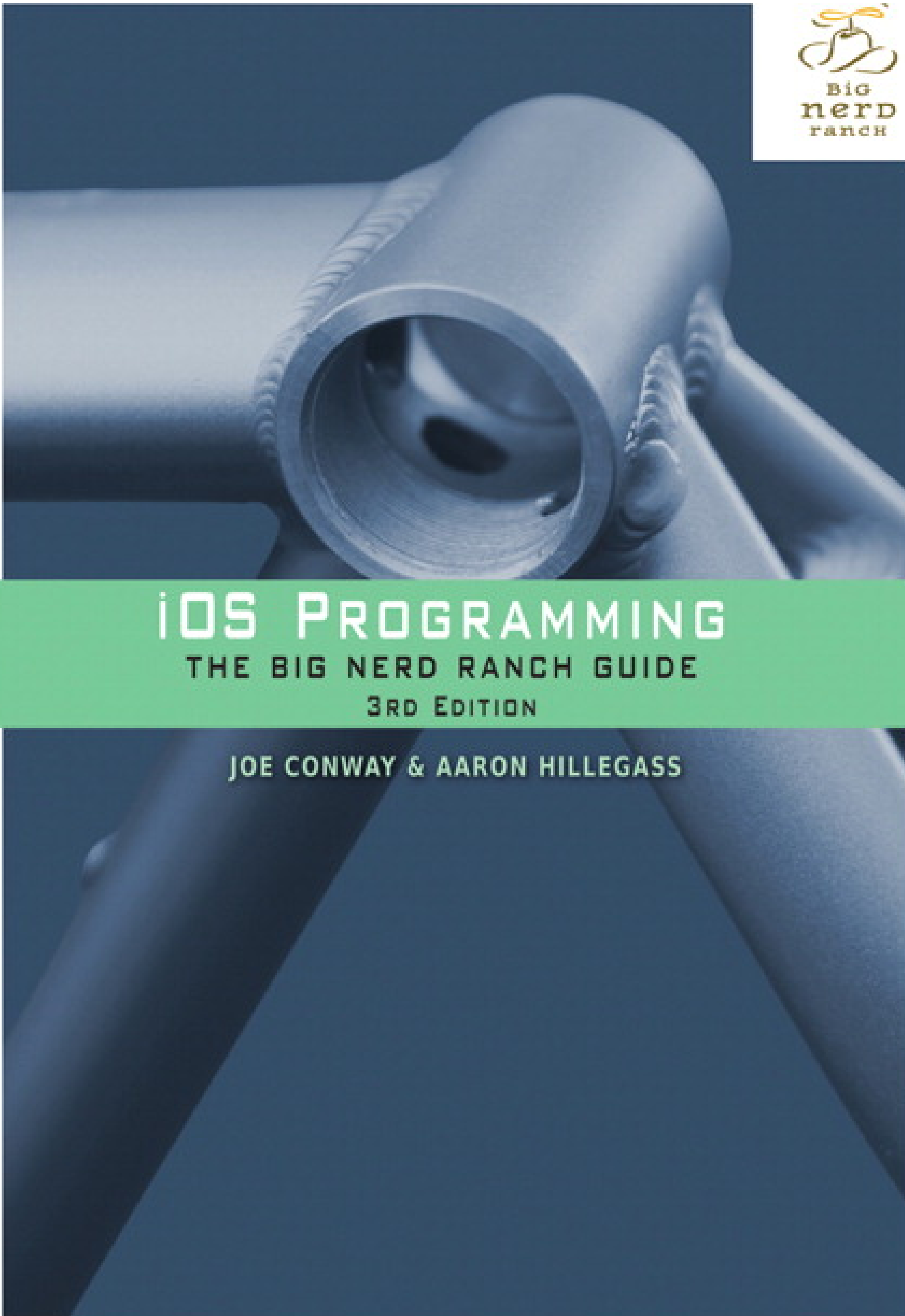


iOS Programming

The Big Nerd Ranch Guide

3rd Edition

Joe Conway & Aaron Hillegass

Big Nerd Ranch

iOS Programming

The Big Nerd Ranch Guide

JOE CONWAY & AARON HILLEGASS

Big Nerd Ranch

iOS Programming: The Big Nerd Ranch Guide

by Joe Conway and Aaron Hillegass

Big Nerd Ranch, Inc.
1989 College Ave.
Atlanta, GA 30317
(404) 478-9005
http://www.bignerdranch.com/
book-comments@bignerdranch.com

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0321821521
ISBN-13 978-0321821522

Third edition, second printing, August 2012

Acknowledgments

While our names appear on the cover, many people helped make this book a reality. We would like to take this chance to thank them.

- The other instructors who teach the iOS Bootcamp fed us with a never-ending stream of suggestions and corrections. They are Scott Ritchie, Brian Hardy, Mikey Ward, Christian Keur, Alex Silverman, Owen Matthews, Brian Turner, Juan Pablo Claude, and Bolot Kerimbaev.
- Our tireless editor, Susan Loper, took our distracted mumblings and made them into readable prose.
- Our technical reviewers, Bill Monk and Jawwad Ahmad, helped us find and fix flaws.
- Ellie Volckhausen designed the cover. (The photo is of the bottom bracket of a bicycle frame.)
- Chris Loper at IntelligentEnglish.com designed and produced the print book and the EPUB and Kindle versions.
- The amazing team at Pearson Technology Group patiently guided us through the business end of book publishing.

The final and most important thanks goes to our students whose questions inspired us to write this book and whose frustrations inspired us to make it clear and comprehensible.

Table of Contents

Introduction

An aspiring iOS developer faces three basic hurdles:

- *You must learn the Objective-C language.* Objective-C is a small and simple extension to the C language. After the first four chapters of this book, you will have a working knowledge of Objective-C.

- *You must master the big ideas.* These include things like memory management techniques, delegation, archiving, and the proper use of view controllers. The big ideas take a few days to understand. When you reach the halfway point of this book, you will understand these big ideas.

- *You must master the frameworks.* The eventual goal is to know how to use every method of every class in every framework in iOS. This is a project for a lifetime: there are over 3000 methods and more than 200 classes available in iOS. To make things even worse, Apple adds new classes and new methods with every release of iOS. In this book, you will be introduced to each of the subsystems that make up the iOS SDK, but we will not study each one deeply. Instead, our goal is get you to the point where you can search and understand Apple's reference documentation.

We have used this material many times at our iOS Development Bootcamp at Big Nerd Ranch. It is well-tested and has helped hundreds of people become iOS application developers. We sincerely hope that it proves useful to you.

Prerequisites

This book assumes that you are already motivated to learn to write iOS apps. We won't spend any time convincing you that the iPhone, the iPad, and the iPod touch are compelling pieces of technology.

We also assume that you know the C programming language and something about object-oriented programming. If this is not true, you should probably start with an introductory book on C and Objective-C, such as *Objective-C Programming: The Big Nerd Ranch Guide*.

What's Changed in the Third Edition?

This edition assumes that the reader is using Xcode 4.3 and running applications on an iOS 5 device or simulator.

With iOS 5, automatic reference counting (ARC) is the default memory management for iOS. We've redone the memory management chapter to address ARC, and we use ARC throughout the book.

You'll find new chapters on using gesture recognizers, storyboards, **`NSRegularExpression`**, and iCloud. We've also added two chapters dedicated to the the Model-View-Controller-Store design pattern, which we use at Big Nerd Ranch and believe is well-suited for many iOS applications.

Besides these obvious changes, we made thousands of tiny improvements that were inspired by questions from our readers and our students. Every page of this book is just a little better than the corresponding page from the second edition.

Our Teaching Philosophy

This book will teach you the essential concepts of iOS programming. At the same time, you'll type in a lot of code and build a bunch of applications. By the end of the book, you'll have knowledge *and* experience. However, all the knowledge shouldn't (and, in this book, won't) come first. That's sort of the traditional way we've all come to know and hate. Instead, we take a learn-while-doing approach. Development concepts and actual coding go together.

Here's what we've learned over the years of teaching iOS programming:

- We've learned what ideas people must have to get started programming, and we focus on that subset.
- We've learned that people learn best when these concepts are introduced *as they are needed*.
- We've learned that programming knowledge and experience grow best when they grow together.
- We've learned that "going through the motions" is much more important than it sounds. Many times we'll ask you to start typing in code before you understand it. We get that you may feel like a trained monkey typing in a bunch of code that you don't fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That's why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.

What does this mean for you, the reader? To learn this way takes some trust. And we appreciate yours. It also takes patience. As we lead you through these chapters, we will try to keep you comfortable and tell you what's happening. However, there will be times when you'll have to take our word for it. (If you think this will bug you, keep reading – we've got some ideas that might help.) Don't get discouraged if you run across a concept that you don't understand right away. Remember that we're intentionally *not* providing all the knowledge you will ever need all at once. If a concept seems unclear, we will likely discuss it in more detail later when it becomes necessary. And some things that aren't clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

People learn differently. It's possible that you will love how we hand out concepts on an as-needed basis. It's also possible that you'll find it frustrating. In case of the latter, here are some options:

- Take a deep breath and wait it out. We'll get there, and so will you.
- Check the index. We'll let it slide if you look ahead and read through a more advanced discussion that occurs later in the book.
- Check the online Apple documentation. This is an essential developer tool, and you'll want plenty of practice using it. Consult it early and often.
- If it's Objective-C or object-oriented programming concepts that are giving you a hard time (or if you think they will), you might consider backing up and reading our *Objective-C Programming: The Big Nerd Ranch Guide*.

How To Use This Book

This book is based on the class we teach at Big Nerd Ranch. As such, it was designed to be consumed in a certain manner.

Set yourself a reasonable goal, like “I will do one chapter every day.” When you sit down to attack a chapter, find a quiet place where you won’t be interrupted for at least an hour. Shut down your email, your Twitter client, and your chat program. This is not a time for multi-tasking; you will need to concentrate.

Do the actual programming. You can read through a chapter first, if you’d like. But the real learning comes when you sit down and code as you go. You will not really understand the idea until you have written a program that uses it and, perhaps more importantly, debugged that program.

A couple of the exercises require supporting files. For example, in the first chapter you will need an icon for your Quiz application, and we have one for you. You can download the resources and solutions to the exercises from `http://www.bignerdranch.com/solutions/iOSProgramming3ed.zip`.

There are two types of learning. When you learn about the Civil War, you are simply adding details to a scaffolding of ideas that you already understand. This is what we will call “Easy Learning”. Yes, learning about the Civil War can take a long time, but you are seldom flummoxed by it. Learning iOS programming, on the other hand, is “Hard Learning,” and you may find yourself quite baffled at times, especially in the first few days. In writing this book, we have tried to create an experience that will ease you over the bumps in the learning curve. Here are two things you can do to make the journey easier:

- Find someone who already knows how to write iOS applications and will answer your questions. In particular, getting your application onto the device the first time is usually very frustrating if you are doing it without the help of an experienced developer.
- Get enough sleep. Sleepy people don’t remember what they have learned.

How This Book Is Organized

In this book, each chapter addresses one or more ideas of iOS development followed by hands-on practice. For more coding practice, we issue challenges towards the end of each chapter. We encourage you to take on at least some of these. They are excellent for firming up the concepts introduced in the chapter and making you a more confident iOS programmer. Finally, most chapters conclude with one or two “For the More Curious” sections that explain certain consequences of the concepts that were introduced earlier.

Chapter 1 introduces you to iOS programming as you build and deploy a tiny application. You’ll get your feet wet with Xcode and the iOS simulator along with all the steps for creating projects and files. The chapter includes a discussion of Model-View-Controller and how it relates to iOS development.

Chapters 2 and 3 provide an overview of Objective-C and memory management. Although you won’t create an iOS application in these two chapters, you will build and debug a tool called RandomPossessions to ground you in these concepts.

In Chapters 4 and 5, you will learn about the Core Location and MapKit frameworks and create a mapping application called Whereami. You will also get plenty of experience with the important design pattern of delegation as well as working with protocols, frameworks, object diagrams, the debugger, and the Apple documentation.

Chapters 6 and 7 focus on the iOS user interface with the Hypnosister and HypnoTime applications. You will get lots of practice working with views and view controllers as well as implementing panning, zooming, and navigating between screens using a tab bar.

In Chapter 8, you will create a smaller application named HeavyRotation while learning about notifications and how to implement autorotation in an application. You will also use autoresizing to make HeavyRotation iPad-friendly.

Chapter 9 introduces the largest application in the book – Homepwner. (By the way, "Homepwner" is not a typo; you can find the definition of "pwn" at www.urbandictionary.com.) This application keeps a record of your possessions in case of fire or other catastrophe. Homepwner will take nine chapters total to complete.

In Chapters 9, 10, and 15, you will build experience with tables. You will learn about table views, their view controllers, and their data sources. You will learn how to display data in a table, how to allow the user to edit the table, and how to improve the interface.

Chapter 11 builds on the navigation experience gained in Chapter 7. You will learn how to use **UINavigationController**, and you will give Homepwner a drill-down interface and a navigation bar.

In Chapter 12, you'll learn how to take pictures with the camera and how to display and store images in Homepwner. You'll use **NSDictionary** and **UIImagePickerController**.

In Chapter 13, you'll learn about **UIPopoverController** for the iPad and modal view controllers. In addition, you will make Homepwner a universal application – an application that runs natively on both the iPhone and the iPad.

Chapter 14 delves into ways to save and load data. In particular, you will archive data in the Homepwner application using the NSCoding protocol. The chapter also explains the transitions between application states, such as active, background, and suspended.

Chapter 16 is an introduction to Core Data. You will change the Homepwner application to store and load its data using an **NSManagedObjectContext**.

Chapter 17 introduces the concepts and techniques of internationalization and localization. You will learn about **NSLocale**, strings tables, and **NSBundle** as you localize parts of Homepwner. This chapter will complete the Homepwner application.

In Chapter 18, you will use **NSUserDefaults** to save user preferences in a persistent manner.

In Chapters 19 and 20, you'll create a drawing application named TouchTracker to learn about touch events. You'll see how to add multi-touch capability and how to use **UIGestureRecognizer** to respond to particular gestures. You'll also get experience with the first responder and responder chain concepts and more practice with **NSDictionary**.

In Chapter 21, you'll learn how to use Instruments to optimize the performance of your applications. This chapter also includes explanations of Xcode schemes and the static analyzer.

Chapters 22 and 23 introduce layers and the Core Animation framework with a brief return to the HypnoTime application to implement animations. You will learn about implicit animations and animation objects, like **CABasicAnimation** and **CAKeyframeAnimation**.

Chapter 24 covers a new feature of iOS for building applications called storyboards. You'll piece together an application using **UIStoryboard** and learn more about the pros and cons of using storyboards to construct your applications.

Chapter 25 ventures into the wide world of web services as you create the Nerdfeed application. This application fetches and parses an RSS feed from a server using **NSURLConnection** and **NSXMLParser**. Nerdfeed will also display a web page in a **UIWebView**.

In Chapter 26, you will learn about **`UISplitViewController`** and add a split view user interface to Nerdfeed to take advantage of the iPad's larger screen size.

Chapter 27 will teach you about the how and why of blocks – an increasingly important feature of the iOS SDK. You'll create a simple application to prepare for using blocks in Nerdfeed in the next chapter.

In Chapters 28 and 29, you will change the architecture of the Nerdfeed application so that it uses the Model-View-Controller-Store design pattern. You'll learn about request logic and how to best design an application that communicates with external sources of data.

In Chapter 30, you'll learn how to enable an application to use iCloud to synchronize and back up data across a user's iOS devices.

Style Choices

This book contains a lot of code. We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple's sample code or code you might find in other books. You may not understand these points now, but it is best that we spell them out before you commit to reading this book:

- There is an alternative syntax for calling accessor methods known as *dot-notation*. In this book, we will explain dot-notation, but we will not use it. For us and for most beginners, dot-notation tends to obfuscate what is really happening.

- In our subclasses of **`UIViewController`**, we always change the designated initializer to **`init`**. It is our opinion that the creator of the instance should not need to know the name of the XIB file that the view controller uses, or even if it has a XIB file at all.

- We will always create view controllers programmatically. Some programmers will instantiate view controllers inside XIB files. We've found this practice leads to projects that are difficult to comprehend and debug.

- We will nearly always start a project with the simplest template project: the empty application. The boilerplate code in the other template projects doesn't follow the rules that precede this one, so we think they make a poor basis upon which to build.

We believe that following these rules makes our code easier to understand and easier to maintain. After you have worked through this book (where you *will* do it our way), you should try breaking the rules to see if we're wrong.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Class names, method names, and function names appear in a bold, fixed-width font. Class names start with capital letters, and method names start with lowercase letters. In this book, method and function names will be formatted the same for simplicity's sake. For example, "In the **`loadView`** method of the **`RexViewController`** class, use the **`NSLog`** function to print the value to the console."

Variables, constants, and types appear in a fixed-width font but are not bold. So you'll see, "The variable `fido` will be of type `float`. Initialize it to `M_PI`."

Applications and menu choices appear in the Mac system font. For example, "Open Xcode and select New Project... from the File menu."

All code blocks will be in a fixed-width font. Code that you need to type in is always bold. For example, in the following code, you would type in everything but the first and last lines. (Those lines are already in the code and appear here to let you know where to add the new stuff.)

```
@interface QuizAppDelegate : NSObject <UIApplicationDelegate> {
    int currentQuestionIndex;

    // The model objects
    NSMutableArray *questions;
    NSMutableArray *answers;

    // The view objects
    IBOutlet UILabel *questionField;
    IBOutlet UILabel *answerField;
    UIWindow *window;
}
```

Necessary Hardware and Software

You can only develop iOS apps on an Intel Mac. You will need to download Apple's iOS SDK, which includes Xcode (Apple's Integrated Development Environment), the iOS simulator, and other development tools.

You should join Apple's iOS Developer Program, which costs $99/year, for three reasons:

- Downloading the latest developer tools is free for members.
- Only signed apps will run on a device, and only members can sign apps. If you want to test your app on your device, you will need to join.
- You can't put an app in the store until you are a member.

If you are going to take the time to work through this entire book, membership in the iOS Developer Program is, without question, worth the cost. Go to `http://developer.apple.com/programs/ios/` to join.

What about iOS devices? Most of the applications you will develop in the first half of the book are for the iPhone, but you will be able to run them on an iPad. On the iPad screen, iPhone applications appear in an iPhone-sized window. Not a compelling use of the iPad, but that's okay when you're starting with iOS. In these first chapters, you'll be focused on learning the fundamentals of the iOS SDK, and these are the same across iOS devices. Later in the book, we'll look at some iPad-only options and how to make applications run natively on both iOS device families.

Excited yet? Good. Let's get started.

3

Managing Memory with ARC

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

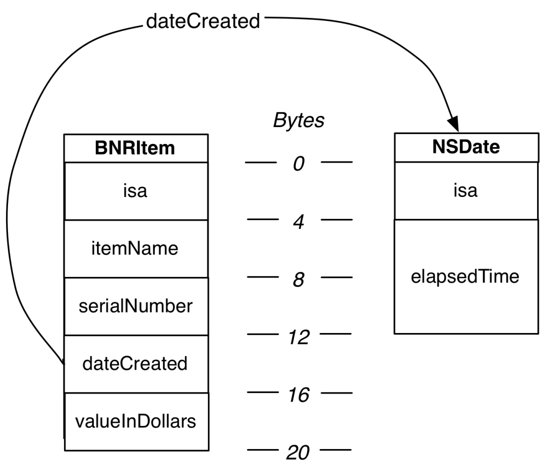
The Heap

All Objective-C objects are stored in a part of memory called *the heap*. When we send an **alloc** message to a class, a chunk of memory is allocated from the heap. This chunk includes space for the object's instance variables.

For example, consider an instance of **NSDate**, which represents a specific point in time. An **NSDate** has two instance variables: a `double` that stores the number of seconds since a fixed reference point in time and the `isa` pointer, which every object inherits from **NSObject**. A `double` is eight bytes, and a pointer is 4 bytes, so each time **alloc** is sent to the **NSDate** class, 12 bytes is allocated from the heap for a new **NSDate** object.

Consider another example: **BNRItem**. A **BNRItem** has five instance variables: four pointers (`isa`, `itemName`, `serialNumber`, and `dateCreated`) and an `int` (`valueInDollars`). The amount of memory needed for an `int` is four bytes, so the total size of a **BNRItem** is 20 bytes (Figure 3.1).

Figure 3.1 Byte count of BNRItem and NSDate instances



Notice in Figure 3.1 that the **NSDate** object does not live inside the **BNRItem**. Objects never live inside one another; they exist separately on the heap. Instead, objects keep references to other objects as needed. These references are the pointer instance variables of an object. Thus, when a **BNRItem**'s `dateCreated` instance variable is set, the *address* of the **NSDate** instance is stored in the **BNRItem**, not the **NSDate** itself. So, if the **NSDate** was 10, 20, or even 1000 bytes, it wouldn't affect the size of the **BNRItem**.)

The Stack

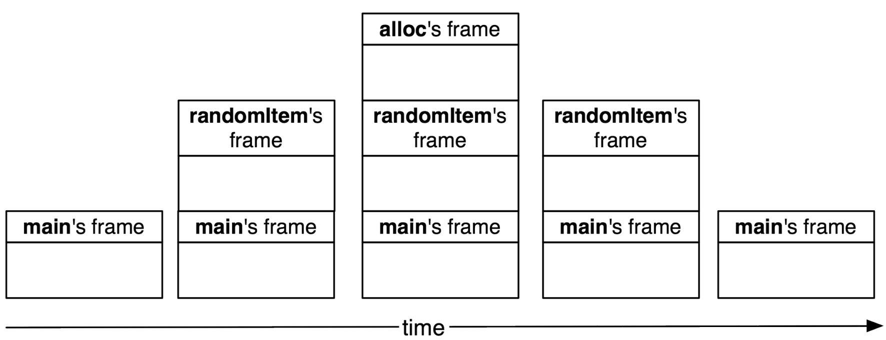
There is another part of memory called the *stack* that is separate from the heap. The reason for the names heap and stack has to do with how we visualize them. The heap is a giant heaping mess of objects, and we use pointers to remember where those objects are stored within the heap. The stack, on the other hand, can be visualized as a physical stack of *frames*.

When a method (or function) is executed, it allocates a chunk of memory from the stack. This chunk of memory is called a frame, and it 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 the **main** function, the frame for **main** is put at the bottom of the stack. When **main** calls another method (or function), the frame for that method is added to the top of the stack. Of course, that method could call another method, and so on, until we have a towering stack of frames. Then, as each method or function finishes, its frame is "popped off" the stack and destroyed. If the method is called again, a brand new frame will be allocated and put on the stack.

For example, in your RandomPossessions application, the **main** function runs **BNRItem**'s **randomItem** method, which in turn runs **BNRItem**'s **alloc** method. The stack would look like Figure 3.2. Notice that **main**'s frame stays alive while the other methods are executing because it has not yet finished executing.

Figure 3.2 Stack growing and shrinking



Recall that the **randomItem** method runs inside of a loop in the **main** function. So with every iteration, the stack grows and shrinks as frames are pushed on and popped off the stack.

Pointer Variables and Object Ownership

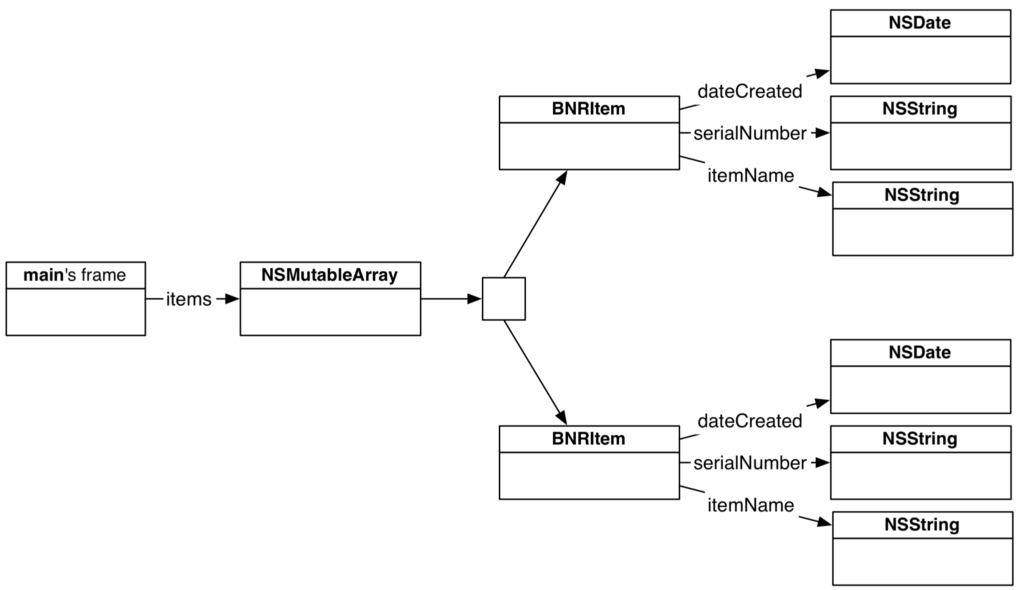
Pointer variables convey *ownership* of the objects that they point to.

- When a method (or function) has a local variable that points to an object, that method 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 RandomPossessions application as a whole. Or, better yet, reopen `RandomPossessions.xcodeproj` and have another look at the code in `main.m`. In this application, an instance of **`NSMutableArray`** is created in the **`main`** function, and then 10 **`BNRItem`** instances are added to the array.

Figure 3.3 shows the objects in RandomPossessions and the pointers that reference them.

Figure 3.3 Objects and pointers in RandomPossessions



The **`NSMutableArray`** is pointed to by the local variable `items` within the **`main`** function, so the **`main`** function owns the **`NSMutableArray`**. Each **`BNRItem`** instance owns the objects pointed to by its instance variables.

In addition, the **`NSMutableArray`** owns the **`BNRItem`**s. Recall that a collection object, like an **`NSMutableArray`**, holds pointers to objects instead of actually containing them. These pointers convey ownership: an array owns the objects it points to.

The relationship between pointers and object ownership is important for understanding memory management in iOS.

Memory Management

If heap memory were infinite, we could create all the objects we needed 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. So it is important to destroy objects that are no longer needed to free up and reuse heap memory. On the other hand, it is critical *not* to destroy objects that *are* still needed.

The idea of object ownership helps us determine whether an object should be destroyed.

- *An object with no owners should 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*.

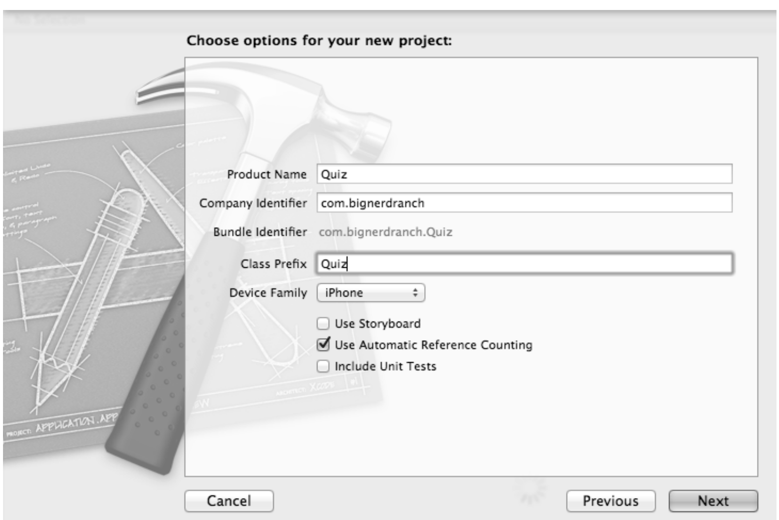
- *An object with at least one owner must not be destroyed.* If an object is destroyed but another object or method still has a pointer to it (or, more accurately, a pointer to where it used to live), then you have a very dangerous situation: sending a message to an object that no longer exists will crash your application. This is called *premature deallocation*.

Using ARC for memory management

The good news is that you don't need to keep track of who owns whom and what pointers still exist. Instead, your application's memory management is handled for you by *automatic reference counting*, or ARC.

In both projects you've built in Xcode so far, you've made sure to Use Automatic Reference Counting when creating the project (Figure 3.4). This won't change; all of your projects in this book will use ARC for managing your application's memory.

Figure 3.4 Naming a new project



(If the Use Automatic Reference Counting box in Figure 3.4 was unchecked, the application would use *manual reference counting* instead, which was the only type of memory management available before iOS 5. For more information about manual reference counting and **retain** and **release** messages, see the For the More Curious section at the end of this chapter.)

ARC can be relied on to manage your application's memory automatically for the most part, but it's important to understand the concepts behind it to know how to step in when you need to. So let's return to the idea of object ownership.

How objects lose owners

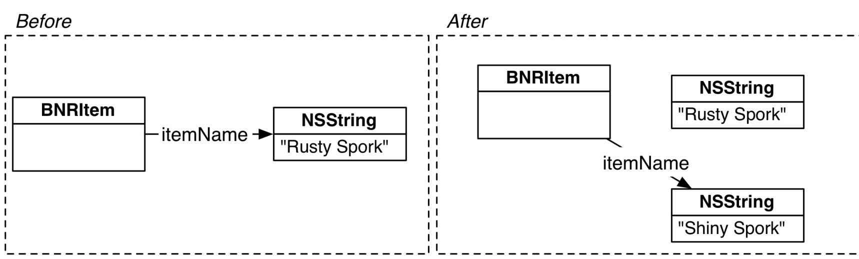
We know that an object is safe to destroy – and should be destroyed – when it no longer has any owners. So how does an object lose an owner?

- 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`.
- A variable that points to the object is itself destroyed.

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

Why might a pointer change the object it points to? Imagine a **BNRItem**. The **NSString** that its `itemName` instance variable points to reads "Rusty Spork." If we polished the rust off of that spork, it would become a shiny spork, and we'd want to change the `itemName` to point at a different **NSString** (Figure 3.5).

Figure 3.5 Changing a pointer



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.

Why would you set a pointer to `nil`? Remember that setting a pointer to `nil` represents the absence of an object. For example, say you have a **BNRItem** that represents a television. Then, someone scratches off the television's serial number. You would then set its `serialNumber` instance variable to `nil`. The **NSString** that `serialNumber` used to point to loses an owner.

When a pointer variable itself is destroyed, the object that the variable was pointing at loses an owner. At what point a pointer variable will get destroyed depends on whether it is a local variable or an instance variable.

Recall that instance variables live in the heap as part of an object. When an object gets destroyed, its instance variables are also destroyed, and any object that was pointed to by one of those instance variables loses an owner.

Local variables live in the method's frame. When a method finishes executing and its frame is popped off the stack, any object that was pointed to by one of these local variables loses an owner.

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

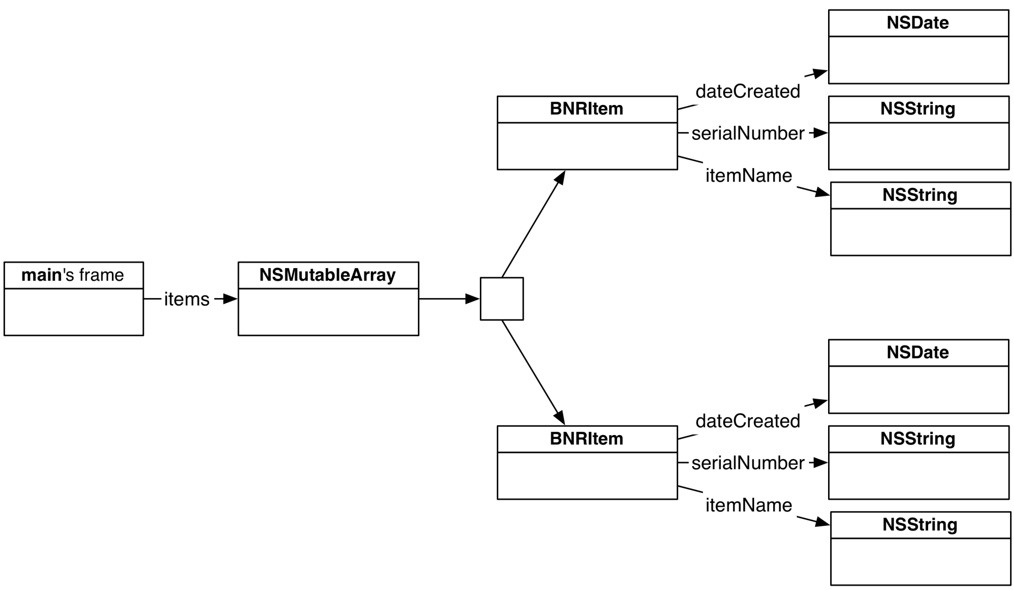
```
[items removeObject:p];    // object pointed to by p loses an owner
```

Keep in mind that losing an owner doesn't necessarily mean that the object gets destroyed; if there is still another pointer to the object somewhere, then that object will continue to exist. However, when an object loses its last owner, it means certain and appropriate death.

Because objects own other 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.

We have an example of this in RandomPossessions. Take another look at the object diagram of this application.

Figure 3.6 Objects and pointers in RandomPossessions



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

But it doesn't stop there. When the **NSMutableArray** is destroyed, all of its pointers to **BNRItem**s are destroyed. Once these variables are gone, no one owns any of the **BNRItem**s, so they are all destroyed.

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

Let's add some code so that we can see this destruction as it happens. **NSObject** implements a **dealloc** method, which is sent to an object when it is about to be destroyed. We can override this method in **BNRItem** to print something to the console when a **BNRItem** is destroyed. In `RandomPossessions.xcodeproj`, 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 **BNRItem**s 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 simply by setting `items` to `nil`. That's the power of ARC.

Strong and Weak References

So far, we've said that anytime a pointer variable stores the address of an object, that object has an owner and will stay alive. This is known as a *strong reference*. However, a variable can optionally *not* take ownership of an object 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 an unusual situation called a *retain cycle*. A retain cycle occurs when two or more objects have strong references to each other. This is bad news. When two objects own each other, they will 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 by virtue of those two strong references.

Thus, a retain 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 retain cycle in RandomPossessions to see how this works. First, we'll give **BNRItem** instances the ability to hold another **BNRItem** (so we can represent things like backpacks and purses). In addition, a **BNRItem** will know which **BNRItem** holds it. In `BNRItem.h`, add two instance variables and accessors

```
@interface BNRItem : NSObject
{
    NSString *itemName;
    NSString *serialNumber;
    int valueInDollars;
    NSDate *dateCreated;
```

```
    BNRItem *containedItem;
    BNRItem *container;
}

+ (id)randomItem;

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

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

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

Implement the accessors in BNRItem.m.

```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    [i setContainer:self];
}

- (BNRItem *)containedItem
{
    return containedItem;
}

- (void)setContainer:(BNRItem *)i
{
    container = i;
}

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

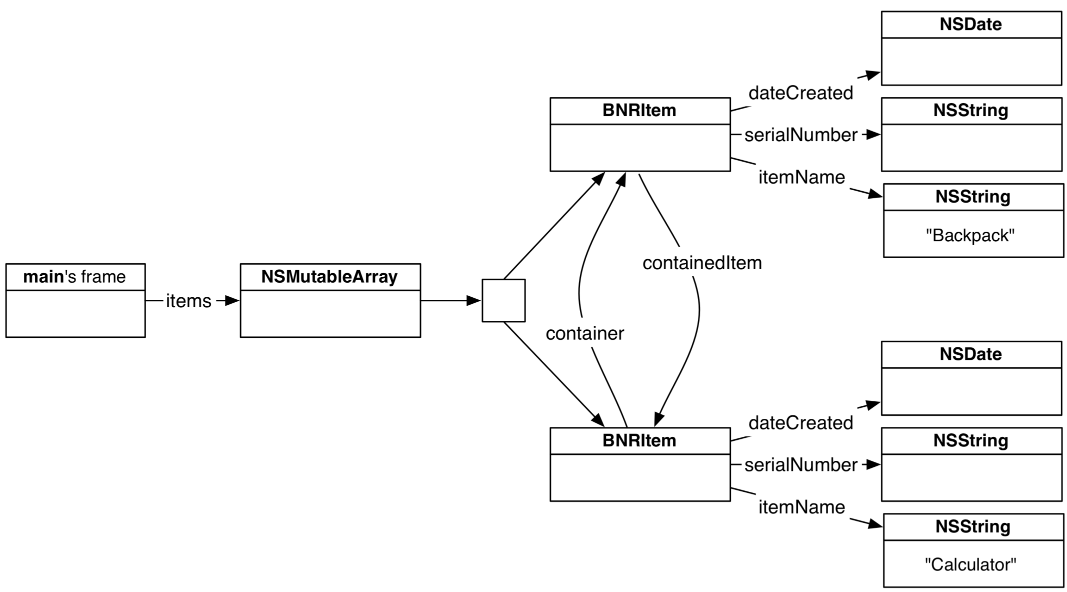
In main.m, remove the code that populated the array with random items. Then create two new items, add them to the array, and make them point at each other.

```
#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 *p = [BNRItem randomItem];
            [items addObject:p];
        }

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

```
        BNRItem *backpack = [[BNRItem alloc] init];
        [backpack setItemName:@"Backpack"];
        [items addObject:backpack];

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

        [backpack setContainedItem:calculator];

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

Here's what the application looks like now:

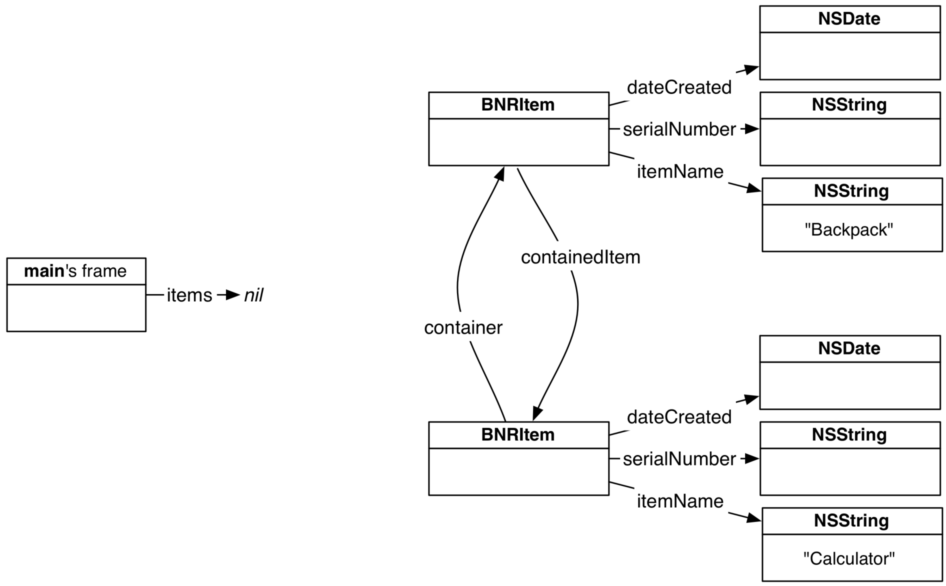
Figure 3.7 RandomPossessions with retain cycle



Per our understanding of memory management so far, both **`BNRItem`**s should be destroyed along with their instance variables when `items` is set to `nil`. Build and run the application. Notice that the console does not report that these objects have been destroyed.

This is a retain cycle: the backpack and the calculator have strong references to one another, so there is no way to destroy these objects. Figure 3.8 shows the objects in the application that are still taking up memory once `items` has been set to `nil`.

Figure 3.8 A retain cycle!



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

To fix this problem, one of the pointers between the **BNRItem**s 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 retain cycle, the backpack is the parent, and the calculator is the child. Thus, the backpack can keep its strong reference to the calculator (`containedItem`), but the calculator's reference to the backpack (`container`) should be weak.

To declare a variable as a weak reference, we 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.

Every retain cycle 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 retain cycle.

A child holding a strong reference to its *parent's* parent also causes a retain 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.

It's good to understand and look out for retain cycles, but keep in mind that they are quite rare. Also, Xcode has a Leaks tool to help you find them. We'll see how to use this tool in Chapter 21.

An interesting property of weak references is that they know when the object they reference is destroyed. Thus, if the backpack is destroyed, the calculator automatically sets its `container` instance variable to `nil`. In `main.m`, make the following changes to see this happen.

```
NSMutableArray *items = [[NSMutableArray alloc] init];

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

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

[backpack setContainedItem:calculator];

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

backpack = nil;

NSLog(@"Container: %@", [calculator container]);

calculator = nil;
```

Build and run the application. Notice that after the backpack is destroyed, the calculator reports that it has no container without any additional work on our part.

A variable can also be declared using the `__unsafe_unretained` attribute. Like a weak reference, an unsafe unretained reference does not take ownership of the object it points to. Unlike a weak reference, an unsafe unretained reference is not automatically set to `nil` when the object it points to is destroyed. This makes unsafe unretained variables, well, unsafe. To see an example, change `container` to be unsafe unretained in `BNRItem.h`.

```
__unsafe_unretained BNRItem *container;
```

Build and run the application. It will most likely crash. The reason? When the calculator was asked for its container within the **NSLog** function call, it obligingly returned its value – the address in memory where the non-existent backpack used to live. Sending a message to a non-existent object resulted in a crash. Oops.

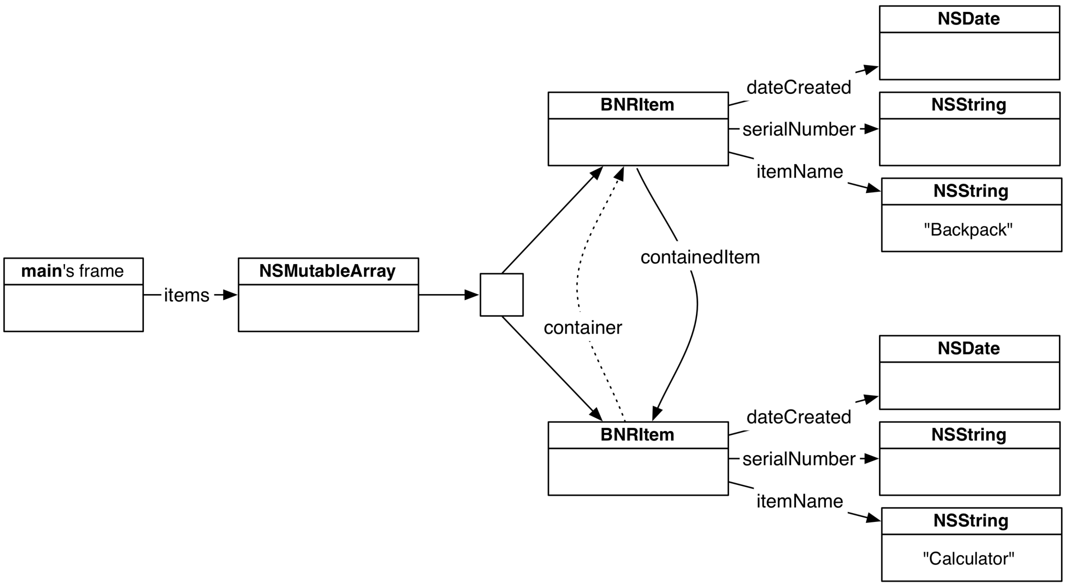
As a novice iOS programmer, you won't use `__unsafe_unretained`. As an experienced programmer, you probably won't use it, either. It exists primarily for backwards compatibility: applications prior to iOS 5 could not use weak references, so to have similar behavior, they must use `__unsafe_unretained`.

Be safe. Change this variable back to `__weak`.

```
__weak BNRItem *container;
```

Here's the current diagram of RandomPossessions. Notice that the arrow representing the `container` pointer variable is now a dotted line. A dotted line denotes a weak (or unsafe unretained reference). Strong references are always solid lines.

Figure 3.9 RandomPossessions with retain cycle avoided



Properties

Each time we've added an instance variable to **`BNRItem`**, we've declared and implemented a pair of accessor methods. Now we're going to see how to use *properties* instead. Properties are a convenient alternative to writing out accessors for instance variables – one that saves a lot of typing and makes your class files much clearer to read.

Declaring properties

A property is declared in the interface of a class where methods are declared. A property declaration has the following form:

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

When you declare a property, you are implicitly declaring a setter and a getter for the instance variable of the same name. So the above line of code is equivalent to the following:

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

Each property has a set of attributes that describe the behavior of the accessor methods. The attributes are declared in parentheses after the `@property` directive. Here is an example:

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

There are three property attributes. Each attribute has two or three options, one of which is the default and does not have to explicitly declared.

The first attribute of a property has two options: nonatomic or atomic. This attribute has to do with multi-threaded applications and is outside the scope of this book. Most Objective-C programmers typically use nonatomic: we do at Big Nerd Ranch, and so does Apple. In this book, we'll use nonatomic for all properties.

Let's change **BNRItem** to use properties instead of accessor methods. In BNRItem.h, replace all of your accessor methods with properties that are nonatomic.

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

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

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

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

- (NSDate *)dateCreated;

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

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

@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
```

Unfortunately, nonatomic is not the default option, so you will always need to explicitly declare your properties to be nonatomic.

The second attribute of a property is either readwrite or readonly. A readwrite property declares both a setter and getter, and a readonly property just declares a getter. The default option for this attribute is readwrite. This is what we want for all of **BNRItem**'s properties with the exception of dateCreated, which should be readonly. In BNRItem.h, declare dateCreated as a readonly property so that no setter method is declared for this instance variable.

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

The final attribute of a property describes its memory management. The default option depends on the type of the property. A property whose type is not a pointer to an object, like int, does not need memory management and thus defaults to assign. **BNRItem** only has one property that is not a pointer to an object, valueInDollars. For pointers to objects, like **NSString ***, this attribute defaults to strong. **BNRItem** has five object pointer properties: four of these will use strong, and the container property will use weak to avoid a retain cycle. With pointers to objects, it is good to be explicit and use the strong property to avoid confusion. In BNRItem.h, update the property declarations as shown.

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

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

Build and run the application. You should see the exact same behavior as the last time you ran it. The only difference is that `BNRItem.h` is much cleaner.

Synthesizing properties

In addition to using a property to declare accessor methods, you can *synthesize* a property to generate the code for the accessor methods in the implementation file. Right now, `BNRItem.m` defines the accessor methods declared by each property. For example, the property `itemName` declares two accessor methods, **itemName** and **setItemName:**, and these are defined in `BNRItem.m` like so:

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

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

When you synthesize a property, you don't have to type out the accessor definitions. You can synthesize a property by using the `@synthesize` directive in the implementation file. In `BNRItem.m`, add a synthesize statement for `itemName` and delete the implementations of **setItemName:** and **itemName**.

```
@implementation BNRItem
@synthesize itemName;

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

You can synthesize properties in the same synthesize statement or split them up into multiple statements. In `BNRItem.m`, synthesize the rest of the instance variables and delete the rest of the accessor implementations.

```
@implementation
@synthesize itemName;
@synthesize containedItem, container, serialNumber, valueInDollars,
    dateCreated;

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

```
{
    return serialNumber;
}

- (void)setValueInDollars:(int)i
{
    valueInDollars = i;
}
- (int)valueInDollars
{
    return valueInDollars;
}

- (NSDate *)dateCreated
{
    return dateCreated;
}

- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    [i setContainer:self];
}
- (BNRItem *)containedItem
{
    return containedItem;
}

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

Usually, synthesized accessors work fine, but sometimes you need an accessor method to do some additional work. This is the case for **`setContainedItem:`**. Here is our original implementation:

```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
    [i setContainer:self];
}
```

The synthesized setter won't include the second line establishing the reciprocal relationship between the `container` and the `containedItem`. Its implementation just looks like this:

```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
}
```

Because we need this setter to do additional work, we cannot rely on the synthesized method and must write the implementation ourselves. Fortunately, writing our own implementation does not conflict with synthesizing the property. Any implementation we add will override the synthesized version. In BNRItem.m, add back the implementation of **setContainedItem:**.

```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
    [i setContainer:self];
}
```

Build and run the application again. It should work the same as always, but your code is much cleaner.

Synthesizing a property that you declared in the header file is optional, but typical. The only reason not to synthesize a property is if both the getter and the setter methods have additional behavior you need to implement.

Instance variables and properties

With properties, we can go even one step further in code clarity. By default, a synthesized property will access the instance variable of the same name. For example, the itemName property accesses the itemName instance variable: the **itemName** method returns the value of the itemName instance variable, and the **setItemName:** method changes the itemName instance variable.

If there is no instance variable that matches the name of a synthesized property, one is automatically created. So declaring an instance variable *and* synthesizing a property is redundant. In BNRItem.h, remove all of the instance variables as well as the curly brackets.

```
@interface BNRItem : NSObject
{
    NSString *itemName;
    NSString *serialNumber;
    int valueInDollars;
    NSDate *dateCreated;

    BNRItem *containedItem;
    __weak BNRItem *container;
}
```

Build and run the application. Notice there are no errors and everything works fine. All of the instance variables (like itemName and dateCreated) still exist even though we no longer explicitly declare them.

Copying

There is one more change we need to make to our properties – specifically, the two properties that point to instances of **NSString**.

In general, when you have a property that points to an instance of a class that has a mutable subclass (like **NSString** or **NSArray**), it is safer to make a copy of the object to point to rather than pointing to an existing object that could have other owners.

For instance, imagine if a **BNRItem** was initialized so that its itemName pointed to an instance of **NSMutableString**.

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

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

This code is valid because an **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 **BNRItem** that also points to it.

You may be wondering why this is a real problem. In your application, you're not going to change this string unless you mean to. However, when you write classes for others to use, you can't be sure how they will use your classes, and you have to program defensively.

In this case, the defense is to declare this property using the memory management attribute `copy` instead of `strong`. In `BNRItem.h`, change the `itemName` and `serialNumber` properties to `copy`.

```
@property (nonatomic, copy) NSString *itemName;
@property (nonatomic, copy) NSString *serialNumber;
```

Now the generated setter method for the synthesized `itemName` property looks like this:

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

Instead of setting `itemName` to point to the **NSString** object pointed to by `str`, this setter sends the message **copy** to that **NSString**. The **copy** method returns a new **NSString** object (not an **NSMutableString**) that has the same values as the original string, and `itemName` is set to point at the new string. 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 doesn't gain or lose an owner, and none of its data changes.

Dot Syntax

We should mention an alternative syntax for sending accessor messages to an object called dot syntax:

```
// Following two lines are exactly equivalent
int value = [item valueInDollars];
int value = item.valueInDollars;

// Following two lines are exactly equivalent
[item setValueInDollars:5];
item.valueInDollars = 5;
```

We have reservations about Objective-C newcomers using dot syntax. We think it hides the fact that you are actually sending a message and can be confusing. Once you are comfortable with Objective-C, it is totally okay to use dot syntax. But while you are learning, it's better to use square brackets to make sure you understand what is really going on.

This book will always use square brackets for sending accessor messages.

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 was a guaranteed 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'll see more about the static analyzer in Chapter 21, 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 memory management problems became a thing of the past.

(Some people have an irrational fear of letting the compiler do their work for them and say they prefer manual memory management for this reason. If someone says something like that to you, open up one of their `.m` files, go to the Product menu, and select Generate Assembly File from the Generate Output menu item. Tell them if they don't trust the compiler, then they should be writing the assembly code they see in front of them.)

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 after using it within the method in which it was returned.

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

```
@autoreleasepool {
    // Get a BNRItem back from a method that created it, method doesn't say alloc/copy
    BNRItem *item = [BNRItem someItem];
} // item loses an owner and is destroyed because nothing else took ownership of it
```

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

Index

Symbols

A

D

E

F

J

K

L

M

O

P

Q

R

S

T

U

V

W

X

Z

ABOUT US

THE BIG NERD STORY

Big Nerd Ranch exists to broaden the minds of our students and the businesses of our clients. Whether we are training talented individuals or developing a company's mobile strategy, our core philosophy is integral to everything we do.

The brainchild of CEO Aaron Hillegass, Big Nerd Ranch has hosted more than 2,000 students at the Ranch since its inception in 2001. Over the past ten years, we have had the opportunity to work with some of the biggest companies in the world such as Apple, Samsung, Nokia, Google, AOL, Los Alamos National Laboratory and Adobe, helping them realize their programming goals. Our team of software engineers are among the brightest in the business and it shows in our work. We have developed dozens of innovative and flexible solutions for our clients.

The Story Behind the Hat

Back in 2001, Big Nerd Ranch founder, Aaron Hillegass, showed up at WWDC (World Wide Developers Conference) to promote the Big Nerd Ranch brand. Without the money to buy an expensive booth, Aaron donned a ten-gallon cowboy hat to draw attention while passing out Big Nerd literature to prospective students and clients. A week later, we landed our first big client and the cowboy hat has been synonymous with the Big Nerd brand ever since. Already easily recognizable at 6'5, Aaron can be spotted wearing his cowboy hat at speaking engagements and conferences all over the world.

The New Ranch – Opening 2012

In the continuing effort to perfect the student experience, Big Nerd Ranch is building its own facility. Located just 20 minutes from the Atlanta airport, the new Ranch will be a monastic learning center that encompasses Aaron Hillegass' vision for technical education featuring a state-of-the-art classroom, fine dining and exercise facilities.

www.bignerdranch.com