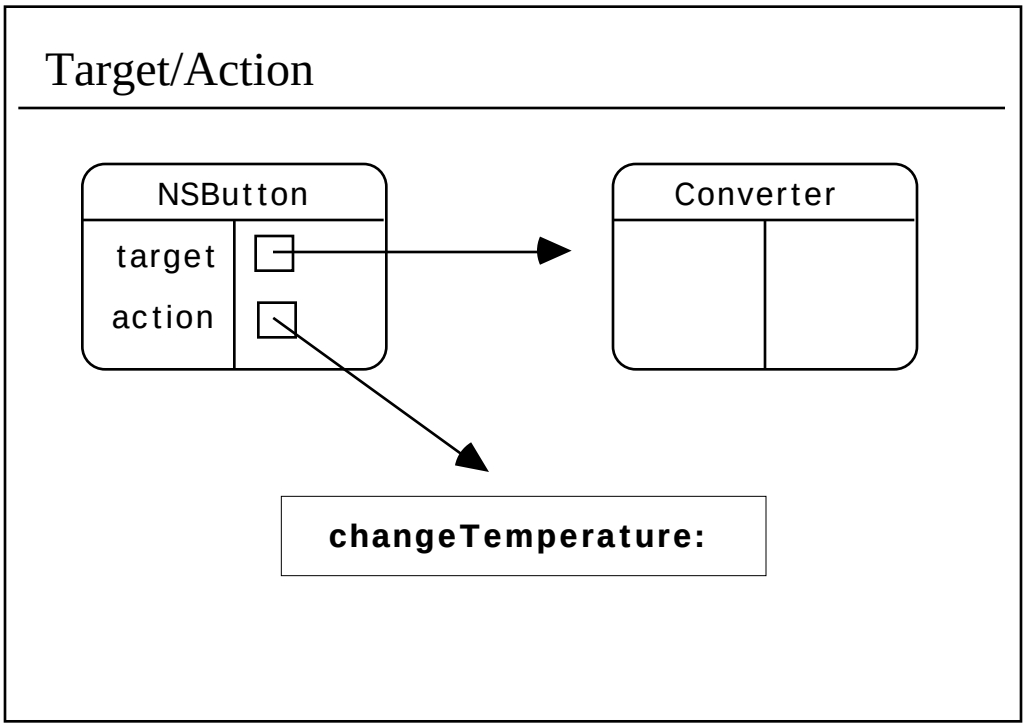


What's a button?

This chapter talks about building user interfaces using the example of one user interface element—the button. A button is one of the simplest user interface elements. It's simply a location on the screen where mouse clicks are treated in a special way. Specifically, when the user pushes a button, the program takes some action.

Buttons are instances of the class NSButton. The framework provides this class so you don't have to worry about how to detect mouse clicks in a certain region of the screen, how to dispatch messages to other objects, or any of the other details about how a button is supposed to operate. Using NSButton you get the functionality of a button without having to write a line of code. In the demonstration and exercise for this chapter, you'll see exactly how this works. You'll also find that many other user interface objects behave a lot like a button—they send a message to some other object when you activate them.

The only thing a button actually does is display itself on the screen and send messages. For example, in the Mail application the Delete button itself doesn't delete anything when you push it. Instead it sends a message to some other object. That other object is responsible for carrying out the deletion.



Target/Action

To implement its message sending behavior, NSButton has two instance variables—**target** and **action**. **target** is a pointer to the button's target. When a user pushes the button, the button sends its message to the target. **action** is the name of the message to send to the target.

This setup allows an instance of NSButton to send a custom message to any type of object. Instead of having a preset method like "**buttonPushed:**" or something equally obtuse, you can use descriptive method names like "**changeTemperature:**".

Once a button has a set **target** and **action**, it simply sits and waits for the user to push it. When the user pushes the button, it sends the message specified in its **action** instance variable to the object specified in its **target** instance variable.

Details: targets and actions

Buttons can have any type of object as their **target**. That is, an instance of any class is a valid target of a button. Variables of type id can refer to objects of any class. Therefore, NSButton's instance variable target is declared to be of type id. Instance variables of type id are often referred to as "outlets".

An action is a special type of method. An action method generally has a name indicating what the method does, and all action methods take a single argument, called **sender**. The **sender** argument is simply the object invoking the action method. So when someone pushes a button, the button sends its action message including itself as the **sender** argument.

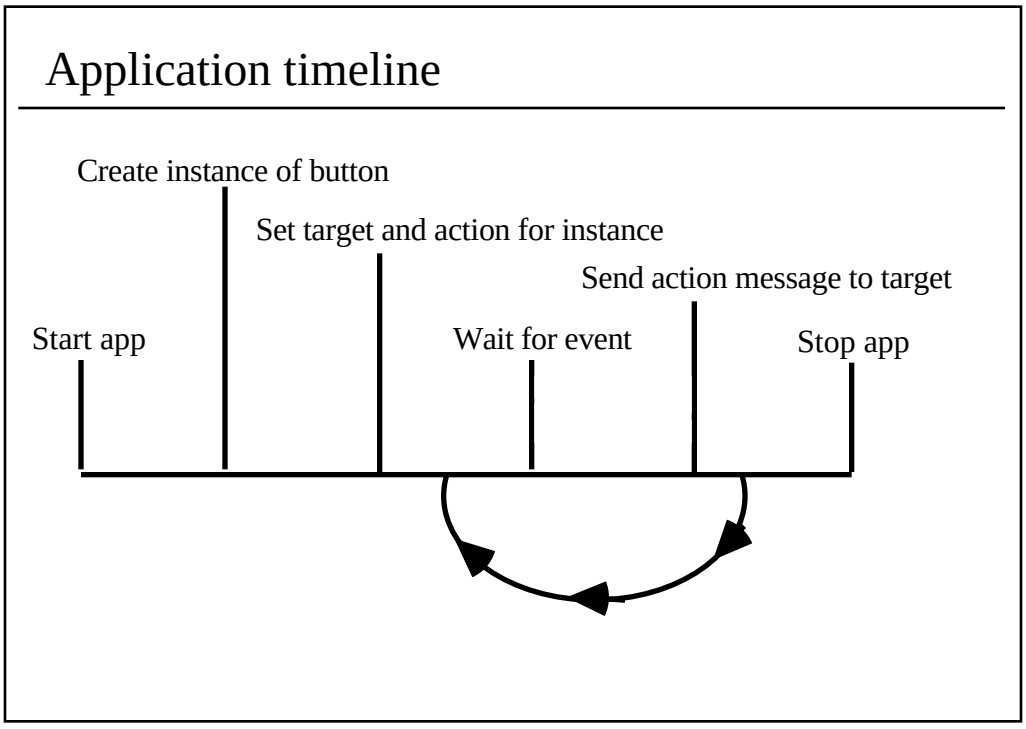
To allow programmers to set which action message to send, NSButton needs a way to store the name of a method. The **action** instance variable stores the name of a method using something called a selector. For every method there is a corresponding selector, used by the Objective-C runtime system to select a method. Selectors can be stored in variables of type SEL. If you know the name of a method, you can get the corresponding selector using the @selector() compiler directive. For example, the following code sets the **methodSelector** variable to the selector for the **raisePrices:** action method.

```
SEL methodSelector;

methodSelector = @selector(raisePrices:);
```

You can set both the **target** and **action** of an NSButton using its **setTarget:** and **setAction:** methods.

NSButton is not the only class with a **target** and **action**. Sliders, text fields, and other types of user interface objects users can activate all have a **target** and **action**. When the user activates the user interface object, it sends its **action** message to the **target**.



Application timeline

This time line depicts the life of an application. An object-oriented application with a graphical user interface has two main tasks when it starts up. First, it must create a network of objects. User interface elements need to know how to communicate with each other and other objects in a complex object network. This task involves creating objects, setting instance variables, and generally doing initialization work. For a graphical application, much of this work has to do with positioning user interface elements in a window, setting their titles, and otherwise making the user interface look right.

The second task is to handle user events. When the user pushes the Delete button, something had better get deleted or the user is going to be confused. This is the interesting part of an application, and this is where developers add the logic that makes their application unique.

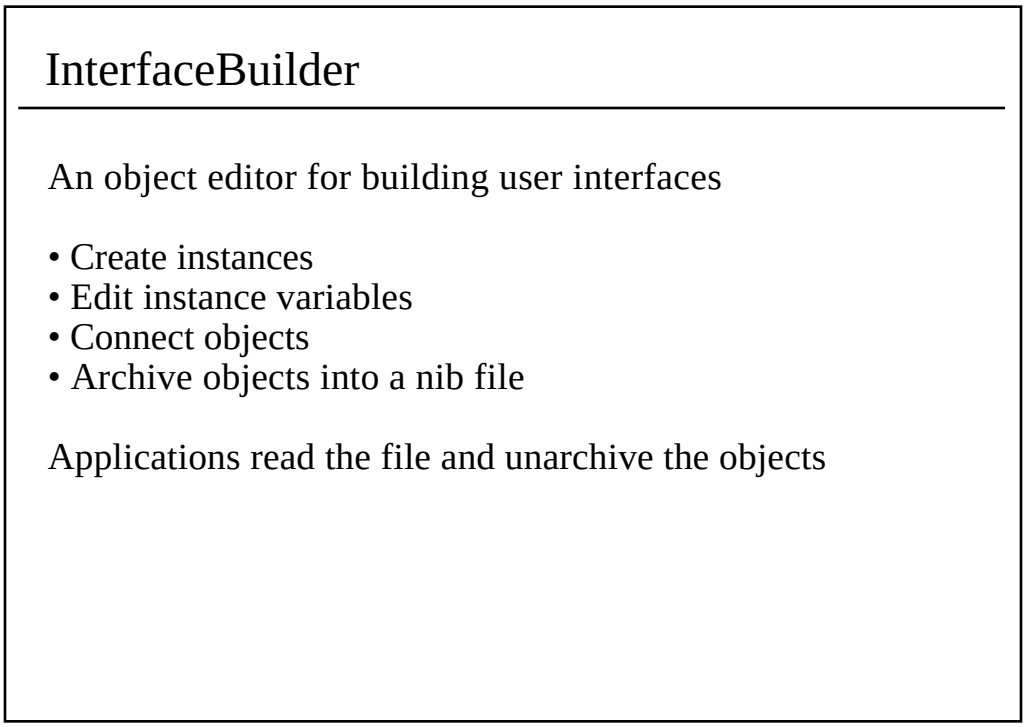
Unfortunately, in many environments the first task, setting up the user interface, takes a majority of the effort. Developers have to write source code to create and initialize all the buttons, windows, and other objects they want to appear in their user interface.

“Where does my code get called?”

The time line raises a very important question. If the user is in charge of an event-driven application with a graphical user interface, where does your code get called?

The answer is that user events initiate messages that call your code. You write a method you want called when the user pushes a button. Say you created a class SomeClass with a method **someMethod:**. You need to set up a network of objects such that a **someMethod:** message is sent whenever the user pushes a certain button. To do this, you create an instance of SomeClass and make it the target of the button. Then you set the action of the button to **someMethod:**.

Now whenever someone pushes the button, it sends a **someMethod:** message to your object.



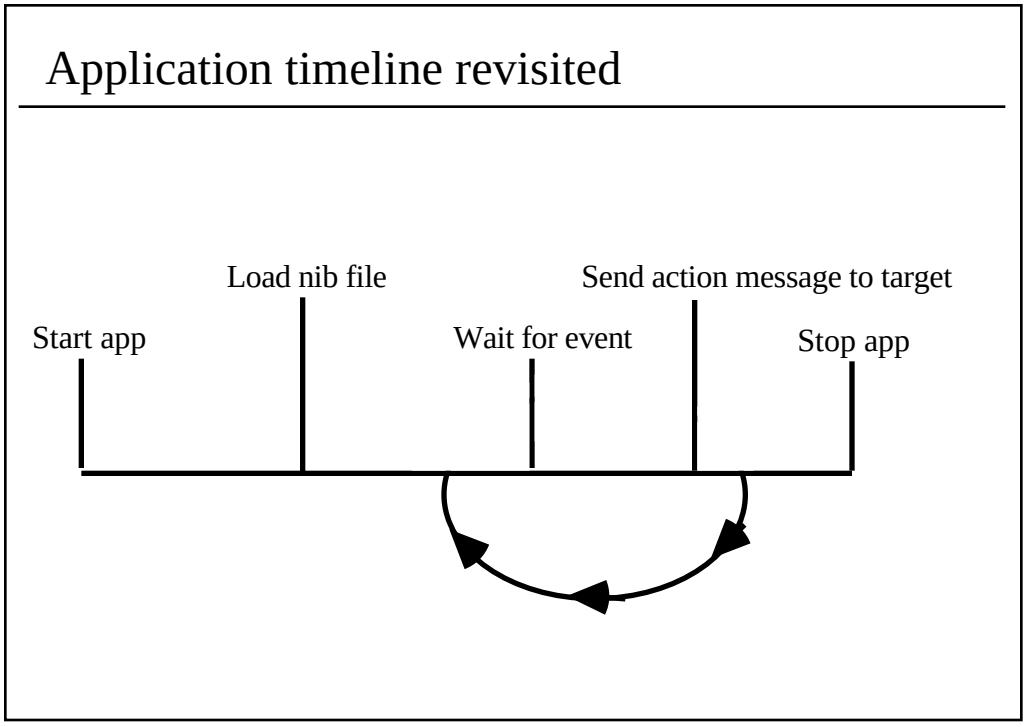
InterfaceBuilder

Setting up the network of objects is a time-consuming task, especially if you had to do it entirely in source code. To make things easier, we have InterfaceBuilder.

InterfaceBuilder is an application that creates and archives a network of interface objects. Using InterfaceBuilder, you can create a button and an instance of your class, then connect them using the button's target instance variable. InterfaceBuilder allows you to set the button's action, resize it, and position it on the screen. InterfaceBuilder saves the important data from the network of objects into a file, known as a nib file.

When an application starts up, it can read in these "freeze-dried" objects and recreate the network of objects. This allows you to create an object network using a graphical, user-friendly tool, instead of having to write source code.

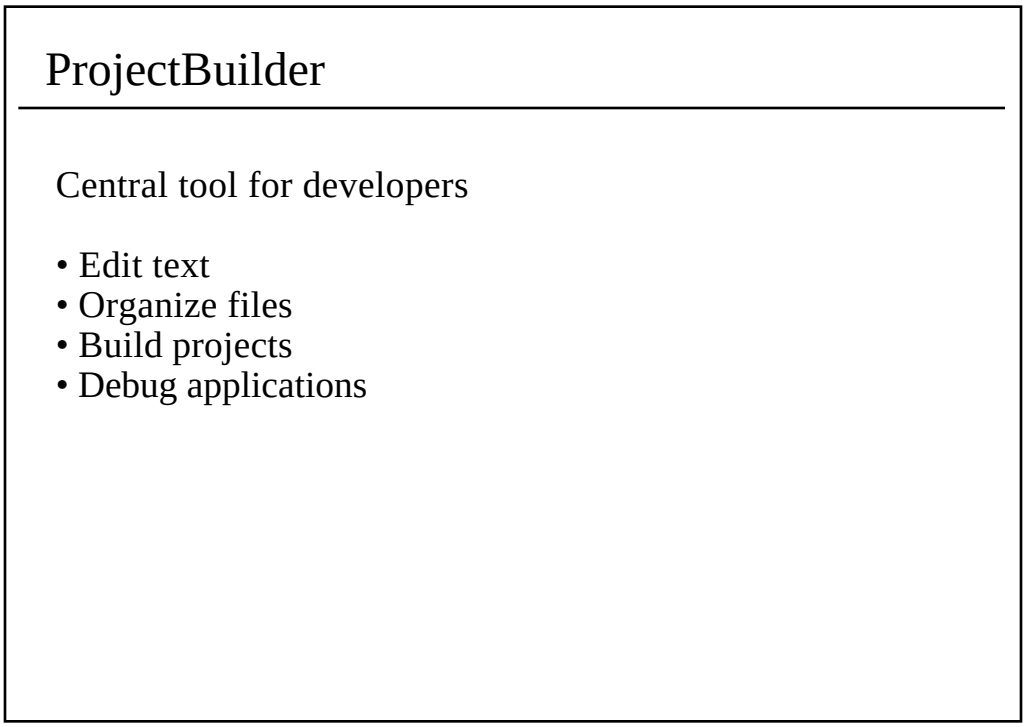
InterfaceBuilder can be found in **/System/Developer/Apps/**



Application timeline revisited

A nib file simplifies the life of an application with a graphical user interface. Instead of programmatically creating the objects of the user interface at run-time, the application simply loads the nib file.

Putting the user interface layout in a separate file makes a lot of sense. The interface layout is conceptually separate from the logic of how the application works. If you want to modify the user interface, you modify the nib file. If you want to change how the application works, you modify the source code.



ProjectBuilder

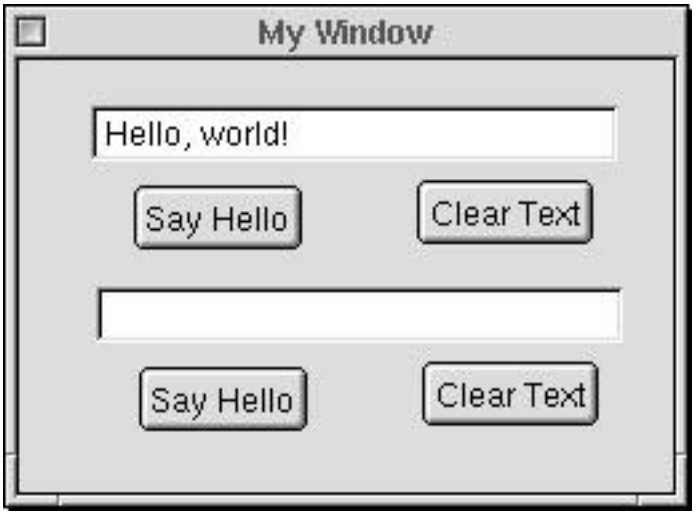
A project contains many different kinds of resources, generally represented by different files. A typical application project contains these types of resources:

- Source code
- nib files
- Images
- Sounds
- Documentation
- Supporting files

ProjectBuilder keeps track of these files and builds them into an application. Many of the tasks of building an application can be done within ProjectBuilder. It serves as an editor for source code files, allows you to build projects, and provides a graphical front-end to a debugger for debugging projects.

ProjectBuilder can be found in **/System/Developer/Apps/**

DEMONSTRATION 2.1 OBJECTS AND INTERFACEBUILDER



A big part of developing an application is coming up with a user interface. However there is a tool that makes this easier—InterfaceBuilder. InterfaceBuilder is more than a screen painter. It lets you create real objects and connect them together, allowing you to build a good deal of functionality into an application without having to write a single line of code.

This demonstration shows off some features of InterfaceBuilder. Led by the instructor, you build an application with a simple user interface incorporating some objects that do not have a direct screen representation. These "controller" objects, of the class HelloController, are provided on a palette. A palette is a collection of objects that can be manipulated in InterfaceBuilder.

Your main task is to connect user interface elements to instances of the HelloController class. When the demonstration is completed, a user of the application will be able to print "Hello, world!" in a text field by clicking a button, and clear the text field with another button.

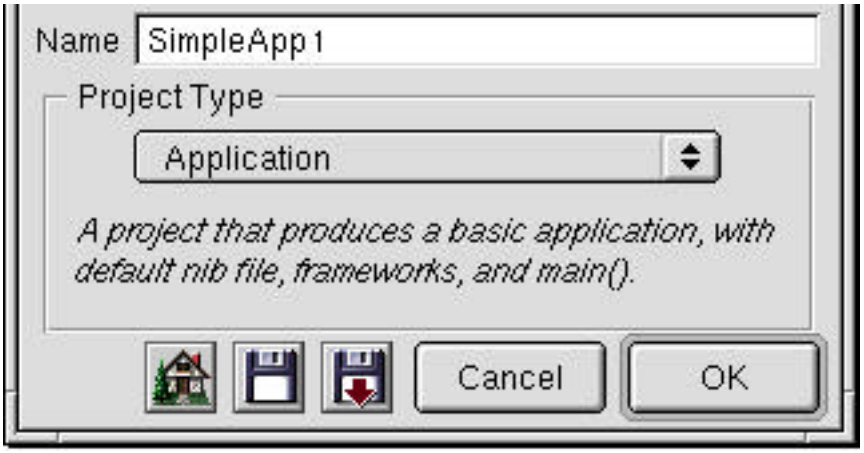
Objectives

After completing this demonstration, you'll be able to:

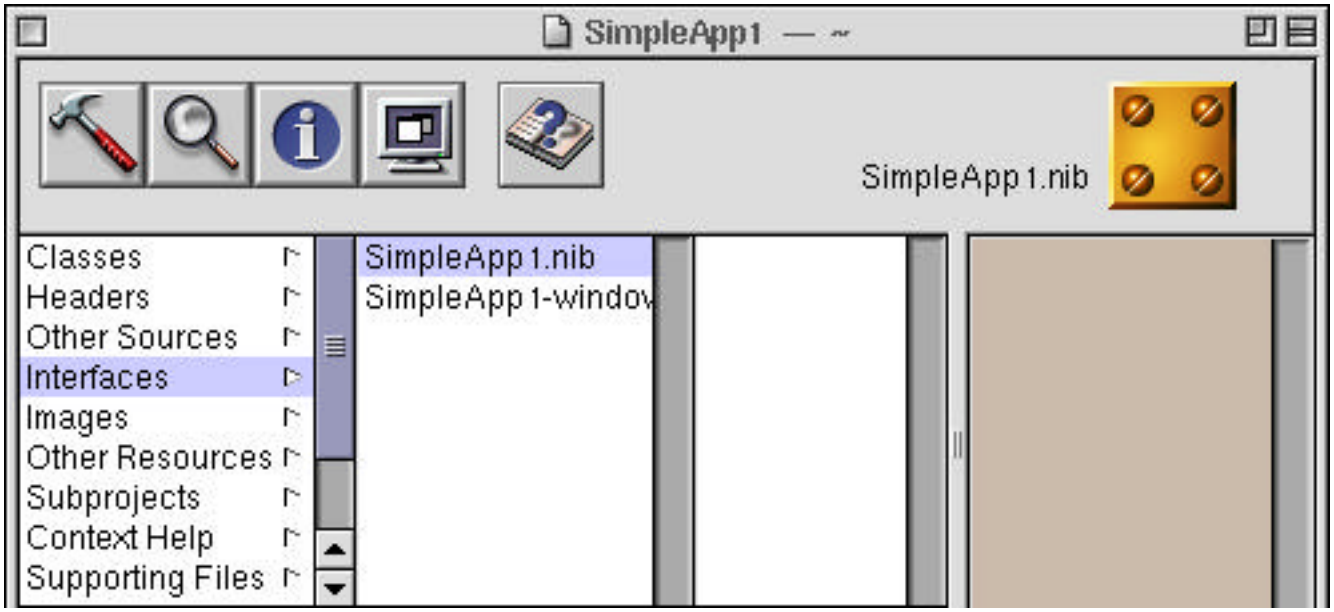
- Create a new application using ProjectBuilder
- Modify a nib file using InterfaceBuilder
- Set targets and actions for Button objects
- Test an interface in InterfaceBuilder
- Build an application
- Start up an application from ProjectBuilder

Demonstration

1. Create a new project. Start up ProjectBuilder.app -- located in /System/Developer/Apps/ -- and select the New command in the Project menu. Name the project SimpleApp1 and select Application in the Project Type pop-up list.



2. Open the user interface file (called a nib file). Select Interfaces in the first column of the project window browser. Double-click SimpleApp1.nib to open the interface.

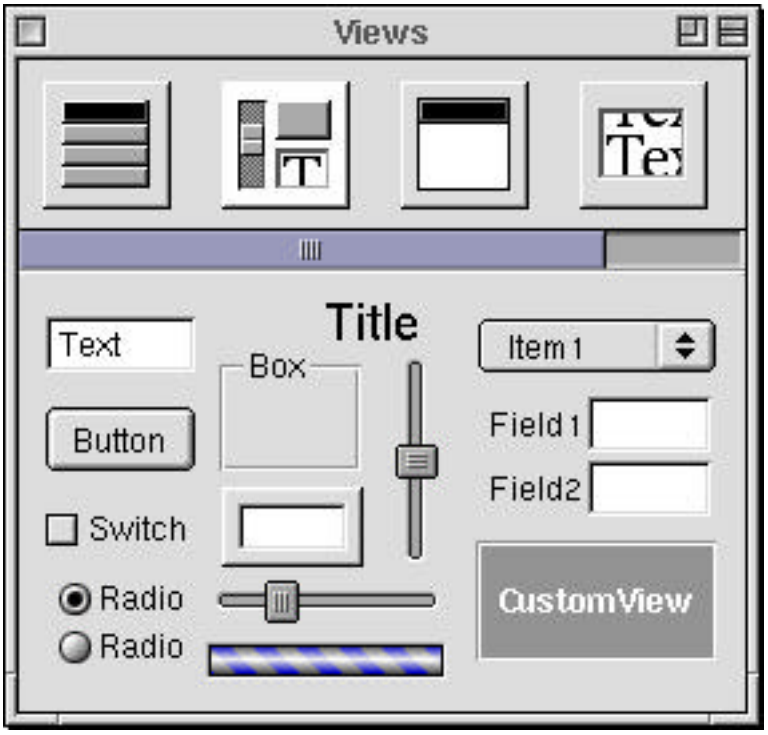


InterfaceBuilder edits nib files. ProjectBuilder launches InterfaceBuilder when you double-click SimpleApp1.nib, and tells InterfaceBuilder to open SimpleApp1.nib. Finally, ProjectBuilder gives control to InterfaceBuilder so you can edit your user interface.

3. Resize the window titled My Window. This is the main window of your interface. Use the window's resize bar to make its dimensions match the sample interface. When you make these changes in InterfaceBuilder, you change the instance variables of a real live NSWindow object.

4. Add a text field to the main window. Apple provides objects pre-loaded into InterfaceBuilder on palettes. You choose a palette in the palettes window, found in the upper right-hand corner of the workspace when you start up InterfaceBuilder. Choose the palette you want by pushing its button in the top portion of the window. The menu bar of the palette window changes to the name of the selected palette, and the bottom portion displays the available objects.

   Choose the Views palette icon. Drag a text field onto the main window.



5. Position and resize the text field. While users can normally resize and reposition windows, they can't resize or reposition other user interface elements. When creating a user interface you need to be able to do this. InterfaceBuilder allows you to do this in the same way a drawing application would.

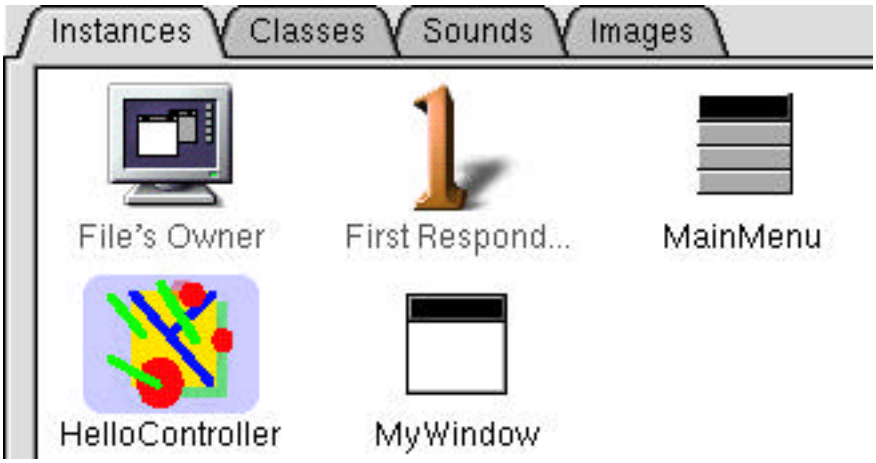
   - Click once on the text field to select it. Small gray squares called control points appear at the corners and midpoints of the text field.
   - Use the control points to resize the text field so it matches the sample interface.

6. Add two buttons to the user interface.
   - Drag a couple of buttons from the Views palette onto your main window
   - Rename the buttons to read Say Hello and Clear Text. You can edit the title of a button by double-clicking on it and typing the new text.

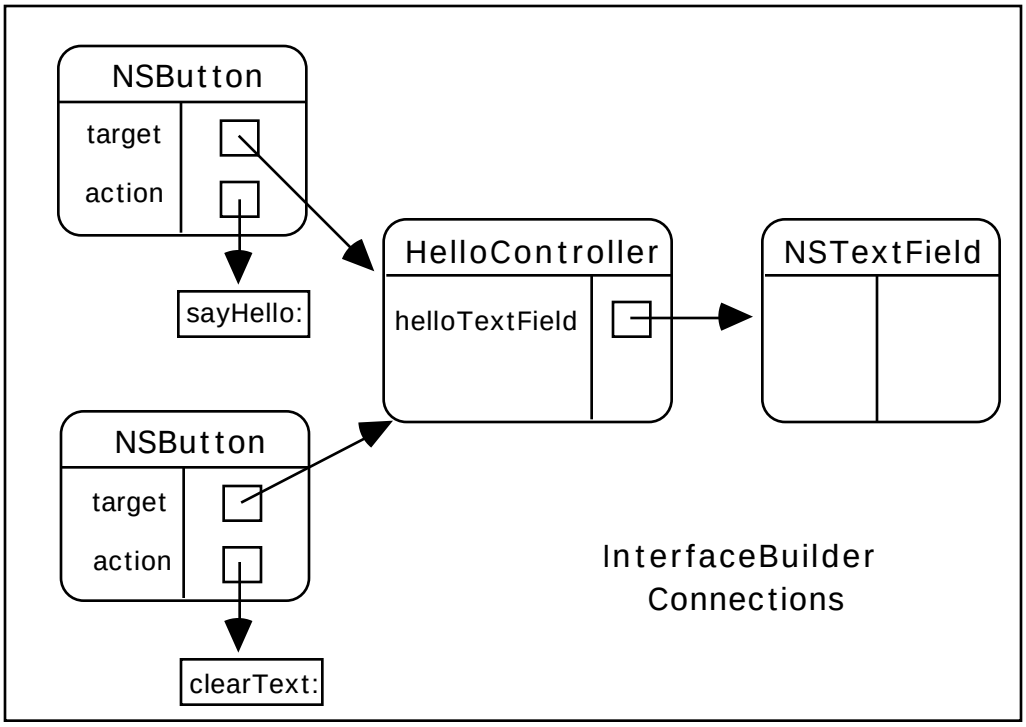
7. Copy the HelloController.framework from the /ExerciseMaterials/Frameworks/HelloController.framework.to your <AccountName>/Library/Frameworks folder (you may have to create this folder yourself). When asked how the links should be copied, check the "Repeat" checkbox and then click the "New Link" button.

8. Add the HelloController palette to InterfaceBuilder. Use the Tools menu in InterfaceBuilder and select Palettes->Open..., then double-click on /ExerciseMaterials/Palettes/HelloControllerPalette.palette.

9. Add an instance of HelloController to your nib file by dragging a HelloController object from the palette to the instances window in the lower left corner of the workspace. Because HelloController objects are not user interface elements, they don't go in your main window. InterfaceBuilder provides an instances window that keeps track of all the objects in your nib file.



10. The next step is to connect the two buttons and text field to one instance of HelloController. Before getting started on the details of setting up the connections, take a moment to study this diagram of how the two buttons, the text field, and the HelloController object are connected:

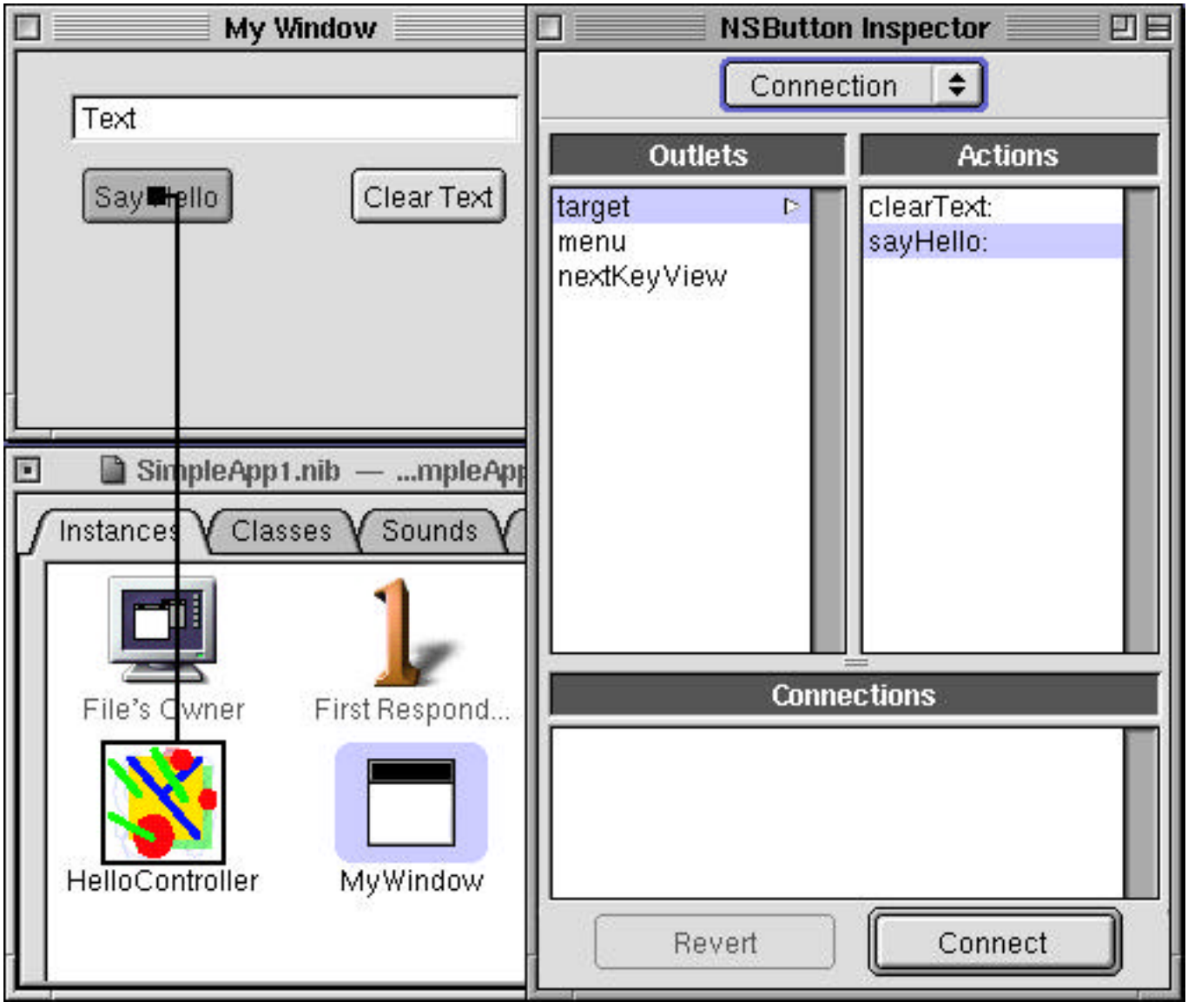


11. Connect the Say Hello button to the HelloController object:

a) Hold down the Control key while dragging from the Say Hello button. Drag to the HelloController icon in the file window and release the mouse button.

b) The Connections Inspector automatically opens. Select **target** in the Outlets column. A list of possible actions appears in the Actions column.

c) Select the **sayHello:** action and push Connect.



InterfaceBuilder uses the word **outlet** to refer to an instance variable of type id. An outlet can be connected to any type of object. InterfaceBuilder recognizes **target** as a special kind of outlet, one that always has an associated action. When you connect a **target** outlet, InterfaceBuilder also lets you set the action.

InterfaceBuilder uses the word outlet to refer to an instance variable of type id. An outlet can be connected to any type of object. InterfaceBuilder recognizes target as a special kind of outlet, one that always has an associated action. When you connect a target outlet, InterfaceBuilder also lets you set the action.

12. Connect the Clear Text button to HelloController. Set the action to **clearText:**.

13. Connect HelloController's **helloTextField** outlet to the text field. Control-drag from HelloController to the text field. Select the **helloTextField** outlet and push Connect in the Connections Inspector.

    For the HelloController object to print out "Hello world!" in the text field, it needs a connection to the text field. Without this connection, pushing the Say Hello and Clear Text buttons has no visible effect. The **sayHello:** and **clearText:** messages are still sent to the HelloController object. However, the HelloController object can't tell the text field to display anything, because HelloController doesn't have a connection to it.

14. Test the interface. Because the user interface elements you create using InterfaceBuilder are real objects, you can test your interface and see how it works. Choose the Test Interface command in the File menu. Try out your interface by pushing the Say Hello and Clear Text buttons.

15. Create another set of buttons and a text field. Position and label them as before.

16. Drag another HelloController object from the HelloController palette. You can have many instances of the same type of object. Notice that InterfaceBuilder automatically names the new object HelloController1, so you can distinguish it from the previous HelloController.

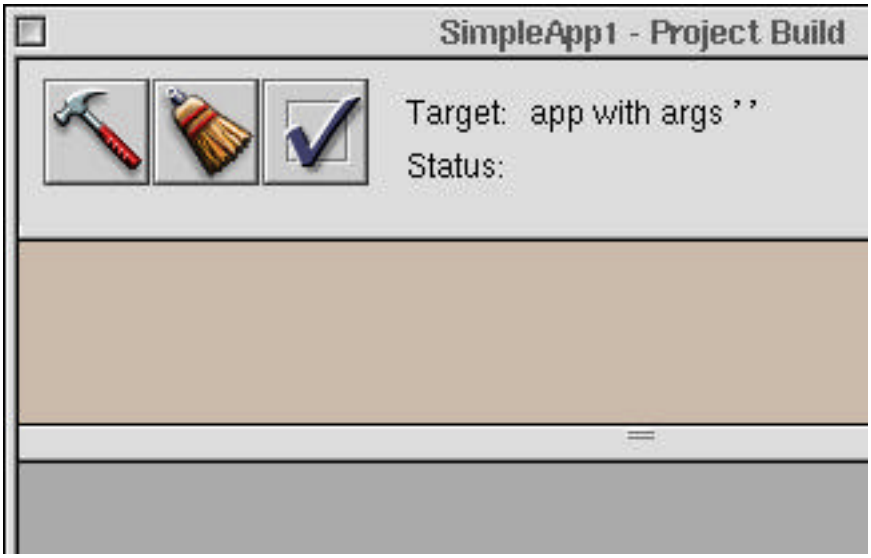
17. Connect HelloController1 to the new text field. Set the target of the new buttons to HelloController1, with appropriate actions.

18. Test your interface. Choose Test Interface in the File menu. Try both sets of Say Hello and Clear Text buttons.

19. Add the code for the HelloController class to your project by including the HelloController.framework to your project.

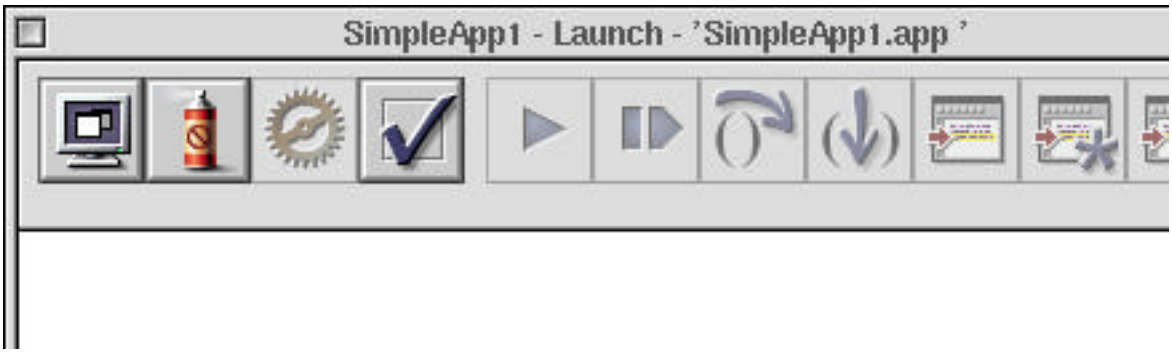
    Switch to ProjectBuilder by double-clicking its icon. Double-click the Frameworks category in the browser—this brings up an open panel. Navigate to your Frameworks folder (from step 7) and select HelloController.framework.

20. Build the application. Bring up the Build Panel by pushing the Build button in the project window. It has a hammer icon. In the Build panel, start building the application by pushing the Build button.



ProjectBuilder's Build panel allows you to easily build an application. All error messages are reported in the Build panel. To make debugging easier, when you click on an error message ProjectBuilder automatically displays the file with the error, at the line where the error occurred.

21. Launch the application. Bring up the Launcher Panel by pushing the Launch button in the project window. Its icon shows application windows on a computer screen. You'll use the Launcher panel for debugging in later exercises.



22. Test the functionality of your application. Do the buttons work the same as when you tested the interface in InterfaceBuilder?

Important ideas from this section

- In an object-oriented application with a graphical user interface, you create a network of objects and then wait for events.
- Creating a network of objects is made easier by InterfaceBuilder.
- An outlet is an instance variable that can point to any object.
- Many user interface objects have the idea of a target and action.
- The target is an outlet.
- The action is the message to send to the target when the user activates the user interface object. For example, a button sends its action to the target when the user pushes the button.

REVIEW OBJECTS AND GRAPHICAL INTERFACES

1. What is InterfaceBuilder used for?

2. What do nib files contain?

3. What is an outlet?

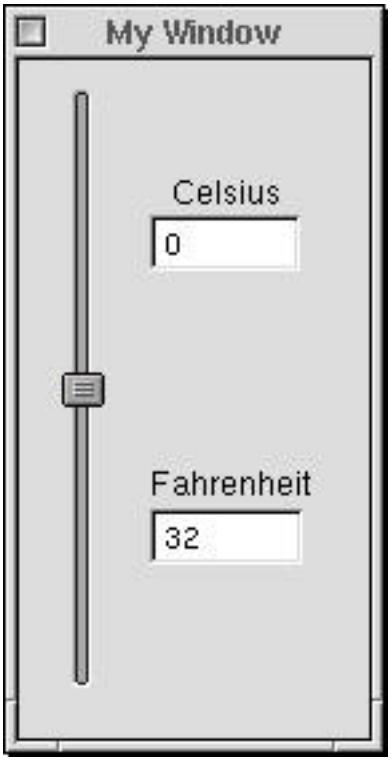
4. What is a target?

5. What is an action?

6. How do targets and actions work together?

7. What is ProjectBuilder used for?

EXERCISE 2.1 TEMPERATURE CONVERTER



Now that you've had a chance to try out InterfaceBuilder and ProjectBuilder, it's time to use them to create your own application. In this exercise you will build a temperature conversion application. The application has a user interface similar to the one shown above. The user moves a slider to change the current temperature, while two text fields display the temperature in Celsius and Fahrenheit.

The application uses a Converter object, provided on a palette, to do the necessary calculation. You connect user interface elements to the Converter using InterfaceBuilder to create a fully functional application.

Objectives

After completing this exercise, you'll be able to:

- Create a user interface using InterfaceBuilder
- Add a controller object from a palette
- Set the target and action of a slider
- Connect a controller object to a user interface element
- Set attributes of a user interface object using InterfaceBuilder's inspector

Exercise

1. Create a new application project named Temperature1 in ProjectBuilder.

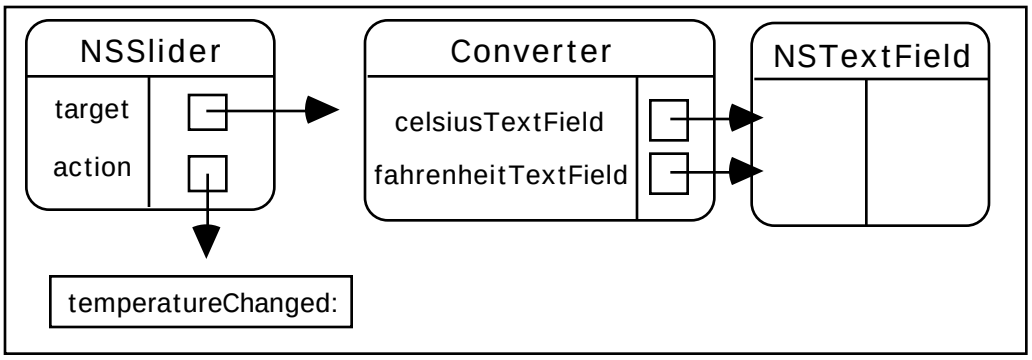
2. Use InterfaceBuilder to set up a user interface with a slider and two text fields for displaying the temperature in Celsius and Fahrenheit.

3. Copy the Converter.framework from the /ExerciseMaterials/Frameworks/ Converter.framework.to your <AccountName>/Library/Frameworks folder (you may have to create this folder yourself). When asked how the links should be copied, check the "Repeat" checkbox and then click the "New Link" button.

4. Add the ConverterPalette to InterfaceBuilder. Use the Tools menu in InterfaceBuilder and select Palettes->Open..., then double-click on /ExerciseMaterials/Palettes/ ConverterPalette.palette.

5. Drag a Converter object from the palette to the your instances window.

6. Connect the objects in your user interface to the Converter object. The diagram below shows what connections should be made. Remember that you make connections in InterfaceBuilder by Control-dragging.



7. Set the limits and default value for the slider. Select the slider. Choose the Inspector command in the Tools menu to bring up the inspector.

A slider is a user interface element that allows the user to select a value from a range of values. As such it has a current value, a maximum value, and a minimum value. InterfaceBuilder allows you to set these initial values without having to write any code.

8. Set the text fields to match the current value. The Converter object takes the value of the slider as the temperature in celsius. Therefore, you should set the celsius text field to read 0 and the fahrenheit text field to read 32.

9. Make the text fields non-editable. One of the attributes of a text field is whether the user is allowed to edit it directly. The program will provide the values in the text fields—the user provides input by moving the slider. You don't want the user to be able to edit the text fields, so use the Attributes Inspector to make them non-editable.

10. Test the interface. Make sure everything works as you expect.

11. Add the code for the Converter class to your project by including the Converter.framework to your project.

    Switch to ProjectBuilder by double-clicking its icon. Double-click the Frameworks category in the browser—this brings up an open panel. Navigate to your Frameworks folder and select Converter.framework.

    Build and run the Temperature1 application.