Table of Contents

iOS Programming

The Big Nerd Ranch Guide

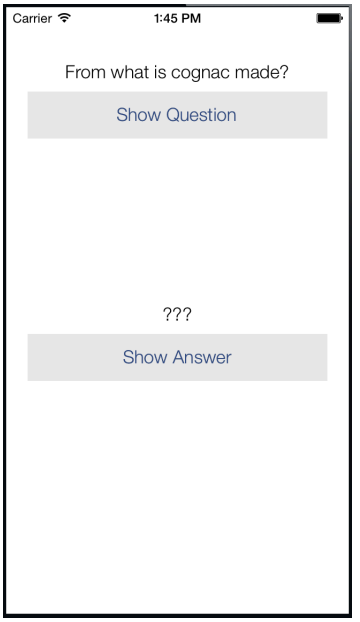
Christian Keur, Aaron Hillegass & Joe Conway

Big nerd ranch

1
A Simple iOS Application

In this chapter, you are going to write an iOS application named Quiz. This application will show a question and then reveal the answer when the user taps a button. Tapping another button will show the user a new question (Figure 1.1).

Figure 1.1 Your first application: Quiz



When you are writing an iOS application, you must answer two basic questions:

- How do I get my objects created and configured properly? (Example: “I want a button here entitled Show Question.”)
- How do I deal with user interaction? (Example: “When the user taps the button, I want this piece of code to be executed.”)

Most of this book is dedicated to answering these questions.

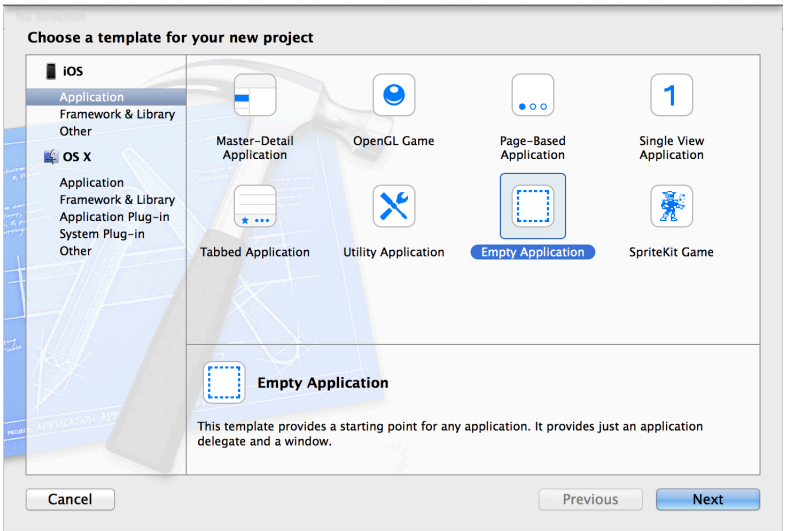
As you go through this first chapter, you will probably not understand everything that you are doing, and you may feel ridiculous just going through the motions. But going through the motions is enough for now. Mimicry is a powerful form of learning; it is how you learned to speak, and it is how you will start iOS programming. As you become more capable, you will experiment and challenge yourself to do creative things on the platform. For now, just do what we show you. The details will be explained in later chapters.

Creating an Xcode Project

Open Xcode and, from the File menu, select New → Project...

A new workspace window will appear, and a sheet will slide down from its toolbar. On the lefthand side, find the iOS section and select Application (Figure 1.2). You will be offered several application templates to choose from. Select Empty Application.

Figure 1.2 Creating a project



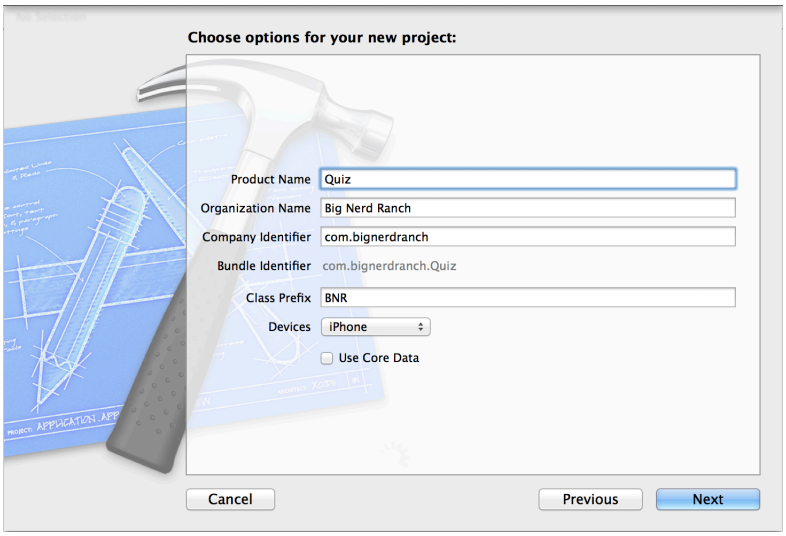
You are using the Empty Application template because it generates the least amount of boilerplate code. Too much boilerplate gets in the way of learning how things work.

This book was created for Xcode 5.0.2. The names of these templates may change with new Xcode releases. If you do not see an Empty Application template, use the simplest-sounding template. Or visit the Big Nerd Ranch forum for this book at `forums.bignerdranch.com` for help working with newer versions of Xcode.

Click Next and, in the next sheet, enter Quiz for the Product Name (Figure 1.3). The organization name and company identifier are required to continue. You can use `Big Nerd Ranch` and `com.bignerdranch`. Or use your company name and `com.yourcompanynamehere`.

In the Class Prefix field, enter `BNR` and, from the pop-up menu labeled Devices, choose iPhone. Make sure that the Use Core Data checkbox is unchecked.

Figure 1.3 Configuring a new project

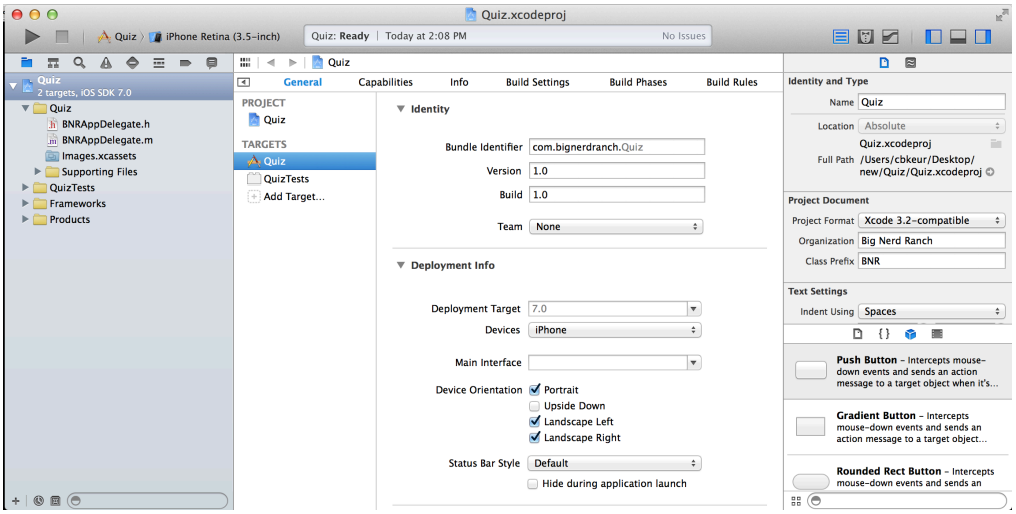


You are creating Quiz as an iPhone application, but it will run on an iPad. It will not look quite right on the iPad's screen, but that is okay for now. For the first part of this book, you will stick to iPhone applications and focus on learning the fundamentals of the iOS SDK, which are the same across devices. Later, you will see some options for iPad-only applications as well as how to make applications run natively on both types of devices.

Click Next and, in the final sheet, save the project in the directory where you plan to store the exercises in this book. You can uncheck the box that creates a local git repository, but keeping it checked will not hurt anything. Click Create to create the Quiz project.

Once the project is created, it will open in the Xcode workspace window (Figure 1.4).

Figure 1.4 Xcode workspace window

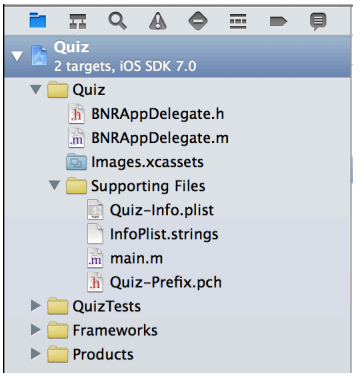


Take a look at the lefthand side of the workspace window. This area is called the *navigator area*, and it displays different *navigators* – tools that show you different parts of your project. You can choose which navigator to use by selecting one of the icons in the navigator selector, which is the bar just above the navigator area.

The navigator currently open is the *project navigator*. The project navigator shows you the files that make up your project (Figure 1.5). You can select a file to open it in the *editor area* to the right of the navigator area.

The files in the project navigator can be grouped into folders to help you organize your project. A few groups have been created by the template for you; you can rename them whatever you want or add new ones. The groups are purely for the organization of files and do not correlate to the filesystem in any way.

Figure 1.5 Quiz application's files in the project navigator



In the project navigator, find the files named `BNRAppDelegate.h` and `BNRAppDelegate.m`. These are the files for a *class* named **`BNRAppDelegate`**. The Empty Application template created this class for you.

A class describes a kind of *object*. iOS development is object-oriented, and an iOS application consists primarily of a set of *objects* working together. When the Quiz application is launched, an object of the **`BNRAppDelegate`** kind will be created. We refer to a **`BNRAppDelegate`** object as an *instance* of the **`BNRAppDelegate`** class.

You will learn much more about how classes and objects work in Chapter 2. Right now, you are going to move on to some application design theory and then dive into development.

Model-View-Controller

Model-View-Controller, or MVC, is a design pattern used in iOS development. In MVC, every object is either a model object, a view object, or a controller object.

- *View objects* are visible to the user. Examples of view objects are buttons, text fields, and sliders. View objects make up an application's user interface. In Quiz, the labels showing the question and answer and the buttons beneath them are view objects.

- *Model objects* hold data and know nothing about the user interface. In Quiz, the model objects will be two ordered lists of strings: one for questions and another for answers.

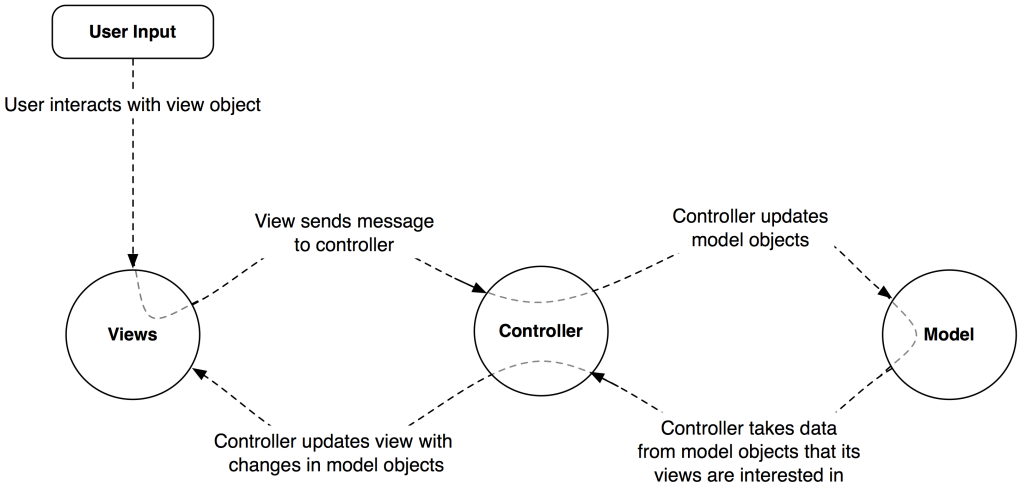
  Usually, the model objects are modeling real things from the world of the user. For example, when you write an app for an insurance company, you will almost certainly end up with a custom model class called **`InsurancePolicy`**.

- *Controller objects* are the managers of an application. Controllers configure the views that the user sees and make sure that the view and model objects keep in sync.

  In general, controllers typically handle "And then?" questions. For example, when the user selects an item from a list, the controller determines what that user sees next.

Figure 1.6 shows the flow of control in an application in response to user input, such as the user tapping a button.

Figure 1.6 MVC pattern



Notice that the models and views do not talk to each other directly; controllers sit squarely in the middle of everything, receiving messages from some objects and dispatching instructions to others.

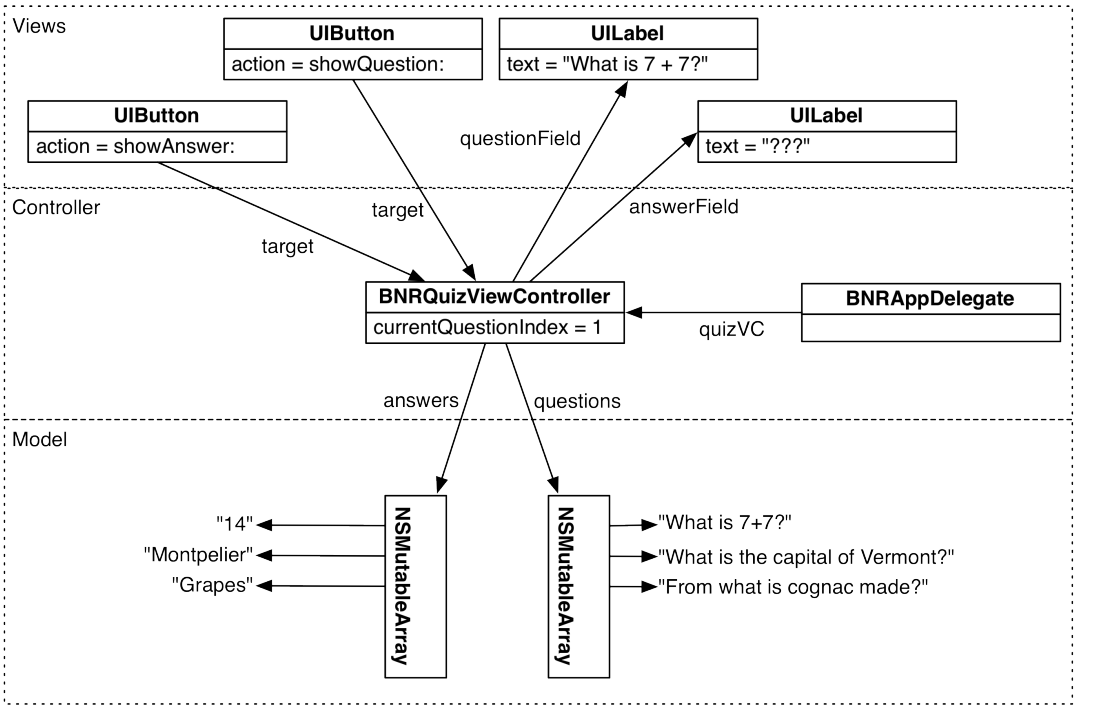
Designing Quiz

You are going to write the Quiz application using the MVC pattern. Here is a break down of the objects you will be creating and working with:

- 4 view objects: two instances of **`UILabel`** and two instances of **`UIButton`**
- 2 controller objects: an instance of **`BNRAppDelegate`** and an instance of **`BNRQuizViewController`**
- 2 model objects: two instances of **`NSMutableArray`**

These objects and their relationships are laid out in the object diagram for Quiz shown in Figure 1.7.

Figure 1.7 Object diagram for Quiz



This diagram is the big picture of how the finished Quiz application will work. For example, when the Show Question button is tapped, it will trigger a *method* in the **`BNRQuizViewController`**. A method is a lot like a function – a list of instructions to be executed. This method will retrieve a new question from the array of questions and ask the top label to display that question.

It is okay if this object diagram does not make sense yet; it will by the end of the chapter. Refer back to it as you continue to see how the app is taking shape.

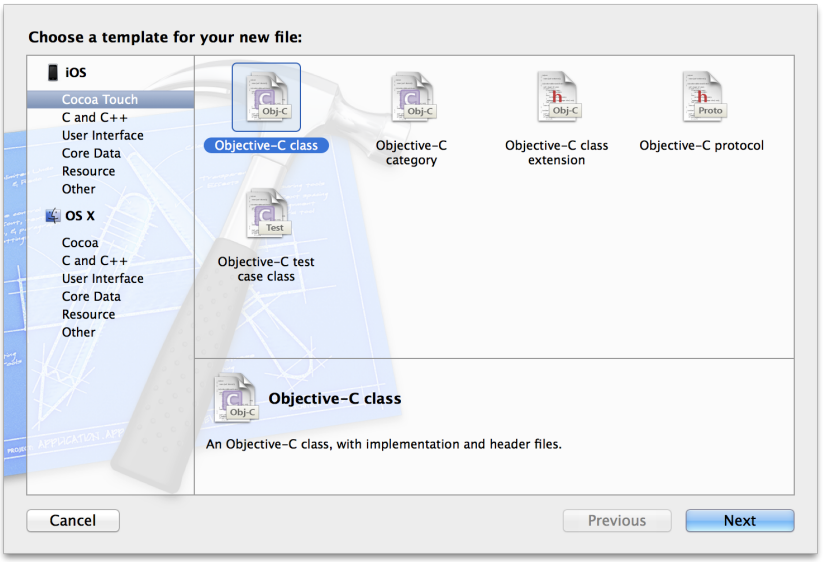
You are going to build Quiz in steps, starting with the controller object that sits in the middle of the app – **`BNRQuizViewController`**.

Creating a View Controller

The **`BNRAppDelegate`** class was created for you by the Empty Application template, but you will have to create the **`BNRQuizViewController`** class. We will talk more about classes in Chapter 2 and more about view controllers in Chapter 6. For now, just follow along.

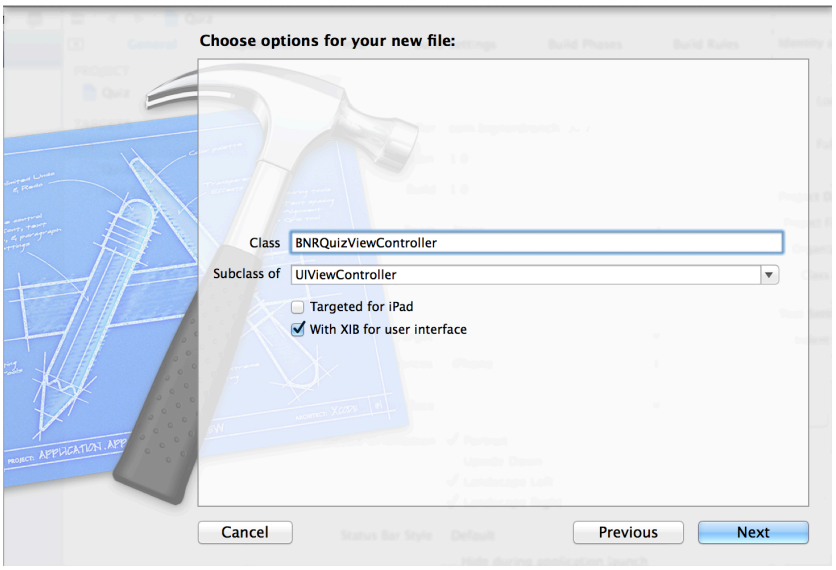
From the File menu, select New → File.... A sheet will slide down asking what type of file you would like to create. On the lefthand side under the iOS section, select Cocoa Touch. Then choose Objective-C Class and click Next.

Figure 1.8 Creating an Objective-C class



On the next sheet, name the class **`BNRQuizViewController`**. For the Subclass of field, click the drop-down menu arrow and select **`UIViewController`**. Select the With XIB for user interface checkbox.

Figure 1.9 Creating a view controller

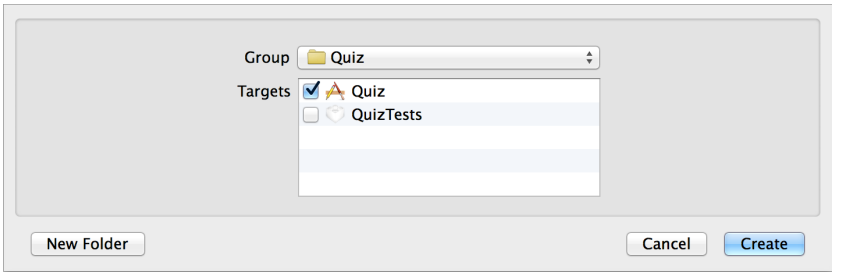


Click Next, and a panel will drop down that prompts you to create the files for this new class. When creating a new class for a project, you want to save the files that describe it inside the project's source

directory on the filesystem. By default, the current project directory is already selected for you. You can also choose the group in the project navigator that these files will be added to. Because these groups are simply for organizing and because this project is very small, just stick with the default.

Make sure the checkbox is selected for the Quiz target. This ensures that the **`BNRQuizViewController`** class will be compiled when the Quiz project is built. Click Create.

Figure 1.10 Quiz target is selected



Building an Interface

In the project navigator, find the class files for **`BNRQuizViewController`**. When you created this class, you checked the box for With XIB for user interface, so **`BNRQuizViewController`** came with a third class file: `BNRQuizViewController.xib`. Find and select `BNRQuizViewController.xib` in the project navigator to open it in the editor area.

When Xcode opens a XIB (pronounced "zib") file, it opens it with Interface Builder, a visual tool where you can add and arrange objects to create a graphical user interface. In fact, XIB stands for XML Interface Builder.

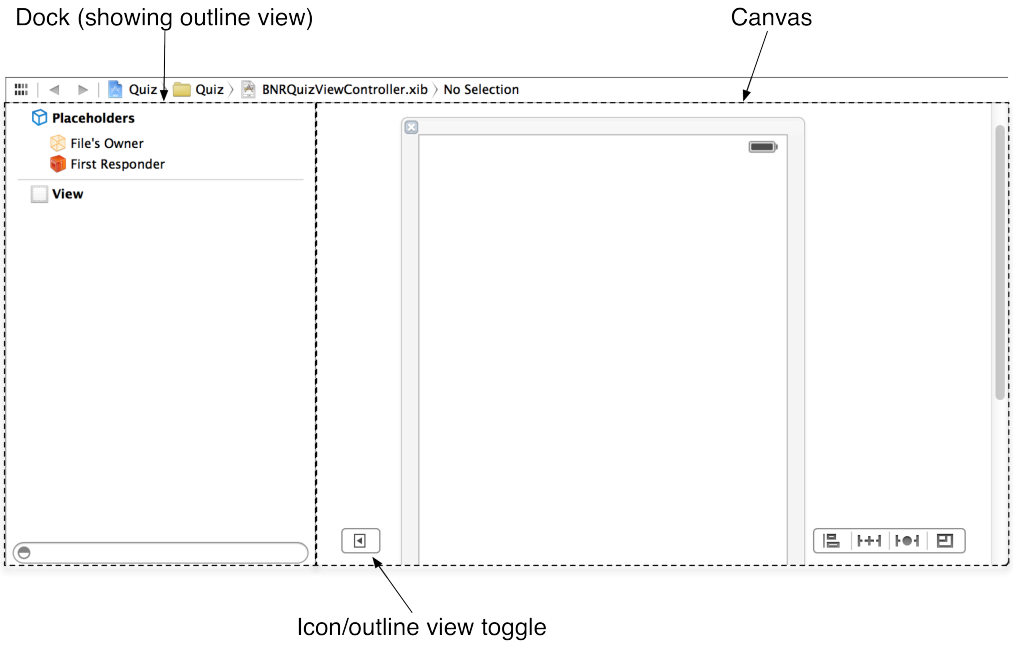
In many GUI builders on other platforms, you describe what you want an application to look like and then press a button to generate a bunch of code. Interface Builder is different. It is an object editor: you create and configure objects, like buttons and labels, and then save them into an archive. The archive is the XIB file.

Interface Builder divided the editor area into two sections: the *dock* is on the lefthand side and the *canvas* is on the right.

The dock lists the objects in the XIB file either as icons (*icon view*) or in words (*outline view*). The icon view is useful when screen real estate is running low. However, for learning purposes, it is easier to see what is going on in the outline view.

If the dock is in icon view, click the disclosure button in the bottom lefthand corner of the canvas to reveal the outline view (Figure 1.11).

Figure 1.11 Editing a XIB file in Interface Builder



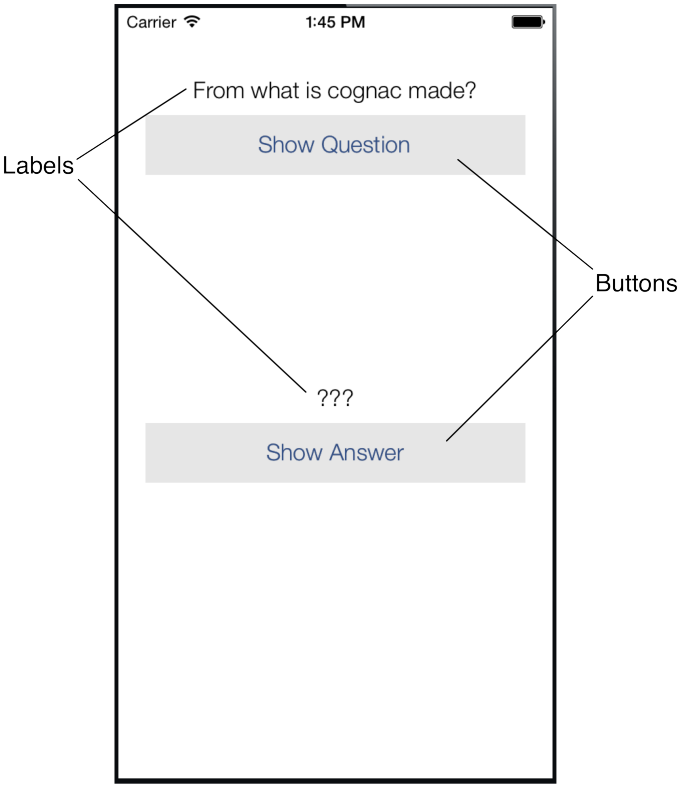
The outline view tells you that `BNRQuizViewController.xib` contains three objects: two placeholders and a View. Ignore the placeholders for now; you will learn about them later.

The View object is an instance of **`UIView`**. This object forms the foundation of your user interface and you can see it displayed on the canvas. The canvas shows how your user interface will appear in the application.

Click on the View object in the document outline to select it in the canvas. You can move the view by dragging it around. Note that moving the view does not change anything about the actual object; it just re-organizes the canvas. You can also close the view by clicking the x in its top left corner. Again, this does not delete the view; it just removes it from the canvas. You can get it back by selecting it in the outline view.

Right now, your interface consists solely of this view object. You need to add four additional view objects for Quiz: two labels and two buttons.

Figure 1.12 Labels and buttons needed



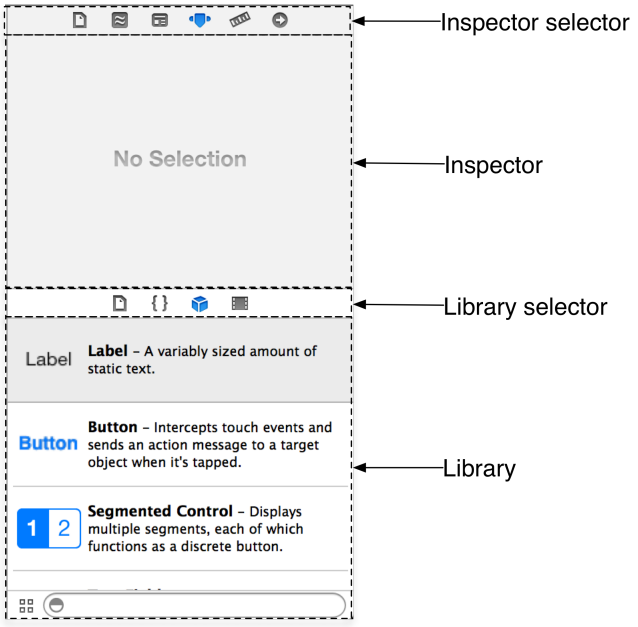
Creating view objects

To add these view objects, you need to get to the object library in the *utility area*.

The utility area is to the right of the editor area and has two sections: the *inspector* and the *library*. The top section is the inspector, which contains settings for the file or object that is selected in the editor area. The bottom section is the library, which lists items that you can add to a file or project.

At the top of each section is a selector for different types of inspectors and libraries (Figure 1.13). From the library selector, select the tab to reveal the *object library*.

Figure 1.13 Xcode utility area



The object library contains the objects that you can add to a XIB file to compose your interface. Find the Label object. (It may be right at the top; if not, scroll down the list or use the search bar at the bottom of the library.) Select this object in the library and drag it onto the view object on the canvas. Position this label in the center of the view, near the top. Drag a second label onto the view and position it in the center, closer to the bottom.

Next, find Button in the object library and drag two buttons onto the view. Position one below each label.

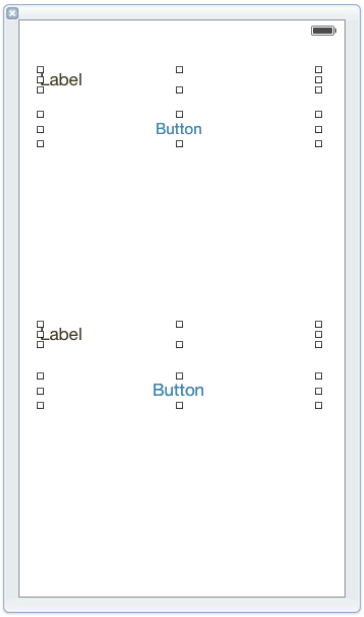
You have now created four view objects and added them to **`BNRQuizViewController`**'s user interface. Confirm this in the document outline.

Configuring view objects

Now that you have created the view objects, you can configure their attributes. Some attributes, like size, position, and text, can be changed directly on the canvas. Others must be changed in the *attributes inspector*, a tool that you will use shortly.

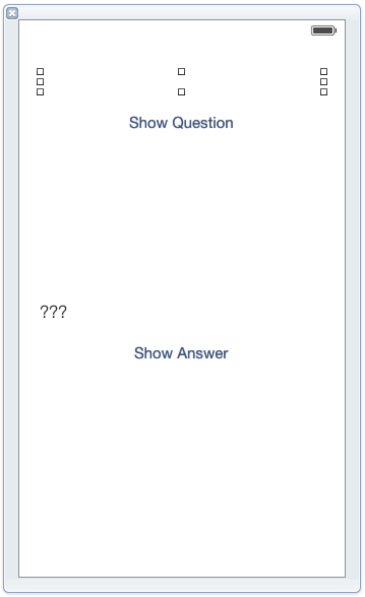
You can resize an object by selecting it on the canvas or in the outline view and then dragging its corners and edges in the canvas. Resize all four of your view objects to span most of the window.

Figure 1.14 Stretching the labels and buttons



You can edit the title of a button or a label by double-clicking it and typing in new text. Change the top button's title to Show Question and the bottom button's title to Show Answer. Change the bottom label to display ???. Delete the text in the top label and leave it blank. (Eventually, this label will display the question to the user.) Your interface should look like Figure 1.15.

Figure 1.15 Setting the text on the labels and buttons

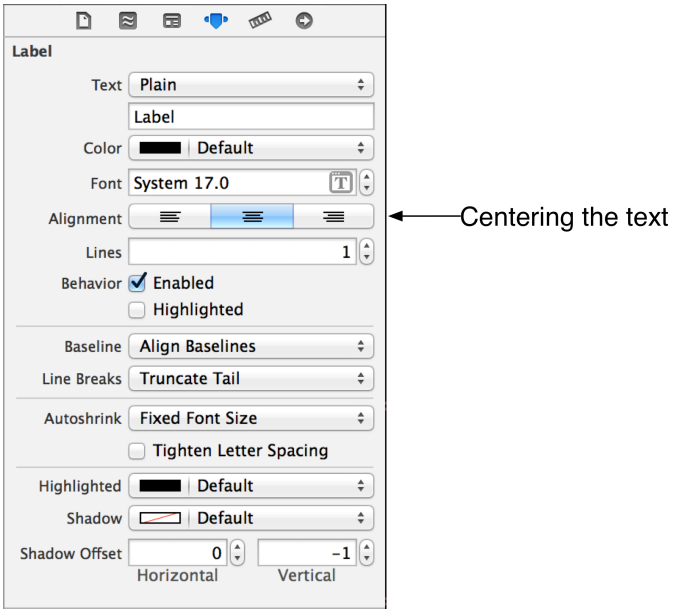


It would be nice if the text in the labels was centered. Setting the text alignment of a label must be done in the attributes inspector.

Select the tab to reveal the attributes inspector. Then select the bottom label on the canvas.

In the attributes inspector, find the segmented control for alignment. Select the centered text option, as shown in Figure 1.16.

Figure 1.16 Centering the label text



Back on the canvas, notice that the ??? is now centered in the bottom label. Select the top label in the canvas and return to the attributes inspector to set its text alignment. (This label has no text to display now, but it will in the running application.)

To inform the user where they are able to tap, you can change the background color of the buttons. Select the Show Question button on the canvas.

In the attributes inspector, scroll down until you see the attributes under the View header. Next to the Background label, click on the color (the white box with a red slash). This will bring up the full color picker. Pick a nice color to go with the button's blue text.

Do the same for the second button, but instead of clicking on the color on the left side, click on the right side which has text and the up and down arrows. This will bring up a list of recently used colors in chronological order as well as some system default colors. Use this to choose the same color for the second button's background color.

NIB files

At this point, you may be wondering how these objects are brought to life when the application is run.

When you build an application that uses a XIB file, the XIB file is compiled into a NIB file that is smaller and easier for the application to parse. Then the NIB file is copied into the application's

bundle. The bundle is a directory containing the application's executable and any resources the executable uses.

At runtime, the application will read in, or load, the NIB file when its interface is needed. Quiz only has one XIB file and thus will have only one NIB file in its bundle. Quiz's single NIB file will be loaded when the application first launches. A complex application, however, will have many NIB files that are loaded as they are needed. You will learn more about how NIB files are loaded in Chapter 6.

Your application's interface now looks like it should. But to begin making it functional, you need to make some connections between these view objects and the **`BNRQuizViewController`** that will be running the show.

Making connections

A *connection* lets one object know where another object is in memory so that the two objects can communicate. There are two kinds of connections that you can make in Interface Builder: outlets and actions. An *outlet* points to an object. (If you are not familiar with "pointers," you will learn about them in Chapter 2.) An *action* is a method that gets triggered by a button or some other view that the user can interact with, like a slider or a picker.

Let's start by creating outlets that point to the instances of **`UILabel`**. Time to leave Interface Builder briefly and write some code.

Declaring outlets

In the project navigator, find and select the file named `BNRQuizViewController.m`. The editor area will change from Interface Builder to Xcode's code editor.

In `BNRQuizViewController.m`, delete any code that the template added between the `@implementation` and `@end` directives so that the file looks like this:

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController ()

@end

@implementation BNRQuizViewController

@end
```

Next, add the following code. Do not worry about understanding it right now; just get it in.

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController ()

@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation

@end
```

Notice the bold type? In this book, code that you need to type in is always bold; the code that is not bold provides context for where to add the new stuff.

In this new code, you declared two properties. You will learn about properties in Chapter 3. For now, focus on the second half of the first line.

```
@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
```

This code gives every instance of **`BNRViewController`** an outlet named `questionLabel`, which it can use to point to a **`UILabel`** object. The `IBOutlet` keyword tells Xcode that you will set this outlet using Interface Builder.

Setting outlets

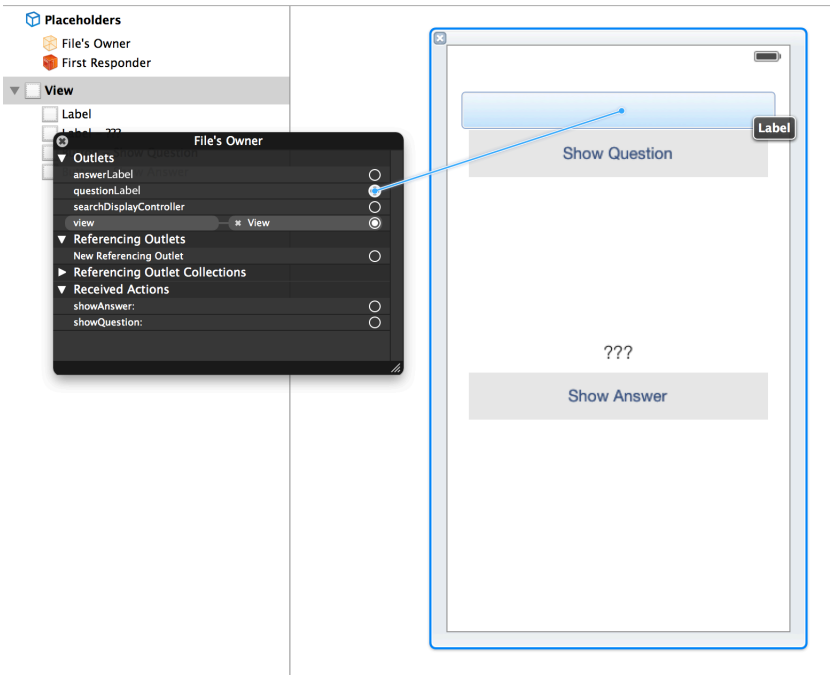
In the project navigator, select `BNRQuizViewController.xib` to reopen Interface Builder.

You want the `questionLabel` outlet to point to the instance of **`UILabel`** at the top of the user interface.

In the dock, find the Placeholders section and the File's Owner object. A placeholder stands in for another object in the running application. In your case, the File's Owner stands in for an instance of **`BNRQuizViewController`**, which is the object responsible for managing the interface defined in `BNRQuizViewController.xib`.

In the dock, right-click or Control-click on the File's Owner to bring up the connections panel (Figure 1.17). Then drag from the circle beside `questionLabel` to the **`UILabel`**. When the label is highlighted, release the mouse button, and the outlet will be set.

Figure 1.17 Setting `questionLabel`

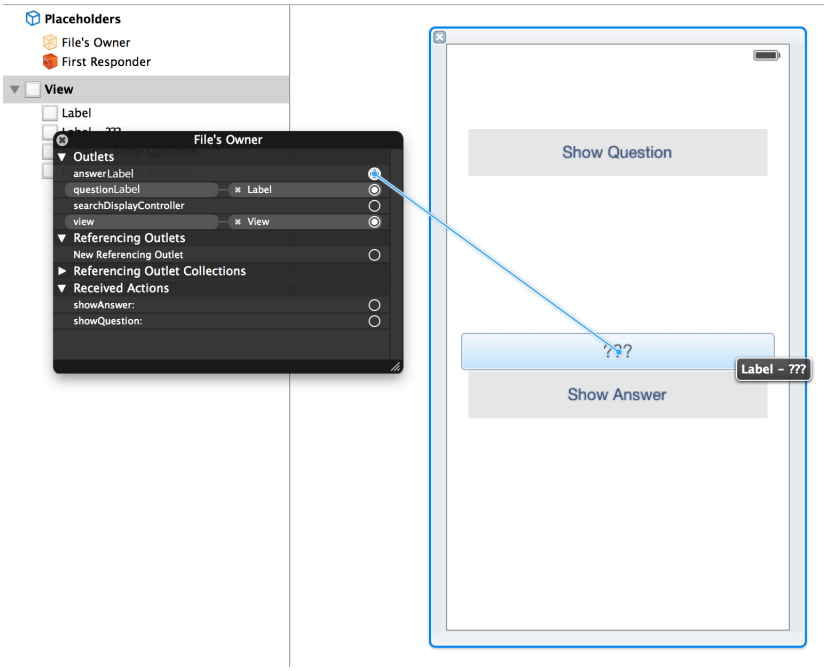


(If you do not see `questionLabel` in the connections panel, double-check your `BNRQuizViewController.m` file for typos.)

Now when the NIB file is loaded, the **`BNRQuizViewController`**'s `questionLabel` outlet will automatically point to the instance of **`UILabel`** at the top of the screen. This will allow the **`BNRQuizViewController`** to tell this label what question to display.

Set the `answerLabel` outlet the same way: drag from the circle beside `answerLabel` to the bottom **`UILabel`** (Figure 1.18).

Figure 1.18 Setting `answerLabel`



Notice that you drag *from* the object with the outlet that you want to set *to* the object that you want that outlet to point to.

Your outlets are all set. The next connections you need to make involve the two buttons.

When a **`UIButton`** is tapped, it sends a message to another object. The object that receives the message is called the *target*. The message that is sent is called the *action*. This action is the name of the method that contains the code to be executed in response to the button being tapped.

In your application, the target for both buttons will be the instance of **`BNRQuizViewController`**. Each button will have its own action. Let's start by defining the two action methods: **`showQuestion:`** and **`showAnwer:`**

Defining action methods

Return to `BNRQuizViewController.m` and add the following code in between `@implementation` and `@end`.

```
@implementation

- (IBAction)showQuestion:(id)sender
{

}

- (IBAction)showAnswer:(id)sender
{

}

@end
```

You will flesh out these methods after you make the target and action connections. The `IBAction` keyword tells Xcode that you will be making these connections in Interface Builder.

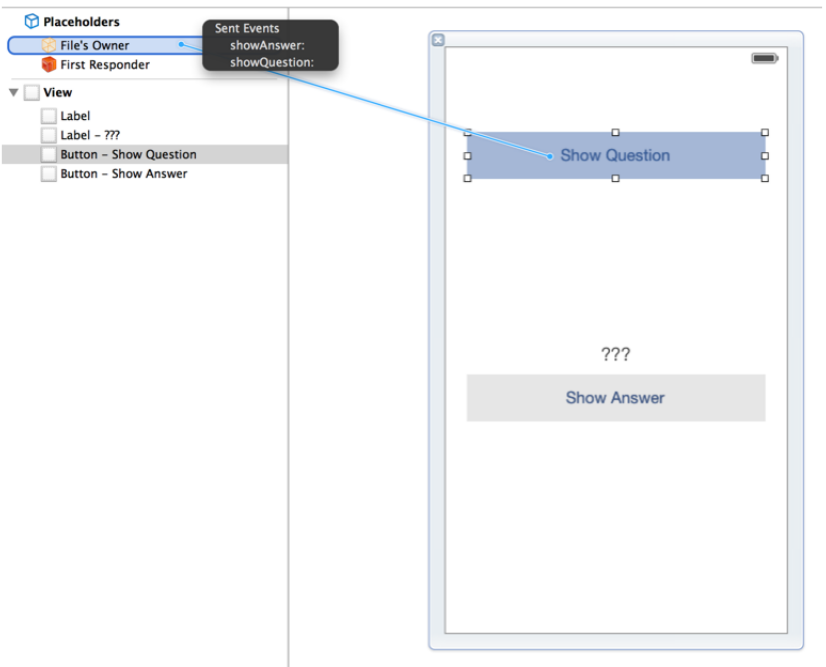
Setting targets and actions

To set an object's target, you Control-drag *from* the object *to* its target. When you release the mouse, the target is set, and a pop-up menu appears that lets you select an action.

Let's start with the Show Question button. You want its target to be **`BNRQuizViewController`** and its action to be **`showQuestion:`**.

Reopen `BNRQuizViewController.xib`. Select the Show Question button in the canvas and Control-drag (or right-click and drag) to the File's Owner. When the File's Owner is highlighted, release the mouse button and choose **`showQuestion:`** from the pop-up menu, as shown in Figure 1.19.

Figure 1.19 Setting Show Question target/action



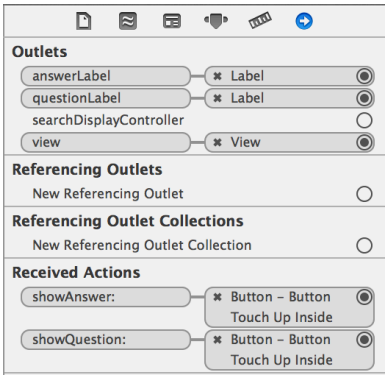
Now for the Show Answer button. Select the button and Control-drag from the button to the File's Owner. Then choose **`showAnswer:`** from the pop-up menu.

Summary of connections

There are now five connections between your **`BNRQuizViewController`** and the view objects. You have set the pointers `answerLabel` and `questionLabel` to point at the label objects – two connections. The **`BNRQuizViewController`** is the target for both buttons – two more. The project's template made one additional connection: the `view` pointer of **`BNRQuizViewController`** is connected to the View object that represents the background of the application. That makes five.

You can check these connections in the *connections inspector*. Select the File's Owner in the outline view. Then in the inspector, click the tab to reveal the connections inspector. (Figure 1.20).

Figure 1.20 Checking connections in the inspector



Your XIB file is complete. The view objects have been created and configured, and all the necessary connections have been made to the controller object. Let's move on to creating and connecting your model objects.

Creating Model Objects

View objects make up the user interface, so developers typically create, configure, and connect view objects using Interface Builder. Model objects, on the other hand, are set up in code.

In the project navigator, select `BNRQuizViewController.m`. Add the following code that declares an integer and pointers to two arrays.

```
@interface BNRQuizViewController ()

@property (nonatomic) int currentQuestionIndex;

@property (nonatomic, copy) NSArray *questions;
@property (nonatomic, copy) NSArray *answers;

@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation

@end
```

The arrays will be ordered lists containing questions and answers. The integer will keep track of what question the user is on.

These arrays need to be ready to go at the same time that the interface appears to the user. To make sure that this happens, you are going to create the arrays right after an instance of **BNRQuizViewController** is created.

When an instance of **BNRQuizViewController** is created, it is sent the message **initWithNibName:bundle:**. In BNRQuizViewController.m, implement the **initWithNibName:bundle:** method.

```
...

@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation BNRQuizViewController

- (instancetype)initWithNibName:(NSString *)nibNameOrNil
                         bundle:(NSBundle *)nibBundleOrNil
{
    // Call the init method implemented by the superclass
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];

    if (self) {
        // Create two arrays filled with questions and answers
        // and make the pointers point to them

        self.questions = @[@"From what is cognac made?",
                           @"What is 7+7?",
                           @"What is the capital of Vermont?"];

        self.answers = @[@"Grapes",
                         @"14",
                         @"Montpelier"];
    }

    // Return the address of the new object
    return self;
}

@end
```

(Scary syntax? Feelings of dismay? Do not panic – you will learn more about the Objective-C language in the next two chapters.)

Using code-completion

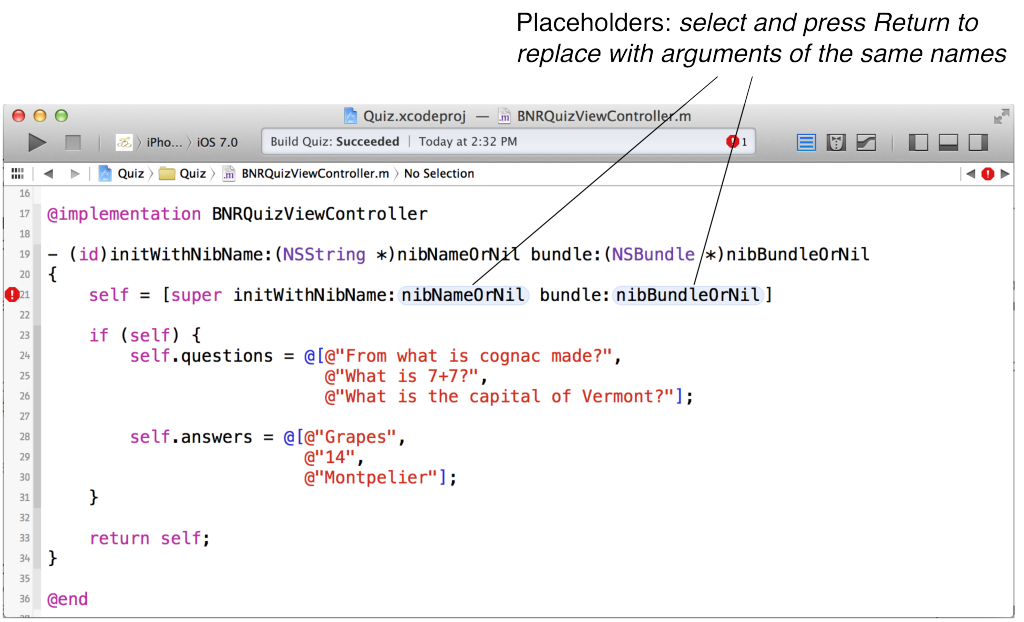
As you work through this book, you will type a lot of code. Notice that as you were typing this code, Xcode was ready to fill in parts of it for you. For example, when you started typing **`initWithNibName:bundle:`**, it suggested this method before you could finish. You can press the Enter key to accept Xcode's suggestion or select another suggestion from the pop-up box that appears.

When you accept a code-completion suggestion for a method that takes arguments, Xcode puts *placeholders* in the areas for the arguments. (Note that this use of the term "placeholder" is completely distinct from the placeholder objects that you saw in the XIB file.)

Placeholders are not valid code, and you have to replace them to build the application. This can be confusing because placeholders often have the same names that you want your arguments to have. So the text of the code looks completely correct, but you get an error.

Figure 1.21 shows two placeholders you might have seen when typing in the previous code.

Figure 1.21 Code-completion placeholders and errors



See the `nibNameOrNil` and `nibBundleOrNil` in the first line of the implementation of **`initWithNibName:bundle:`**? Those are placeholders. You can tell because they are inside slightly-shaded, rounded rectangles. The fix is to select the placeholders, type the correct argument name, and press the Enter key. The rounded rectangles will go away, and your code will be correct and valid.

Because the placeholders have the correct text in this case, you can simply select a placeholder and press Enter to have Xcode replace it with an argument of the same name.

When using code-completion, watch out for names that look *almost* like what you want. Cocoa Touch uses naming conventions which often cause distinct methods, types, and variables to have very similar names. Thus, do not blindly accept the first suggestion Xcode gives you without verifying it. Always double-check.

Pulling it all Together

You have created, configured, and connected your view objects and their view controller. You have created the model objects. Two things are left to do for Quiz:

- finish implementing the action methods **`showQuestion:`** and **`showAnswer:`** in **`BNRQuizViewController`**
- add code to **`BNRAppDelegate`** that will create and display an instance of **`BNRQuizViewController`**

Implementing action methods

In `BNRQuizViewController.m`, finish the implementations of **`showQuestion:`** and **`showAnswer:`**.

```
    ...
    // Return the address of the new object
    return self;
}

- (IBAction)showQuestion:(id)sender
{
    // Step to the next question
    self.currentQuestionIndex++;

    // Am I past the last question?
    if (self.currentQuestionIndex == [self.questions count]) {

        // Go back to the first question
        self.currentQuestionIndex = 0;
    }

    // Get the string at that index in the questions array
    NSString *question = self.questions[self.currentQuestionIndex];

    // Display the string in the question label
    self.questionLabel.text = question;

    // Reset the answer label
    self.answerLabel.text = @"???";
}

- (IBAction)showAnswer:(id)sender
{
    // What is the answer to the current question?
    NSString *answer = self.answers[self.currentQuestionIndex];

    // Display it in the answer label
    self.answerLabel.text = answer;
}

@end
```

Getting the view controller on the screen

If you were to run Quiz right now, you would not see the interface that you created in `BNRQuizViewController.xib`. Instead, you would see a blank white screen. To get your interface on screen, you have to connect your view controller with the application's other controller – **`BNRAppDelegate`**.

Whenever you create an iOS application using an Xcode template, an *app delegate* is created for you. The app delegate is the starting point of an application, and every iOS application has one.

The app delegate manages a single top-level **`UIWindow`** for the application. To get the **`BNRQuizViewController`** on screen, you need to make it the *root view controller* of this window.

When an iOS application is launched, it is not immediately ready for the user. There is some setup that goes on behind the scenes. Right before the app is ready for the user, the app delegate receives the message **`application:didFinishLaunchingWithOptions:`**. This is your chance to prepare the application for action. In particular, you want to make sure that your interface is ready before the user has a chance to interact with it.

In the project navigator, find and select `BNRAppDelegate.m`. Add the following code to the **`application:didFinishLaunchingWithOptions:`** method to create an instance of **`BNRQuizViewController`** and to set it as the root view controller of the app delegate's window.

```
#import "BNRAppDelegate.h"
#import "BNRQuizViewController.h"

- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    BNRQuizViewController *quizVC = [[BNRQuizViewController alloc] init];
    self.window.rootViewController = quizVC;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Now whenever the app is launched, an instance of **`BNRQuizViewController`** will be created. This instance will then receive the **`initWithNibName:bundle:`** message, which will trigger loading the NIB file compiled from `BNRQuizViewController.xib` and the creation of the model objects.

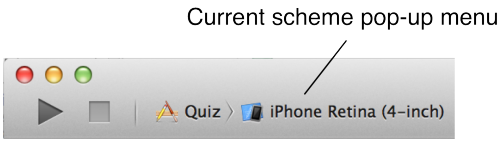
Your Quiz is complete. Time to try it out.

Running on the Simulator

First, you are going to run Quiz on Xcode's iOS simulator. Later, you will see how to run it on an actual device.

To prepare Quiz to run on the simulator, find the current scheme pop-up menu on the Xcode toolbar (Figure 1.22).

Figure 1.22 iPhone Retina (4-inch) scheme selected



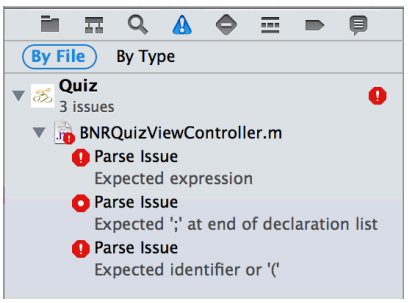
If it says something generic like iPhone Retina (4-inch), then the project is set to run on the simulator and you are good to go. If it says something like Christian's iPhone, then click it and choose iPhone Retina (4-inch) from the pop-up menu.

For this book, use the iPhone Retina (4-inch) scheme. The only difference between that and iPhone Retina (3.5-inch) is the height of the screen. If you select the 3.5-inch simulator, parts of your user interface might get cut off. We will discuss how to scale interfaces for both iPhone screen sizes (and iPad as well) in Chapter 15.

Next, click the iTunes-esque play button in the toolbar. This will build (compile) and then run the application. You will be doing this often enough that you may want to learn and use the keyboard shortcut Command-R.

If building turns up any errors, you can view them in the *issue navigator* by selecting the ⚠ tab in the navigator area (Figure 1.23).

Figure 1.23 Issue navigator with example errors and warnings



You can click on any error or warning in the issue navigator to be taken to the file and the line of code where the issue occurred. Find and fix any problems (i.e., code typos!) by comparing your code with the book's. Then try running the application again. Repeat this process until your application compiles.

Once your application has compiled, it will launch in the iOS simulator. If this is the first time you have used the simulator, it may take a while for Quiz to appear.

Play around with the Quiz application. You should be able to tap the Show Question button and see a new question in the top label; tapping Show Answer should show the right answer. If your application is not working as expected, double-check your connections in `BNRQuizViewController.xib`.

Deploying an Application

Now that you have written your first iOS application and run it on the simulator, it is time to deploy it to a device.

To install an application on your development device, you need a developer certificate from Apple. Developer certificates are issued to registered iOS Developers who have paid the developer fee. This certificate grants you the ability to sign your code, which allows it to run on a device. Without a valid certificate, devices will not run your application.

Apple's Developer Program Portal (`http://developer.apple.com`) contains all the instructions and resources to get a valid certificate. The interface for the set-up process is continually being updated by Apple, so it is fruitless to describe it here in detail. Instead, visit our guide at `http://www.bignerdranch.com/iOS_device_provisioning` for instructions.

If you are curious about what exactly is going on here, there are four important items in the provisioning process:

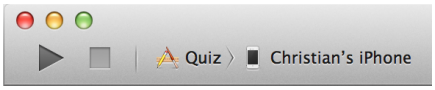
Developer Certificate	This certificate file is added to your Mac's keychain using Keychain Access. It is used to digitally sign your code.
App ID	The application identifier is a string that uniquely identifies your application on the App Store. Application identifiers typically look like this: `com.bignerdranch.AwesomeApp`, where the name of the application follows the name of your company. The App ID in your provisioning profile must match the *bundle identifier* of your application. A development profile can have a wildcard character (*) for its App ID and therefore will match any bundle identifier. To see the bundle identifier for the Quiz application, select the project in the project navigator. Then select the Quiz target and the General pane.
Device ID (UDID)	This identifier is unique for each iOS device.
Provisioning Profile	This is a file that lives on your development device and on your computer. It references a Developer Certificate, a single App ID, and a list of the device IDs for the devices that the application can be installed on. This file is suffixed with `.mobileprovision`.

When an application is deployed to a device, Xcode uses a provisioning profile on your computer to access the appropriate certificate. This certificate is used to sign the application binary. Then, the development device's UDID is matched to one of the UDIDs contained within the provisioning profile, and the App ID is matched to the bundle identifier. The signed binary is then sent to your development device, where it is confirmed by the same provisioning profile on the device and, finally, launched.

Open Xcode and plug your development device (iPhone, iPod touch, or iPad) into your computer. This should automatically open the Organizer window, which you can re-open at any time using the Window menu's Organizer item. In the Organizer window, you can select the Devices tab to view all of the provisioning information.

To run the Quiz application on your device, you must tell Xcode to deploy to the device instead of the simulator. Return to the current scheme description in Xcode's toolbar. Click the description and then choose iOS Device from the pop-up menu (Figure 1.24). If iOS Device is not an option, find the choice that reads something like Christian's iPhone.

Figure 1.24 Choosing the device



Build and run your application (Command-R), and it will appear on your device.

Application Icons

While running the Quiz application (on your development device or the simulator), return to the device's Home screen. You will see that its icon is a boring, default tile. Let's give Quiz a better icon.

An *application icon* is a simple image that represents the application on the iOS home screen. Different devices require different sized icons, and these requirements are shown in Table 1.1.

Table 1.1 Application icon sizes by device

Device	Application icon sizes
iPhone / iPod touch (iOS 7)	120x120 pixels (@2x)
iPhone / iPod touch (iOS 6 and earlier)	57x57 pixels 114x114 pixels (@2x)
iPad (iOS 7 and earlier)	72x72 pixels 144x144 pixels (@2x)

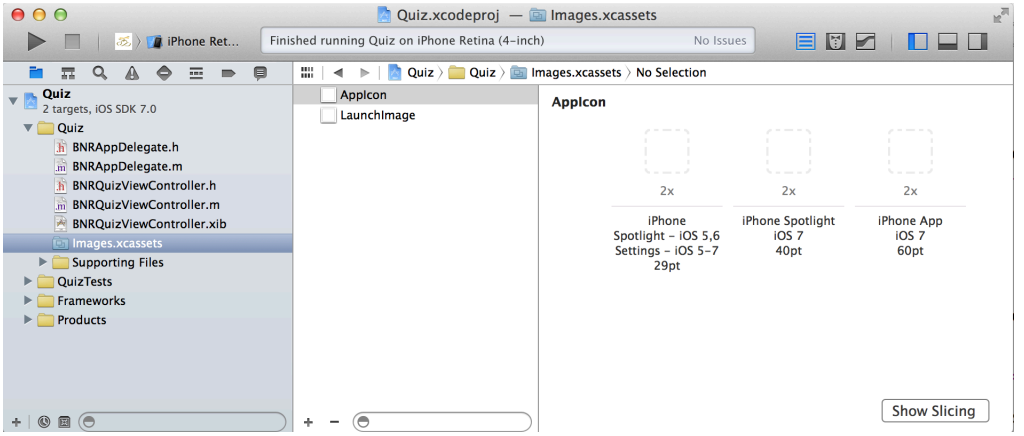
Every application you deploy to the App Store is required to have icons for every device class on which it can run. For example, if you are supporting just iPhone and iPod touch running iOS 7 or later, then you just need to supply one image (at the resolution listed above). On the other extreme, if you have a universal application which supports iOS 6 and above, you will need to have five different resolution app icons, the two necessary for iPad and the three others necessary for iPhone and iPod touch.

We have prepared an icon image file (size 120x120) for the Quiz application. You can download this icon (along with resources for other chapters) from `http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip`. Unzip `iOSProgramming4ed.zip` and find the `Icon@2x.png` file in the `Resources` directory of the unzipped folder.

You are going to add this icon to your application bundle as a *resource*. In general, there are two kinds of files in an application: code and resources. Code (like `BNRQuizViewController.h` and `BNRQuizViewController.m`) is used to create the application itself. Resources are things like images and sounds that are used by the application at runtime. XIB files, which are compiled into NIB files that are read in at runtime, are also resources.

In the project navigator, find `Images.xcassets`. Select this file to open it and then select AppIcon from the resource list on the left side.

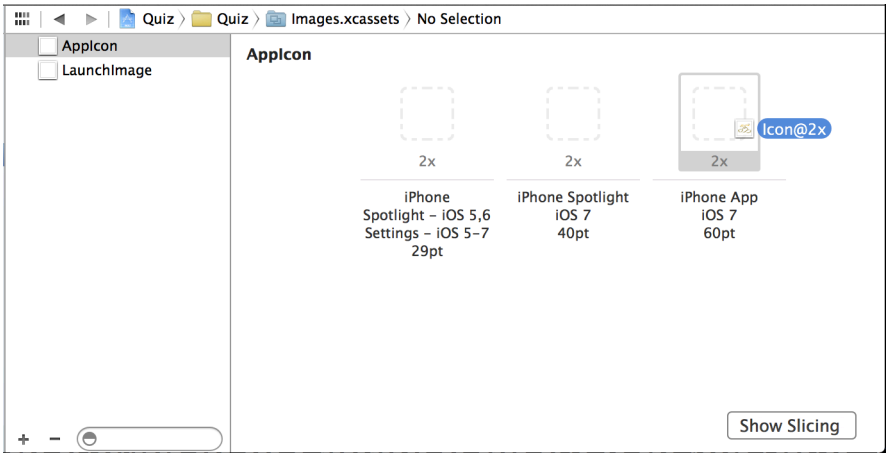
Figure 1.25 Showing the Asset Catalog



This panel is the *Asset Catalog* where you can manage all of the various images that your application will need.

Drag the `Icon@2x.png` file from Finder onto the margins of the AppIcon section. This will copy the file into your project's directory on the filesystem and add a reference to that file in the Asset Catalog. (You can Control-click on a file in the Asset Catalog and select the option to Show in Finder to confirm this.)

Figure 1.26 Adding the App Icon to the Asset Catalog



Build and run the application again. Exit the application and look for the Quiz application with the BNR logo.

(If you do not see the icon, delete the application and then build and run again to redeploy it. On a device, do this as you would any other application. On the simulator, the easiest option is to reset the simulator. With the simulator open, find its menu bar. Select iOS Simulator and then Reset Content and Settings.... This will remove all applications and reset the simulator to its default settings. You should see the app icon the next time you run the application.)

Launch Images

Another resource that you can manage from the Asset Catalog is the application launch image, which appears while an application is loading. The launch image has a specific role on iOS: it conveys to the user that the application is indeed launching and depicts the user interface that the user will interact with once the application has finished launching. Therefore, a good launch image is a content-less screenshot of the application. For example, the Clock application's launch image shows the four tabs along the bottom, all in the unselected state. Once the application loads, the correct tab is selected and the content becomes visible. (Keep in mind that the launch image is replaced after the application has launched; it does not become the background image of the application.)

In the `Resources` directory where you found the app icons, there are two launch images: `Default@2x.png` and `Default-568h@2x.png`. Open the LaunchImage item in the Asset Catalog, and then drag these images in to the Asset Catalog like you did with the app icons.

Build and run the application. As the application launches, you will briefly see the launch image.

Why two launch images? A launch image must fit the screen of the device it is being launched on. Therefore you need one launch image for 3.5 inch Retina displays, and another launch image for 4 inch Retina displays. Note that if you are supporting iOS6 or earlier on iPhone and iPod touch, you will need to include a third non-Retina launch image. Table 1.2 shows the different size images you will need for each type of device.

Table 1.2 Launch image sizes by device

Device	**Launch image size**
iPhone/iPod touch without Retina display	320x480 pixels (Portrait only)
iPhone/iPod touch with Retina display (3.5 inch)	640x960 pixels (Portrait only)
iPhone/iPod touch with Retina display (4 inch)	640x1136 pixels (Portrait only)
iPad without Retina display	768x1024 pixels (Portrait) 1024x768 pixels (Landscape)
iPad with Retina display	1536x2048 pixels (Portrait) 2048x1536 pixels (Landscape)

(Note that Table 1.2 lists the screen resolutions of the devices; the real status bar is overlaid on top of the launch image.)

Congratulations! You have written your first application and installed it on your device. Now it is time to dive into the big ideas that make it work.

2
Objective-C

iOS applications are written in the Objective-C language using the Cocoa Touch frameworks. Objective-C is an extension of the C language, and the Cocoa Touch frameworks are collections of Objective-C classes.

In this chapter, you will learn the basics of Objective-C and create an application called RandomItems. Even if you are familiar with Objective-C, you should still go through this chapter to create the **`BNRItem`** class that you will use later in the book.

This book assumes you know some C and understand the basic ideas of object-oriented programming. If C or object-oriented programming makes you uneasy, we recommend starting with *Objective-C Programming: The Big Nerd Ranch Guide*.

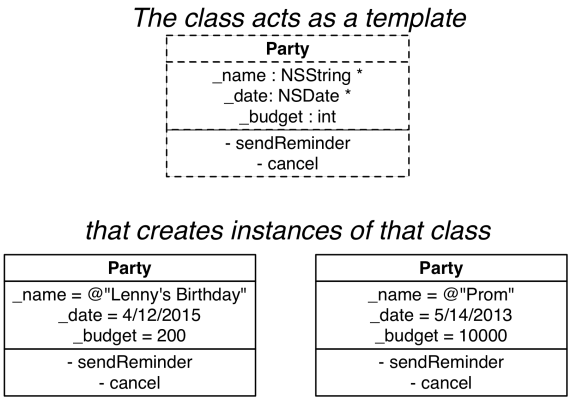
Objects

Let's say you need a way to represent a party. Your party has a few attributes that are unique to it, like a name, a date, and a list of invitees. You can also ask the party to do things, like send an email reminder to all the invitees, print name tags, or cancel the party altogether.

In C, you would define a structure to hold the data that describes a party. The structure would have data members – one for each of the party's attributes. Each data member would have a name and a type. To create an individual party, you would use the function **`malloc`** to allocate a chunk of memory large enough to hold the structure.

In Objective-C, instead of using a structure to represent a party, you use a *class*. A class is like a cookie-cutter that produces objects. The **`Party`** class creates objects, and these objects are instances of the **`Party`** class. Each instance of the **`Party`** class can hold the data for a single party (Figure 2.1).

Figure 2.1 A class and its instances



An instance of **`Party`**, like any object, is a chunk of data in memory, and it stores the values for its attributes in *instance variables*. (You also may see these referred to as "ivars" in some places.) In Objective-C, we typically put an underscore at the beginning of the instance variable name. So an instance of **`Party`** might have the instance variables `_name`, `_date`, and `_budget`.

A C structure is a chunk of memory, and an object is a chunk of memory. A C structure has data members, each with a name and a type. Similarly, an object has instance variables, each with a name and a type.

The important difference between a C structure and an Objective-C class is that a class has *methods*. A method is similar to a function: it has a name, a return type, and a list of parameters that it expects. A method also has access to an object's instance variables. If you want an object to run the code in one of its methods, you send that object a *message*.

Using Instances

To use an instance of a class, you must have a variable that points to that object. A pointer variable stores the location of an object in memory, not the object itself. (It "points to" the object.) A variable that points to a **`Party`** object is declared like this:

```
Party *partyInstance;
```

Creating this pointer does not create a **`Party`** object – only a variable that can point to a **`Party`** object.

This variable is named `partyInstance`. Notice that this variable's name does not start with an underscore; it is not an instance variable. It is meant to be a pointer to an instance of **`Party`**.

Creating objects

An object has a life span: it is created, sent messages, and then destroyed when it is no longer needed.

To create an object, you send an **`alloc`** message to a class. In response, the class creates an object in memory (on the heap, just like **`malloc()`** would) and gives you the address of the object, which you store in a variable:

```
Party *partyInstance = [Party alloc];
```

You create a pointer to an instance so that you can send messages to it. The first message you *always* send to a newly allocated instance is an initialization message. Although sending an **`alloc`** message to a class creates an instance, the instance is not ready for work until it has been initialized.

```
Party *partyInstance = [Party alloc];
[partyInstance init];
```

Because an object must be allocated *and* initialized before it can be used, you always combine these two messages in one line.

```
Party *partyInstance = [[Party alloc] init];
```

Combining two messages in a single line of code is called a *nested message send*. The innermost brackets are evaluated first, so the message **`alloc`** is sent to the class **`Party`** first. This returns a pointer to a new, uninitialized instance of **`Party`** that is then sent the message **`init`**. This returns a pointer to the initialized instance that you store in your pointer variable.

Sending messages

What do you do with an instance that has been initialized? You send it more messages.

First, let's take a closer look at message anatomy. A message is always contained in square brackets. Within a pair of square brackets, a message has three parts:

receiver a pointer to the object being asked to execute a method

selector the name of the method to be executed

arguments the values to be supplied as the parameters to the method

For example, a party might have a list of attendees that you can add to by sending the party the message **`addAttendee:`**.

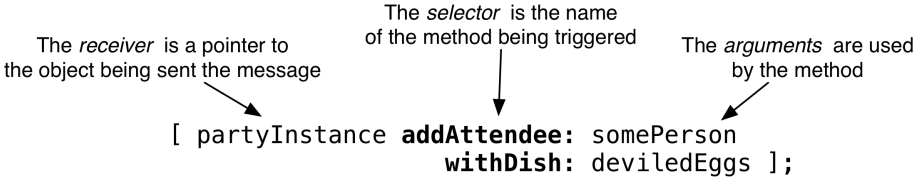
```
[partyInstance addAttendee:somePerson];
```

Sending the **`addAttendee:`** message to `partyInstance` (the receiver) triggers the **`addAttendee:`** method (named by the selector) and passes in `somePerson` (an argument).

The **`addAttendee:`** message has one argument. Objective-C methods can take a number of arguments or none at all. The message **`init`**, for instance, has no arguments.

An attendee to a party might need to RSVP and inform the host what dish the attendee will bring. Thus, the **`Party`** class may have another method named **`addAttendee:withDish:`**. This message takes two arguments: the attendee and the dish. Each argument is paired with a label in the selector, and each label ends with a colon. The selector is all of the labels taken together (Figure 2.2).

Figure 2.2 Parts of a message send



The pairing of labels and arguments is an important feature of Objective-C. In other languages, this method might look something like this:

```
partyInstance.addAttendeeWithDish(somePerson, deviledEggs);
```

In those languages, it is not completely obvious what each of the arguments sent to this function are. In Objective-C, however, each argument is paired with the appropriate label.

```
[partyInstance addAttendee:somePerson withDish:deviledEggs];
```

It takes some getting used to, but eventually Objective-C programmers appreciate the clarity of arguments being interposed into the selector. The trick is to remember that for every pair of square brackets, there is only one message being sent. Even though **`addAttendee:withDish:`** has two labels, it is still only one message, and sending that message results in only one method being executed.

In Objective-C, the name of a method is what makes it unique. Therefore, a class cannot have two methods with the same name. Two method names can contain the same individual label, as long as the name of each method differs as a whole. For example, our **`Party`** class has two methods, **`addAttendee:`** and **`addAttendee:withDish:`**. These are two distinct methods, and they do not share any code.

Also, notice the distinction being made between a *message* and a *method*: a method is a chunk of code that can be executed, and a message is the act of asking a class or object to execute a method. The name of a message always matches the name of the method to be executed.

Destroying objects

To destroy an object, you set the variable that points to it to `nil`.

```
partyInstance = nil;
```

This line of code destroys the object pointed to by the `partyInstance` variable and sets the value of the `partyInstance` variable to `nil`. (It is actually a bit more complicated than that, and you will learn about memory management in the next chapter.)

The value `nil` is the zero pointer. (C programmers know it as `NULL`. Java programmers know it as `null`.) A pointer that has a value of `nil` is typically used to represent the absence of an object. For example, a party could have a venue. While the organizer of the party is still determining where to host the party, `venue` would point to `nil`. This allows you to do things like:

```
if (venue == nil) {
    [organizer remindToFindVenueForParty];
}
```

Objective-C programmers typically use the shorthand form of determining if a pointer is `nil`:

```
if (!venue) {
    [organizer remindToFindVenueForParty];
}
```

Since the `!` operator means “not,” this reads as “if there is not a venue” and will evaluate to true if `venue` is `nil`.

If you send a message to a variable that is `nil`, nothing happens. In other languages, sending a message to the zero pointer is illegal, so you see this sort of thing a lot:

```
// Is venue non-nil?
if (venue) {
    [venue sendConfirmation];
}
```

In Objective-C, this check is unnecessary because a message sent to `nil` is ignored. Therefore, you can simply send a message without a `nil`-check:

```
[venue sendConfirmation];
```

If the venue has not yet been chosen, you will not send a confirmation anywhere. (A corollary: if your program is not doing anything when you think it should be doing something, an unexpected `nil` pointer is often the culprit.)

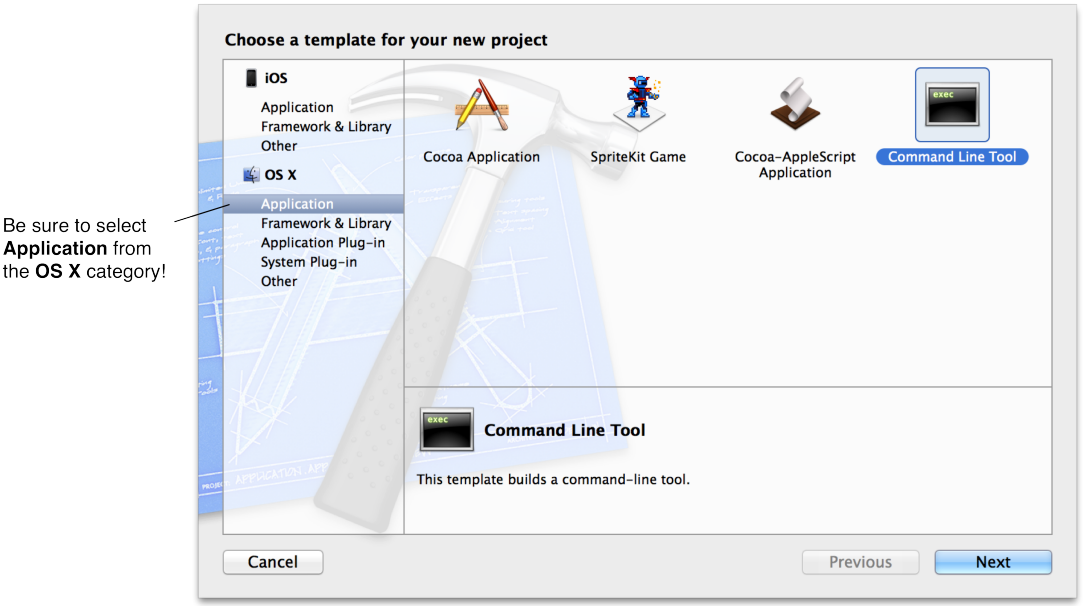
Enough theory. Time for some practice and a new project.

Beginning RandomItems

This new project is not an iOS application; it is a command-line program. Building a command line program will let you focus on Objective-C without the distraction of a user interface. Still, the concepts that you will learn here and in Chapter 3 are critical for iOS development. You will get back to building iOS applications in Chapter 4.

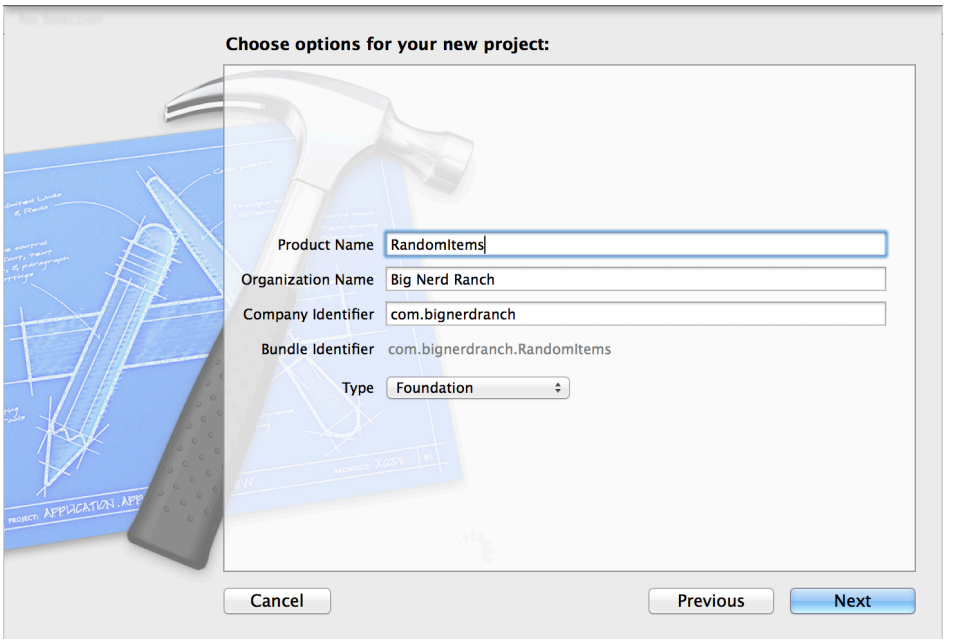
Open Xcode and select File → New → Project.... In the lefthand table in the OS X section, click Application and then select Command Line Tool from the upper panel, as shown in Figure 2.3. Then click Next.

Figure 2.3 Creating a command line tool



On the next sheet, name the product RandomItems and choose Foundation as its type (Figure 2.4).

Figure 2.4 Naming the project

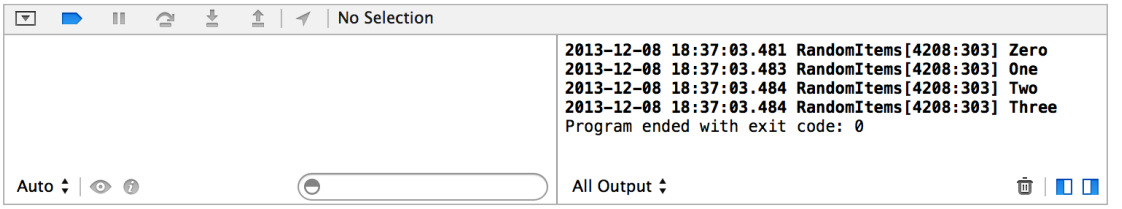


Click Next and you will be prompted to save the project. Save it some place safe – you will be reusing parts of this code in future projects.

In the initial version of RandomItems, you will create an *array* of four strings. An array is an ordered list of pointers to other objects. Pointers in an array are accessed by an index. (Other languages might call it a list or a vector.) Arrays are zero-based; the first item in the array is always at index `0`.

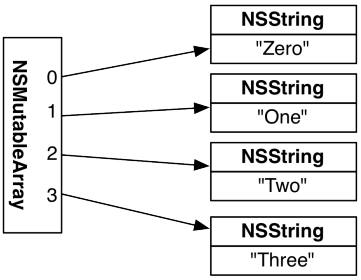
After the array is created, you will loop through the array and print each string. Its output will appear in Xcode's console:

Figure 2.5 Console output



Your program will need five objects: one instance of **`NSMutableArray`** and four instances of **`NSString`** (Figure 2.6).

Figure 2.6 **`NSMutableArray`** instance containing pointers to **`NSString`** instances

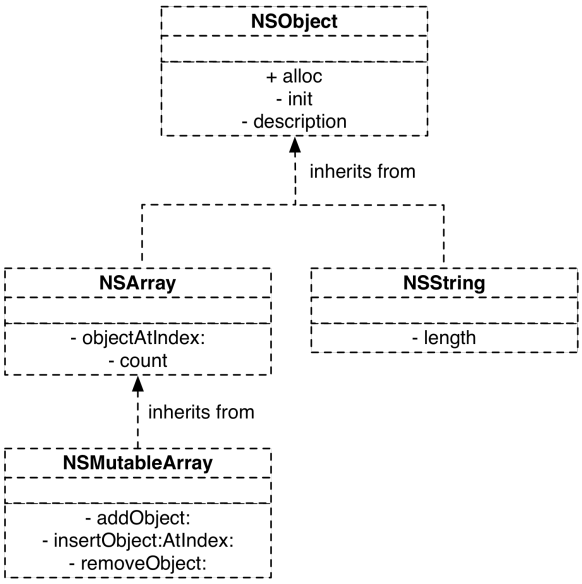


In Objective-C, an array does not contain the objects that belong to it; instead it holds a pointer to each object. When an object is added to an array, the address of that object in memory is stored inside the array.

Now let's consider the **`NSMutableArray`** and **`NSString`** classes. **`NSMutableArray`** is a *subclass* of **`NSArray`**. Classes exist in a hierarchy, and every class has exactly one superclass – except for the root class of the entire hierarchy: **`NSObject`**. A class inherits the behavior of its superclass.

Figure 2.7 shows the class hierarchy for **`NSMutableArray`** and **`NSString`** along with a few methods implemented by each class.

Figure 2.7 Class hierarchy



As the top superclass, **`NSObject`**'s role is to implement the basic behavior of every object in Cocoa Touch. Every class inherits the methods and instance variables defined in **`NSObject`**. Two methods that

NSObject implements are **alloc** and **init** (Figure 2.7). Thus, these methods can be used to create an instance of any class.

A subclass adds methods and instance variables to extend the behavior of its superclass:

- **NSString** adds behavior for storing and handling strings, including the method **length** that returns the number of characters in a string.
- **NSArray** adds behavior for ordered lists, including accessing an object at a given index (**objectAtIndex:**) and getting the number of objects in an array (**count**).
- **NSMutableArray** extends **NSArray**'s abilities by adding the abilities to add and remove pointers.

Creating and populating an array

Let's use these classes in some real code. In the project navigator, find and select the file named `main.m`. When it opens, you will see that some code has been written for you – most notably, a **main** function that is the entry point of any C or Objective-C application.

In `main.m`, delete the line of code that **NSLog**s "Hello, World!" and replace it with code that creates an instance of **NSMutableArray**, adds a total of four objects to the mutable array, and then destroys the array.

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {

        // insert code here...
        NSLog(@"Hello, World!");

        // Create a mutable array object, store its address in items variable
        NSMutableArray *items = [[NSMutableArray alloc] init];

        // Send the message addObject: to the NSMutableArray pointed to
        // by the variable items, passing a string each time
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];

        // Send another message, insertObject:atIndex:, to that same array object
        [items insertObject:@"Zero" atIndex:0];

        // Destroy the mutable array object
        items = nil;

    }
    return 0;
}
```

The objects that you have added and inserted into the array are instances of **NSString**. You can create an instance of **NSString** by prefixing a character string with an `@` symbol:

```
NSString *myString = @"Hello, World!";
```

Iterating over an array

Now that you have an `items` array with strings in it, you are going to iterate over the array, access each string, and output it to the console.

You could write this operation as a `for` loop:

```
for (int i = 0; i < [items count]; i++) {
    NSString *item = [items objectAtIndex:i];
    NSLog(@"%@", item);
}
```

Because arrays are zero-based, the counter starts at `0` and stops at one less than the result of sending the items array the message **`count`**. Within the loop, you then send the message **`objectAtIndex:`** to retrieve the object at the current index before printing it out.

The information returned from **`count`** is important because if you ask an array for an object at an index that is equal to or greater than the number of objects in the array, an exception will be thrown. (In some languages, exceptions are thrown and caught all the time. In Objective-C, exceptions are considered programmer errors and should be fixed in code instead of handled at runtime. We will talk more about exceptions at the end of this chapter.)

This code would work fine, but Objective-C provides a better option for iterating over an array called *fast enumeration*. Fast enumeration is shorter syntax than a traditional `for` loop and far less error-prone. In some cases, it will be faster.

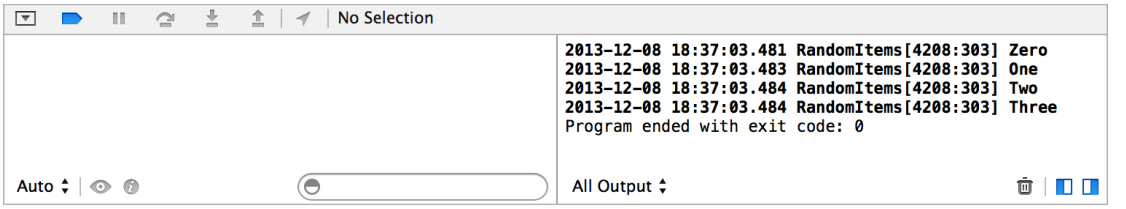
In `main.m`, add the following code that uses fast enumeration to iterate over the `items` array.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {

        // Create a mutable array object, store its address in items variable
        NSMutableArray *items = [[NSMutableArray alloc] init];

        // Send the message addObject: to the NSMutableArray pointed to
        // by the variable items, passing a string each time
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];

        // Send another message, insertObject:atIndex:, to that same array object
        [items insertObject:@"Zero" atIndex:0];

        // For every item in the items array ...
        for (NSString *item in items) {
            // Log the description of item
            NSLog(@"%@", item);
        }

        items = nil;
    }
    return 0;
}
```

Fast enumeration has a limitation: you cannot use it if you need to add or remove objects within the loop. Trying to do so will cause an exception to be thrown. If you need to add or remove objects, you must use a regular `for` loop with a counter variable.

Build and run the application (Command-R). A new pane will appear at the bottom of the Xcode window. This is the *debug area*. On the righthand side of this area is the *console* and your output.

Figure 2.8 Output in console



If you need to, you can resize the debug area and its panels by dragging their frames. (In fact, you can resize any area in the workspace window this way.)

You have completed the first phase of the RandomItems program. Before you continue with the next phase, let's take a closer look at the **`NSLog`** function and format strings.

Format strings

`NSLog()` takes a variable number of arguments and prints a string to the log. In Xcode, the log appears in the console. The first argument of **`NSLog()`** is required. It is an instance of **`NSString`** and is called the *format string*.

A format string contains text and a number of *tokens*. Each token (also called a format specification) is prefixed with a percent symbol (`%`). Each additional argument passed to the function replaces a token in the format string.

Tokens specify the type of the argument that they correspond to. Here is an example:

```
int a = 1;
float b = 2.5;
char c = 'A';
NSLog(@"Integer: %d Float: %f Char: %c", a, b, c);
```

The output would be

```
Integer: 1 Float: 2.5 Char: A
```

Objective-C format strings work the same way as in C. However, Objective-C adds one more token: `%@`. This token corresponds to an argument whose type is "a pointer to any object."

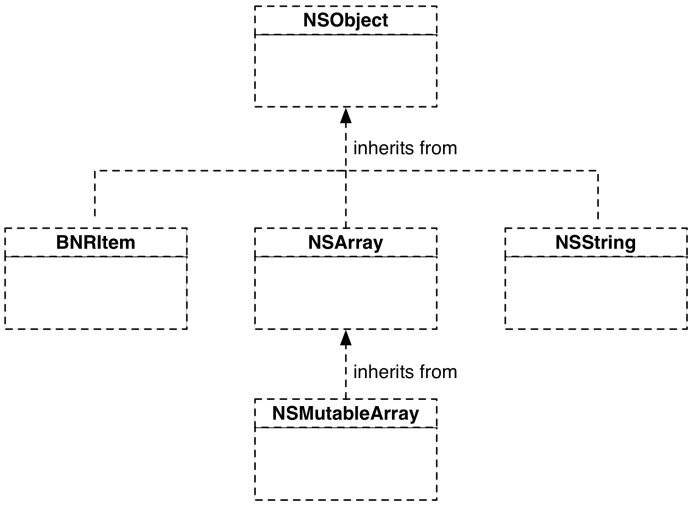
When `%@` is encountered in the format string, instead of the token being replaced by the corresponding argument, that argument is sent the message **`description`**. The **`description`** method returns an instance of **`NSString`** that replaces the token.

Because the argument that corresponds to the `%@` token is sent a message, that argument must be an object. Back in Figure 2.7, you can see that **`description`** is a method on the **`NSObject`** class. Thus, every class implements **`description`**, which is why you can use the `%@` token with any object.

Subclassing an Objective-C Class

In this section, you are going to create a new class named **`BNRItem`**. **`BNRItem`** will be a subclass of **`NSObject`**.

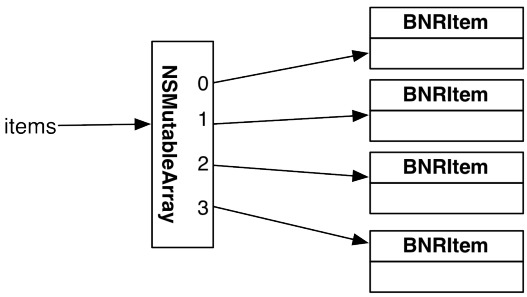
Figure 2.9 Class hierarchy including **BNRItem**



An instance of **BNRItem** will represent something that a person owns in the real world, like a laptop, a bicycle, or a backpack. In terms of Model-View-Controller, **BNRItem** is a model class. An instance of **BNRItem** stores information about a possession.

Once you have created the **BNRItem** class, you will populate the `items` array with instances of **BNRItem** instead of **NSString**.

Figure 2.10 A different class of items

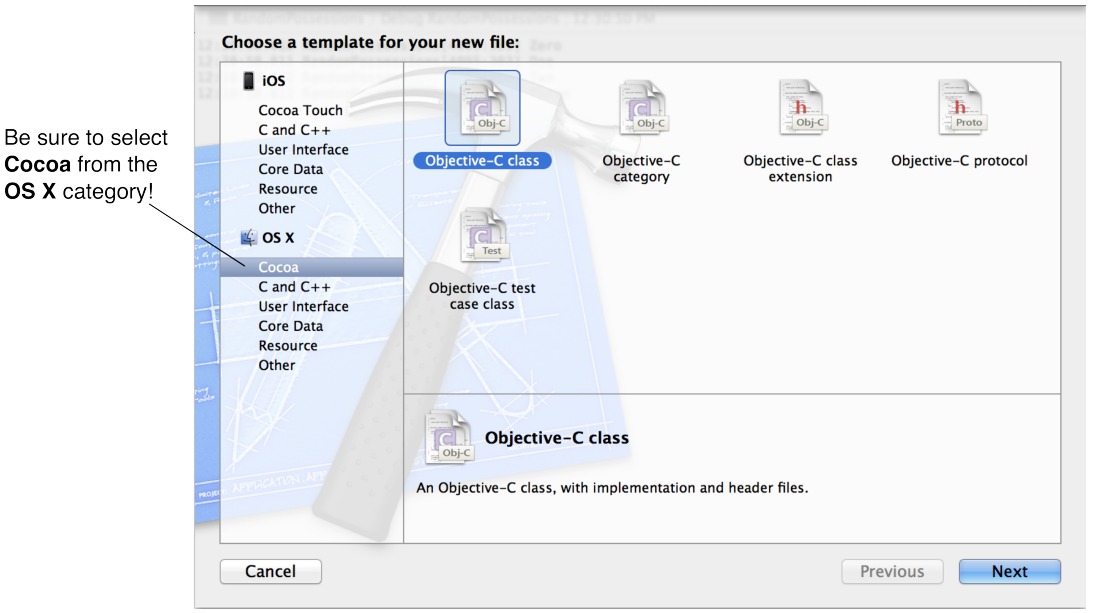


Later in the book, you will reuse **BNRItem** in a complex iOS application.

Creating an NSObject subclass

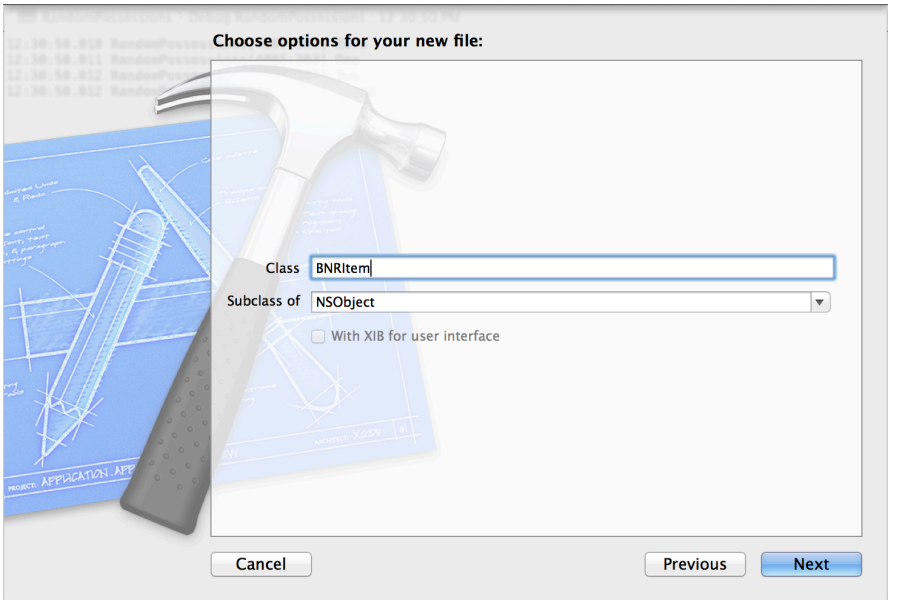
To create a new class in Xcode, choose File → New → File.... In the lefthand table of the panel that appears, select Cocoa from the OS X section. Then select Objective-C class from the upper panel and click Next (Figure 2.11).

Figure 2.11 Creating a class



On the next panel, name this new class **`BNRItem`** and enter **`NSObject`** as its superclass, as shown in Figure 2.12.

Figure 2.12 Choosing a superclass



Click Next and you will be asked where to save the files for this class. The default location is fine. Make sure the checkbox is selected for the RandomItems target. Click Create.

In the project navigator, find the class files for **BNRItem** ▷ BNRItem.h and BNRItem.m:

- BNRItem.h is the *header file* (also called the *interface file*). This file declares the name of the new class, its superclass, the instance variables that each instance of this class has, and any methods this class implements.

- BNRItem.m is the implementation file, and it contains the code for the methods that the class implements.

Every Objective-C class has these two files. You can think of the header file as a user manual for an instance of a class and the implementation file as the engineering details that define how it really works.

Select BNRItem.h in the project navigator. The contents of the file look like this:

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

@interface BNRItem : NSObject

@end
```

To declare a class in Objective-C, you use the keyword @interface followed by the name of the new class. After a colon comes the name of the superclass. Objective-C only allows single inheritance, so every class can only have one superclass:

```
@interface ClassName : SuperclassName
```

The @end directive indicates that the class declaration has come to an end.

Notice the @ prefixes. Objective-C retains the keywords of the C language. Additional keywords specific to Objective-C are distinguishable by the @ prefix.

Instance variables

An “item,” in our world, is going to have a name, a serial number, a value, and a date of creation. These will be the instance variables of **BNRItem**.

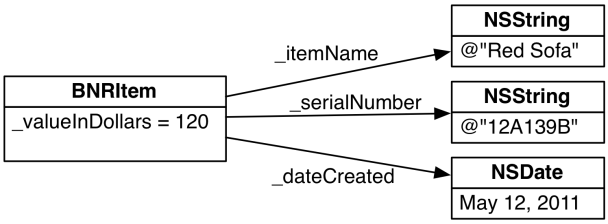
Instance variables for a class are declared between curly braces immediately after the class declaration.

In BNRItem.h, add a set of curly braces and four instance variables to the **BNRItem** class:

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

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

@end
```

Now every instance of **BNRItem** will have one spot for a simple integer and three spots for pointers to objects, specifically two **NSString** instances and one **NSDate** instance. (Remember, the * denotes that the variable is a pointer.) Figure 2.13 shows an example of a **BNRItem** instance after its instance variables have been given values.

Figure 2.13 A **BNRItem** instance



Notice that Figure 2.13 shows a total of four objects: the instance of **BNRItem**, two instances of **NSString**, and an instance of **NSDate**. Each object exists independently and outside of the others. The **BNRItem** object's instance variables are the *pointers* to the other objects, not the objects themselves.

For example, every **BNRItem** instance has a pointer instance variable named `_itemName`. The `_itemName` of the **BNRItem** object shown in Figure 2.13 points to an **NSString** object whose contents are `"Red Sofa"`. The `"Red Sofa"` string does not live inside the **BNRItem** object. The **BNRItem** object only knows where the `"Red Sofa"` string lives in memory and stores that address as `_itemName`. One way to think of this relationship is "the **BNRItem** object calls this string its `_itemName`."

The story is different for the instance variable `_valueInDollars`. This instance variable is *not* a pointer to another object; it is just an `int`. The `int` itself does live inside the **BNRItem** object.

The idea of pointers is not easy to understand at first. In the next chapter, you will learn more about objects, pointers, and instance variables, and throughout this book you will see object diagrams like Figure 2.13 to drive home the difference between an object and a pointer to an object.

Accessing instance variables

Now that instances of **BNRItem** have instance variables, you need a way to get and set their values. In object-oriented languages, we call methods that get and set instance variables *accessors*. Individually, we call them *getters* and *setters*. Without these methods, an object cannot access the instance variables of another object.

In `BNRItem.h`, declare accessor methods for the instance variables of the **BNRItem** class. You need getters and setters for `_valueInDollars`, `_itemName`, and `_serialNumber`. The `_dateCreated` instance variable will be read-only, so it only needs a getter method.

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

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

- (void)setItemName:(NSString *)str;
- (NSString *)itemName;

- (void)setSerialNumber:(NSString *)str;
- (NSString *)serialNumber;

- (void)setValueInDollars:(int)v;
- (int)valueInDollars;

- (NSDate *)dateCreated;
@end
```

In Objective-C, the name of a setter method is **set** plus the capitalized name of the instance variable it is changing – in this case, **setItemName:**. In other languages, the name of the getter method would likely be **getItemName**. However, in Objective-C, the name of the getter method is just the name of the instance variable. Some of the cooler parts of the Cocoa Touch library make the assumption that your classes follow this convention; therefore, stylish Cocoa Touch programmers always do so.

(For those of you with some experience in Objective-C, we will talk about properties in the next chapter.)

Next, open **BNRItem**'s implementation file, `BNRItem.m`.

At the top of any implementation file, the header file of that class is always imported. The implementation of a class needs to know how it has been declared. (Importing a file is the same as including a file in C except that it ensures that the file will only be included once.)

After the import statement is the implementation block that begins with the `@implementation` keyword followed by the name of the class that is being implemented. All of the method definitions in the implementation file are inside this implementation block. Methods are defined until you close out the block with the `@end` keyword.

In `BNRItem.m`, delete anything that the template may have added between `@implementation` and `@end`. Then define the accessor methods for the instance variables that you declared in `BNRItem.h`.

```
#import "BNRItem.h"

@implementation BNRItem

- (void)setItemName:(NSString *)str
{
    _itemName = str;
}
- (NSString *)itemName
{
    return _itemName;
}

- (void)setSerialNumber:(NSString *)str
{
    _serialNumber = str;
}

- (NSString *)serialNumber
{
    return _serialNumber;
}

- (void)setValueInDollars:(int)v
{
    _valueInDollars = v;
}

- (int)valueInDollars
{
    return _valueInDollars;
}

- (NSDate *)dateCreated
{
    return _dateCreated;
}

@end
```

Notice that each setter method sets the instance variable to whatever is passed in as an argument, and each getter method returns the value of the instance variable.

At this point, check for and fix any errors in your code that Xcode is warning you about. Some possible culprits are typos and missing semicolons.

Let's test out your new class and its accessor methods. In `main.m`, first import the header file for the **`BNRItem`** class.

```
#import <Foundation/Foundation.h>
#import "BNRItem.h"

int main (int argc, const char * argv[])
{
    ...
```

Why do you import the class header `BNRItem.h` but not, say, `NSMutableArray.h`? The **`NSMutableArray`** class comes from the Foundation framework, so it is included when you import

`Foundation/Foundation.h`. On the other hand, the **`BNRItem`** class exists in its own file, so you have to explicitly import it into `main.m`. If you do not, the compiler will not know that it exists and will complain loudly.

Next, create an instance of **`BNRItem`** and log its instance variables to the console.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {

        NSMutableArray *items = [[NSMutableArray alloc] init];
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];
        [items insertObject:@"Zero" atIndex:0];

        // For every item in the array pointed to by items...
        for (NSString *item in items) {
            // print a description
            NSLog(@"%@", item);
        }

        BNRItem *item = [[BNRItem alloc] init];
        NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
                              [item serialNumber], [item valueInDollars]);


        items = nil;
    }

    return 0;
}
```

Build and run the application. At the end of your output, you will see three `(null)` strings and a `0`. These are the values of the instance variables of your freshly-minted instance of **`BNRItem`**.

Figure 2.14 Instance variables' values in console

```
2013-12-08 18:43:11.237 RandomItems[4239:303] Zero
2013-12-08 18:43:11.239 RandomItems[4239:303] One
2013-12-08 18:43:11.239 RandomItems[4239:303] Two
2013-12-08 18:43:11.240 RandomItems[4239:303] Three
2013-12-08 18:43:11.240 RandomItems[4239:303] (null) (null) (null) 0
Program ended with exit code: 0
```

When an object is created, all of its instance variables are "zeroed-out." A pointer to an objects points to `nil`; a primitive like `int` has the value of `0`.

To give the **`BNRItem`** object's instance variables more interesting values, you need to create new objects and pass them as arguments to the setter methods.

In `main.m`, type in the following code:

```
// Notice we are omitting some of the surrounding code
...

BNRItem *item = [[BNRItem alloc] init];

// This creates an NSString, "Red Sofa" and gives it to the BNRItem
[item setItemName:@"Red Sofa"];

// This creates an NSString, "A1B2C" and gives it to the BNRItem
[item setSerialNumber:@"A1B2C"];

// This sends the value 100 to be used as the valueInDollars of this BNRItem
[item setValueInDollars:100];

NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
                      [item serialNumber], [item valueInDollars]);

...
```

Build and run the application. You will see the values of the three instance variables. You will still see `(null)` for `dateCreated`. Later in the chapter, you will take care of giving this instance variable a value when an object is created.

Figure 2.15 More interesting values

```
2013-12-08 18:44:56.551 RandomItems[4254:303] Zero
2013-12-08 18:44:56.553 RandomItems[4254:303] One
2013-12-08 18:44:56.553 RandomItems[4254:303] Two
2013-12-08 18:44:56.554 RandomItems[4254:303] Three
2013-12-08 18:44:56.554 RandomItems[4254:303] Red Sofa (null) A1B2C 100
Program ended with exit code: 0
```

Using dot syntax

To get and set an instance variable, you can send explicit accessor messages:

```
BNRItem *item = [[BNRItem alloc] init];

// set valueInDollars by sending an explicit message
[item setValueInDollars:5];

// get valueInDollars by sending an explicit message
int value = [item valueInDollars];
```

Or you can use *dot syntax*, also called *dot notation*. Here is the same code using dot syntax:

```
BNRItem *item = [[BNRItem alloc] init];

// set valueInDollars using dot syntax
item.valueInDollars = 5;

// get valueInDollars using dot syntax
int value = item.valueInDollars;
```

The receiver (`item`) is followed by a `.` followed by the name of the instance variable without the leading underscore (`valueInDollars`).

Notice that the syntax is the same for both setting and getting the instance variable (`item.valueInDollars`); the difference is in which side of the assignment operator it is on.

There is no difference at runtime between accessor messages and dot syntax; the compiled code is the same and either syntax will invoke the **`valueInDollars`** and **`setValueInDollars:`** methods that you just implemented.

These days, stylish Objective-C programmers tend to use dot syntax for invoking accessors. It makes code easier to read, especially when there would traditionally be nested message calls. It is also consistent with Apple's code. It is what we will do in this book.

In `main.m`, update your code to use dot syntax to set the instance variables and to get them as part of the format string.

```
...

BNRItem *item = [[BNRItem alloc] init];

// This creates an NSString, "Red Sofa" and gives it to the BNRItem
[item setItemName:@"Red Sofa"];
item.itemName=@"Red Sofa";

// This creates an NSString, "A1B2C" and gives it to the BNRItem
[item setSerialNumber:@"A1B2C"];
item.serialNumber=@"A1B2C";

// This sends the value 100 to be used as the valueInDollars of this BNRItem
[item setValueInDollars:100];
item.valueInDollars = 100;

NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
                      [item serialNumber], [item valueInDollars]);
NSLog(@"%@ %@ %@ %d", item.itemName, item.dateCreated,
                      item.serialNumber, item.valueInDollars);

...
```

Class vs. instance methods

Methods come in two types: *instance methods* and *class methods*. A class method typically either creates a new instance of the class or retrieves some global property of the class. An instance method operates on a particular instance of the class. For instance, the accessors that you just implemented are all instance methods. You use them to set or get the instance variables of a particular object.

To invoke an instance method, you send the message to an instance of the class. To invoke a class method, you send the message to the class itself.

For example, when you created an instance of **`BNRItem`**, you sent **`alloc`** (a class method) to the **`BNRItem`** class and then **`init`** (an instance method) to the instance of **`BNRItem`** returned from **`alloc`**.

The **`description`** method is an instance method. In the next section, you are going to implement **`description`** in **`BNRItem`** to return an **`NSString`** object that describes an instance of **`BNRItem`**. Later in the chapter, you will implement a class method to create an instance of **`BNRItem`** using random values.

Overriding methods

A subclass can also override methods of its superclass. For example, sending the **description** message to an instance of **NSObject** returns the object's class and its address in memory as an instance of **NSString** that looks like this:

```
<BNRQuizViewController: 0x4b222a0>
```

A subclass of **NSObject** can override this method to return an **NSString** object that better describes an instance of that subclass. For example, the **NSString** class overrides **description** to return the string itself. The **NSArray** class overrides **description** to return the description of every object in the array.

Because **BNRItem** is a subclass of **NSObject** (the class that originally declares the **description** method), when you re-implement **description** in **BNRItem**, you are *overriding* it.

When overriding a method, all you need to do is define it in the implementation file; you do not need to declare it in the header file because it has already been declared by the superclass.

In `BNRItem.m`, override the **description** method. The code for a method implementation can go anywhere between `@implementation` and `@end`, as long as it is not inside the curly braces of an existing method.

```
- (NSString *)description
{
    NSString *descriptionString =
        [[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, recorded on %@",
                          self.itemName,
                          self.serialNumber,
                          self.valueInDollars,
                          self.dateCreated];

    return descriptionString;

}
```

Note what you are not doing here: you are not passing the instance variables by name (e.g.., `_itemName`). Instead you are invoking accessors (via dot syntax). It is good practice to use accessor methods to access instance variables even inside a class. It is possible that an accessor method may change something about the instance variable that you are trying to access, and you want to make sure it gets the chance to do what it needs to.

Now whenever you send the message **description** to an instance of **BNRItem**, it will return an instance of **NSString** that better describes the instance.

In `main.m`, replace the statement that prints out the instance variables individually with a statement that relies on **BNRItem**'s implementation of **description**.

```
...

item.valueInDollars = 100;

NSLog(@"%@ %@ %@ %d", item.itemName, item.dateCreated,
                      item.serialNumber, item.valueInDollars);

// The %@ token is replaced with the result of sending
// the description message to the corresponding argument
NSLog(@"%@", item);

items = nil;
```

Build and run the application and check your results in the console.

Figure 2.16 An instance of **BNRItem** described

```
2013-12-08 18:49:05.934 RandomItems[4277:303] Zero
2013-12-08 18:49:05.935 RandomItems[4277:303] One
2013-12-08 18:49:05.936 RandomItems[4277:303] Two
2013-12-08 18:49:05.936 RandomItems[4277:303] Three
2013-12-08 18:49:05.937 RandomItems[4277:303] Red Sofa (A1B2C): Worth $100, recorded on (null)
Program ended with exit code: 0
```

What if you want to create an entirely new instance method, one that you are not overriding from the superclass? You declare the new method in the header file and define it in the implementation file. Let's see how that works by creating two new instance methods to initialize an instance of **BNRItem**.

Initializers

Right now, the **BNRItem** class has only one way to initialize an instance – the **init** method, which it inherits from the **NSObject** class. In this section, you are going to write two additional initialization methods, or *initializers*, for **BNRItem**.

In `BNRItem.h`, declare two initializers.

```
    NSDate *_dateCreated;
}

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setItemName:(NSString *)str;
```

(Wondering about `instancetype`? Hold on – we will get there shortly.)

Each initializer begins with the word **init**. Naming initializers this way does not make these methods different from other instance methods; it is only a naming convention. However, the Objective-C community is all about naming conventions, which you should strictly adhere to. (Disregarding naming conventions in Objective-C results in problems that are worse than you might imagine.)

An initializer takes arguments that the object can use to initialize itself. Often, a class has multiple initializers because instances can have different initialization needs. For instance, the first initializer that you declared takes three arguments that it uses to configure the item's name, value, and serial number. So you need all of this information to initialize an instance with this method. What if you only know the item's name? Then you can use the second initializer.

The designated initializer

For each class, regardless of how many initialization methods there are, one method is chosen as the *designated initializer*. The designated initializer makes sure that every instance variable of an object is valid. ("Valid" in this context means "when you send messages to this object after initializing it, you can predict the outcome and nothing bad will happen.")

Typically, the designated initializer has parameters for the most important and frequently used instance variables of an object. The **BNRItem** class has four instance variables, but only three are writeable. Therefore, **BNRItem**'s designated initializer should accept three arguments and provide a value within its implementation for `_dateCreated`.

In `BNRItem.h`, add a comment naming the designated initializer:

```
    NSDate *_dateCreated;
}

// Designated initializer for BNRItem
- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setItemName:(NSString *)str;
```

instancetype

The return type for both initializers is `instancetype`. This keyword can only be used for return types, and it matches the return type to the receiver. **init** methods are always declared to return `instancetype`.

Why not make the return type `BNRItem *`? That would cause a problem if the **BNRItem** class was ever subclassed. The subclass would inherit all of the methods from **BNRItem**, including this initializer and its return type. If an instance of the subclass was sent this initializer message, what would be returned? Not a pointer to a **BNRItem** instance, but a pointer to an instance of the subclass. You might think, "No problem. I will override the initializer in the subclass to change the return type." But in Objective-C, you cannot have two methods with the same selector and different return types (or arguments). By specifying that an initialization method returns "an instance of the receiving object," you never have to worry what happens in this situation.

id

Before the `instancetype` keyword was introduced in Objective-C, initializers returned `id` (pronounced "eye-dee"). This type is defined as "a pointer to any object." (`id` is a lot like `void *` in C.). As of this writing, Xcode class templates still use `id` as the return type of initializers added in boilerplate code. We imagine that this will change soon.

Unlike `instancetype`, `id` can be used as more than just a return type. You can declare variables or method parameters of type `id` when you are unsure what type of object the variable will end up pointing to.

```
id objectOfUnknownType;
```

You can use `id` when using fast enumeration to iterate over an array of multiple or unknown types of objects:

```
for (id item in items) {
    NSLog(@"%@", item);
}
```

Note that because `id` is defined as "a pointer to any object," you do not include an `*` when declaring a variable or method parameter of this type.

Implementing the designated initializer

In `BNRItem.m`, implement the designated initializer within the implementation block.

```
@implementation BNRItem

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber
{
    // Call the superclass's designated initializer
    self = [super init];

    // Did the superclass's designated initializer succeed?
    if (self) {
        // Give the instance variables initial values
        _itemName = name;
        _serialNumber = sNumber;
        _valueInDollars = value;
        // Set _dateCreated to the current date and time
        _dateCreated = [[NSDate alloc] init];
    }

    // Return the address of the newly initialized object
    return self;
}
```

There is a lot to talk about in this code. First, notice that you set the `_dateCreated` instance variable to point a new instance of **`NSDate`**, which represents the current date and time.

Next, consider the first line of code in this implementation. In the designated initializer, the first thing you always do is call the superclass's designated initializer using `super`. The last thing you do is return a pointer to the successfully initialized object using `self`. So to understand what is going on in an initializer, you need to know about `self` and `super`.

self

Inside a method, `self` is an implicit local variable. There is no need to declare it, and it is automatically set to point to the object that was sent the message. (Most object-oriented languages have this concept, but some call it `this` instead of `self`.) Typically, `self` is used so that an object can send a message to itself:

```
- (void)chickenDance
{
    [self pretendHandsAreBeaks];
    [self flapWings];
    [self shakeTailFeathers];
}
```

In the last line of an **`init`** method, you always return the newly initialized object so that the caller can assign it to a variable:

```
return self;
```

super

When you are overriding a method, you often want to keep what the method of the superclass is doing and have your subclass add something new on top of that. To make this easier, there is a compiler directive in Objective-C called `super`:

```
- (void)someMethod
{
    [super someMethod];
    [self doMoreStuff];
}
```

How does `super` work? Usually when you send a message to an object, the search for a method of that name starts in the object's class. If there is no such method, the search continues in the superclass of the object. The search will continue up the inheritance hierarchy until a suitable method is found. (If it gets to the top of the hierarchy and no method is found, an exception is thrown.)

When you send a message to `super`, you are sending a message to `self`, but the search for the method skips the object's class and starts at the superclass. In the case of **`BNRItem`**'s designated initializer, you send the **`init`** message to `super`. This calls **`NSObject`**'s implementation of **`init`**.

Confirming initialization success

Now let's look at the next line where you confirm what the superclass's initializer returned. If an initializer message fails, it will return `nil`. Therefore, it is a good idea to save the return value of the superclass's initializer into the `self` variable and confirm that it is not `nil` before doing any further initialization.

Instance variables in initializers

Now we get to the core of this method where the instance variables are given values. Earlier we told you not to access instance variables directly and to use accessor methods. Now we are asking you to break that rule when writing initializers.

While an initializer is being executed, the object is being born, and you cannot be sure that its instance variables have all been set to usable values. When writing a method, you typically assume that all of an object's instance variables have been set to usable values. Thus, invoking a method (like an accessor) at a time when this may not be the case is unsafe. At Big Nerd Ranch, we typically set the instance variables directly in initializers, instead of calling accessor methods.

Some very good Objective-C programmers do use accessors in initializers. They argue that if the accessor does something complicated, you want that code in exactly one place; replicating it in your initializer is bad. We are not religiously devoted to either approach, but in this book we will set instance variables directly in initializers.

Other initializers and the initializer chain

Let's implement the second initializer for the **`BNRItem`** class. In this initializer's definition, you are not going to replicate the code in the designated initializer. Instead, this initializer will simply call the designated initializer, passing the information it is given for the `_itemName` and default values for the other arguments.

In `BNRItem.m`, implement **`initWithItemName:`**.

```
- (instancetype)initWithItemName:(NSString *)name
{
    return [self initWithItemName:name
                       valueInDollars:0
                         serialNumber:@""];
}
```

The **BNRItem** class already has a third initializer – **init**, which it inherits from **NSObject**. If **init** is used to initialize an instance of **BNRItem**, none of the stuff that you put in the designated initializer will happen. Therefore, you must override **init** in **BNRItem** to link to **BNRItem**'s designated initializer.

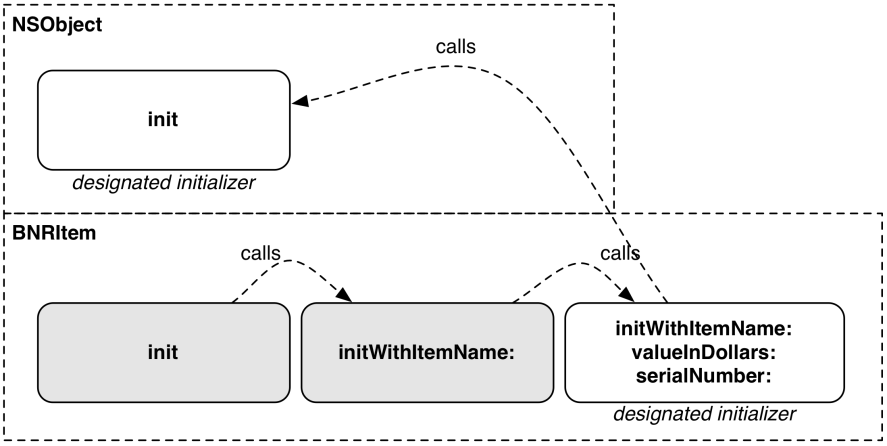
In BNRItem.m, override **init** to call **initWithItemName:**, passing a default value for the item's name.

```
- (instancetype)init
{
    return [self initWithItemName:@"Item"];
}
```

Now when **init** is sent to an instance of **BNRItem**, the method will call **initWithItemName:** with a default value for _itemName, which will call the designated initializer, **initWithItemName:valueInDollars:serialNumber:** with default values for _valueInDollars and _serialNumber.

The relationships between **BNRItem**'s initializers are shown in Figure 2.17; the designated initializers are white, and the additional initializers are gray.

Figure 2.17 A chain of initializers



Using initializers in a chain reduces the possibility of error and makes maintaining code easier. The programmer who created the class makes it clear which initializer is the designated initializer. You only write the core of the initializer once in the designated initializer, and other initialization methods simply call the designated initializer (directly or indirectly) with default values.

Let's form some simple rules for initializers from these ideas.

- A class inherits all initializers from its superclass and can add as many as it wants for its own purposes.
- Each class picks one initializer as its *designated initializer*.
- The designated initializer calls the superclass's designated initializer (directly or indirectly) before doing anything else.
- Any other initializers call the class's designated initializer (directly or indirectly).
- If a class declares a designated initializer that is different from its superclass, the superclass's designated initializer must be overridden to call the new designated initializer (directly or indirectly).

Using initializers

Now that you have a designated initializer for **`BNRItem`**, you can use it instead of setting instance variables individually.

In `main.m`, remove the creation of the single **`BNRItem`** instance and all three setter messages. Then add code that creates an instance and sets its instance variables using the designated initializer.

```
...

// For every item in the items array ...
for (NSString *item in items) {
    // ... print a description of the current item
    NSLog(@"%@", item);
}

~~BNRItem *item = [[BNRItem alloc] init];~~

~~item.itemName = @"Red Sofa";~~
~~item.serialNumber = @"A1B2C";~~
~~item.valueInDollars = 100;~~


BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                     valueInDollars:100
                                       serialNumber:@"A1B2C"];

NSLog(@"%@", item);

...
```

Build and run the application. Notice that the console now prints a single **`BNRItem`** instance that was instantiated with the values passed to the **`BNRItem`** class's designated initializer.

Let's confirm that your other two initializers work as expected. In `main.m`, create two additional instances of **`BNRItem`** using **`initWithItemName:`** and **`init`**.

```
        ...

        BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                           valueInDollars:100
                                             serialNumber:@"A1B2C"];
        NSLog(@"%@", item);

        BNRItem *itemWithName = [[BNRItem alloc] initWithItemName:@"Blue Sofa"];
        NSLog(@"%@", itemWithName);

        BNRItem *itemWithNoName = [[BNRItem alloc] init];
        NSLog(@"%@", itemWithNoName);

        items = nil;
    }
    return 0;
}
```

Build and run the application and check the console to confirm that **BNRItem**'s initialization chain is working.

Figure 2.18 Three initializers at work

```
2013-12-08 18:55:18.620 RandomItems[4302:303] Zero
2013-12-08 18:55:18.622 RandomItems[4302:303] One
2013-12-08 18:55:18.623 RandomItems[4302:303] Two
2013-12-08 18:55:18.623 RandomItems[4302:303] Three
2013-12-08 18:55:18.628 RandomItems[4302:303] Red Sofa (A1B2C): Worth $100, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Blue Sofa (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Item (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
Program ended with exit code: 0
```

There is only one thing left to do to complete the **BNRItem** class. You are going to write a method that creates an instance and initializes it with random values. This method will be a class method.

Class methods

Class methods typically either create new instances of the class or retrieve some global property of the class. Class methods do not operate on an instance or have any access to instance variables.

Syntactically, class methods differ from instance methods by the first character in their declaration. An instance method uses the `-` character just before the return type, and a class method uses the `+` character.

In `BNRItem.h`, declare a class method that will create a random item.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;
```

Notice the order of the declarations in the header file. Instance variables come first, followed by class methods, followed by initializers, followed by any other instance methods. This convention makes header files easier to read.

In `BNRItem.m`, implement **`randomItem`** to create, configure, and return a **`BNRItem`** instance. (Make sure this method is between the `@implementation` and `@end`.)

```
+ (instancetype)randomItem
{
    // Create an immutable array of three adjectives
    NSArray *randomAdjectiveList = @[@"Fluffy", @"Rusty", @"Shiny"];

    // Create an immutable array of three nouns
    NSArray *randomNounList = @[@"Bear", @"Spork", @"Mac"];

    // Get the index of a random adjective/noun from the lists
    // Note: The % operator, called the modulo operator, gives
    // you the remainder. So adjectiveIndex is a random number
    // from 0 to 2 inclusive.
    NSInteger adjectiveIndex = arc4random() % [randomAdjectiveList count];
    NSInteger nounIndex = arc4random() % [randomNounList count];

    // Note that NSInteger is not an object, but a type definition
    // for "long"

    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
                            [randomAdjectiveList objectAtIndex:adjectiveIndex],
                            [randomNounList objectAtIndex:nounIndex]];

    int randomValue = arc4random() % 100;

    NSString *randomSerialNumber = [NSString stringWithFormat:@"%c%c%c%c%c",
                                    '0' + arc4random() % 10,
                                    'A' + arc4random() % 26,
                                    '0' + arc4random() % 10,
                                    'A' + arc4random() % 26,
                                    '0' + arc4random() % 10];


    BNRItem *newItem = [[self alloc] initWithItemName:randomName
                                    valueInDollars:randomValue
                                      serialNumber:randomSerialNumber];
    return newItem;
}
```

First, at the beginning of this method, notice the syntax for creating the two arrays `randomAdjectiveList` and `randomNounList` – an `@` symbol followed by square brackets. Within the brackets is a comma-delimited list of objects that will populate the array. (In this case, the objects are instances of **`NSString`**.) This syntax is shorthand for creating instances of **`NSArray`**. Note that it always creates an immutable array. You can only use this shorthand if you do not need the resulting array to be mutable.

After creating the arrays, **`randomItem`** creates a string from a random adjective and noun, a random integer value, and another string from random numbers and letters.

Finally, the method creates an instance of **`BNRItem`** and sends it the designated initializer message with these randomly-created objects and `int` as parameters.

In this method, you also used **stringWithFormat:**, which is a class method of **NSString**. This message is sent directly to the **NSString** class, and the method returns an **NSString** instance with the passed-in parameters. In Objective-C, class methods that return an object of their type (like **stringWithFormat:** and **randomItem**) are called *convenience methods*.

Notice the use of `self` in **randomItem**. Because **randomItem** is a class method, `self` refers to the **BNRItem** class itself instead of an instance. Class methods should use `self` in convenience methods instead of their class name so that a subclass can be sent the same message. In this case, if you create a subclass of **BNRItem** called **BNRToxicWasteItem**, you could do this:

```
BNRToxicWasteItem *item = [BNRToxicWasteItem randomItem];
```

Testing your subclass

For the final version of RandomItems in this chapter, you are going to fill the `items` array with 10 randomly-created instances of **BNRItem**. Then you will loop through the array and log each item (Figure 2.19).

Figure 2.19 Random items

```
2013-10-29 18:17:42.880 RandomItems[64653:303] Rusty Spork (8Q2U8): Worth $73, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.880 RandomItems[64653:303] Shiny Spork (5Y2V3): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Spork (2F9Z7): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Bear (8G5V6): Worth $99, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Shiny Spork (3P9B1): Worth $10, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Rusty Mac (6R5C1): Worth $93, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Spork (3E400): Worth $1, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Mac (3A6T4): Worth $30, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Shiny Spork (8S3I1): Worth $77, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Rusty Spork (4F6F9): Worth $65, recorded on 2013-10-29 22:17:42 +0000
Program ended with exit code: 0
```

In `main.m`, delete all of the code except for the creation and destruction of the `items` array. Then add 10 random **BNRItem** instances to the array and log them.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];
        [items insertObject:@"Zero" atIndex:0];

        // For every item in the items array ...
        for (NSString *item in items) {
            // ... print a description of the current item
            NSLog(@"%@", item);
        }

        BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                           valueInDollars:100
                                             serialNumber:@"A1B2C"];

        NSLog(@"%@", item);

        BNRItem *itemWithName = [[BNRItem alloc] initWithItemName:@"Blue Sofa"];
        NSLog(@"%@", itemWithName);

        BNRItem *itemWithNoName = [[BNRItem alloc] init];
        NSLog(@"%@", itemWithNoName);


        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        for (BNRItem *item in items) {
            NSLog(@"%@", item);
        }


        items = nil;
    }
    return 0;
}
```

Notice that you do not use fast enumeration in the first loop because you are adding to the array within the loop.

Build and run your application and then check the output in the console.

More on NSArray and NSMutableArray

You will frequently use arrays when developing iOS applications, so let's go over some more array-related details.

An Objective-C array can contain objects of different types. For example, although your `items` array currently only contains instances of **`BNRItem`**, you could add an instance of **`NSDate`** or any other Objective-C object. This is different from most strongly typed languages where an array can only hold objects of a single type.

Objective-C arrays can only hold references to Objective-C objects. Primitives and C structures cannot be added to an Objective-C array. If you need to add primitives or C structures, you can "wrap" them in Objective-C objects written for this purpose, including **NSNumber**, **NSValue**, and **NSData**.

Note that you cannot add `nil` to an array. If you need to add "holes" to an array, you must use **NSNull**. **NSNull** is a class whose only instance is meant to stand in for `nil` and is used specifically for this task.

```
    [items addObject:[NSNull null]];
```

When accessing members of an array, you have used the **objectAtIndex:** message with the index of the object you want returned. This, like many other elements of Objective-C, is very verbose. Thus, there exists a shorthand syntax for accessing members of an array:

```
NSString *foo = items[0];
```

This line of code is equivalent to sending **objectAtIndex:** to `items`.

```
NSString *foo = [items objectAtIndex:0];
```

In `BNRItem.m`, update **randomItem** to use this syntax when creating the random name.

```
+ (instancetype)randomItem
{
    ...

    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
                                [randomAdjectiveList objectAtIndex:adjectiveIndex],
                                [randomNounList objectAtIndex:nounIndex]];

    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
                                randomAdjectiveList[adjectiveIndex],
                                randomNounList[nounIndex]];

    int randomValue = arc4random() % 100;

    ...

    return newItem;
}
```

Build and run to confirm that the program works the same as before.

The nested brackets that you end up with can make things confusing because they are used in two distinct ways: one use sends a message and the other use accesses items in an array. Sometimes, it can be clearer to stick with sending the typed-out message. Other times, it is nice to avoid typing the finger-numbing **objectAtIndex:**.

Whichever syntax you use, it is important to understand that there is no difference in your application: the compiler turns the shorthand syntax into code that sends the **objectAtIndex:** message.

In an **NSMutableArray**, you can use a similar shorthand syntax to add and replace objects.

```
NSMutableArray *items = [[NSMutableArray alloc] init];
items[0] = @"A"; // Add @"A"
items[1] = @"B"; // Add @"B"
items[0] = @"C"; // Replace @"A" with @"C"
```

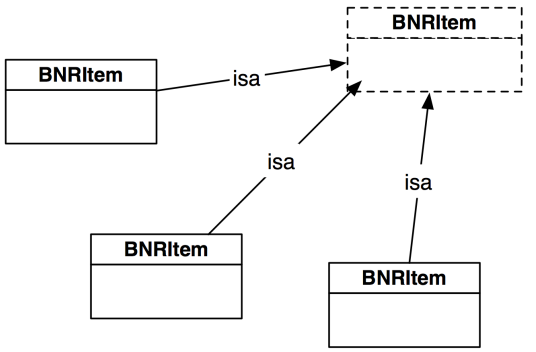
These lines are equivalent to sending **`insertObject:atIndex:`** and **`replaceObjectAtIndex:withObject:`** messages to `items`.

Exceptions and Unrecognized Selectors

At runtime, when a message is sent to an object, that object goes to the class that created it and says, "I was sent this message. Run the code for the matching method." This is different than in most compiled languages, where the method to be executed is determined at compile time.

How does an object know which class created it? It uses its `isa` pointer. Every object has an instance variable called `isa`. When an object is created, the class sets the `isa` instance variable of the returned object to point back at that class (Figure 2.20). It is called the `isa` pointer because an object "is a" instance of that class. Although you probably will never explicitly use the `isa` pointer, its existence gives Objective-C gets much of its power.

Figure 2.20 The `isa` pointer



An object only responds to a message if its class (pointed to by its `isa` pointer) implements the associated method. Because this happens at runtime, Xcode cannot always figure out at compile time (when the application is built) whether an object will respond to a message. Xcode will give you an error if it thinks you are sending a message to an object that will not respond, but if it is not sure, it will let the application build.

If, for some reason (and there are many possibilities), you end up sending a message to an object that does not respond, your application will throw an *exception*. Exceptions are also known as *run-time errors* because they occur once your application is running, as opposed to *compile-time errors* that show up when your application is being built, or compiled.

To practice dealing with exceptions, you are going to cause one in RandomItems.

In `main.m`, get the last item in the array using the **`lastObject`** method of **`NSArray`**. Then send this item a message that it will not understand:

```
#import <Foundation/Foundation.h>
#import "BNRItem.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        id lastObj = [items lastObject];

        // lastObj is an instance of BNRItem and will not understand the count message
        [lastObj count];

        for (BNRItem *item in items) {
            NSLog(@"%@", item);
        }

        items = nil;
    }
    return 0;
}
```

Build and run the application. Your application will compile, start running, and then halt. Check your console and find the line that looks like this:

```
2014-01-19 12:23:47.990 RandomItems[10288:707] ***
Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
'-[BNRItem count]: unrecognized selector sent to instance 0x100117280'
```

This is what an exception looks like. What exactly is it saying? First, it tells you the date, time, and name of the application. You can ignore that information and focus on what comes after the "***." That line tells you that an exception occurred and the reason.

The reason is the most important piece of information an exception gives you. Here the reason tells you that an *unrecognized selector* was sent to an instance. You know that selector means message. You sent a message to an object, and the object does not implement that method.

The type of the receiver and the name of the message are also in this output, which makes it easier to debug. An instance of **`BNRItem`** was sent the message **`count`**. The `-` at the beginning tells you the receiver was an instance of **`BNRItem`**. A `+` would mean the class itself was the receiver.

There are two important lessons to take away from this. First, always check the console if your application halts or crashes; errors that occur at runtime (exceptions) are just as important as those that occur during compiling. Second, remember that *unrecognized selector* means the message you are sending is not implemented by the receiver. You will make this mistake more than once, and you will want to be able to diagnose it quickly.

Some languages use try and catch blocks to handle exceptions. While Objective-C has this ability, we do not use it very often in application code. Typically, an exception is a programmer error and should be fixed in the code instead of handled at runtime.

In `main.m`, remove the exception-causing code.

```
        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:p];
        }

        id lastObj = [items lastObject];
        [lastObj count];

        for (BNRItem *item in items) {
            NSLog(@"%@", item);
        }
```

Challenges

Most chapters in this book will finish with at least one challenge that encourages you to take your work in the chapter one step further and prove to yourself what you have learned. We suggest that you tackle as many of these challenges as you can to cement your knowledge and move from *learning* iOS development from us to *doing* iOS development on your own.

Challenges come in three levels of difficulty:

- Bronze challenges typically ask you to do something very similar to what you did in the chapter. These challenges reinforce what you learned in the chapter and force you to type in similar code without having it laid out in front of you. Practice makes perfect.
- Silver challenges require you to do more digging and more thinking. You will need to use methods, classes, and properties that you have not seen before, but the tasks are still similar to what you did in the chapter.
- Gold challenges are difficult and can take hours to complete. They require you to understand the concepts from the chapter and then do some quality thinking and problem-solving on your own. Tackling these challenges will prepare you for the real-world work of iOS development.

Before beginning any challenge, *always make a copy of your project directory in Finder and attack the challenge in that copy*. Many chapters build on previous chapters, and working on challenges in a copy of the project assures you will be able to progress through the book.

Bronze Challenge: Bug Finding

Create a bug in your program by asking for the eleventh item in the array. Run it and note the exception that gets thrown.

Silver Challenge: Another Initializer

Create another initializer method for the **`BNRItem`** class. This initializer is *not* the designated initializer of **`BNRItem`**. It takes an instance of **`NSString`** that identifies the `itemName` of the item and an instance of **`NSString`** that identifies the `serialNumber`.

Gold Challenge: Another Class

Create a subclass of **`BNRItem`** named **`BNRContainer`**. An instance of **`BNRContainer`** should have an array of `subitems` that contains instances of **`BNRItem`**. Printing the description of a **`BNRContainer`** object should show you the name of the container, its value in dollars (a sum of all items in the container plus the value of the container itself), and a list of every instance of **`BNRItem`** it contains. A properly-written **`BNRContainer`** class can contain instances of **`BNRContainer`**. It can also report back its full value and every contained item properly.

Are You More Curious?

In addition to Challenges, many chapters will conclude with one or more "For the More Curious" sections. These sections offer deeper explanations of or additional information about the topics presented in the chapter. The knowledge in these sections is not absolutely essential to get you where you are going, but we hope you will find it interesting and useful.

For the More Curious: Class Names

In simple applications like RandomItems, you only need a few classes. As applications grow larger and more complex, the number of classes grows. At some point, you will run into a situation where you have two classes that could easily be named the same thing. This is bad news. If two classes have the same name, it is impossible for your program to figure out which one it should use. This is known as a *namespace collision*.

Other languages solve this problem by declaring classes inside a *namespace*. You can think of a namespace as a group to which classes belong. To use a class in these languages, you have to specify both the class name and the namespace.

Objective-C has no notion of namespaces. Instead, class names are prefixed with two or three letters to keep them distinct. For example, in this exercise, the class was named **`BNRItem`** instead of **`Item`**.

Stylish Objective-C programmers always prefix their classes. The prefix is typically related to the name of the application you are developing or the library that it belongs to. For example, if I were writing an application named "MovieViewer," I would prefix all classes with **`MOV`**. Classes that you will use across multiple projects typically bear a prefix that is related to your name (**`CBK`**), your company's name (**`BNR`**), or a portable library (a library for dealing with maps might use **`MAP`**).

Notice that Apple's classes have prefixes, too. Apple's classes are organized into frameworks, and each framework has its own prefix. For instance, the **`UILabel`** class belongs to the UIKit framework. The classes **`NSArray`** and **`NSString`** belong to the Foundation framework. (The **`NS`** stands for NeXTSTEP, the platform for which these classes were originally designed.)

For your classes, you should use three-letter prefixes. Two-letter prefixes are reserved by Apple for use in framework classes. Although nothing is stopping you from creating a class with a two-letter prefix, you should use three-letter prefixes to eliminate the possibility of namespace collisions with Apple's present and future classes.

For the More Curious: #import and @import

When Objective-C was new, the system did not ship with many classes. Eventually, however, there were enough classes that it become necessary to organize them into frameworks. In your source code, you would typically `#import` the master header file for a framework:

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

And that file would #import all the headers in that framework, like this:

```
#import <Foundation/NSArray.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSBundle.h>
#import <Foundation/NSByteOrder.h>
#import <Foundation/NSCalendar.h>
#import <Foundation/NSCharacterSet.h>
...
```

And then you would explicitly link that framework into your program at compile time.

This was easy to implement; it was using the existing C preprocessor to copy all these headers into the file that was about to be compiled.

This approach worked pretty well for about a decade. Then, as more classes were added to the frameworks and more frameworks went into each project, we noticed that the compiler was spending most of its time parsing and processing those same standard headers again and again. So, the *precompiled header file* was added to every project. The first time you compiled your project, the headers listed in the that file would be compiled once and the result would be cached away. Having this pre-digested clump of headers made compiling all the other files much, much faster. The project you just created has the file `RandomItems-Prefix.pch` and it forces the build system to precompile the headers for the Foundation framework:

```
#ifdef __OBJC__
    #import <Foundation/Foundation.h>
#endif
```

You still had to explicitly link that framework into your program at compile time.

That worked pretty well for another decade, but recently Apple realized that developers were not maintaining their `.pch` files effectively. So, they made the compiler smarter and the `@import` directive was introduced:

```
@import Foundation;
```

This tells the compiler, “Hey, I’m using the Foundation module. You figure out how to make that work.” The compiler is given a lot of freedom to optimize the preprocessing and caching of header files. (This also eliminates the need to explicitly link the module into the program – when the compiler sees the `@import`, it makes a note to link in the appropriate module.)

As we write this, only Apple can create modules that can be used with `@import`. To use classes and frameworks that you create, you will still need to use `#import`.

We are writing this book on Xcode 5.0, and `#import` still appears in the template projects and files, but we are certain that in the near future `@import` will be ubiquitous.

3

Managing Memory with ARC

In this chapter, you will learn how memory is managed in iOS and the concepts that underlie *automatic reference counting*, or ARC. Let's start with some basics of application memory.

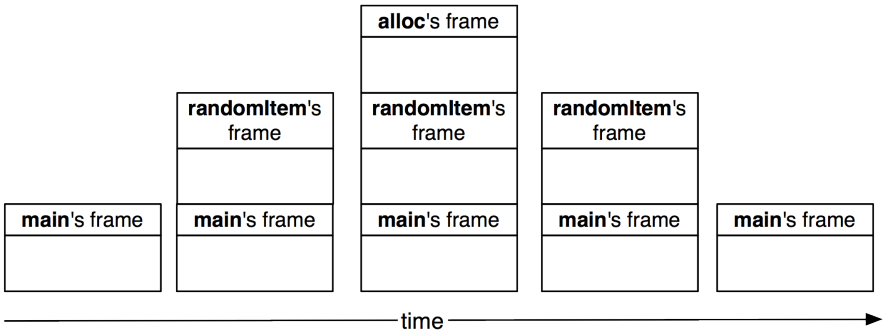
The Stack

When a method (or function) is executed, a chunk of memory is allocated from a part of memory called the *stack*. This chunk of memory is called a *frame*, and the frame stores the values for variables declared inside the method. A variable declared inside a method is called a *local variable*.

When an application launches and runs **`main()`**, the frame for **`main()`** is put on the stack. When **`main()`** calls another method (or function), the frame for that method is put on top of the stack. Of course, that method could call another method, and so on, until you have a towering stack of frames. As each method or function finishes, its frame is "popped off" the top of the stack and destroyed. If the method is called again, a new frame will be allocated and put on the stack.

For example, in the RandomItems application, the **`main`** function runs **`BNRItem`**'s **`randomItem`** method, which in turn runs **`alloc`**. The stack would look like Figure 3.1. Notice that **`main()`**'s frame stays alive while the other methods are executing because it has not yet finished executing.

Figure 3.1 Stack growing and shrinking



The **`randomItem`** method runs inside a loop in **`main()`**. With every iteration of the loop, the stack grows and shrinks as frames are put on and popped off the stack.

The Heap

There is another part of memory called the *heap* that is separate from the stack. The reason for the names "heap" and "stack" has to do with how you visualize them. The stack can be visualized as an

orderly stack of frames. The heap, on the other hand, is where all Objective-C objects live. It is a giant heaping mess of objects. You use pointers to keep track of where those objects are stored in the heap.

When you send the **`alloc`** message to a class, a chunk of memory is allocated from the heap. This chunk is your object, and it includes space for the object's instance variables. An instance of **`BNRItem`** has five instance variables: four pointers (`isa`, `_itemName`, `_serialNumber`, and `_dateCreated`) and an `int` (`_valueInDollars`). Thus, the chunk of memory that is allocated includes space for one `int` and four pointers. These pointers store addresses of other objects in the heap.

An iOS application creates objects at launch and will typically continue to create objects for as long as the application is running. If heap memory were infinite, the application could create all the objects that it wanted to and have them exist for the entire run of the application.

But an application gets only so much heap memory, and memory on an iOS device is especially limited. Thus, this resource must be managed: It is important to destroy objects that are no longer needed to free up heap memory so that it can be reused to create new objects. On the other hand, it is critical not to destroy objects that are still needed.

ARC and memory management

The good news is that you do not need to keep track of which objects should live and die. Your application's memory management is handled for you by ARC, which stands for Automatic Reference Counting. All of the applications in this book will use ARC. Before ARC was available, applications used *manual reference counting*. There is more information about manual reference counting at the end of the chapter.

ARC can be relied on to manage your application's memory automatically for the most part. However, it is important to understand the concepts behind it to know how to step in when you need to. Let's start with the idea of object ownership.

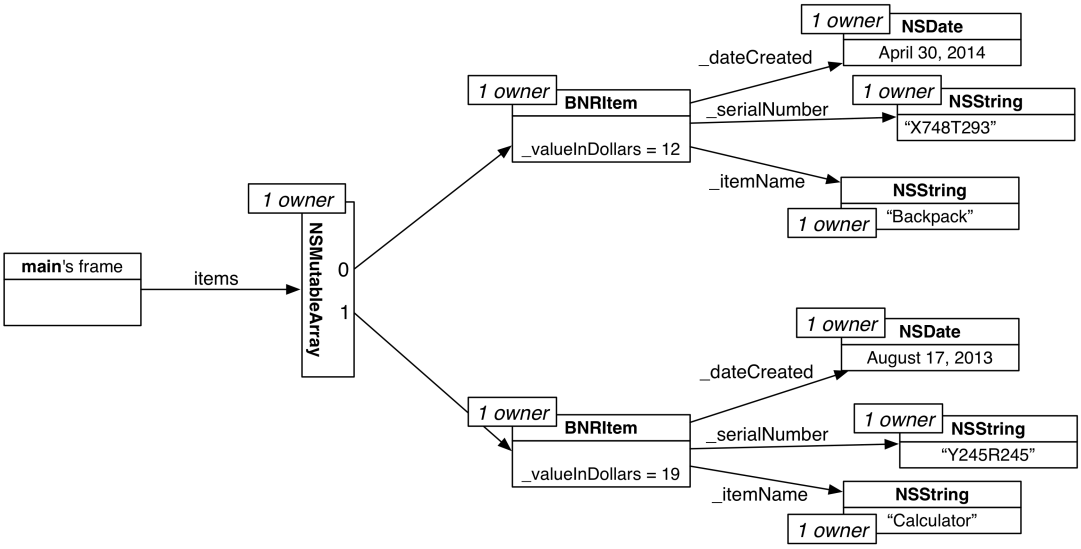
Pointer Variables and Object Ownership

Pointer variables imply *ownership* of the objects that they point to.

- When a method (or function) has a local variable that points to an object, that variable is said to *own* the object being pointed to.
- When an object has an instance variable that points to another object, the object with the pointer is said to *own* the object being pointed to.

Think back to your RandomItems application. In this application, an instance of **`NSMutableArray`** is created in **`main()`** and then ten **`BNRItem`** instances are added to it. Figure 3.2 shows some of the objects in RandomItems and the pointers that reference them.

Figure 3.2 RandomItems object diagram (with only two items)



Within **`main()`**, the local variable `items` points to an instance of **`NSMutableArray`**, so **`main()`** owns that **`NSMutableArray`** instance.

The array, in turn, owns the **`BNRItem`** instances. A collection object, like an instance of **`NSMutableArray`**, holds pointers to objects instead of actually containing them, and these pointers imply ownership: an array always owns the objects that are "in" the array.

Finally, each **`BNRItem`** instance owns the objects pointed to by its instance variables.

The idea of object ownership is useful for determining whether an object will be destroyed so that its memory can be reused.

- *An object with no owners will be destroyed.* An ownerless object cannot be sent messages and is isolated and useless to the application. Keeping it around wastes precious memory. This is called a *memory leak*.

- *An object with one or more owners will not be destroyed.* If an object is destroyed but another object or method still has a pointer to it (or, more accurately, a pointer that stores the address where the object *used* to live), then you have a dangerous situation: sending a message via this pointer may crash your application. Destroying an object that is still needed is called *premature deallocation*. It is also known as a *dangling pointer* or a *dangling reference*.

How objects lose owners

Here are the ways that an object can lose an owner:

- A variable that points to the object is changed to point to another object.

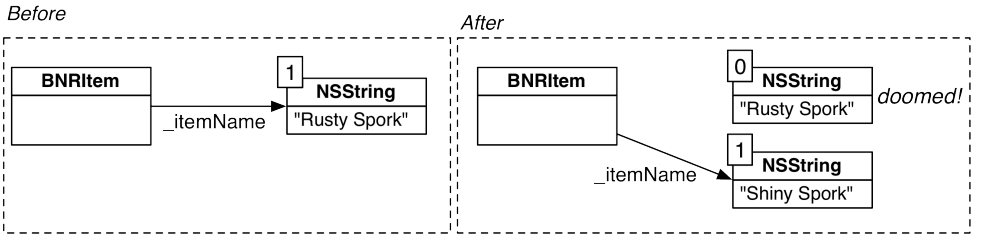
- A variable that points to the object is set to `nil`.

- The owner of the object is itself destroyed.
- An object in a collection, like an array, is removed from that collection.

Let's take a look at each of these situations.

Changing a pointer

Imagine an instance of **`BNRItem`**. Its `_itemName` instance variable points to an **`NSString`** instance `@"RustySpork"`. If you polished the rust off that spork, it would become a shiny spork, and you would want to change the `_itemName` to point at a different **`NSString`**.



When the value of `_itemName` changes from the address of the "Rusty Spork" string to the address of the "Shiny Spork" string, the "Rusty Spork" string loses an owner. If it has no other owners, then it will be destroyed.

Setting a pointer to nil

Setting a pointer to `nil` represents the absence of an object. For example, say you have a **`BNRItem`** instance that represents a television. Then, someone scratches off the television's serial number. You would set its `_serialNumber` instance variable to `nil`. The **`NSString`** instance that `_serialNumber` previously pointed to loses an owner.

The owner is destroyed

When an object is destroyed, the objects that it owns lose an owner. In this way, one object being deallocated can cause a cascade of object deallocations.

Through its local variables, a method or a function can own objects. When the method or function is done executing and its frame is popped off the stack, the objects it owns will lose an owner.

Removing an object from a collection

There is one more important way an object can lose an owner. An object in a collection object is owned by the collection object. When you remove an object from a mutable collection object, like an instance of **`NSMutableArray`**, the removed object loses an owner.

```
[items removeObject:item];    // Object pointed to by item loses an owner
```

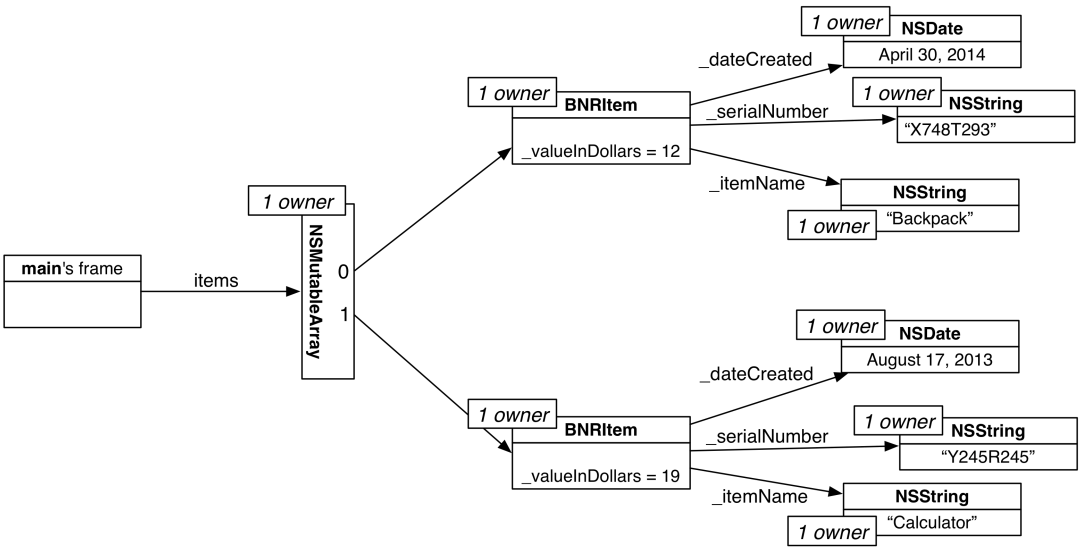
Keep in mind that losing an owner by any of these means does not necessarily result in the object being destroyed; if there is still another pointer to the object somewhere, then the object will continue to exist. When an object loses its last owner, the result is certain and appropriate death.

Ownership chains

Because objects own objects, which can own other objects, the destruction of a single object can set off a chain reaction of loss of ownership, object destruction, and freeing up of memory.

There is an example of this in RandomItems. Take another look at the object diagram for this application.

Figure 3.3 Objects and pointers in RandomItems



In `main.m`, after you finish printing out the array, you set the `items` variable to `nil`. Setting `items` to `nil` causes the array to lose its only owner, so the array is destroyed.

But the destruction does not stop there. When the array is destroyed, all of the pointers to the **`BNRItem`** instances are destroyed. Once these variables are gone, no one owns any of the items, so they are destroyed.

Finally, destroying a **`BNRItem`** destroys its instance variables, which leaves the objects pointed to by those variables unowned. So they, too, get destroyed.

Let's add some code so that you can see this destruction as it happens. **`NSObject`** implements a **`dealloc`** method, which is sent to an object just before it is destroyed. You can override **`dealloc`** in **`BNRItem`** to print something to the console when an item is destroyed.

In the RandomItems project, open `BNRItem.m` and override **`dealloc`**.

```
- (void)dealloc
{
    NSLog(@"Destroyed: %@", self);
}
```

In `main.m`, add the following line of code.

```
NSLog(@"Setting items to nil...");
items = nil;
```

Build and run the application. After the items print out, you will see the message announcing that `items` is being set to `nil`. Then, you will see the destruction of each **`BNRItem`** logged to the console.

At the end, there are no more objects taking up memory, and only the **`main`** function remains. All this automatic clean-up and memory recycling occurs as the result of setting `items` to `nil`. That is the power of ARC.

Strong and Weak References

We have said that anytime a pointer variable points to an object, that object has an owner and will stay alive. This is known as a *strong reference*.

A variable can optionally *not* take ownership of an object that it points to. A variable that does not take ownership of an object is known as a *weak reference*.

A weak reference is useful for preventing a problem called a *strong reference cycle* (also known as a *retain cycle*.) A strong reference cycle occurs when two or more objects have strong references to each other. This is bad news. When two objects own each other, they can never be destroyed by ARC. Even if every other object in the application releases ownership of these objects, these objects (and any objects that they own) will continue to exist inside their bubble of mutual ownership.

Thus, a strong reference cycle is a memory leak that ARC needs your help to fix. You fix it by making one of the references weak.

Let's introduce a strong reference cycle in RandomItems to see how this works. First, you are going to give an instance of **`BNRItem`** the ability to hold another **`BNRItem`** (to represent something like a backpack or a purse). In addition, an item will know which other item holds it.

In `BNRItem.h`, declare two instance variables and their accessors.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    BNRItem *_container;
}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;

- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;
```

In `BNRItem.m`, implement the accessors.

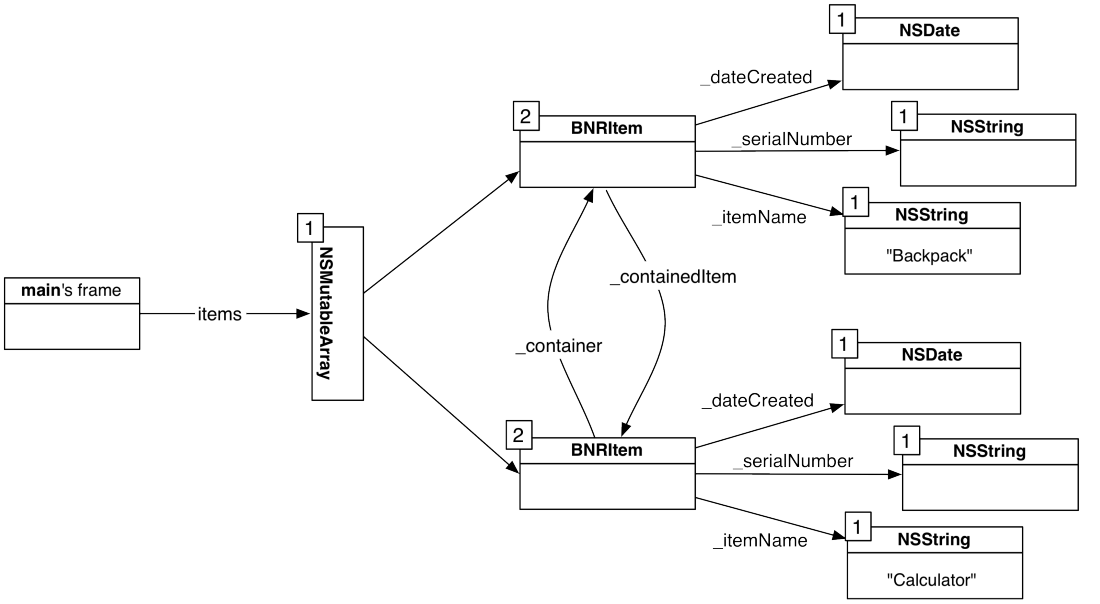
```
- (void)setContainedItem:(BNRItem *)item
{
    _containedItem = item;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    item.container = self;
}

- (BNRItem *)containedItem
{
    return _containedItem;
}

- (void)setContainer:(BNRItem *)item
{
    _container = item;
}

- (BNRItem *)container
{
    return _container;
}
```

In `main.m`, remove the code that populates the array with random items. Then create two new items, add them to the array, and make them point at each other.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        BNRItem *backpack = [[BNRItem alloc] initWithItemName:@"Backpack"];
        [items addObject:backpack];

        BNRItem *calculator = [[BNRItem alloc] initWithItemName:@"Calculator"];
        [items addObject:calculator];

        backpack.containedItem = calculator;

        backpack = nil;
        calculator = nil;

        for (BNRItem *item in items)
            NSLog(@"%@", item);

        NSLog(@"Setting items to nil...");
        items = nil;
    }
    return 0;
}
```

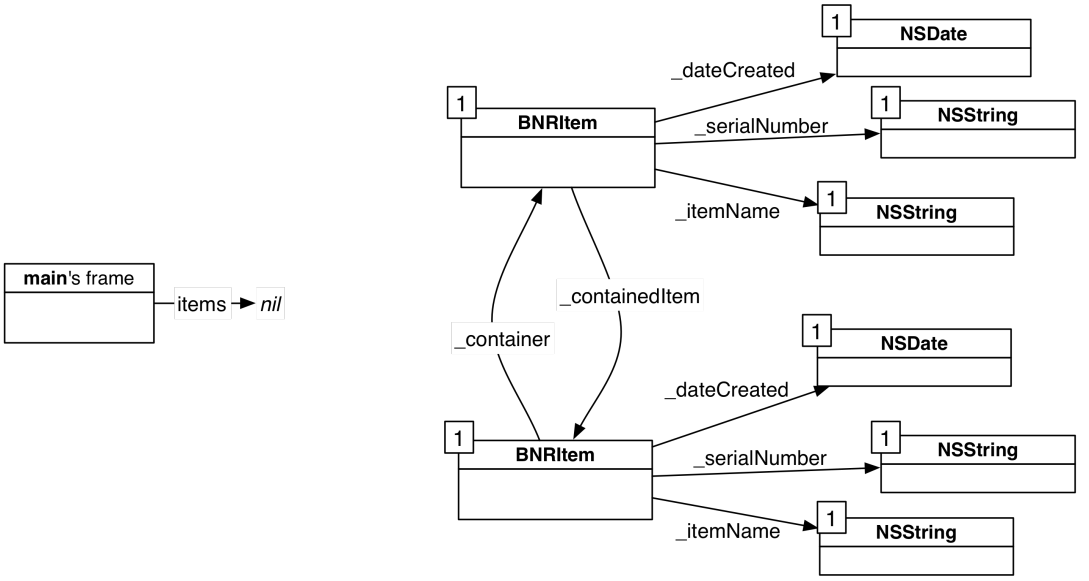
Here is what the application looks like now:

Figure 3.4 RandomItems with strong reference cycle



Build and run the application. This time, you will not see any messages reporting the destruction of the **`BNRItem`** objects. This is a strong reference cycle: the backpack and the calculator have strong references to one another, so there is no way to destroy these objects. Figure 3.5 shows the objects in the application that are still taking up memory once `items` has been set to `nil`.

Figure 3.5 Memory leak!



The two items cannot be accessed by any other part of the application (in this case, **`main()`**), yet they still exist, doing nothing useful. Moreover, because they cannot be destroyed, neither can the objects that their instance variables point to.

To fix this problem, one of the pointers between the items needs to be a weak reference. To decide which one should be weak, think of the objects in the cycle as being in a parent-child relationship. In this relationship, the parent can own its child, but a child should never own its parent. In our strong reference cycle, the backpack is the parent, and the calculator is the child. Thus, the backpack can keep its strong reference to the calculator (the `_containedItem` instance variable), but the calculator's reference to the backpack (the `_container` instance variable) should be weak.

To declare a variable as a weak reference, you use the `__weak` attribute. In `BNRItem.h`, change the `container` instance variable to be a weak reference.

```
__weak BNRItem *_container;
```

Build and run the application again. This time, the objects are destroyed properly.

Most strong reference cycles can be broken down into a parent-child relationship. A parent typically keeps a strong reference to its child, so if a child needs a pointer to its parent, that pointer must be a weak reference to avoid a strong reference cycle.

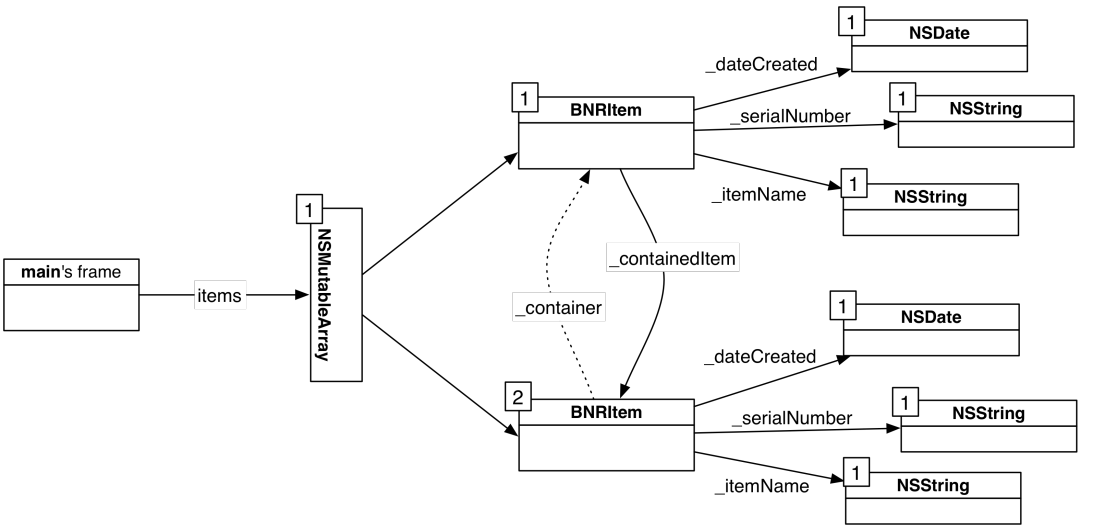
A child holding a strong reference to its *parent's* parent also causes a strong reference cycle. So the same rule applies in this situation: if a child needs a pointer to its parent's parent (or its parent's parent's parent, etc.), then that pointer must be a weak reference.

Apple's development tools includes a Leaks tool to help you find strong reference cycles. You will see how to use this tool in Chapter 14.

A weak reference knows when the object that it points to is destroyed and responds by setting itself to `nil`. Thus, if the backpack is destroyed, the calculator's `_container` instance variable will be automatically set to `nil`. This is convenient. If `_container` was not set to `nil`, then destroying the object would leave you with a dangling pointer, which could crash your application.

Here is the current diagram of RandomItems. Notice that the arrow representing the `container` pointer variable is now a dotted line. A dotted line denotes a weak reference. Strong references are always solid lines.

Figure 3.6 RandomItems with strong reference cycle avoided



Properties

Each time you have declared an instance variable in **BNRItem**, you have declared and implemented a pair of accessor methods. Now you are going to learn to use *properties*, a convenient alternative to writing out accessors methods that saves a lot of typing and makes your class files easier to read.

Declaring properties

A property declaration has the following form:

```
@property NSString *itemName;
```

By default, declaring a property will get you three things: an instance variable and two accessors for the instance variable. Take a look at Table 3.1, which shows a class not using properties on the left and the equivalent class with properties on the right.

Table 3.1 With and without properties

	Without properties	**With properties**

BNRThing.h	`@interface BNRThing : NSObject`
	`{`
	`  NSString *_name;`
	`}`
	`- (void)setName:(NSString *)n;`
	`- (NSString *)name;`
	`@end`
		`@interface BNRThing : NSObject`
		
		`@property NSString *name;`
		
		`@end`

BNRThing.m	`@implementation BNRThing`
	
	`- (void)setName:(NSString *)n`
	`{`
	`  _name = n;`
	`}`
	
	`- (NSString *)name`
	`{`
	`  return _name;`
	`}`
	
	`@end`
		`@implementation BNRThing`
		`@end`

These two classes in Table 3.1 are exactly the same: each has one instance variable for the name of the instance and a setter and getter for the name. On the left, you type out these declarations and instance variables yourself. On the right, you simply declare a property.

You are going to replace your instance variables and accessors in **BNRItem** with properties.

In BNRItem.h, delete the instance variable area and the accessor method declarations. Then, add the property declarations that replace them.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    __weak BNRItem *_container;
}


@property BNRItem *containedItem;
@property BNRItem *container;

@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setItemName:(NSString *)str;
- (NSString *)itemName;

- (void)setSerialNumber:(NSString *)str;
- (NSString *)serialNumber;

- (void)setValueInDollars:(int)v;
- (int)valueInDollars;

- (NSDate *)dateCreated;

- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;

- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;

@end
```

Now, BNRItem.h is much easier to read:

```
@interface BNRItem : NSObject

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

@property BNRItem *containedItem;
@property BNRItem *container;

@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;

@end
```

Notice that the names of the properties are the names of the instance variables minus the underscore. The instance variable generated by a property, however, does have an underscore in its name.

Let's look at an example. When you declared the property named `itemName`, you got an instance variable named `_itemName`, a getter method named **`itemName`**, and a setter method named **`setItemName:`**. (Note that these declarations will not appear in your file; they are declared by the compiler behind the scenes.) Thus, the rest of the code in your application can work as before.

Declaring these properties also takes care of the implementations of the accessors. In `BNRItem.m`, delete the accessor implementations.

```
- (void)setItemName:(NSString *)str
{
    _itemName = str;
}
- (NSString *)itemName
{
    return _itemName;
}
- (void)setSerialNumber:(NSString *)str
{
    _serialNumber = str;
}
- (NSString *)serialNumber
{
    return _serialNumber;
}
- (void)setValueInDollars:(int)p
{
    _valueInDollars = p;
}
- (int)valueInDollars
{
    return _valueInDollars;
}
```

```
- (NSDate *)dateCreated
{
    return _dateCreated;
}
- (void)setContainedItem:(BNRItem *)item
{
    _containedItem = item;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    item.container = self;
}
- (BNRItem *)containedItem
{
    return _containedItem;
}

- (void)setContainer:(BNRItem *)item
{
    _container = item;
}
- (BNRItem *)container
{
    return _container;
}
```

You may be wondering about the implementation of **`setContainedItem:`** that you just deleted. This setter did more than just set the `_containedItem` instance variable. It also set the `_container` instance variable of the passed-in item. To replicate this functionality, you will shortly write a custom setter for the `containedItem` property. But first, let's discuss some property basics.

Property attributes

A property has a number of attributes that allow you to modify the behavior of the accessor methods and the instance variable it creates. The attributes are declared in parentheses after the `@property` directive. Here is an example:

```
@property (nonatomic, readwrite, strong) NSString *itemName;
```

Each attribute has a set of possible values, one of which is the default and does not have to be explicitly declared.

Multi-threading attribute

The multi-threading attribute of a property has two values: `nonatomic` or `atomic`. (Multi-threading is outside the scope of this book, but you still need to know the values for this attribute.) Most iOS programmers typically use `nonatomic`. We do at Big Nerd Ranch, and so does Apple. In this book, you will use `nonatomic` for all properties.

Unfortunately, the default value for this attribute is `atomic`, so you have to specify that you want your properties to be `nonatomic`.

In `BNRItem.h`, change all of your properties to be `nonatomic`.

```
@interface BNRItem : NSObject

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
       valueInDollars:(int)value
         serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;

@property (nonatomic) BNRItem *containedItem;
@property (nonatomic) BNRItem *container;

@property (nonatomic) NSString *itemName;
@property (nonatomic) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic) NSDate *dateCreated;
@end
```

Read/write attribute

The read/write attribute's value, `readwrite` or `readonly`, tells the compiler whether to implement a setter method for the property. A `readwrite` property implements both a setter and getter. A `readonly` property just implements a getter. The default option for this attribute is `readwrite`. This is what you want for all of **`BNRItem`**'s properties except `dateCreated`, which should be `readonly`.

In `BNRItem.h`, declare `dateCreated` as a `readonly` property so that no setter method is generated for this instance variable.

```
@property (nonatomic, readonly) NSDate *dateCreated;
```

Memory management attribute

The memory management attribute's values are `strong`, `weak`, `copy`, and `unsafe_unretained`. This attribute describes the type of reference that the object with the instance variable has to the object that the variable is pointing to.

For properties that do not point to objects (like the `int valueInDollars`), there is no need for memory management, and the only option is `unsafe_unretained`. This is direct assignment. You may also see the value `assign` in some places, which was the term used before ARC.

(The "unsafe" part of `unsafe_unretained` is misleading when dealing with non-object properties. It comes from contrasting unsafe unretained references with weak references. Unlike a weak reference, an unsafe unretained reference is not automatically set to `nil` when the object that it points to is destroyed. This is unsafe because you could end up with dangling pointers. However, the issue of dangling pointers is irrelevant when dealing with non-object properties.)

As the only option, `unsafe_unretained` is also the default value for non-object properties, so you can leave the `valueInDollars` property as is.

For properties that manage a pointer to an Objective-C object, all four options are possible. The default is `strong`. However, Objective-C programmers tend to explicitly declare this attribute. (One reason is that the default value has changed in the last few years, and that could happen again.)

In `BNRItem.m`, set the memory management attribute as `strong` for the `containedItem` and `dateCreated` properties and `weak` for the `container` property.

```
@property (nonatomic, strong) BNRItem *containedItem;
@property (nonatomic, weak) BNRItem *container;

@property (nonatomic) NSString *itemName;
@property (nonatomic) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic, readonly, strong) NSDate *dateCreated;
```

Setting the `container` property to `weak` prevents the strong reference cycle that you caused and fixed earlier.

What about the `itemName` and `serialNumber` properties? These point to instances of **`NSString`**. When a property points to an instance of a class that has a mutable subclass (like **`NSString`**/**`NSMutableString`** or **`NSArray`**/**`NSMutableArray`**), you should set its memory management attribute to `copy`.

In `BNRItem.m`, set the memory management attribute for `itemName` and `serialNumber` as `copy`.

```
@property (nonatomic, strong) BNRItem *containedItem;
@property (nonatomic, weak) BNRItem *container;

@property (nonatomic, copy) NSString *itemName;
@property (nonatomic, copy) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic, readonly, strong) NSDate *dateCreated;
```

Here is what the generated setter for `itemName` will look like:

```
- (void)setItemName:(NSString *)itemName
{
    _itemName = [itemName copy];
}
```

Instead of setting `_itemName` to point to the incoming object, this setter sends the message **`copy`** to that object. The **`copy`** method returns an immutable **`NSString`** object that has the same values as the original string, and `_itemName` is set to point at the new string.

Why is it safer to do this for **`NSString`**? It is safer to make a copy of the object rather than risk pointing to a possibly mutable object that could have other owners who might change the object without your knowledge.

For instance, imagine if an item was initialized so that its `itemName` pointed to an **`NSMutableString`**.

```
NSMutableString *mutableString = [[NSMutableString alloc] init];

BNRItem *item = [[BNRItem alloc] initWithItemName:mutableString
                                    valueInDollars:5
                                      serialNumber:@"4F2W7"]];
```

This code is valid because an instance of **`NSMutableString`** is also an instance of its superclass, **`NSString`**. The problem is that the string pointed to by `mutableString` can be changed without the knowledge of the item that also points to it.

In your application, you are not going to change this string unless you mean to. However, when you write classes for others to use, you cannot control how they will be used, and you have to program defensively.

In this case, the defense is to declare `itemName` with the `copy` attribute.

In terms of ownership, `copy` gives you a strong reference to the object pointed to. The original string is not modified in any way: it does not gain or lose an owner, and none of its data changes.

While it is wise to make a copy of an mutable object, it is wasteful to make a copy of an immutable object. An immutable object cannot be changed, so the kind of problem described above cannot occur. To prevent needless copying, immutable classes implement **`copy`** to quietly return a pointer to the original and immutable object.

Custom accessors with properties

By default, the accessors that a property implements are very simple and look like this:

```
- (void)setContainedItem:(BNRItem *)item
{
    _containedItem = item;
}
- (BNRItem *)containedItem
{
    return _containedItem;
}
```

For most properties, this is exactly what you want. However, for the `containedItem` property, the default setter method is not sufficient. The implementation of **`setContainedItem:`** needs an extra step: it should also set the `container` property of the item being contained.

You can replace the default setter by implementing the setter yourself in the implementation file.

In `BNRItem.m`, add back an implementation for **`setContainedItem:`**.

```
- (void)setContainedItem:(BNRItem *)containedItem
{
    _containedItem = containedItem;
    self.containedItem.container = self;
}
```

When the compiler sees that you have implemented **`setContainedItem:`**, it will not create a default setter for `containedItem`. It will still create the getter method, **`containedItem`**.

Note that if you implement both a custom setter and a custom getter (or just a custom getter on a read-only property), then the compiler will not create an instance variable for your property. If you need one, you must declare it yourself.

Note the moral: sometimes the default accessors do not do what you need, and you will need to implement them yourself.

Now you can build and run the application. The leaner **`BNRItem`** works in the exact same way.

For the More Curious: Property Synthesis

When explaining properties in this chapter, we noted that a property automatically generates the implementation for the accessor methods and it declares and creates an instance variable. While this is true, we omitted the fact that this behavior is only the default and you have other options.

Declaring a property in a class interface only declares the accessor methods in a class interface. In order for a property to automatically generate an instance variable and the implementations for its methods, it must be *synthesized*, either implicitly or explicitly. Properties are implicitly synthesized by default. A property is explicitly synthesized by using the `@synthesize` directive in an implementation file:

```
@implementation Person

// Generates the code for -setAge: and -age,
// and creates the instance variable _age
@synthesize age = _age;


// Other methods go here

@end
```

This is how properties are automatically synthesized. The first attribute (`age`) says "create methods named **`age`** and **`setAge:`**," and the second attribute (`_age`) says "the instance variable that backs these methods should be `_age`."

You can optionally leave off the variable name, which creates a backing variable with the same name as the accessors.

```
@synthesize age;
// Is the same as:
@synthesize age = age;
```

There are cases where you do not want an instance variable to back a property and therefore do not want a property to automatically generate the accessor method implementations. Consider a **`Person`** class with three properties, `spouse`, `lastName`, and `lastNameOfSpouse`:

```
@interface Person : NSObject
@property (nonatomic, strong) Person *spouse;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, copy) NSString *lastNameOfSpouse;
@end
```

In this somewhat contrived example, it makes sense for both the `spouse` and `lastName` properties to be backed by an instance variable. After all, this is information that each **`Person`** needs to hang onto. However, it does not make sense to hold onto the last name of the spouse as an instance variable. A **`Person`** can just ask their `spouse` for their `lastName`, so storing this information in both **`Person`** instances is redundant and therefore prone to error. Instead, the **`Person`** class would implement the getter and setter for the `lastNameOfSpouse` property like so:

```
@implementation Person
- (void)setLastNameOfSpouse:(NSString *)lastNameOfSpouse
{
    self.spouse.lastName = lastNameOfSpouse;
}

- (NSString *)lastNameOfSpouse
{
    return self.spouse.lastName;
}
@end
```

In this case, because you have implemented both accessors, the compiler will not automatically synthesize an instance variable for `lastNameOfSpouse`. Which is exactly what you would hope for.

For the More Curious: Autorelease Pool and ARC History

Before automatic reference counting (ARC) was added to Objective-C, we had *manual reference counting*. With manual reference counting, ownership changes only happened when you sent an explicit message to an object.

```
[anObject release]; // anObject loses an owner

[anObject retain]; // anObject gains an owner
```

This was a bummer: Forgetting to send **`release`** to an object before setting a pointer to point at something else would create a memory leak. Sending **`release`** to an object if you had not previously sent **`retain`** to the object was a premature deallocation. A lot of time was spent debugging these problems, which could become very complex in large projects.

During the dark days of manual reference counting, Apple was contributing to an open source project known as the Clang static analyzer and integrating it into Xcode. You will see more about the static analyzer in Chapter 14, but the basic gist is that it could analyze code and tell you if you were doing something silly. Two of the silly things it could detect were memory leaks and premature deallocations. Smart programmers would run their code through the static analyzer to detect these problems and then write the necessary code to fix them.

Eventually, the static analyzer got so good that Apple thought, "Why not just let the static analyzer insert all of the retain and release messages?" Thus, ARC was born. People rejoiced in the streets, and most memory management problems became a thing of the past.

Another thing programmers had to understand in the days of manual reference counting was the *autorelease pool*. When an object was sent the message **`autorelease`**, the autorelease pool would take ownership of an object temporarily so that it could be returned from the method that created it without burdening the creator or the receiver with ownership responsibilities. This was crucial for convenience methods that created a new instance of some object and returned it:

```
+ (BNRItem *)someItem
{
    BNRItem *item = [[[BNRItem alloc] init] autorelease];

    return item;
}
```

Because you had to send the **`release`** message to an object to relinquish ownership, the caller of this method had to understand its its ownership responsibilities. But it was easy to get confused.

```
BNRItem *item = [BNRItem someItem]; // I guess I own this now?

NSString *string = [item itemName]; // Well, if I own that, do I own this?
```

Thus, objects created by methods other than **`alloc`** and **`copy`** would be sent **`autorelease`** before being returned, and the receiver of the object would take ownership as needed or just let it be destroyed when the autorelease pool was drained.

With ARC, this is done automatically (and sometimes optimized out completely). An autorelease pool is created by the `@autoreleasepool` directive followed by curly braces. Inside those curly braces, any newly instantiated object returned from a method that does not have **`alloc`** or **`copy`** in its name is placed in that autorelease pool. When the curly brace closes, any object in the pool loses an owner.

```
@autoreleasepool {
    // Get a BNRItem back from a method that created it,
    // method does not say alloc/copy
    BNRItem *item = [BNRItem someItem];
} // Pool is drained, item loses an owner and is destroyed
```

iOS applications automatically create an autorelease pool for you, and you really do not have to concern yourself with it. But isn't it nice to know what that `@autoreleasepool` is for?

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X