Chapter 3

Creating Classes

CHAPTER 3 CREATING CLASSES

In Objective-C, you define objects by defining their class. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every member of the class, and it defines a set of methods that all objects in the class can use.

—Object-Oriented Programming and the Objective-C Language,

Goal

To understand how to write classes.

Prerequisites

A basic understanding of objects and instance variables.

Objectives

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

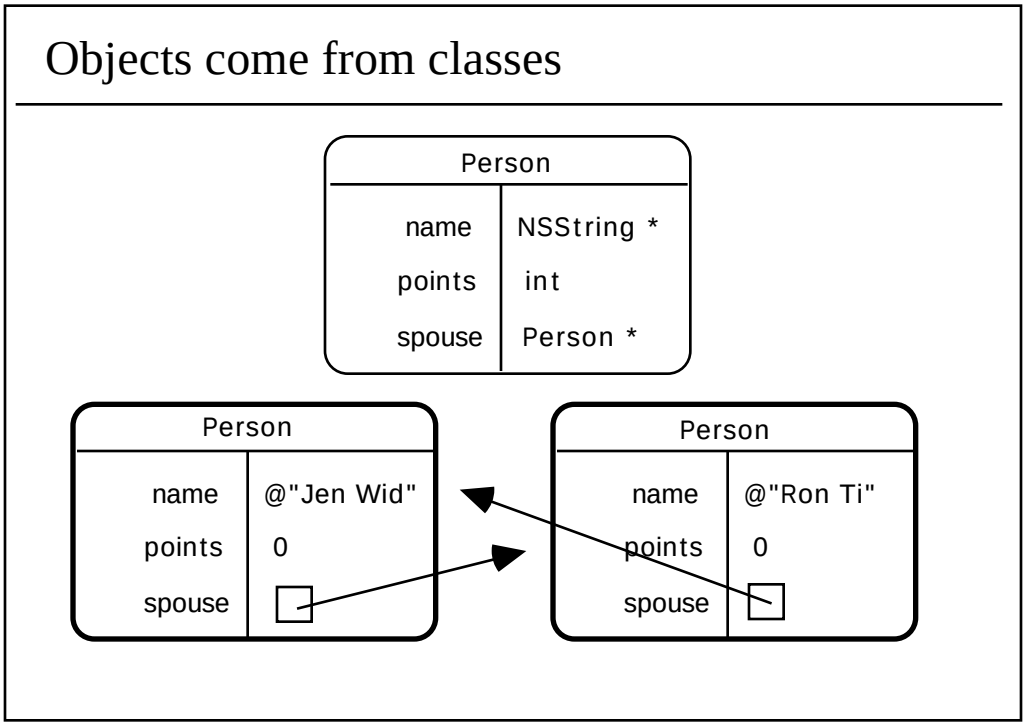
- Send a message to an object in Objective-C
- Write a simple class in Objective-C

Reading:

These references contain more information about creating classes:

/System/Documentation/Developer/TasksAndConcepts/ DevEnvGuide/Chapters/ (Subclass)

/System/Documentation/Developer/TasksAndConcepts/ ObjectiveC



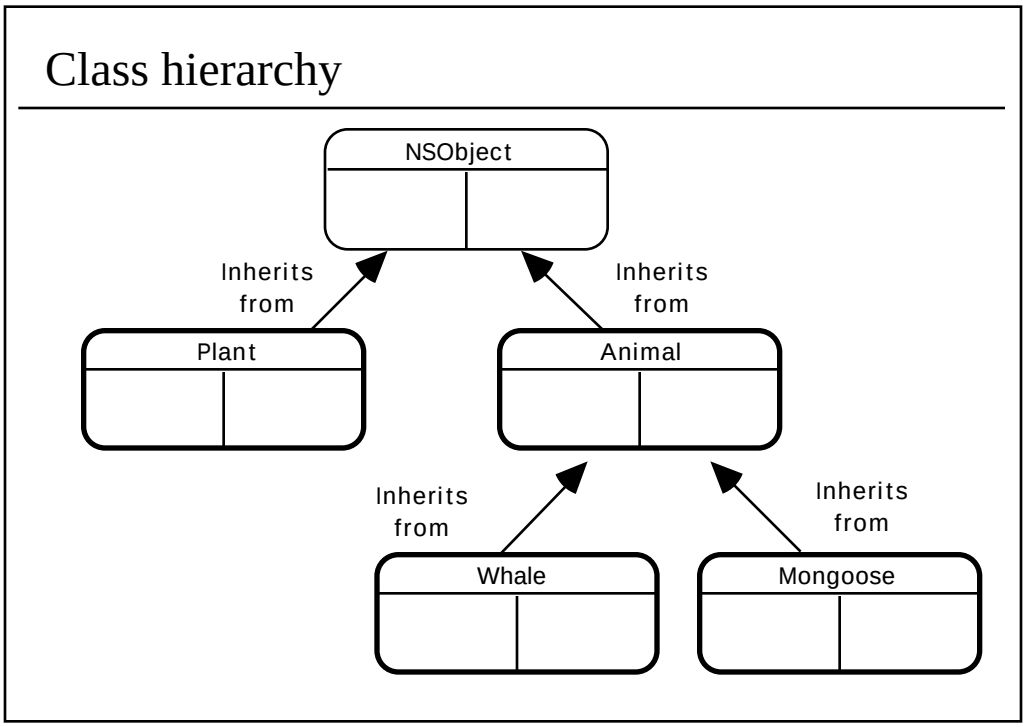
Objects come from classes

In Objective-C, you program by writing classes. The class definition is a prototype for a kind of object. It declares the instance variables that become part of every instance of the class, and it defines a set of methods all instances of the class can use.

When the class definition is compiled, it creates a class. This class exists in the program at runtime and acts as an object factory.

All instances of a class have access to the same set of methods, and they all have a set of instance variables cut from the same mold. Each object gets its own instance variables.

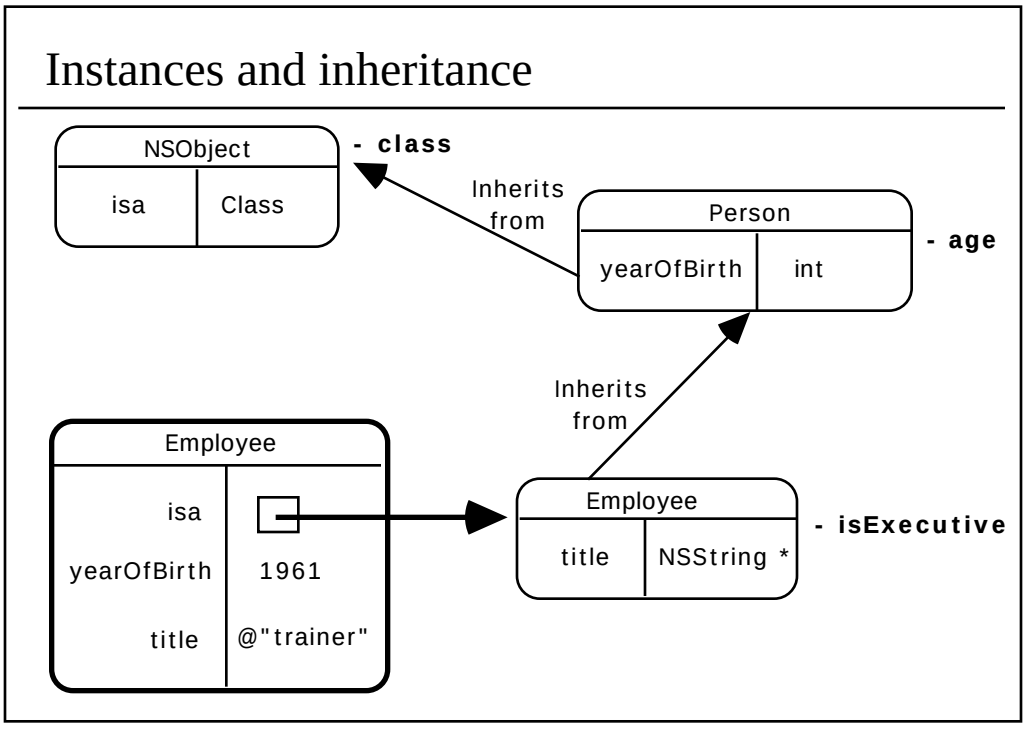
By convention, class names begin with an uppercase letter, for example, “Person.” The names of instances typically begin with a lowercase letter, for example, “aPerson.”



Class hierarchy

Classes are arranged in a hierarchy. More general classes are arranged at the top of the hierarchy, more specialized classes near the bottom. In the example, Whale is a **subclass** of Animal. Animal is the **superclass** of Whale. The subclass to superclass relationship is a direct child-parent relationship. There can be no intervening classes in the hierarchy between a subclass and its superclass.

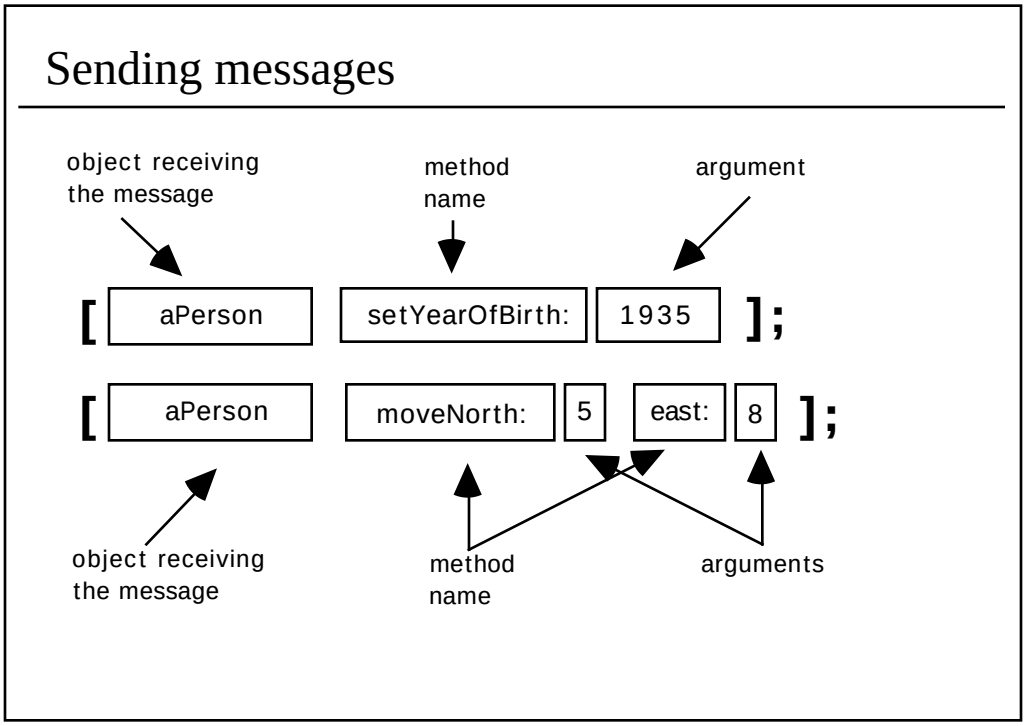
NSObject is an **ancestor** of Whale. Whale is a **descendant** of NSObject. The descendant to ancestor relationship is an eventual child-parent relationship. There can be several intervening classes in the hierarchy between a descendant and its ancestor.



Instances and inheritance

Class definitions are additive. Each new class you define is based on another class from which it inherits methods and instance variables. The new class simply adds to or modifies what it inherits. It doesn't need to duplicate inherited code.

Inheritance links all classes together in a hierarchical tree with a single class, NSObject, at its root. Every class except NSObject has a superclass one step nearer the root, and any class, including NSObject, can be the superclass for any number of subclasses one step farther from the root. When you define a class, you link it to the hierarchy by declaring its superclass. Every class you create must be the subclass of another class.



Sending messages

To get an object to do something, you send it a message telling it to apply a method. In Objective-C, message expressions are enclosed in square brackets:

```
[thePerson receiveRaise];
```

The receiver is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any arguments passed to it. When a message is sent, the runtime system selects the appropriate method from the receiver's repertoire and invokes it.

If there are any arguments, they appear after colons in the method name.

```
[aPerson receiveRaise:10];

[aPerson moveNorth:5 east:8];
```

Remember the difference between a method and a message. A **method** is a set of instructions to execute. It's associated with the class of the object. A **message** is the stimulus that causes a method to execute. It's transitory, generally triggered by a user event like pushing a button or in source code.

Examples of messages

Here are several lines of a program, with explanations of what they do.

```
Employee *aWorker;
Department *anOffice;
```

Two variables are declared. Assume that **anOffice** is initialized to contain the address of an instance of the Department class.

```
[anOffice raiseMorale];
```

This sends the message **raiseMorale** to **anOffice**. **raiseMorale** is a simple method—it takes no arguments and returns nothing.

```
aWorker = [anOffice personAtDesk:5];
```

Here **aWorker** is set to the person at desk number 5. This code sends the message **personAtDesk:** with an argument of **aNum** to **anOffice**. The return value of this message is assigned to the variable **aWorker**. **personAtDesk**: is a somewhat more complex method—it takes a single argument, and returns an object as its return value.

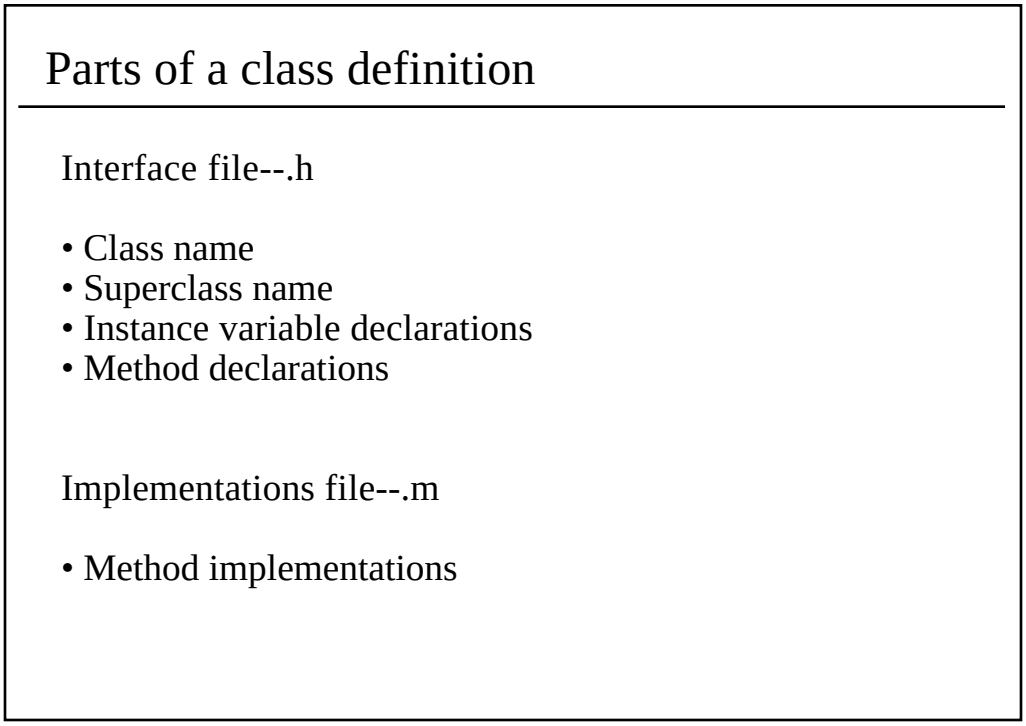
```
[aWorker moveLeft:1 forward:4];
```

This code moves **aWorker** one space to the left and four spaces forward by sending the message **moveLeft:forward:** with two arguments—1 and 4. **moveLeft:forward:** takes two arguments, but doesn't return anything. Generally, messages that make an object do something don't return a value. Messages that ask an object for information about itself do.

```
[[anOffice boss] receiveRaise];
```

This code sends the message **boss** to **anOffice**. **boss** is a method that returns the person object for the department's boss. The result of the initial message, the department's boss, is then sent the message **receiveRaise**. Presumably the boss has done a good job and deserves a raise.

This last example shows how messages can be nested. Nesting messages makes your code more compact, and sometimes makes it easier to read. However, you should beware of overusing this feature. It's much easier to debug a program where intermediate results are stored in variables, instead of nested deep within a complex message expression.



Parts of a class definition

To write a class, you need to create two files—an interface file and an implementation file. You generally name these files with the name of the class they define. A class called Employee would have these two files:

» **Employee.h**—the interface file

» **Employee.m**—the implementation file

The familiar extension .h indicates a header file. The .m extension indicates a file containing Objective-C source code.

Separating the implementation from the interface maintains encapsulation. Someone who wants to use your class needs access to its interface. If you had both the interface and implementation in a single file, they'd be able to see the implementation. By separating the implementation and interface into two files, you can give someone just the interface without the implementation.

Interface example: Employee.h

Here is an example interface file. It's a simple class, so the interface file is short.

```
#import "Person.h"

@interface Employee : Person
{
   NSString *title;
}

- (BOOL)isExecutive;

@end
```

The declaration of a class interface begins with the compiler directive @interface and ends with the directive @end. All Objective-C compiler directives begin with "@". The @interface directive names the class and its superclass, separated by a colon.

Before the declaration of the class, you must import the interface file for the superclass. This way the compiler knows which instance variables and methods are inherited from the super class.

The #import compiler directive is similar to #include, but #import ensures that a file is only included once.

The general format of the interface file for a class is as follows:

```
#import "SuperClass.h"

@interface ClassName : SuperClass
{
   instance variable declarations
}

method declarations

@end
```

Implementation example: Employee.m

Here is the implementation file that corresponds to the previous interface example. Again, it's quite short.

```
#import "Employee.h"

@implementation Employee

- (BOOL)isExecutive
{
   if ([title isEqual:@"President"]) {
      return YES;
   } else {
      return NO;
   }
}

@end
```

The first thing you have to do in the implementation file is import the interface for the class. The interface contains important information that the compiler needs to do its job.

The definition of a class starts with @implementation and ends with @end. @implementation simply names the class being defined. There's no need to name the superclass, as that's defined in the interface.

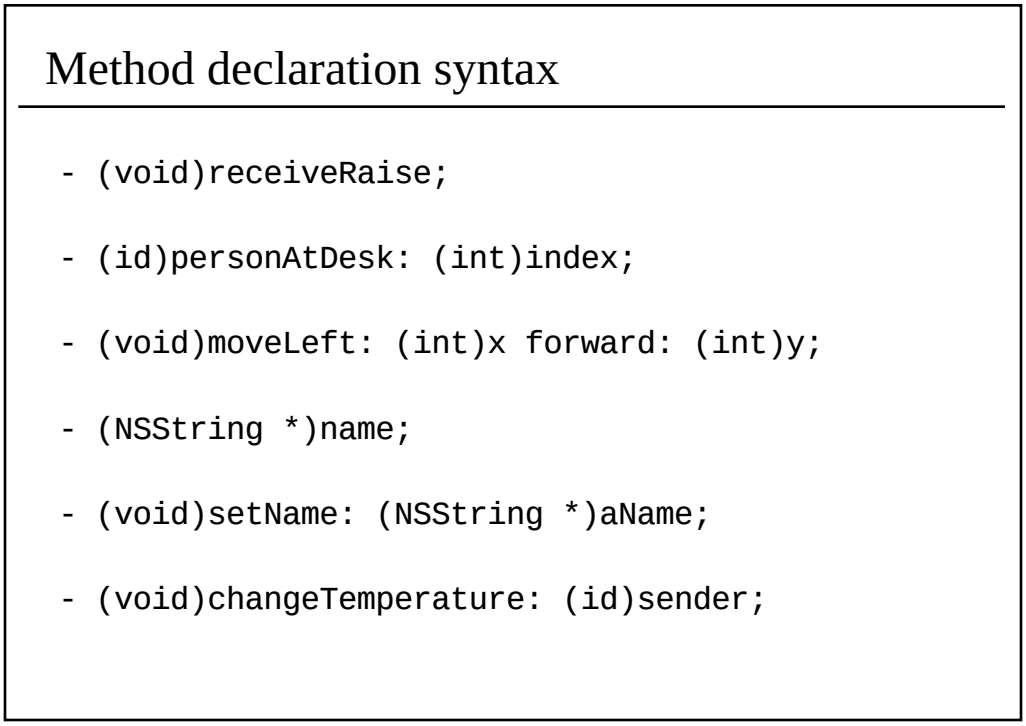
The main contents of the implementation file is the implementation of all the methods defined in the class. The general syntax of an implementation file is as follows:

```
#import "SomeClass.h"

@implementation SomeClass

method definitions

@end
```



Method declaration syntax

Method return types are declared using parentheses. This is similar to the C casting operator:

```
- (int)tag;
```

Parameter types are declared in the same way:

```
- setTag:(int)anInt;
```

If a return or argument type isn't explicitly declared, it's assumed to be the default type for methods and messages— id. It's good coding practice to always provide explicit types.

When there's more than one argument, they're declared within the method name after the colons. Arguments break the name apart in the declaration, just as in a message. For example:

```
- (void)moveLeft:(int)x forward:(int)y;
```

You can determine the number of arguments by counting the number of colons in the method name.

Example of method implementation

A method is simply a list of instructions. It's much like a function, but has access to the object's instance variables. This method accesses two instance variables: **celsiusField** and **fahrenheitField**.

```
- (void)temperatureChanged:(id)sender
{
   int celsius; // degrees Celsius
   double fahrenheit; // degrees Fahrenheit

   celsius = [sender intValue];
   fahrenheit = (1.8 * celsius) + 32.0;
   [celsiusField setIntValue:celsius];
   [fahrenheitField setIntValue:fahrenheit];
}
```

Notice the message to **sender**. **temperatureChanged:** is an action method. When a button or other user interface element sends an action message, it includes a pointer to itself as the **sender** parameter. In this case, the sender is a slider object. **temperatureChanged:** uses the **sender** parameter to ask the slider its current integer value.

It doesn't matter if the user interface later changes to use some other slider, or a completely different user interface element. As long as it understands **intValue**, **temperatureChanged:** will still work properly. This concept of action messages including sender makes source code much more portable to different user interfaces.

Accessor methods

Because encapsulation prevents access to instance variables from outside an object, you often need to create accessor methods for your instance variables. An **accessor method** allows you to read or set an object's instance variable from outside the object.

By convention, the accessor method for reading an instance variable has the same name as the instance variable. The accessor method for setting an instance variable precedes the instance variable name with "set." For example, accessor methods for the instance variable **name** would be declared like this:

```
- (NSString *)name;
- (void)setName:(NSString *)aName;
```

The implementation for accessor methods tends to be quite straightforward.

```
- (NSString *)name
{
   return name;
}

- (void)setName:(NSString *)aName
{
   if (name != aName) {
      [name release];
      name = [aName copy];
   }
}
```

Notice the if clause. Without it, **setName** would send a **release** message to **aName**–its retain count could, potentially, go to 0, and **aName** would be deallocated. There would be no argument to retain since name and **aName** pointed to the same object. The if clause solves the problem.

Assuming the argument is different from the object in our instance variable, the old name is released. The **release** call in **setName**: tells the old string object that it can disappear. This is how useless objects get cleaned out of memory.

self

If object **myObject** of class SomeClass is executing a method **myMethod**, the method can access several variables:

» Global variables

» **myObject**'s instance variables

» **myMethod**'s temporary variables

» self

self is a pointer to the object executing the method. In other words, **self** is a pointer to yourself—in this case, **myObject**. **self** allows the object to send itself messages. The most common use of **self** is to call accessor methods.

```
[self setName:@"Max Funk"]
```

Why? Couldn't you simply say:

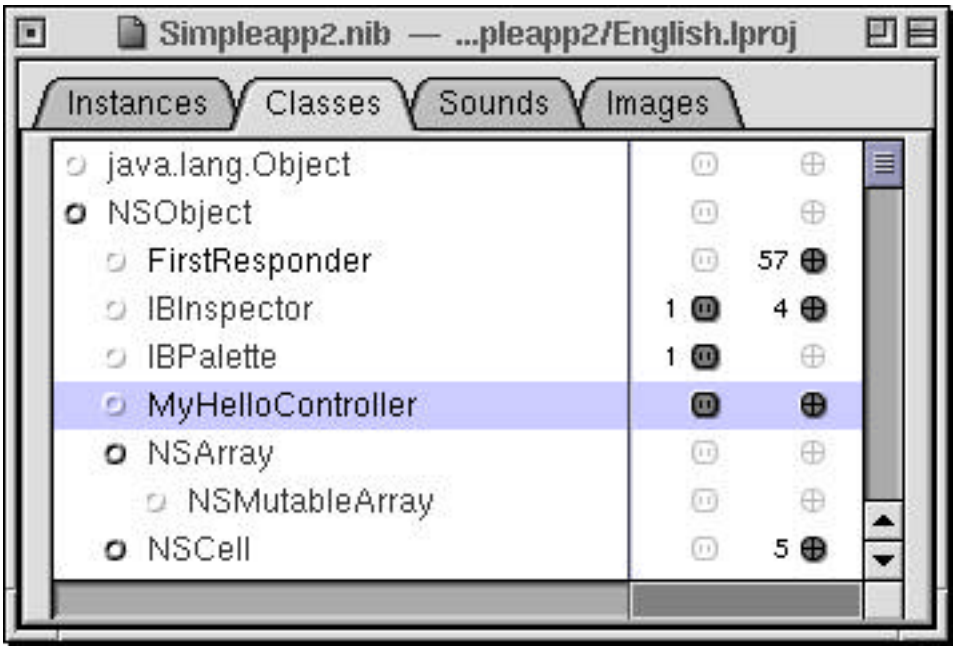
```
name = @"Aaron";
```

You could, but you would have forgotten to release the old name. This would be a memory leak. The accessor method takes care of this for you:

```
- (void) setName:(NSString *)aName
{
   if (name != aName) {
      [name release];
      name = [aName copy];
   }
}
```

By using the accessor method, you centralize the code necessary for setting a new name. You don't have to remember to release the name every time you want to set a new one—you just call s**etName:** and that's it.

DEMONSTRATION 3.1 WRITING A CLASS AND USING IT IN INTERFACE BUILDER



In an earlier demonstration you created a simple application using InterfaceBuilder and a class provided on a palette. In this demonstration, you will create the same application without the palette.

Objectives

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

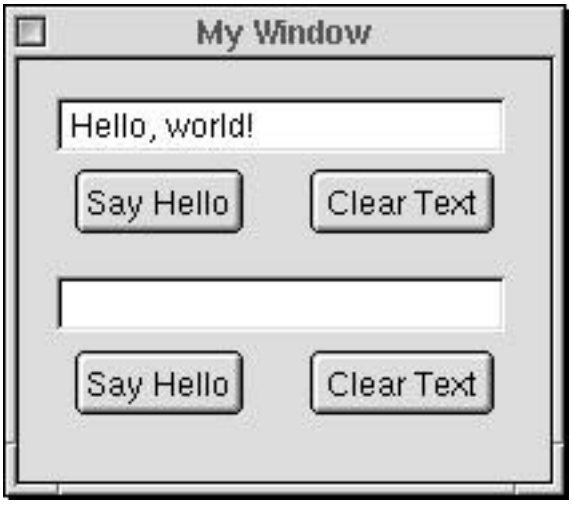
» Add a custom class to InterfaceBuilder

» Write the code for a custom class and add it to a project

Demonstration

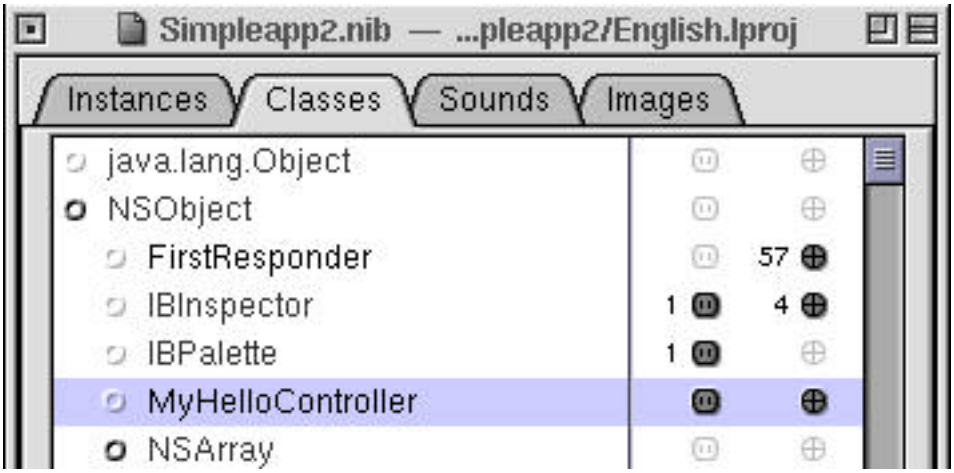
1. Use ProjectBuilder to create a new application project named SimpleApp2.

2. Create the user interface using InterfaceBuilder. The interface should match the interface from the demonstration in Chapter 2.



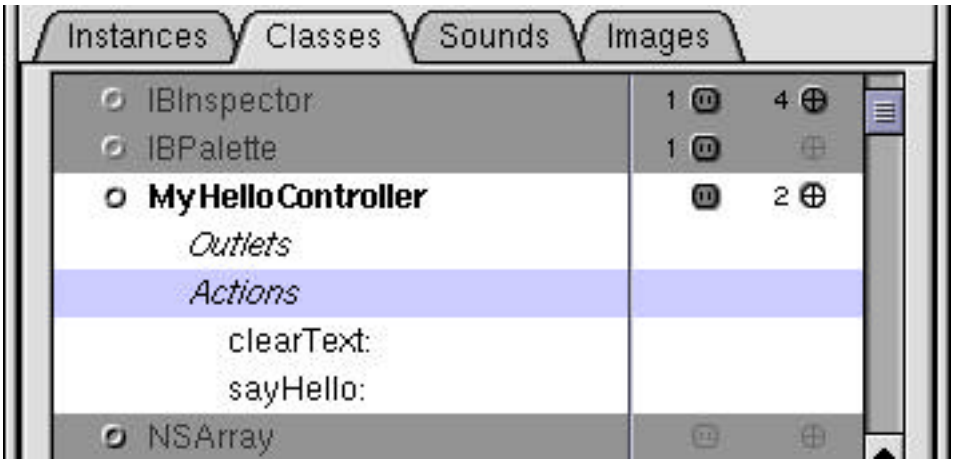
3. Make a subclass of NSObject called MyHelloController. InterfaceBuilder allows you to add new classes to the list of classes it knows about. You can add these classes by loading palettes, but sometimes you won't have a palette—for example, when you're first writing a new class. You add new classes from the Classes view in InterfaceBuilder's instances window.

    - Click the Classes tab in the instances window
    - Select NSObject
    - Select Subclass in the classes menu of InterfaceBuilder
    - Change the name of the new class to MyHelloController



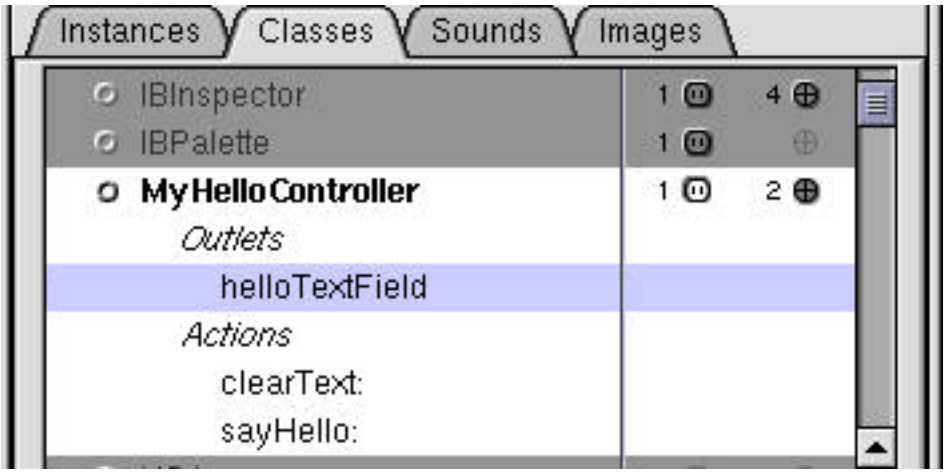
4. Add two actions, sayHello: and clearText:, to the MyHelloController class. Instances of MyHelloController need to respond to these messages when the user clicks on the buttons in your user interface. InterfaceBuilder won't let you set these messages as actions unless you tell InterfaceBuilder that MyHelloController knows how to respond. You do this in the Classes view.

    - Push the crosshair icon to the right of the word MyHelloController. Press return.
    - Change the name of the new action to sayHello:. Press return twice.
    - Change the name of the new action to clearText:.



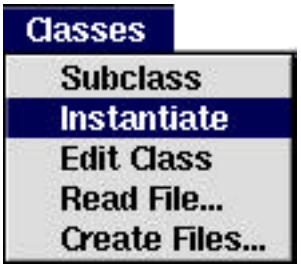
5. Add an outlet called helloTextField to MyHelloController. Instances of MyHelloController need access the text field in which they're supposed to print, "Hello world!" You provide that access through an outlet in InterfaceBuilder. An outlet is simply a variable of type id.

    - Select the outlet icon under the class MyHelloController
    - Press return
    - Change the name of the new outlet to helloTextField

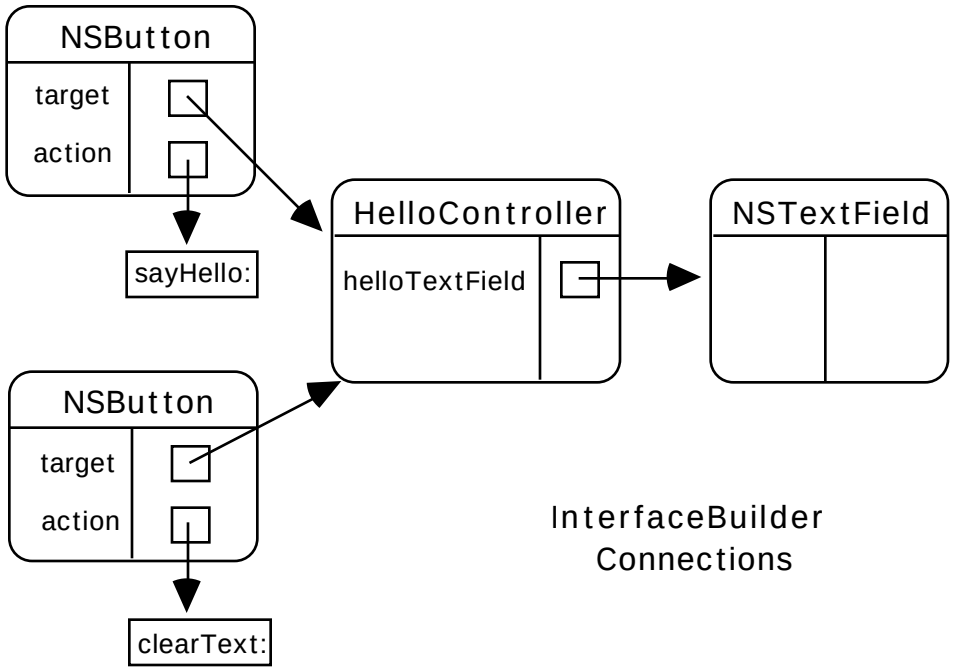


6. Add an instance of MyHelloController to your nib file. You don't have a palette to drag the instances from, so InterfaceBuilder gives you another way to instantiate MyHelloController.

    - Select MyHelloController in the class browser
    - In the Classes menu for InterfaceBuilder, choose Instantiate



7. Connect the top text field and buttons to the instance of MyHelloController. MyHelloController has the same actions and outlets as the HelloController class on the palette. Therefore, you connect the user interface elements the same as in the Demonstration for Chapter 2. As a reminder, here is a diagram showing the connections and actions:



8. Create a second instance of MyHelloController. Connect the other buttons and text field to the second MyHelloController.

9. Test your interface. Choose Test Interface in the Document menu. Try the Say Hello and Clear Text buttons. Wait a second—nothing happened! What's wrong?

    Actually, nothing's wrong. It's just that InterfaceBuilder doesn't have the code for your MyHelloController class. You've provided InterfaceBuilder with the skeleton of that code. You've told InterfaceBuilder what outlets your MyHelloController objects have and what actions they respond to, but you haven't written the code yet!

10. Add MyHelloController.m and MyHelloController.h to your project. Based on the information you've provided, InterfaceBuilder creates skeleton .m and .h files where you can add your code.

    - Select MyHelloController in InterfaceBuilder's class browser
    - Select Create Files in the Classes menu
    - Push Yes to add the files to your project

11. Edit MyHelloController.m and MyHelloController.h in ProjectBuilder.

Here is MyHelloController.h:

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

@interface MyHelloController : NSObject

{
   id helloTextField;
}

- (void)sayHello:(id)sender;
- (void)clearText:(id)sender;

@end
```

Here is MyHelloController.m:

```
#import "MyHelloController.h"

@implementation MyHelloController

- (void)sayHello:(id)sender
{
   [helloTextField setStringValue:@"Hello, World"];
}

- (void)clearText:(id)sender
{
   [helloTextField setStringValue:@""];
}

@end
```

12. Build the application and run it. Verify that everything works as expected.

DEMONSTRATION 3.2 USING THE DEBUGGER

Any sort of serious development effort requires debugging. No complicated source code is ever fully bug free when first written. A debugger makes this effort much less painful by giving you tools to help debug programs. ProjectBuilder provides a graphical interface to gdb, a command-line debugger that understands Objective-C. In this demonstration you learn how to do simple operations with the debugger.

Objectives

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

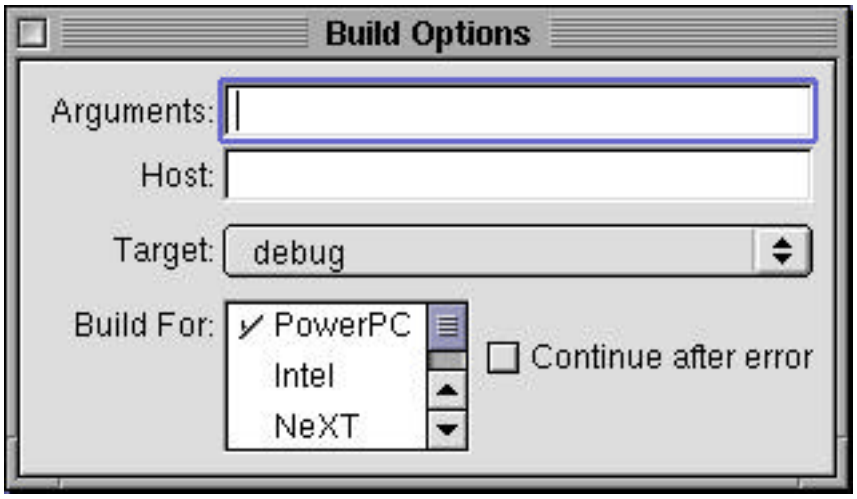
- Build a project for debugging
- Run an application in the debugger
- Set a breakpoint
- Inspect the values of variables

Demonstration

1. Open the interface file of your SimpleApp2 project.

2. Bring up the Connections inspector for one instance of MyHelloController. Select the helloTextField outlet and push the Disconnect button. This is the bug you'll use the debugger to diagnose.

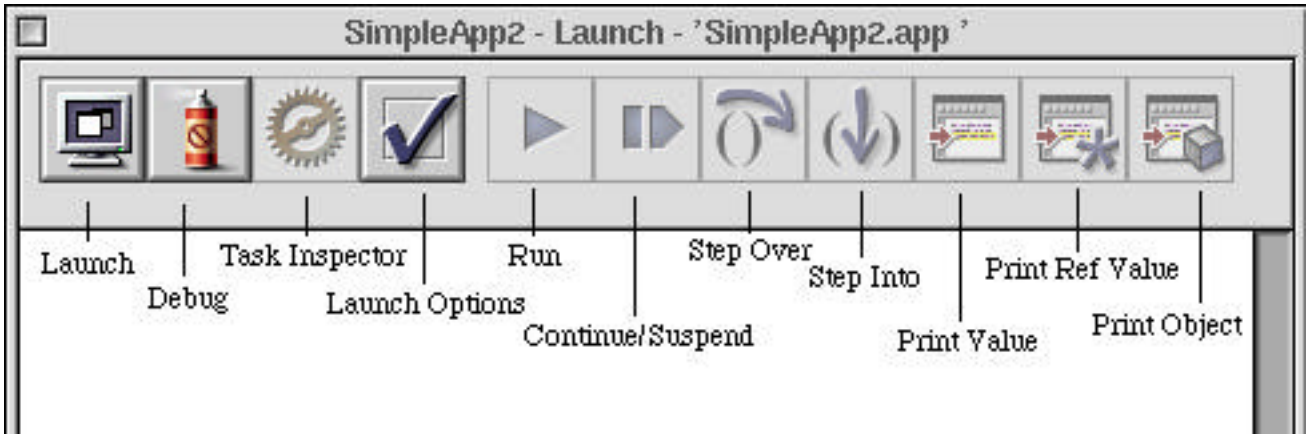
3. Switch back to ProjectBuilder and bring up the Build panel. Push the Build Options button to bring up the options panel. Choose the debug target, then build the project.



Build Options button 

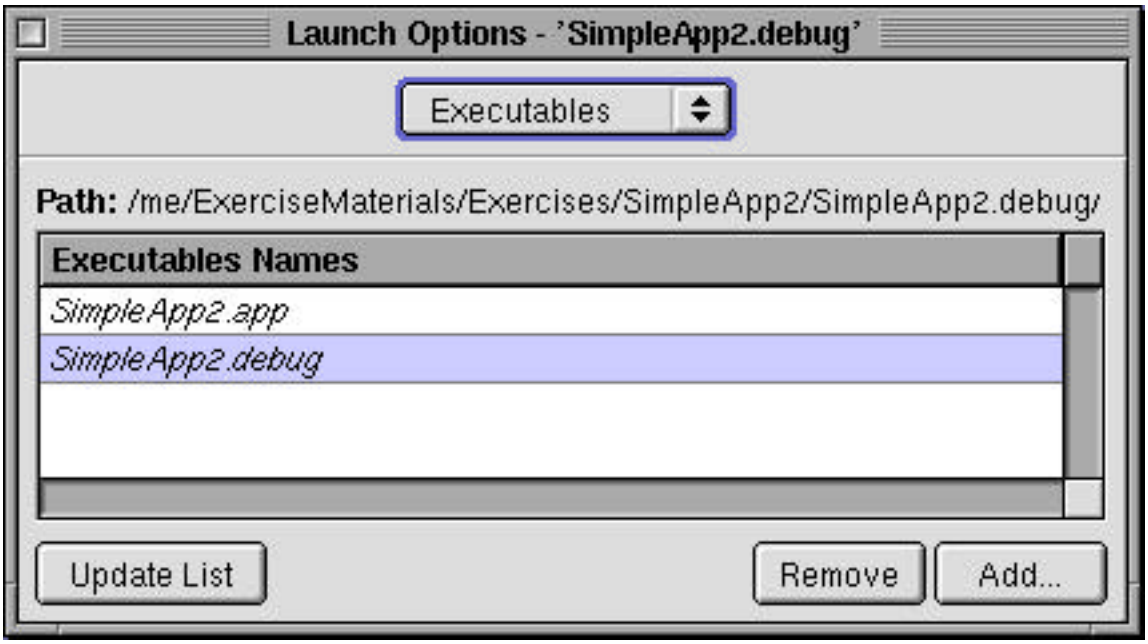
The debugger needs extra debugging symbols in an application's executable to effectively do its job. It's also easier to debug code that hasn't been optimized by the compiler. ProjectBuilder provides a special debug target for the build process that includes the debugging symbols. The debug target builds your application with .debug as the extension, instead of .app. In this case, it builds an application named SimpleApp2.debug.

4. Bring up the Launcher panel. The Launcher panel is used for debugging, as well as running applications.



5. Push the Launch options button to bring up the options panel. Tell the Launcher to debug SimpleApp2.debug, not SimpleApp2.app.

6. Push Update List to update the list of executables. Select SimpleApp2.debug. The title bar of the Launcher panel should change to reflect the new selection.



7. Push the Debug application button. The Launcher starts up gdb and displays it in the Launcher window. Because you're running a debugger, the Launcher panel enables the debugging buttons.

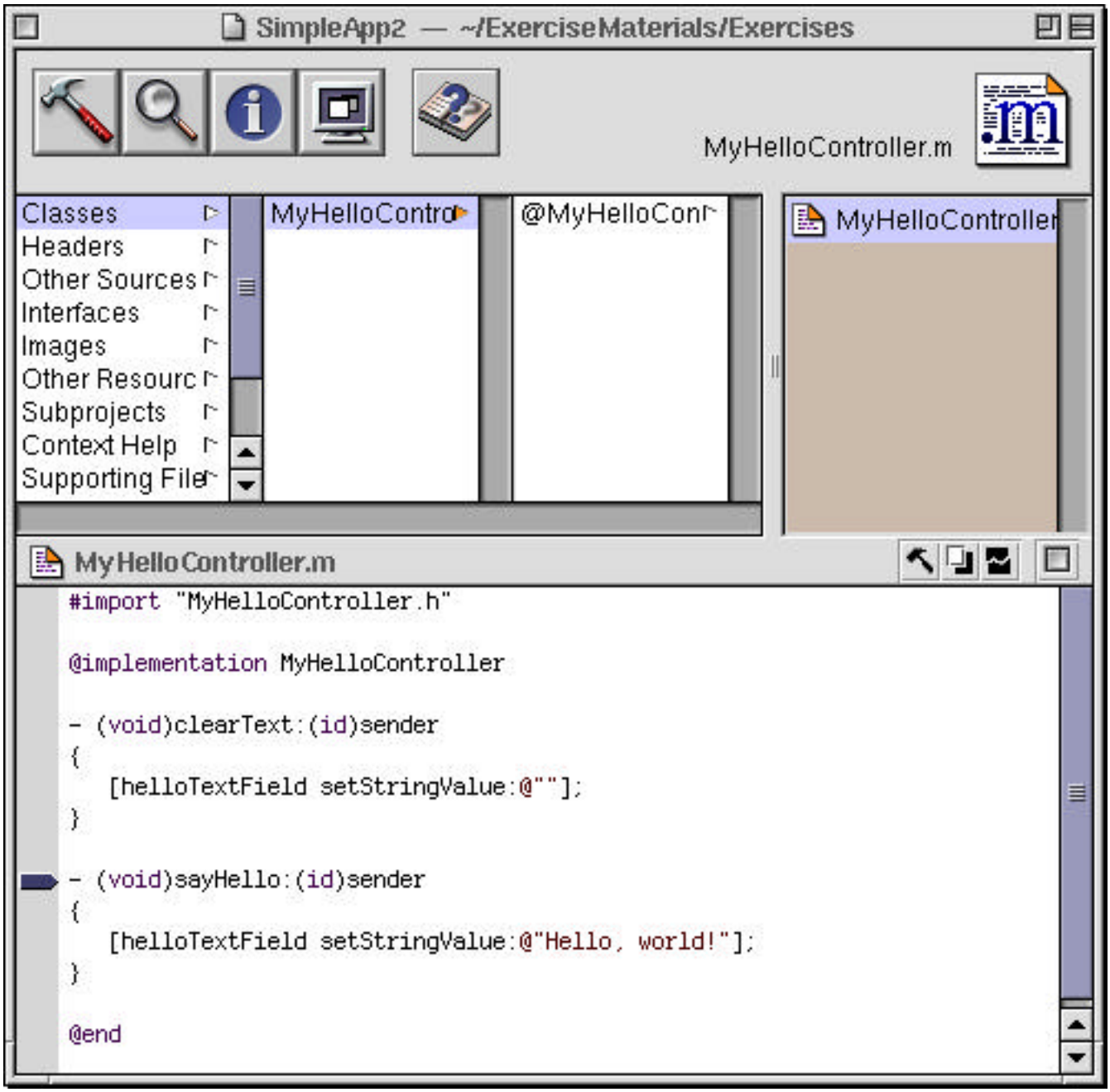
   This is a fully functional version of gdb. If you want, you can type text in the window and gdb will execute it, just as if you'd run gdb from a Bourne Shell window.

8. Push the Run button to start up the application. Try using the application. What happens when you push the Say Hello or Clear Text buttons associated with the instance of MyHelloController whose connection you broke in step 2?

9. Set a breakpoint for the sayHello: method. It might be that when you push the Say Hello button, sayHello: never gets called. Switch to ProjectBuilder and bring up the project browser window for SimpleApp2. Push the pause button in the Launcher panel, to issue commands to the debugger.

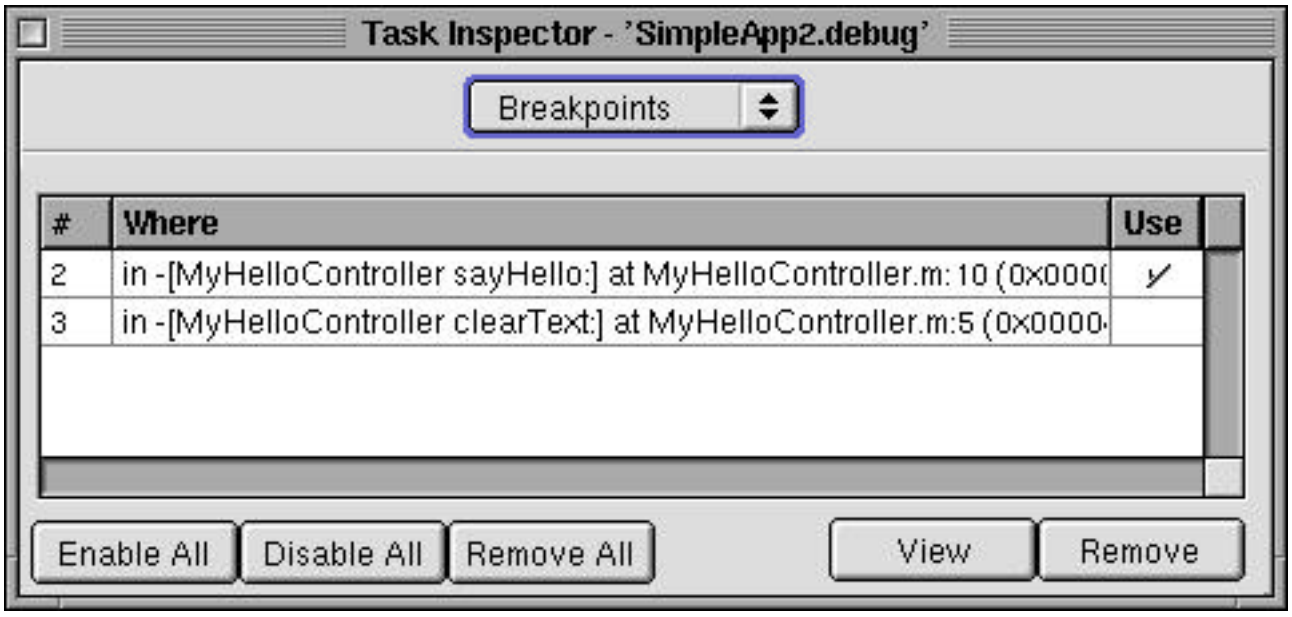
   Display the MyHelloController.m file. Notice the new column along the left-hand side of the text window. This column is used for debugging. It displays the current point of execution and lets you set breakpoints.

   Double-click in the column to the left of the sayHello: method. An arrow indicates that you've set a breakpoint. Program execution will stop when it reaches this point.



10. Set another breakpoint for the clearText: method. Double-click in the column to the left of the clearText: method. On second thought, let's disable this breakpoint for now. We only want to debug one thing at a time. Double-click on the arrow to disable the breakpoint—it dims to show it's been disabled.

11. Bring up the Task Inspector. Notice that it displays your breakpoints in a single central location. You can enable, disable, and remove breakpoints from this window.



12. Choose Stack from the pop-up list in the Task Inspector. This displays the stack of nested messages, including arguments, that were called to get to the point where execution was paused. Investigating the stack can show what other method is calling your method, which can be very revealing.

13. Push the Continue button, then switch back to SimpleApp2 and push the Say Hello button. When execution reaches the breakpoint, SimpleApp2 stops at the breakpoint.

14. Switch back to ProjectBuilder. The browser window displays the current point of execution with a red arrow. It also highlights the line of code about to be executed. Because the program stopped, you know that it's successfully sending the sayHello: message to the instance of MyHelloController. So that must not be the problem.

15. Step Into the sayHello: method and then double-click on helloTextField to select it. ProjectBuilder's variable inspection buttons work on the currently selected text. Push the Print Object button to see the description of helloTextField.

16. Notice that gdb thinks helloTextField is a nil object. This is because helloTextField is set to nil. You broke the connection from MyHelloController1 to the text field in step 2. So when sayHello: tries to send a message to helloTextField, it can't—helloTextField is nil.

    But if that's the case, shouldn't the application crash? A message is being sent to a nil object, after all. The reason the application doesn't crash is because Objective-C allows messages to nil objects—they're ignored. This makes it much easier to write methods that might message a nil object. For example, accessor methods need to autorelease the old object before retaining or copying the new one. But what if the old object doesn't exist? Being able to send release to a nil object makes accessor methods much simpler to write.

17. Push the Continue button to start SimpleApp2 executing again.

    See the documentation for more details on using the debugger.

References

/System/Documentation/Developer/Reference/DevTools/Debugger

Important ideas from this section

» To write a class, you must:

Give it a name

Give it a superclass

Declare and define its methods

» Messages are made up of:

A receiver object

method name

Arguments

» Method declarations are made up of:

A return type

A selector

Parameter names and types

» Method definitions are a set of instructions to be executed

» Accessor methods are used for setting and getting the values of instance variables

» **self** is a pointer to the object executing the method

» Generally, use accessor methods instead of accessing instance variables directly

REVIEW CREATING CLASSES

1. Do more specialized classes appear at the top or at the bottom of the class hierarchy?

2. What is the root object for all class hierarchies?

3. What two things do subclasses inherit from their superclass?

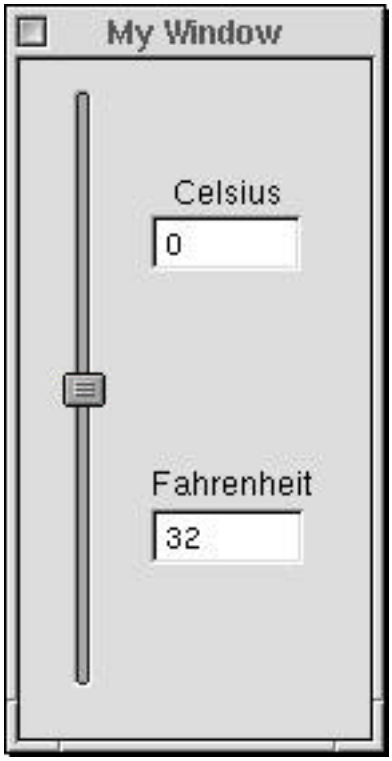
4. What are the three parts of a message?

5. What two files must be created to make a class? What does each contain?

6. What is self?

7. What are accessor methods for?

EXERCISE 3.1 TEMPERATURE CONVERTER, TAKE TWO



In this exercise you revisit the Temperature application and recreate it with your own MyConverter class. Most applications that you write need custom classes. Seldom can you just wire together a bunch of objects from palettes and get the functionality you require. It's fairly common to start an application by developing a user interface in InterfaceBuilder, adding custom classes as you go.

Objectives

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

- Add a custom class to InterfaceBuilder
- Create skeleton header and implementation files using InterfaceBuilder
- Write the code for a custom class and add it to a project

Exercise

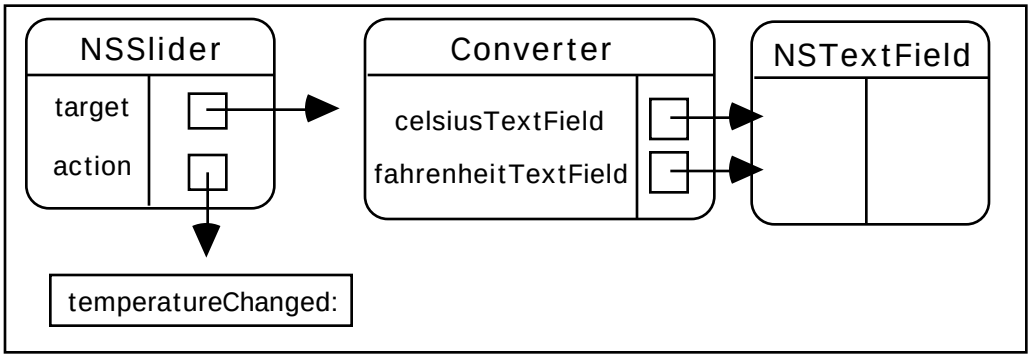
1. Create a new application project named Temperature2.

2. Make the user interface using InterfaceBuilder. The interface should have a slider and two text fields for displaying the temperature in Celsius and Fahrenheit.

3. Create a new class called MyConverter using InterfaceBuilder's Classes view. Converter should have two outlets named celsiusTextField and fahrenheitTextField. It should also have an action named temperatureChanged:.

4. Use the Instantiate command to get an instance of Converter.

5. Connect the slider and two text fields to your Converter. The diagram below shows the connections and related action.



6. Set the limits and value of the slider using InterfaceBuilder's Attributes Inspector. The slider has a range from -100 to 100, with an initial value of 0.

7. Set the text fields to match the initial value of the slider. Remember that the Converter interprets the value of the slider as the temperature in degrees Celsius.

8. Make the text fields non-editable so the user won't be able to change their values except by moving the slider.

9. Test your interface. Choose Test Interface in the File menu. Try moving the slider around. Wait a second. Nothing happened!
   What's wrong?

   Actually, nothing's wrong. It's just that InterfaceBuilder doesn't have the code for your Converter class. You've provided InterfaceBuilder with the skeleton of that code. That is, you've told InterfaceBuilder what outlets your Converter object has and what actions it responds to, but you haven't written the code yet!

10. Generate Conveter.h and Conveter.m. Use the Create Files command in InterfaceBuilder's Classes pull-down menu. Make sure you select the Converter class first!

11. Alter MyConveter.m to read:

```
#import "MyConverter.h"

@implementation MyConverter

- (void)sliderChanged:(id)sender
{
   int celsius;
   double fahrenheit;

   celsius = [sender intValue];
   [celsiusTextField setIntValue:celsius];
   fahrenheit = 1.8 * celsius + 32.0;
   [fahrenheitTextField setIntValue:fahrenheit];
}

@end
```

12. Build your project and run the application. Make sure everything works as expected.