CHAPTER 16
Instruments and the Debugger

As anyone who's written software knows, designing and implementing the features of an application is only a fraction of the work. Once the app's done and performs all the tasks it's meant to do, you need to make sure that it runs *well*.

Performance is a feature that many developers neglect, but it's something that influences people's decisions to buy your software or not. Many of our friends and family prefer to use Pages instead of Microsoft Word, because despite Page's relative lack of features, it's a more nimble and zippy application.

Not paying attention to performance also has more tangible implications for your code. For example, an application that is careless with memory will end up being force-quit by the system on iOS, and an app that consumes lots of CPU time will exhaust your user's battery and make the system run hot. There are other resources as well that your application needs to be careful with, including network bandwidth and disk space.

To help monitor your application's performance and use of resources, the developer tools include *Instruments*, an application that's able to inspect and report on just about every aspect of an application.

Instruments works by inserting diagnostic hooks into a running application. These hooks are able to do things like analyze the memory usage of an application, monitor how much time the app spends in various methods, and examine other data. This information is then collected, analyzed, and presented to you, allowing you to figure out what's going on inside your application.

In this chapter, you'll learn how to get around in Instruments, how to analyze an application, and how to spot and fix memory issues and performance problems using the information that Instruments gives you. You'll also learn how to use Xcode's debugger to track down problems and fix them.

Getting Started with Instruments

To get started with Instruments, we'll load a sample application and examine how it works.

The application that we'll be examining is *TextEdit*, which is the built-in text editor that comes with OS X. TextEdit is a great sample app to modify because it's a rather complex little app—it's effectively an entire word processor, with support for images, Microsoft Word import and export, and a lot more. You've probably used it before; if you haven't, you can find it by either searching Spotlight for "TextEdit" or by looking in the Applications folder.

The source code to TextEdit is available from the Apple Mac Developer site and you can find it by going to *http://bit.ly/1gMPNC9*.

The Developer site also contains a great deal of other example code and resources in the Mac Developer Library available at *http://bit.ly/1fHRMVr*.

1. *Open the TextEdit project.* Double-click the *TextEdit.xcodeproj* file to open it in Xcode.
2. *Run the application.* Click the Run button or press ⌘-R.

   The app will launch. Play around with it by writing some text, and saving and opening some documents.

We'll now use Instruments to examine what TextEdit is doing in memory as it runs.

3. *Quit TextEdit.* You can do this by pressing ⌘-Q or choosing Quit from the TextEdit menu.
4. *Tell Xcode to profile the application.* To do this, choose Profile from the Product menu. You can also press ⌘-I.

   Xcode will rebuild the application, and then launch Instruments. When Instruments launches, it presents a window that lets you choose which aspects of the app you'd like to inspect (Figure 16-1).
5. *Select the Allocations instrument, and click Choose.* Instruments will launch TextEdit, and start recording memory usage information from the application.

   At this point, you're in Instruments proper, so it's worthwhile to stop and take a look around (Figure 16-2).

The Instruments Interface

When you work with Instruments, you're working with one or more individual modules that are responsible for analyzing different parts. Each module is also called

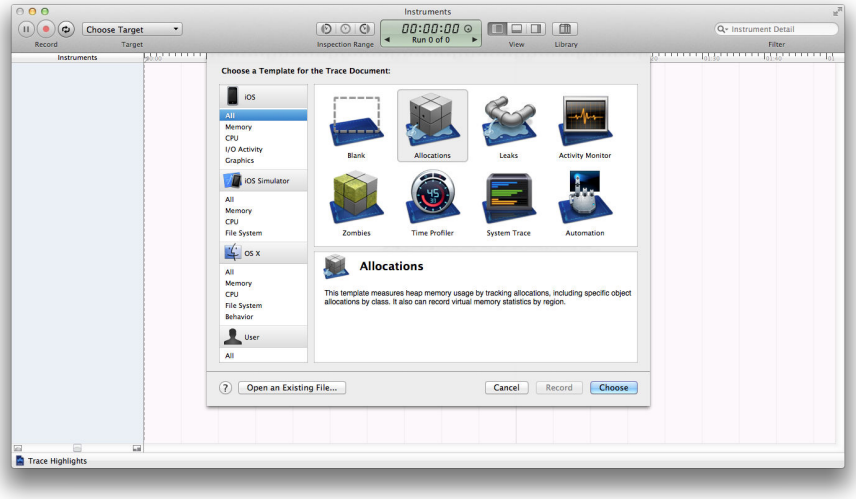

Figure 16-1. The Instruments template chooser

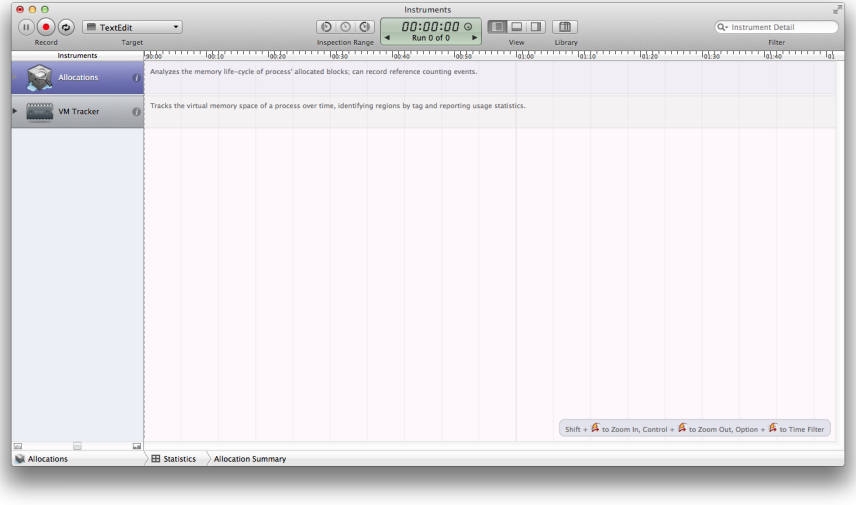

Figure 16-2. The Instruments window

an instrument—so, for example, there's the Allocations instrument (for measuring memory performance), the Time Profiler instrument (for measuring runtime performance), the VM Tracker instrument (for measuring memory consumption), and so on.

Each instrument is listed in the Instruments pane on the lefthand side of the window. As an application runs, information from each instrument is shown in the Track pane. If you select an instrument in the Instruments pane, more detailed information is shown in the Detail pane at the bottom of the window. The Track pane is useful for giving you a high-level overview of the information that is being reported, but the majority of useful information is kept in the Detail pane.

The Detail pane shows different kinds of information, depending on the instrument. To choose which information to present, you can use the navigation bar, which separates the window horizontally.

To configure how an instrument collects its data, you can click the *i* button at the right of each row of the Instruments pane. From there, you can set the various options that affect what information the instrument collects.

By default, the Instruments pane and the Detail pane are visible. You can also bring up a third pane, known as the Extended Detail pane. This pane, as its name suggests, displays extended detail information on whatever is selected in the Detail pane. For example, if you're using the Allocations instrument, the Detail pane could be showing the objects currently active in your application. You could then select a specific object, and the Extended Detail pane would show exactly where that object was created in your code.

You can also control what Instruments is doing through the Record buttons. The large red button in the center is the main one that we care about—clicking on it will launch the application that's currently under investigation, and start recording data. Clicking it again will quit the application and stop recording, though the data that was collected remains. If you click Record again, a new set of data will be recorded—if you want to see past runs, you can navigate among them by clicking on the arrows in the display in the middle of the toolbar.

To open and close the various panes, click on the view buttons in the toolbar.

Observing Data

We'll now do some work inside TextEdit and watch how the data is collected.

1. *Start recording, if the app isn't open already.* If TextEdit isn't running, hit the Record button to launch it again.

When the application starts up, it immediately allocates some memory as it gets going. When it needs to store more information, it allocates more. We'll now cause the app to start allocating more memory by adding text to the document.

2. *Enter some text in the document.* Go to the TextEdit window and start typing. Because text isn't very large, we won't see much of a difference in what's being displayed unless we enter quite a lot of text.

   So, to quickly enter lots of text, type something, select it all, copy, and paste. Then select all again, and copy and paste again. Repeat until you've got a huge amount of text in the document.

3. *Observe the memory usage of TextEdit climbing.* Go back to Instruments, and you'll notice that the amount of memory used by the application has increased quite a lot (Figure 16-3).

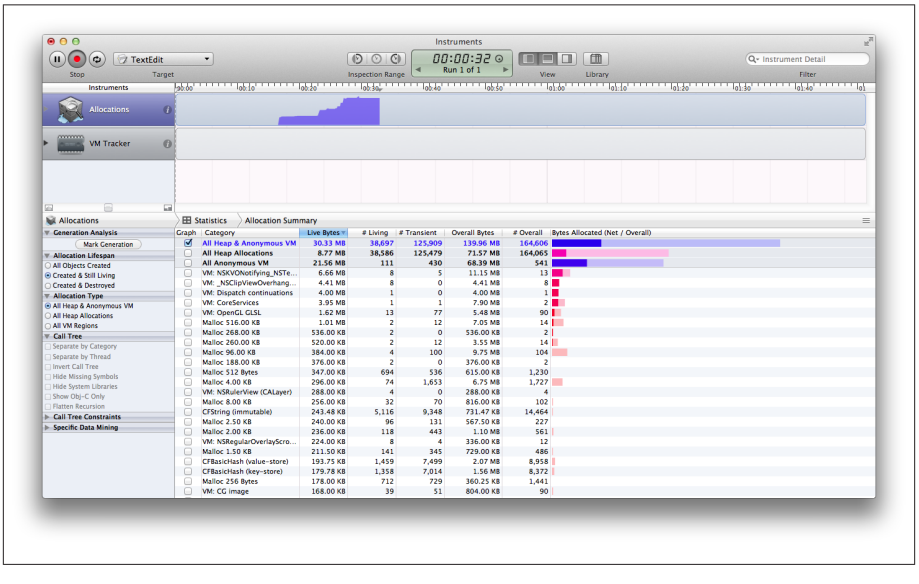

Figure 16-3. Instruments records an increase in memory usage as the application is used

Here, the consumption of memory is OK, because we deliberately stress-tested the application. However, if you see similar spikes in memory usage in your application from regular use, you probably have a problem to solve.

Adding Instruments from the Library

While Instruments provides a selection of templates that you can use to get started (such as the Allocations template we used earlier), you can add more instruments to your trace to help hunt down issues.

To add an instrument to your trace document, select the instrument you want to use from the Library. To open the Library, click the Library button, choose Library from the Window menu, or press ⌘-L (Figure 16-4).

The Library lists all of the available instruments that you can use, as well as information on what each one does. To add an instrument to your trace, drag and drop an instrument into the Instruments pane, or double-click the instrument.

Not all instruments work on all platforms. For example, the OpenGL ES analyzer instrument only works on iOS.

Combining different kinds of instruments allows you to zoom in on specific problems. For example, if your application is being slow and you think it's because it's loading and processing lots of information at once, you can use a Reads/Writes instrument alongside a Time Profiler. If the slowdowns occur while both of these instruments indicate heavy activity, then your slowdowns are being caused by your application working the disk too hard while using lots of CPU time.

Fixing Problems with Instruments

To demonstrate how to detect and solve problems with Instruments, we'll create an application that has a large memory problem, and then use Instruments to find and fix it.

This iOS application will create and display a large gallery of images and let the user smoothly scroll between them. We'll develop and run it on the iOS Simulator, and then see how well it does on a real device.

The application will consist of a single scroll view, which will have a number of image views added to it. The user will be able to scroll around inside the view to see the different images.

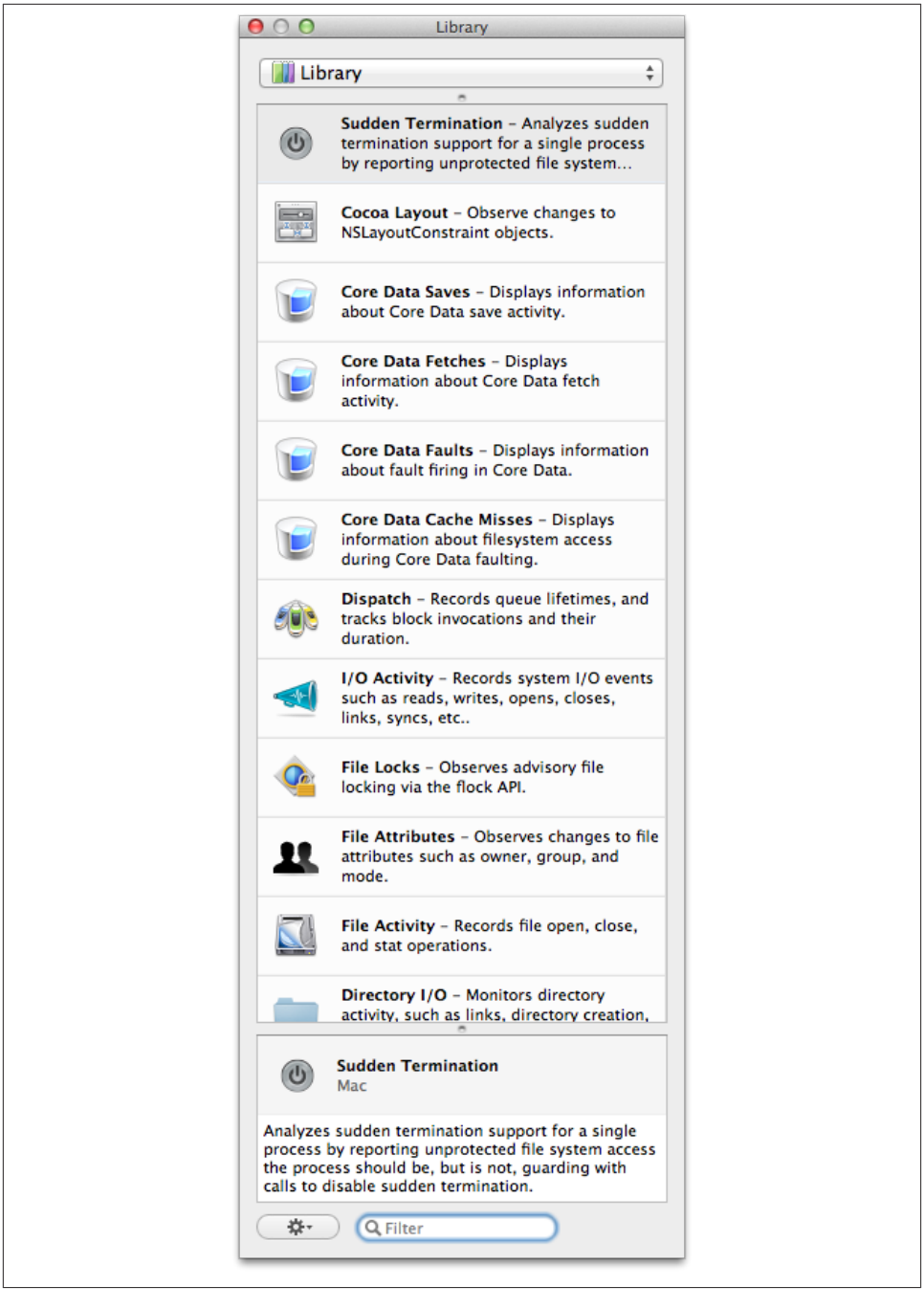

Figure 16-4. The Instruments Library

1. *Create a new single-view application for iOS.* Call this application **`MemoryDemo`**. Make it use storyboards and ARC, and make it run on the iPhone.
2. *Open the main storyboard.* Open *MainStoryboard.storyboard* in the project navigator.
3. *Add a scroll view to the window.* Make it fill the entire screen. While you have it selected, turn Paging Enabled on. This means that the scroll view will behave much like the home screen on the iPhone, where all scrolling snaps to the width of the scroll view.
4. *Connect the scroll view to the view controller class.* Open the assistant, and control-drag from the scroll view into `ViewController`'s interface. Create a new outlet called **`imagesContainer`**.
5. *Add the code that sets up the application.* Add the following code to *View Controller.m*, and remove the previous implementation of `viewDidLoad`:

```
- (UIImage*) imageWithNumber:(NSInteger)number {

    CGRect imageRect = self.imagesContainer.frame;

    UIGraphicsBeginImageContext(imageRect.size);

    // Inset the image by 30px so that we can see the rounded corners
    imageRect = CGRectInset(imageRect, 30, 30);

    // Draw a rounded rectangle
    UIBezierPath* path = [UIBezierPath bezierPathWithRoundedRect:imageRect
        cornerRadius:10];

    [path setLineWidth:20];
    [[UIColor blackColor] setStroke];
    [[UIColor scrollViewTexturedBackgroundColor] setFill];

    [path fill];
    [path stroke];

    // Draw the number
    NSString* label = [NSString stringWithFormat:@"%i", number];

    UIFont* font = [UIFont systemFontOfSize:50];
    CGPoint labelPoint = CGPointMake(50, 50);

    [[UIColor whiteColor] setFill];
    [label drawAtPoint:labelPoint withFont:font];

    // Get the finished image and return it
    UIImage* returnedImage = UIGraphicsGetImageFromCurrentImageContext();

    UIGraphicsEndImageContext();
```

```
    return returnedImage;
}

- (void) loadPageWithNumber:(NSInteger) number {

    // If an image view already exists for this page, don't do anything
    if ([self.imagesContainer viewWithTag:number])
        return;

    // Get the image for this page
    UIImage* image = [self imageWithNumber:number];

    // Create and prepare the image view for this page
    UIImageView* imageView = [[UIImageView alloc] initWithImage:image];
    CGRect imageViewFrame = [self.imagesContainer bounds];
    imageViewFrame.origin.x = imageViewFrame.size.width * (number - 1);
    imageView.frame = imageViewFrame;

    // Add it to the scroll view
    [self.imagesContainer addSubview:imageView];

    // Mark this new image view with a tag so that we can
    // easily refer to it later
    imageView.tag = number;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSInteger pageCount = 1000;

    for (int i = 1; i < pageCount; i++) {
        [self loadPageWithNumber:i];
    }

    CGSize contentSize;
    contentSize.height = self.imagesContainer.bounds.size.height;
    contentSize.width = self.imagesContainer.bounds.size.width * pageCount;


    self.imagesContainer.contentSize = contentSize;

}
```

6. *Run the application.* The application runs fine on the simulator, but if you try to run it on the device, it will appear to hang for a while and finally exit without showing the app.

To find out why this happens, we'll run this inside Instruments.

1. *Set the Scheme to launch on your iOS device.* We want Instruments to run on the device, not the simulator. (If you don't have an iOS device to test on, that's OK—you can still use the simulator, but the numbers won't be representative of how it would work on a real iPhone or iPad.)
2. *Launch the application inside Instruments.* Do this by choosing Profile from the Product menu, or pressing ⌘-I.
3. *Select the Allocations template.* We want to keep an eye on how memory is being used. Select the Allocations template, and click Choose.
4. *When the Instruments window appears, stop the recording.* We want to make some adjustments to how the data is being collected.

The Allocations template includes two instruments: the Allocations instrument, which keeps track of memory allocations made by your application, and the VM Tracker instrument, which monitors the total usage of memory. Some memory allocations, such as images, don't show up in the Allocations instrument because this memory is handled on the graphics card or in other regions of memory.

Therefore, to keep a close eye on the total amount of memory used by the app, we'll use the VM Tracker instrument. The VM Tracker instrument scans all of the memory used by the app, and reports on how much is being used and where. Normally, the VM Tracker instrument will only scan memory when you explicitly tell it to, because performing a memory scan causes the app to hang for a moment while the scan takes place. Because we want to stay apprised of memory usage as the app launches, we'll instruct the VM Tracker to automatically scan the memory.

5. *Select the VM Tracker in the Instruments pane.* The Detail pane will update to show information gathered by that instrument.
6. *Turn on Automatic Snapshotting and set the Snapshot Interval to 1 second.* This will cause the instrument to scan the memory usage of the app every second.
7. *Start recording.* The application will launch, and the VM Tracker will show how memory is consumed while it starts up.

As the application launches, you'll notice that the amount of memory used by the app steadily increase. After a while, the app will start receiving memory warnings (you'll see a bunch of black flags pop up in the timeline), and will then quit.

Clearly, the problem is that the application consumes too much memory. There's an additional problem—the number of images being drawn during startup is causing a huge slowdown. The application is creating and inserting a thousand image views onto the screen. Each image displayed by the image views needs to be kept in memory, which means that the app rapidly runs out of space and is forced to exit.

A better way to handle this is to only display the images that the user is able to see, rather than loading all of them at once. At minimum, there are only three images that need to be present—the one currently being shown, and the two on either side of it. Because of the size of the image views, it's possible for this app to be showing one or two images at the same time, but never three.

To fix the problem, therefore, we need to make the application update the image views while the user is scrolling. If an image view isn't visible by the user, the app should remove it from the screen, which frees up memory.

To do this, we'll add a method that makes sure that the image views for the previous, current, and next pages are present, and then removes all other image views. This method will be called every time the scroll view scrolls, meaning that as far as the user is concerned, every image is on the screen when she needs to see it.

First, we'll set up the view controller to be notified when the scroll view scrolls, and then add the code that checks the image views. Finally, we'll update the `viewDidLoad` method to make it only display the first set of image views.

1. *Open the storyboard and make the scroll view use the view controller as its delegate.*

   Control-drag from the scroll view onto the view controller's icon. Choose "delegate" from the list that pops up.

2. *Open* ViewController.h. We now need to make the class conform to the `UIScrollViewDelegate` protocol. Replace the class's definition with the following line of code:

   ```
   @interface ViewController : UIViewController <UIScrollViewDelegate>
   ```

3. *Add the additional methods to* ViewController.m.

Finally, we'll update the code to update the collection of image views when the scroll view scrolls.

Add the following methods to *ViewController.m*, above the `viewDidLoad` method:

```
- (void) updatePages {
    int pageNumber = imagesContainer.contentOffset.x /
        imagesContainer.bounds.size.width + 1;

    // Load the image previous to this one
    [self loadPageWithNumber:pageNumber - 1];
    // Load the current page
```

```
    [self loadPageWithNumber:pageNumber];
    // Load the next page
    [self loadPageWithNumber:pageNumber+1];

    // Remove all image views that aren't on this page or the pages adjacent
    // to it
    for (UIImageView* imageView in imagesContainer.subviews) {
        if (imageView.tag < pageNumber - 1 || imageView.tag > pageNumber + 1)
            [imageView removeFromSuperview];
    }
}

- (void)scrollViewDidScroll:(UIScrollView *)scrollView{
    [self updatePages];
}
```

4. *Replace the* `viewDidLoad` *method.* Replace the method with the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSInteger pageCount = 1000;

    [self updatePages];

    CGSize contentSize;
    contentSize.height = self.imagesContainer.bounds.size.height;
    contentSize.width = self.imagesContainer.bounds.size.width * pageCount;

    self.imagesContainer.contentSize = contentSize;
}
```

Once you've made these changes, try running the app on the device again. You'll find that its behavior is identical to how it used to run, with the added bonus that the application doesn't run out of memory and crash on launch.

Retain Cycles and Leaks

The Automatic Reference Counting feature built into the compiler is great at reducing problems caused by memory leaks, but it's not foolproof. In this section, we'll look at using Instruments to detect a memory leak.

Automatic Reference Counting releases an object from memory when the last strong reference to that object goes away. However, if two objects have strong references *to each other*, the number of strong references to each object will never be zero, and the objects will stay in memory—even if nothing else in the application has a reference to those objects. This is called a *retain cycle*, so named because if you were to draw a graph showing how each object refers to each other, you'd end up drawing a circle.

You can use Instruments to detect leaked memory, again by using the Allocations instrument. To demonstrate this, we'll build a sample application that has a built-in retain cycle.

This will be a master-detail application that has a custom `UIView` on the detail view controller. This view uses the view controller as a delegate, whereby it asks which color to use as the background.

1. *Create a new master-detail application for iOS.* Name the application **`RetainCycle`**.
2. *Create a new file.* Make a new Objective-C subclass, and make it a subclass of `UIView.` Name it **`ExampleView`**.
3. *Set up the* `ExampleView` *class.* Make *ExampleView.h* look like the following code. In this code, we're creating a new delegate protocol for other classes to conform to, and adding a property on the `ExampleView` class so that it has a delegate.

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

@class ExampleView;

@protocol ExampleViewDelegate <NSObject>
- (UIColor*)colorForView:(ExampleView*)view;
@end

@interface ExampleView : UIView
@property (strong) IBOutlet id <ExampleViewDelegate> delegate;
@end
```

4. *Make the* `ExampleView` *class use its delegate when it's loaded.* Add the following method to *ExampleView.m*:

```
- (void)awakeFromNib {
    self.backgroundColor = [self.delegate colorForView:self];
}
```

5. *Add an* `ExampleView` *to the detail view controller.* Open *Main.storyboard* and remove the label.

   Drag in a new `UIView` and open its Identity inspector (it's the third button from the left at the top of the Inspector pane).

   Set its class to `ExampleView`.

6. *Make the* `ExampleView` *use the view controller as its delegate.*

   Control-drag from the `ExampleView` to the view controller's icon, and select "delegate" from the menu that pops up.

   The view will now use the view controller as its delegate, which means we need to make the view controller function as the delegate.

7. *Update DetailViewController.h*. Make the file look like the following code. We've modified it to make it conform to the `ExampleViewDelegate` protocol, which declares that it has a method called `colorForView:` for the `ExampleView` to call.

```
#import <UIKit/UIKit.h>
#import "ExampleView.h"

@interface DetailViewController : UIViewController <ExampleViewDelegate>

@property (strong, nonatomic) id detailItem;
@property (strong, nonatomic) IBOutlet UILabel *detailDescriptionLabel;

@end
```

8. *Update DetailViewController.m*. All we're doing here is implementing the `colorForView:` method. Add it to the file:

```
-(UIColor *)colorForView:(ExampleView *)view {
    return [UIColor greenColor];
}
```

Once this is done, we're ready to test it.

9. *Launch the application inside Instruments.* Do this by choosing Profile from the Product menu or pressing ⌘-I.
10. *Select the Allocations template.* We want to keep an eye on how memory is being used. Select the Allocations template and click Choose.

    This application contains a retain cycle. We'll now use Instruments to figure out what's being leaked and where.
11. *Add and Select a Detail row.* Press the + button in the navigation bar to add a new Detail row. A new row will be added to the tableview. Select the row and a `DetailViewController` will be created and added to the navigation stack.
12. *Search for* `DetailViewController` *in Instruments.* In Instruments, select the Allocations instrument and type **`DetailViewController`** in the search field at the top-right corner of the screen.

    The list of objects shown in the Detail pane will show only `DetailViewController` objects. Currently, there's just one.
13. *Tap the back button in the app.* The `DetailViewController` is removed from the screen, and should therefore be removed from memory because nothing else is referring to it.

However, it doesn't go away—it still shows up in the list of objects.

Go back and forth between the detail view controller and the master view controller a few times. Every time you open the detail view controller, a new one is created, and when you close it, it's not being removed from memory.

The `DetailViewController` class is being leaked. The problem lies in this line in *ExampleView.h*:

```
@property (strong) IBOutlet id <ExampleViewDelegate> delegate;
```

The `ExampleView` has a strong reference to the view controller (its delegate) due to the use of the `strong` keyword. The view controller also has a strong reference to the view, because the view controller's view has the `ExampleView` as a subview.

To solve this problem, change the line listed above to this:

```
@property (weak) IBOutlet id <ExampleViewDelegate> delegate;
```

A weak reference means that the `ExampleView` is still able to refer to its delegate, but the view does not own the delegate and doesn't need it to stay in memory. If the delegate does get removed from memory, the reference is set to `nil` to avoid dangling pointers.

Using the Debugger

Xcode includes a source debugger called LLDB. Like all debuggers, LLDB allows you to observe your code as it runs, set breakpoints and watchpoints, and inspect the contents of memory.

The debugger is deeply integrated into Xcode, and Xcode lets you create very specific actions to run when your code does certain things. You can, for example, ask Xcode to speak some text the third time that a specific line of code gets run.

Setting Breakpoints

There are a few ways to set a breakpoint in Xcode. The most common method is to set a breakpoint on a line of code—when execution reaches that point, the debugger stops the program.

To set a breakpoint at a line, click the gray gutter at the left of the code editor. A blue breakpoint arrow will appear (Figure 16-5).

It's easier to add breakpoints, and navigate your code in general, if you turn on line numbers in Xcode. To do this, open Preferences by pressing ⌘-, and open the Text Editing tab. Turn on the Line Numbers checkbox.

```
103 - (id)init {
104     if ((self = [super init])) {
105         [[self undoManager] disableUndoRegistration];
106
107     textStorage = [[NSTextStorage allocWithZone:[self zone]] init];
108
109     [self setBackgroundColor:[NSColor whiteColor]];
110     [self setEncoding:NoStringEncoding];
111     [self setEncodingForSaving:NoStringEncoding];
112     [self setScaleFactor:1.0];
113     [self setDocumentPropertiesToDefaults];
114     inDuplicate = NO;
```

Figure 16-5. A breakpoint

After a breakpoint has been added to your code, you can drag the arrow to move the breakpoint. To remove the breakpoint, drag it out of the gutter.

When the program hits a breakpoint, Xcode shows the backtrace of all threads in the debug navigator. From there, you can see how a breakpoint was hit, and what functions were called that led to the program hitting that breakpoint.

Controlling program flow

When the program execution hits a breakpoint, you can choose to simply resume execution, or step through the code line by line.

To control program flow, you use the buttons in the debugger bar. The debugger bar is at the top of the debug area, which you can open and close by clicking the middle segment of the View control, at the top-right of the toolbar (Figure 16-6).

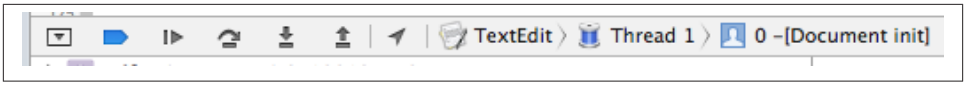

Figure 16-6. The debugger bar

From left to right, the buttons in the debugger bar perform the following actions:

- Close the debug area
- Pause or resume execution
- Step Over (continue to the next line of code)
- Step Into (if the current line of code is a method call, continue into it)
- Step Out (continue until execution leaves the current method)

When you add a breakpoint, it appears in the breakpoints navigator. From there, you can see all of the currently set breakpoints—you can also jump directly to a breakpoint, disable it, or delete it.

Custom breakpoints

Normally, a breakpoint just pauses execution when hit. However, you can customize your breakpoints to perform specific actions.

To customize a breakpoint, right-click the arrow or the breakpoint's entry in the breakpoints navigator, and choose Edit. The Edit Breakpoint window will appear (Figure 16-7).

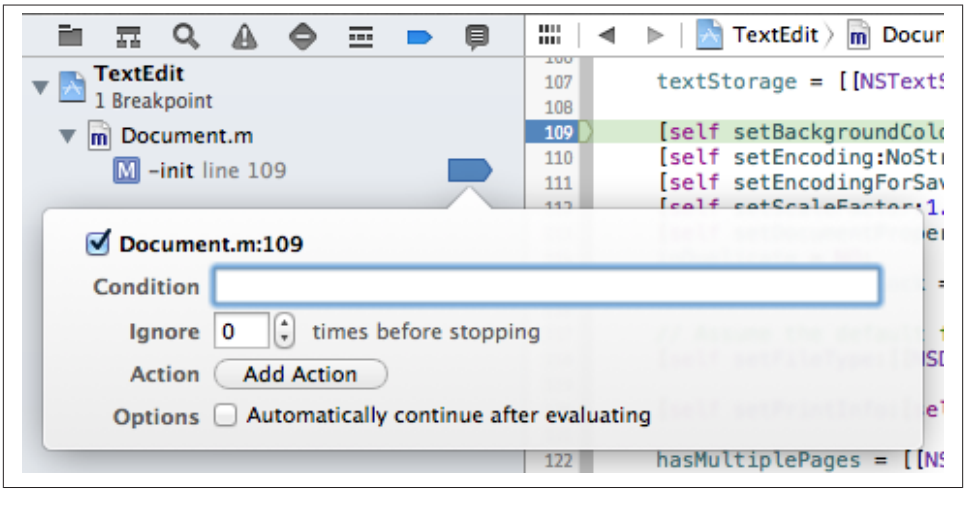

Figure 16-7. Editing a breakpoint

The Edit Breakpoint window allows you to customize when the breakpoint should trigger and what happens when it does. You can also indicate how many times the breakpoint should be ignored, what actions the breakpoint should run, and whether the breakpoint should pause the execution of the program.

Adding actions to a breakpoint allows you to run some AppleScript, speak a line of text, play a sound, or other actions. To add an action, click the "Click to add an action" button and choose the action that should be run.

This is a tremendously flexible feature, as it allows you to get additional information about how your program is running without the program stopping and starting.

Special breakpoints

The breakpoints navigator also allows you to add special breakpoints for exceptions, symbols, OpenGL ES Errors, and Test Failures.

An *exception breakpoint* stops the program when an Objective-C or C++ exception is thrown. For example, if you have an `NSArray` with two items and you try to access the third one, an exception is thrown and your program exits. Normally, Xcode stops the program at the point where the exception is caught, which isn't often the place where it is thrown. This makes it difficult to work out where the problem is. To solve this problem, you can add an exception breakpoint that stops the program at the instant the exception is thrown.

A *symbolic breakpoint* stops the program when a specific, named function or Objective-C method is entered. This is mostly useful when you want to stop execution at a function that you might not have the source code for, and then view the backtrace.

An *OpenGL ES Error breakpoint* stops the program whenever an OpenGL ES error is encountered. This is mostly useful when debugging graphics heavy iOS applications such as games.

A *Test Failure breakpoint* stops the program when a test assertion fails. This breakpoint is designed to be used in conjunction with unit tests, letting you see exactly when, where and hopefully why, your tests are failing.

To add one of the special breakpoints, click the + button at the bottom-left of the breakpoints navigator. Then choose which type of breakpoint you want to add, and Xcode asks you to configure the new breakpoint (Figure 16-8).

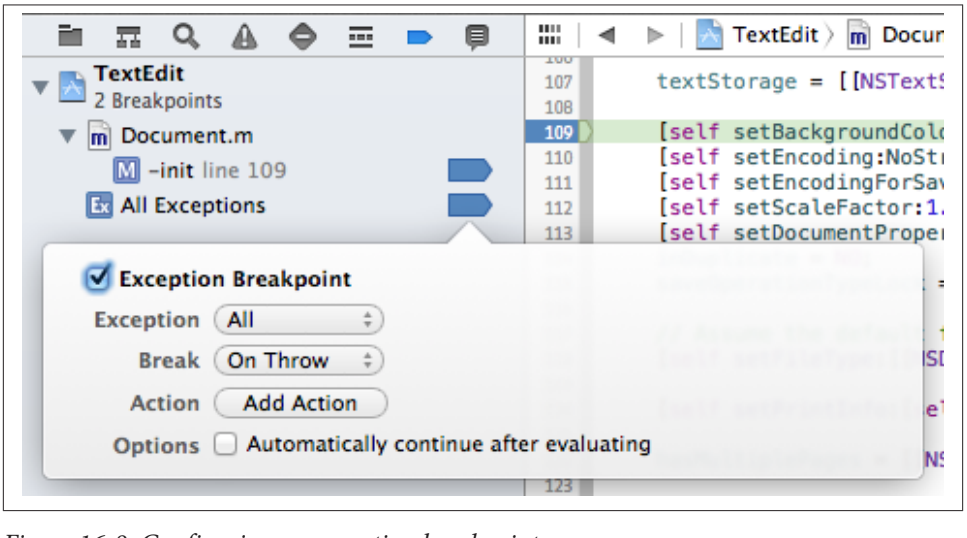

Figure 16-8. Configuring an exception breakpoint

If you're creating an exception breakpoint, you can choose whether you want to stop on Objective-C exceptions, C++ exceptions, or both. You can also choose whether the breakpoint should stop when the exception is thrown or caught.

Inspecting Memory Contents

When the program is stopped in the debugger, you can see the current state of objects and variables in memory.

The variables view is the lefthand section of the debug area. When the program is stopped, the variables view shows the variables that exist at that point.

The variables view shows the value of the variables. If the variables are simple types like integers or `BOOL`s, their values are shown; if the variables are things like `NSString`s or `NSArray`s, then summary information about them is shown, like the content of the string or the number of items in the array.

If you use the flow control buttons while the program is stopped, the variables view updates to show any changes. If a variable changes, it gets highlighted in blue.

The variables view also allows you to quickly send the `description` message to any object and see the results. To do this, right-click on a variable and choose Print Description.

Working with the Debugger Console

At the righthand side of the debug area is the console. The console is the command-line interface to the debugger, and allows you to directly access some of the debugger's powerful, lower-level features.

Working with LLDB via the console is a subject large enough to fill its own book, but in this section we'll talk about how to use the console for its arguably most powerful purpose: running custom Objective-C code to work with your program's variables.

Let's assume that the debugger has stopped at a breakpoint in a method. In this method, `myArray` is an `NSArray`, and you want to check its contents.

To see how many items are in `myArray`, you'd type this into the console:

```
print (int)[myArray count]
```

Note the lack of a semicolon. In the console, you don't put a semicolon at the end of lines.

When you call an Objective-C method in the debugger, you must specify what type of data the method will return. In the above example, `count` returns an `int`.

You can send arbitrary Objective-C messages to objects. For example, if you wanted to get the second object in the `myArray` array, you'd do this:

```
print-object [myArray objectAtIndex:1]
```

The `print-object` command takes an Objective-C object and sends it a `description` message. It then returns the string that comes back.