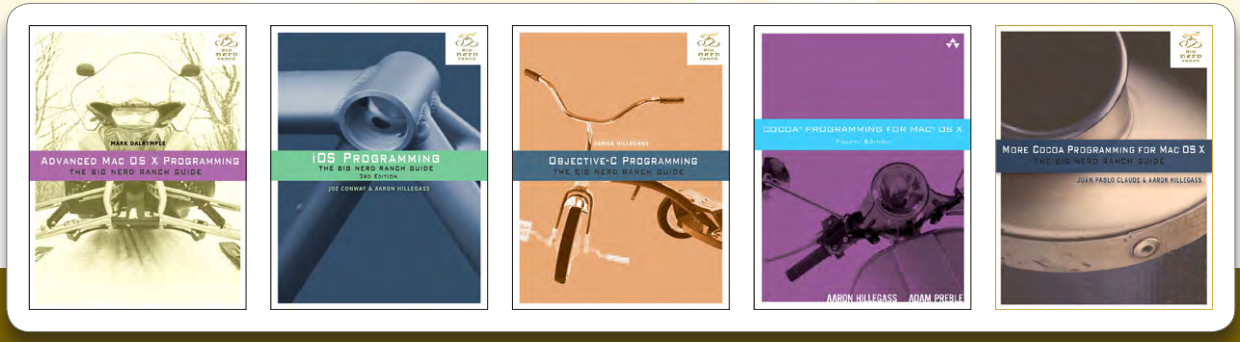
Free eSampler

THE 2012 BIG NERD RANCH eSampler



BUY FROM INFORMIT AND SAVE UP TO 40%

ENTER THE DISCOUNT CODE

LEARNMAC2012 DURING CHECKOUT

Big nerd ranch

AARON HILLEGASS
SERIES EDITOR

SHARE + SHOP @ informit.com

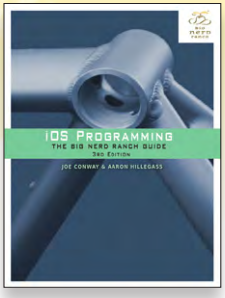
ALWAYS LEARNING

PEARSON

UPPER SADDLE RIVER, NJ | BOSTON | INDIANAPOLIS | SAN FRANCISCO | NEW YORK | TORONTO | MONTREAL
LONDON | MUNICH | PARIS | MADRID | CAPETOWN | SYDNEY | TOKYO | SINGAPORE | MEXICO CITY

Table of Contents

THE 2012 BIG NERD RANCH eSampler



Advanced Mac OS X Programming	iOS Programming	Objective-C Programming	Cocoa Programming for Mac OS X, 4th Ed.	More Cocoa Programming for Mac OS X
CHAPTER 10 Performance Tuning	CHAPTER 25 Web Services and UIWebView	CHAPTER 2: Your First Program	CHAPTER 27 Blocks	CHAPTER 23 Status Items

BUY FROM INFORMIT AND SAVE UP TO 40%

ENTER THE DISCOUNT CODE

LEARNMAC2012 DURING CHECKOUT

BUY NOW

SHARE + SHOP @ informit.com

CHAPTER 10 Performance Tuning

ADVANCED MAC OS X PROGRAMMING: THE BIG NERD RANCH GUIDE, by Mark Dalrymple

TABLE OF CONTENTS

AVAILABLE FORMATS

- 9780321706256 Book
- 9780132931052 eBook
- 9780321706560 Safari Books Online

Advanced Mac OS X Programming

THE BIG NERD RANCH GUIDE

MARK DALRYMPLE

BIG
nerd
ranch

10
Performance Tuning

It has happened to all of us: you subject your program to real world data and discover that performance is sub-optimal, ranging from “could be faster” to “locks up instantly and CPU fans reach Mach 3.” Finding out what the performance problem is can be a difficult task. Many times we think we *know* where the problem is, but we turn out to be wrong. Luckily, there are a number of tools available to give definite metrics of where the program spends its time and what kind of pressure it puts on the OS in general.

Approaches To Performance

The key to keeping on top of your performance is to use profiling tools. Profile early, and profile often. Catch performance issues early in your development so you don’t build a lot of code around an inefficient core. If you use Instruments, Shark, or other performance tools regularly, you can see possible performance issues on the horizon before they come close and bite you.

Be sure to profile with each new revision of the OS and on new hardware as it comes out. As Apple changes Mac OS X under the hood, things that were optimal may now be suboptimal and vice-versa. Hardware changes can change the game performance-wise. Consider look-up tables, which are a common way to avoid doing calculation. On PowerPC G4 and older processors, using a look-up table was often a big win, but the G5 could do a lot of calculation in the time it took to load data from memory. Having situations like this can be a real problem if you have to support older versions of the OS or if you want to optimally target vastly different hardware.

Be careful to not totally contort your design early on in a noble quest for Optimal Performance. You might be addressing performance issues that do not have a real impact on your final product. Reports from profilers are not gospel. A report may highlight a performance problem, but the problem may be something that doesn’t need fixing. If a problem highlighted by a profiler will not affect the user experience of your program or if it is something that rarely happens, you can put your energies into optimizing something else.

Finally, do not just profile your development builds. If you use different compiler flags for deployment builds, especially with higher optimization levels, you will want to do some profiling on your final build so that you do not waste time fixing code paths that will change with compiler optimizations.

Major Causes of Performance Problems

Performance problems typically come from one or more of 5 major areas: algorithms, memory, CPU, disk, and graphics. Granted, that is pretty much everything your program interacts with in the machine. You can use performance tools to look at each aspect of computer performance in isolation to get a

better handle on your overall performance issues, even if one problem is causing problems in several categories.

Memory

Even though modern machines have vast amounts of memory, RAM is still a scarce resource. Once your app or other apps on the system fill up memory, Mac OS X starts sending memory pages to disk, destroying performance. On iOS devices, your program may be killed outright in low-memory situations.

Typically if you optimize to reduce your memory usage (optimizing for space), you will often get reductions in execution time because the processor is not waiting for that extra data to arrive from memory. Also, because Mac OS X is a shared system with daemons running, with each user running lots of programs of their own, and potentially multiple users logged in, it is good to be conservative with your memory usage. This can be a tough discipline when each process has its own wide-open address space to play in, especially when using 64-bit addressing.

Locality of Reference

“Locality of reference” describes memory accesses that happen near each other. Reading a hundred bytes off one 4k page is faster than reading one byte off a hundred different pages scattered across the address space. When you ask for data from memory, the processor actually grabs a sequence of bytes, known as a cache line, under the assumption that you will be accessing memory in a contiguous manner. From the processor’s point of view, it is just as fast to grab a 64-byte cache line as it is to grab a 4-byte integer. So, if you set up your loops to operate on memory sequentially, you can see a performance boost.

Example 10.1 creates a large two dimensional global array and accesses it in two different ways.

Example 10.1 locality.m

```
// locality.m -- time locality of reference

#include <stdio.h>      // for printf
#include <stdlib.h>     // for EXIT_SUCCESS
#include <time.h>       // for time_t, time()

// gcc -g -std=c99 -o locality locality.m

#define ARRAYSIZE 20000
int a[ARRAYSIZE][ARRAYSIZE]; // make a huge array

int main (int argc, char *argv[]) {
    // Walk the array in row-major order, so that once we're done
    // with a page we never bother with it again.

    time_t starttime = time(NULL);
    for (int i = 0; i < ARRAYSIZE; i++){
        for(int j = 0; j < ARRAYSIZE; j++){
            a[i][j] = 1;
        }
    }

    time_t endtime = time (NULL);
```

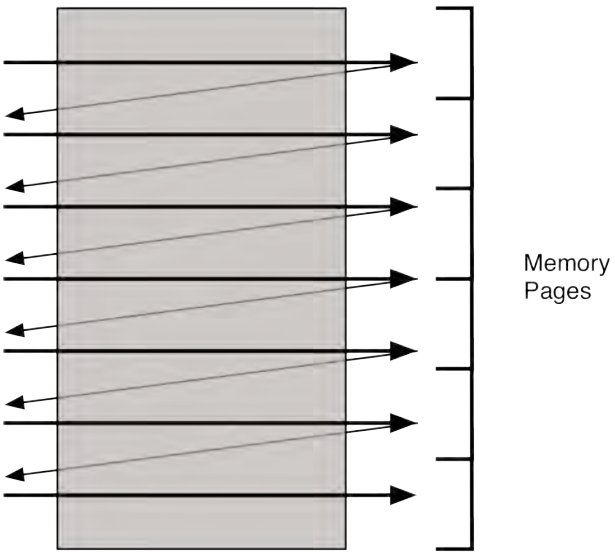
```
    printf("row-major: %d operations in %ld seconds.\n",
           ARRAYSIZE * ARRAYSIZE, endtime - starttime);

    // Walk the array in column-major order.  It ends up touching a bunch of
    // pages multiple times.

    starttime = time(NULL);
    for (int j = 0; j < ARRAYSIZE; j++){
        for(int i = 0; i < ARRAYSIZE; i++){
            a[i][j] = 1;
        }
    }

    endtime = time (NULL);

    printf("column-major: %d operations in %ld seconds.\n",
           ARRAYSIZE * ARRAYSIZE, endtime - starttime);

    return EXIT_SUCCESS;

} // main
```

Here is a sample run:

```
$ ./locality
row-major: 400000000 operations in 3 seconds.
column-major: 400000000 operations in 27 seconds.
```

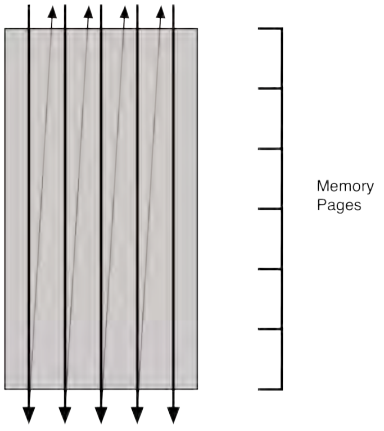
A simple reversal of the `for` loops can result in a 9x performance penalty! The first loop follows the way that C has the array's memory organized, as shown in Figure 10.1. This loop accesses adjacent bytes, and as it works through the array, it has good locality of reference. Memory pages are accessed only once, and after the loop has stopped manipulating memory on a page, that page is no longer used.

Figure 10.1 Good memory access pattern



The second loop works “across the grain,” as shown in Figure 10.2. It ends up hitting every page used by the array every time through the loop. This puts a lot of pressure on the virtual memory system because every page stays “warm,” causing the kernel to keep shuffling its least-recently-used page lists. The first loop, because it does not touch a page once the work is done, is nicer to the kernel. Once the page ages out of the kernel’s data structures, it is never seen again.

Figure 10.2 Bad memory access pattern

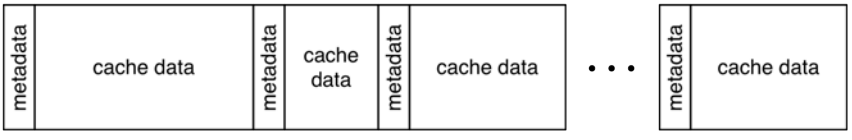


Caches

One common way to boost performance is caching, where you keep around some loaded or calculated data in memory. If you are not careful, this technique can have drawbacks in a system that employs virtual memory and paging. Recall that memory that hasn’t been accessed recently can be paged out to disk and the space in RAM can be used by other processes. iOS devices do not page data to disk, so dirty pages are always resident.

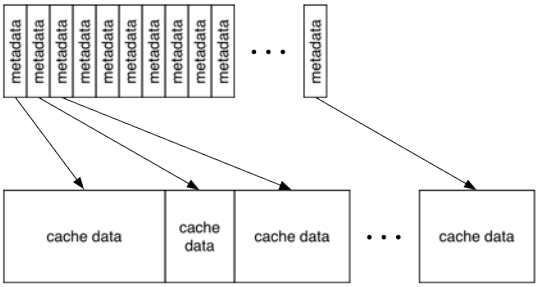
If do you choose to cache information, it is best to split up your cache data and the metadata that describes the cached data. You don’t want to use an architecture like Figure 10.3, which mixes the cache data and the metadata.

Figure 10.3 Bad locality of reference



Instead, organize your data as shown in Figure 10.4. Keep your metadata together because you will have good locality of reference when walking through your cache looking for expired objects. You can even do your own form of virtual memory: if a cache entry has not been used in a while or if you get an iOS memory warning, you can remove your data blob from memory and then load it again when needed.

Figure 10.4 Good locality of reference



Memory is the New I/O

The motivation that drives programmers to cache data read from disk is that I/O from disk is hideously expensive. Waiting for one disk I/O can cost hundreds of thousands (or more) CPU cycles that could be put to better use.

With today's processors, memory subsystems, and bus architectures, RAM has become like I/O. Sometimes accessing memory can be extremely slow compared to CPU speed. For example, according to an Apple tech note, a G5 could do 16 to 50 vector adds in the time it takes to load a cache line from memory. And the situation can get even worse with modern processors.

The "precalculate and store in memory" technique can become a bottleneck compared to brute-force calculations. The CPU can grind through some calculations faster than the fetch from memory, and the look-up table can force more important data out of the CPU caches. The tech note goes on to say "In one example, vector code that converts unsigned char data to float and then applies a 9th order polynomial to it is still marginally faster than hand tuned scalar code that does a lookup into a 256 entry lookup table containing floats."

Level-1 Cache, the cache memory nearest the CPU logic units, has an area for instructions, but it is only 32 to 64 kilobytes large, per core. Optimizations that increase code length, like loop unrolling and 64-bit code in general, can blow out this cache, requiring code to be continually brought in from RAM. It becomes a balancing act between the size of code, what you calculate, and what you store and retrieve. Sometimes trial-and-error is the way to go to see what technique results in the best performance, especially if you have high-performance scientific modeling that will have long runtimes or if you are dealing with large volumes of data quickly, such as when processing video.

Semantics of the C language can get in the way, optimization-wise, especially with regards to memory. If the compiler knows how a particular chunk of memory is being accessed, it can cache the values in registers or even avoid loading data that has already been loaded. Because C has pointers, there can be aliasing problems. There may be a pointer elsewhere in the process that is pointing to (and could conceivably modify) a piece of memory that the compiler could otherwise optimize access to. This is why languages that do not have pointers, like FORTRAN, can perform much more aggressive optimizations.

When using a data structure through a pointer or a global variable in a loop, the compiler will emit code to reload that location in memory each time through the loop. It does this just in case the value was changed by someone else, either in another thread or by a function called inside the loop. Making a local variable to hold the global's value lets the compiler figure out that this data is not going to change and thus avoids the memory hit each time through the loop.

CPU

CPU usage is the metric that most programmers think about first when confronted with an optimization issue. “My app is pegging the CPU, and I need to speed it up.” Typically when CPU usage becomes a dominant factor, the root cause is a slow algorithm. It might have a high level of complexity, or it might just be a poor implementation. In almost all cases, changing your algorithm will give you more speedups than most other kinds of code or system tweaking. The classic example is changing from a bubble-sort, an order $O(N^2$ algorithm), to a quicksort or merge sort, which is O(n log n).

Sometimes a bad implementation of an algorithm can wreak havoc. For instance, a programming error turned **`strstr()`** in one version of SunOS 4.1.x from an O(N) operation to a worthless $O(N^2)$ one:

```
while (c < strlen(string)) {
    // do stuff with string[c]
}
```

Recall that C strings do not store their length. A string is just a sequence of bytes terminated by zero. **`strlen()`** has to traverse the entire string counting characters. There are tricks you can use to do the work in greater than one-byte chunks, but it’s still an O(N) operation. In this particular case, the length of the string is not going to change, so there is no reason to take the length every time through the loop.

Luckily, high CPU usage can be easily discovered by noticing that the CPU meter is pegged in Activity Monitor, that **top** is showing your app consuming 99% of the available CPU power, or that your laptop case has started glowing. The sampling and profiling tools discussed later in this chapter are ideal for tracking down the cause of these problems.

Disk

Disk access is very slow – many orders of magnitude slower than accessing memory. In general, if you can avoid disk I/O, do so. If you are planning on caching data from disk, remember that the virtual memory system also uses the disk. If you cache a large amount of data, you could end up causing the VM system to do disk I/O. This is a very bad situation because you have now exchanged one disk read (from disk into memory) into a read and a write to page it out and then another read to page it back in from the disk into memory.

Locality of reference plays a part when optimizing disk access when VM paging involved. With bad locality of reference, you end up touching lots of pages. These pages cause other pages to “age out” of the VM cache and get sent to disk. Eventually you will touch them again which could cause disk I/O to retrieve the data on those pages.

You can avoid some of the expense of disk I/O by not doing the work at all. Putting windows into different `.nib` files and loading them on demand is a common technique. If you do not need to show the window, there is no reason to load it in memory.

Similarly, if you have a large database of information, accessing it piecemeal can yield significant speedups over loading the whole thing into memory. Using memory-mapped files can avoid disk activity because only the parts of the file being touched will make their way into memory.

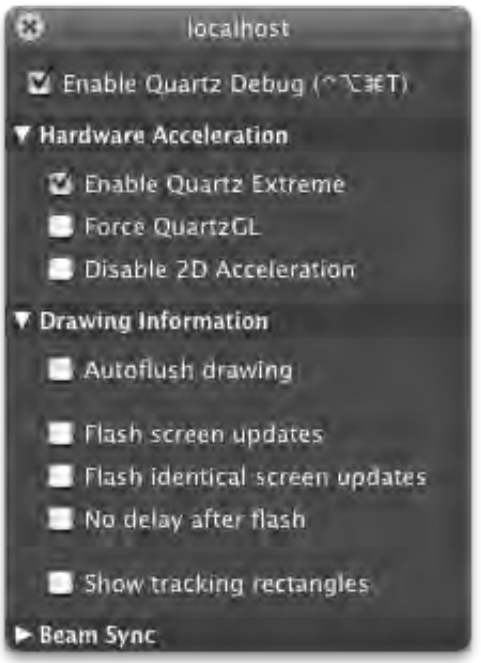
Graphics

The Quartz graphics engine in Mac OS X puts a lot of pressure on the memory system. Quartz uses large graphic buffers, one for each window visible on the screen. There are also compositing

operations to render the user's desktop. Quartz also uses the CPU to do some of its drawing effects, although many of these operations have been migrated to the graphics processing units on the graphics card. There are some operations that do not work well on the GPU, so these must be done on the CPU.

The key to optimizing graphics is to avoid drawing when you can. Use the Quartz Debug utility, shown in Figure 10.5, to see where you are doing unnecessary drawing. The most commonly used features are in the Drawing Information panel. Autoflush drawing causes drawing operations to appear on the screen as soon as they happen rather than being batched for the next update. Areas to the screen that are drawn to can be made to flash so you can see where you are drawing. Identical screen updates are highlighted in a different color so you can see where redundant work is happening.

Figure 10.5 Quartz Debug



`NSView` has some features that let you decide which parts of the view need to be redrawn and which ones do not. You can hit-test the rectangle that is passed to **`NSView`**'s **`drawRect:`** method and only perform drawing calls for items that live in that rectangle. This rectangle tends to be the union of all of the area that needs redrawing, so you can consult **`getRectsBeingDrawn:`** and **`needsToDrawRect:`** to hit-test against the areas that need to be redrawn.

One aspect of Quartz drawing that catches some programmers off guard is that overlapping lines in a single path can be very expensive. A lot of work happens at line crossings, such as anti-aliasing the intersections, as well as making sure that transparent colors do not get "painted" multiple times at the crossings and appear darker. If you need to draw lots of overlapping lines, especially if you are using opaque colors and do not care about antialiasing, you can get much better performance by drawing a bunch of small paths instead.

Before using any of the profiling tools

Forget any assumptions you may have about where the performance problems may be. Programmers are notoriously bad about predicting where performance problems are; otherwise, the problems would already be fixed. One programmer I worked with was convinced that file loading and disk I/O was the slow part of his program when loading files, and he was spending a lot of effort to optimize disk access. After a quick session with Shark, the problem actually turned out to be the marshaling of data into a tree so that **`NSOutlineView`** could use it. The time spent in actual file I/O was minimal.

Keep good notes on what you do and the measurements you make so that you can apply the optimizations to other situations. By keeping a record of execution times (for instance), you can tell if your optimization attempts are helping or are actually making the problem worse.

When tracking down performance problems, throw a large data set at your application. With the file-loading issue mentioned earlier, some of the test data were 5K files that took a second or two to load. That's too small a window in which to figure anything out. If your application is designed to edit 50-page research papers, then throw a 500- or 5000-page document at it. The larger data sets should make $O(N^2)$ algorithms stand out like the proverbial sore thumb. If your program is responsive when editing 5000-page documents, it should give the user a really nice experience when they are using it to edit 50-page documents. Do not bother with more than two or three orders of magnitude more data, since that much more data will probably require a redesign of your data structures and may become suboptimal for smaller data sets.

There is some debate over when you should optimize. One school of thought is "premature optimization is the root of all evil," and you should wait until the end of your development cycle to identify and fix performance problems. Unfortunately, that can require re-engineering large chunks of the product if there is a deeply-rooted bottleneck. Another school of thought is to act like you are on a diet and adopt a constant discipline about performance. The downside to that is that premature optimization can obscure the design and the code and make it harder to track down program errors before shipping.

As with most everything in life, the middle ground is a good place to live. Keep an eye out for algorithms that can be improved, but do not obfuscate code to trim every cycle you can too early in the development process. Throw large data sets at your program often. Do not wait until right before a trade show or a launch to subject your program to what the customer will throw at it. Keep an eye on your memory usage so that it does not grow too large too quickly. Also be sure to run the program in the user's environment. If you are writing a desktop app, be sure to have Safari and iTunes running, since the user will probably be using those apps, too. If your application is a memory pig and makes iTunes skip, you will definitely get some user complaints.

Command-Line Tools

Mac OS X comes with a number of command-line tools for tracking down particular types of performance problems. The nice thing about the command line is that you can remotely log into a machine and watch things as they happen. They also don't interfere with your application's user interface.

(Mac OS X also comes with a number of powerful GUI performance tools, which are easier to use than the command-line tools. We'll look more closely at those in the next chapter.)

time

The simplest tool is **time**. It times command execution and shows you clock time, CPU time in userspace, and CPU time spent in the kernel. Here is a run of `/usr/bin/time` on TextEdit. The time measured was the time starting TextEdit up, loading `/usr/share/dict/words`, and then scrolling from the top to the bottom.

```
$ time /Applications/TextEdit.app/Contents/MacOS/TextEdit
real   0m14.619s
user   0m1.257s
sys    0m0.180s
```

This is 14 seconds of clock time, one second in user space, and less than a second in the kernel.

The C shell has its own version of time that gives more information:

```
% time /Applications/TextEdit.app/Contents/MacOS/TextEdit
2.515u 0.226s 0:15.01 18.1% 0+0k 22+43io 0pf+0w
```

This is 2.5 seconds in user space, 0.2 seconds in kernel space, and fifteen seconds clock time. The 18.1% is a utilization percentage: the ratio of user + system times to real time. Following the time information is memory information: shared + unshared memory usage, input + output operations, number of pagefaults and swaps. OS X seems not to report the shared + unshared memory usage.

time is very handy when comparing optimizations. Run a baseline or two with **time**, make the optimization, then try **time** again. If you are optimizing CPU usage and discover CPU time figures going up, you should reconsider that particular optimization.

dtruss

Many Unix systems have a utility that will show all of the system calls a program makes. On Solaris, it is called **truss**; on Linux, it's **strace**. Mac OS X 10.4 has **ktrace** (kernel tracing), and Mac OS X 10.5 and later have a similar utility, **dtruss**, based on DTrace.

dtruss requires root privileges to run because DTrace requires them. Run it like this:

```
$ sudo dtruss ls
```

This will generate a couple of hundred lines of output, like

```
SYSCALL(args)    = return
getpid(0x7FFF5FBFF600, 0x7FFFFFE00050, 0x0)   = 9234 0
open_nocancel("/dev/urandom\0", 0x0, 0x0)   = 3 0
read_nocancel(0x3, "...", 0x6C)   = 108 0
close_nocancel(0x3)   = 0 0
issetugid(0x100000000, 0x7FFF5FBFF8C8, 0x7FFF5FC40530)   = 0 0
geteuid(0x100000000, 0x7FFF5FBFF8C8, 0x0)   = 0 0
...
```

The output is not terribly easy to read, but there is a lot of information there. As **ls** started up, it called **`getpid()`**, which returned the value 9234, the process ID for **ls**. After the function exited, `errno` was zero. If there was an error, **dtruss** would print a result like this:

```
stat64("grausenstein\0", 0x7FFF5FBFEFB0, 0x1)   = -1 Err#2
```

with a return value of -1, and `errno` set to 2.

Being able to see the system call traffic can be a great debugging aid, especially if you have a program that will not start. You can see if the program is trying to load a missing shared library or if it needs some configuration file that is not supplied.

System call tracking can be a performance-tuning aid, too. You might discover a lot of one-byte writes that can be coalesced into a single operation, you may have given a bad timeout to **`kevent()`** so that it returns a lot sooner than you expect, or you can see why your program is blocking unexpectedly.

fs_usage and sc_usage

fs_usage and **sc_usage** are programs run as the root user that also show system call activity. **fs_usage** shows file system information, and **sc_usage** shows system call information.

Here is BigShow, the Big Nerd Ranch slide show application, about to start paging through slides:

```
$ sudo fs_usage
password:
18:38:06 open  /Preferences/com.apple.dock.plist 0.00005   BigShow
18:38:06 fstat                                   0.00000   BigShow
18:38:06 read                                    0.00029   BigShow
18:38:06 close                                   0.00002   BigShow
18:38:06 open  com.apple.dock.0003931024a6.plist 0.00015   BigShow
18:38:06 PAGE_IN                                 0.00070 W BigShow
18:38:06 open  /Library/Preferences/Network      0.00008   BigShow
18:38:06 open  com.apple.systempreferences.plist 0.00005   BigShow
```

Part of Cocoa is looking at the plist for the dock, presumably for getting size and location information so that it can properly place a window. You can see the open, a stat to get the size of the file, the file being read, and its close. Unlike **dtruss**, there is not an easy way to correlate specific calls like a **`read()`** with the file descriptor it is using, but **fs_usage** does show you how much time it took. **fs_usage** can be run on a system-wide basis, which can be handy if you have a problem that is slowing the entire machine down. **fs_usage** is also useful when you have a program that accesses the hard drive unexpectedly and you want to track down who is responsible.

One really snazzy feature of **fs_usage** can be seen when used on applications that make Carbon file-system calls. If you set the environment variable `DYLD_IMAGE_SUFFIX` to `_debug`, **fd_usage** will show the Carbon calls being made. Here is a peek at an old copy of Mozilla running:

```
18:34:38 GetCatInfo                          0.000174  LaunchCFMApp
18:34:38 PBMakeFSSpec  (0, 0x0, 0x0, 0x0)                LaunchCFMApp
18:34:38 getattrlist   .vol/280763/Mozilla.app 0.000032  LaunchCFMApp
18:34:38 PBMakeFSSpec                        0.000064  LaunchCFMApp
18:34:38 GetCatInfo    (-100, 0x0, 0x0, 0x0)             LaunchCFMApp
18:34:38 getattrlist   .vol/280763/Mozilla.app 0.000046  LaunchCFMApp
```

sc_usage shows system calls for a program in a manner like **top**, with a continually updating display. Here is a snapshot from Safari:

```
Safari       12 preemptions  189 context switches    8 threads    16:05:42
              0 faults       706 system calls                     0:00:07

TYPE                        NUMBER        CPU_TIME   WAIT_TIME
-------------------------------------------------------------------------------
System         Idle                                  00:06.608(00:00.922)
System         Busy                                  00:00.368(00:00.065)
Safari         Usermode                 00:00.109
```

```
mach_msg_trap              2004(382)     00:00.013  00:17.878(00:02.978) 3
kevent                       20(3)       00:00.000  00:05.531(00:01.005) W
semwait_signal                2(1)       00:00.000  00:05.496(00:01.003) W
select                       13          00:00.000  00:05.076(00:01.004) W

CURRENT_TYPE          LAST_PATHNAME_WAITED_FOR     CUR_WAIT_TIME THRD# PRI
-------------------------------------------------------------------------
mach_msg_trap                                      00:00.016     0     46
mach_msg_trap                                      00:00.150     1     46
semwait_signal                                     00:00.496     2     47
kevent                                             00:00.481     3     49
workq_ops                                          00:00.478     4     47
select                                             00:01.391     5     46
mach_msg_trap                                      00:01.391     6     62
bsdthread_terminate                                00:05.019     7     47
```

The `CPU_TIME` column is the amount of CPU time consumed, and `WAIT_TIME` is the absolute time the process waits.

If you think you have I/O performance problems, these two programs can help you track down the specific calls that could be causing problems.

top

Unix systems are complex beasts composed of multiple programs interacting. Sometimes performance problems manifest as overall system slowness while each program looks just fine in isolation. The **dtruss** and **sc_usage** utilities are useful for monitoring system calls in a particular program. **top**, on the other hand, can be used to monitor all the programs on the system. Running **top** without arguments will show the familiar OS information (memory distributions, load average). By default, it orders programs by launch order (most recent program listed first). This is useful if you are monitoring a recently launched program. The `-u` flag will sort the list by CPU usage.

top can also count and show system-wide events. `top -e` shows VM (virtual memory), network activity, disk activity, and messaging stats:

```
$ top -e

Processes: 70 total, 4 running, 66 sleeping, 260 threads            16:09:30
Load Avg: 0.19, 0.28, 0.24  CPU usage: 6.27% user, 5.51% sys, 88.20% idle
SharedLibs: 7424K resident, 7276K data, 0B linkedit.
MemRegions: 9361 total, 451M resident, 17M private, 290M shared.
PhysMem: 798M wired, 1062M active, 427M inactive, 2287M used, 1809M free.
VM: 168G vsize, 1041M framework vsize, 339589(0) pageins, 278055(0) pageouts.
Networks: packets: 2550949/3311M in, 1667316/160M out.
Disks: 561048/8214M read, 976911/21G written.

PID   COMMAND    %CPU TIME     #TH  #WQ #POR #MRE RPRVT  RSHRD  RSIZE  VPRVT
9317  top        3.8  00:00.63 1/1  0   24   33   920K   244K   1496K  17M
9316- WebKitPlug 0.3  00:03.47 6    2   110  245  9584K  30M    15M    49M
9306  Safari     0.2  00:12.02 11   2   167  533  97M    53M    153M   265M
9299  csh        0.0  00:00.04 1    0   15   26   612K   592K   1160K  17M
9268  mdworker   0.0  00:00.36 3    1   50   77   4388K  19M    12M    29M
9202  bash       0.0  00:00.03 1    0   17   25   320K   244K   976K   9576K
9198  csh        0.0  00:00.02 1    0   17   26   580K   592K   1164K  9648K
9197  login      0.0  00:00.02 1    0   22   54   472K   312K   1608K  10M
9180  ssh        0.0  00:00.04 1    0   22   25   508K   244K   1812K  9588K
8900  Preview    0.0  00:16.27 2    1   125  268  20M    73M    45M    28M
```

```
8886  WebKitPlug 0.0  00:00.01 2    2   28   51   544K   244K   988K   40M
8850  Activity M 0.0  00:35.63 2    1   106  243  5360K  75M    15M    25M
8700  VDCAssista 0.0  00:00.15 4    1   90   73   384K   19M    3104K  23M
8697- Snak       1.5  07:03.51 5/1  1   192  232  3788K  43M    11M    40M
```

There is a lot of information here. 70 processes, 260 threads system-wide. Shared libraries take about 7 megabytes of memory, 1.8 gigs of physical memory free, 168 gigs of virtual memory allocated, network and disk I/Os. Each process has information such as the number of threads, work queues, mach ports, and memory regions. Resize your terminal window to see more columns, such as virtual size, process state, page faults, bsd system calls made, etc.

top -e shows cumulative output, while **top -d** will show things in a delta mode. The update interval is one second. That can be changed by using the `-s` flag to control the number of seconds between intervals.

Stochastic profiling

One useful low-tech tool is "stochastic profiling," where you run the program in the debugger and interrupt it occasionally to see what is on the call stack. If you see the same function(s) on the stack over and over again, you know where to start looking. This technique is handy if you are on a platform or in a situation where traditional performance tools are not available or do not work. Plus, it's fast and easy, especially if you are already running your program in a debugger.

sample

You can do some profiling from the command-line to answer quick-and-dirty "what is happening here?" kinds of questions. The **sample** program will sample a process at 10-millisecond intervals and then build a snapshot of what the program was doing. You can give sample a pid or give it the partial name of a program:

```
$ sample iTunes 5
Sampling process 216 each 10 msecs 500 times
Sample analysis of process 216 written file /tmp/iTunes_216.sample.txt
```

The resulting trace file shows a bunch of call stacks, one for each thread, along with the number of times it found those particular functions on a call stack. Here's an example of one thread that is waiting in a run loop.

```
434 Thread_1103
  434 _pthread_body
    434 dyld_stub_binding_helper
      434 CFRunLoopRun
        434 CFRunLoopRunSpecific
          434 __CFRunLoopRun
            434 mach_msg
              434 mach_msg_trap
                434 mach_msg_trap
```

This is the same output you get when sampling processes in Activity Monitor.

Precise Timing with mach_absolute_time()

Command-line tools are a great place to benchmark snippets of code, which is useful for those cases where you can isolate an algorithm or a programming technique out of your full application. A dozen-

line or a couple-hundred-line command-line tool is a much more tractable problem than a million-line application. Not every problem can be put into a little benchmark, but enough of them can to make it a useful technique.

The nice thing about command-line programs is you can use the **time** command to get absolute figures of the running time of the program making it easy to compare and contrast changes you make to your target program.

But sometimes the **time** command does not have enough granularity. You might want more precise timing, or you may just be interested in timing a specific part of your program. You might not be interested in the time it takes to load the data to feed your algorithm. If loading the data takes 3 times as long as it takes the algorithm to run, you will want to do timing inside of the program yourself.

Mach, Mac OS X's kernel, provides some functions you can use for precise timing. **`mach_absolute_time()`** reads the CPU time base register and reports the value back to you. This time base register serves as the basis for other time measurements in the OS:

```
uint64_t mach_absolute_time (void);
```

`mach_absolute_time()` returns values based on the CPU time, so it is not directly usable for getting time values because you do not know what time span each increment of the counter represents.

To translate **`mach_absolute_time()`**'s results to nanoseconds, use **`mach_timebase_info()`** to get the scaling of **`mach_absolute_time()`**'s values:

```
kern_return_t mach_timebase_info (mach_timebase_info_t info);
```

Where `mach_timebase_info_t` is a pointer to this struct:

```
struct mach_timebase_info {
    uint32_t    numer;
    uint32_t    denom;
};
```

`mach_timebase_info()` fills in the struct with the fraction to multiply the result of **`mach_absolute_time()`** by to calculate nanoseconds. Multiply the result of **`mach_absolute_time()`** by `numer` and divide by `denom`.

Example 10.2 shows how to use these two functions. The code times how long it takes to call **`mach_timebase_info()`** and **`printf()`**. For real-life code, you would want to put something more interesting in there to time.

Example 10.2 machtime.m

```
// machtime.m -- exercise mach_absolute_time()

#import <mach/mach_time.h>  // for mach_absolute_time() and friends
#import <stdio.h>           // for printf()
#import <stdlib.h>          // for abort()

// gcc -g -Wall -o machtime machtime.m

int main (void) {
    uint64_t start = mach_absolute_time ();

    mach_timebase_info_data_t info;
```

```
    if (mach_timebase_info (&info) == KERN_SUCCESS) {
        printf ("scale factor : %u / %u\n", info.numer, info.denom);
    } else {
        printf ("mach_timebase_info failed\n");
        abort ();
    }

    uint64_t end = mach_absolute_time ();
    uint64_t elapsed = end - start;
    uint64_t nanos = elapsed * info.numer / info.denom;

    printf ("elapsed time was %lld nanoseconds\n", nanos);

    return 0;

} // main
```

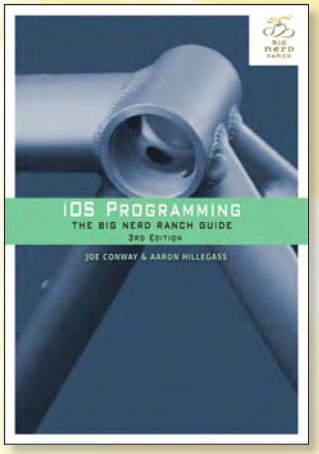
And here it is in action:

```
$ ./machtime
scale factor : 1 / 1
elapsed time was 55055 nanoseconds
$ ./machtime
scale factor : 1 / 1
elapsed time was 95363 nanoseconds
$ ./machtime
scale factor : 1 / 1
elapsed time was 46839 nanoseconds
```

On this system, a 2010 Macbook Pro, the numerator and denominator of the conversion are both one. Some older machines, such as the original TiBook, had the numerator of the conversion at 1,000,000,000 and the denominator at 24,965,716, resulting in a scale value of 40.05. So there were about 40 nanoseconds for each increment of **`mach_absolute_time()`**.

Outside of the second run, it takes about 50,000 nanoseconds, or 50 microseconds to do the work between the two timings. So what's up with that middle run being twice as long as the others? When you are dealing with time values this short, *anything* can perturb them. Maybe some dynamic library lookup was necessary for that second run. Maybe iTunes was running and was loading a new track. Maybe Time Machine kicked in. For a real benchmark, you would run it for a longer period of time to hide those small one-time-only blips. And of course, you would run the benchmark a couple of times to get a good average and iron out the noise.

In the next chapter, we'll examine the GUI performance tools that Mac OS X provides, including the Activity Monitor and Instruments, Apple's suite of profiling tools.



CHAPTER 25
Web Services and UIWebView

iOS PROGRAMMING: THE BIG NERD RANCH GUIDE, THIRD EDITION, by Joe Conway and Aaron Hillegass

TABLE OF CONTENTS

AVAILABLE FORMATS

- 9780321821522 Book
- 9780132978750 eBook
- 9780132978767 Safari Books Online

iOS Programming

The Big Nerd Ranch Guide

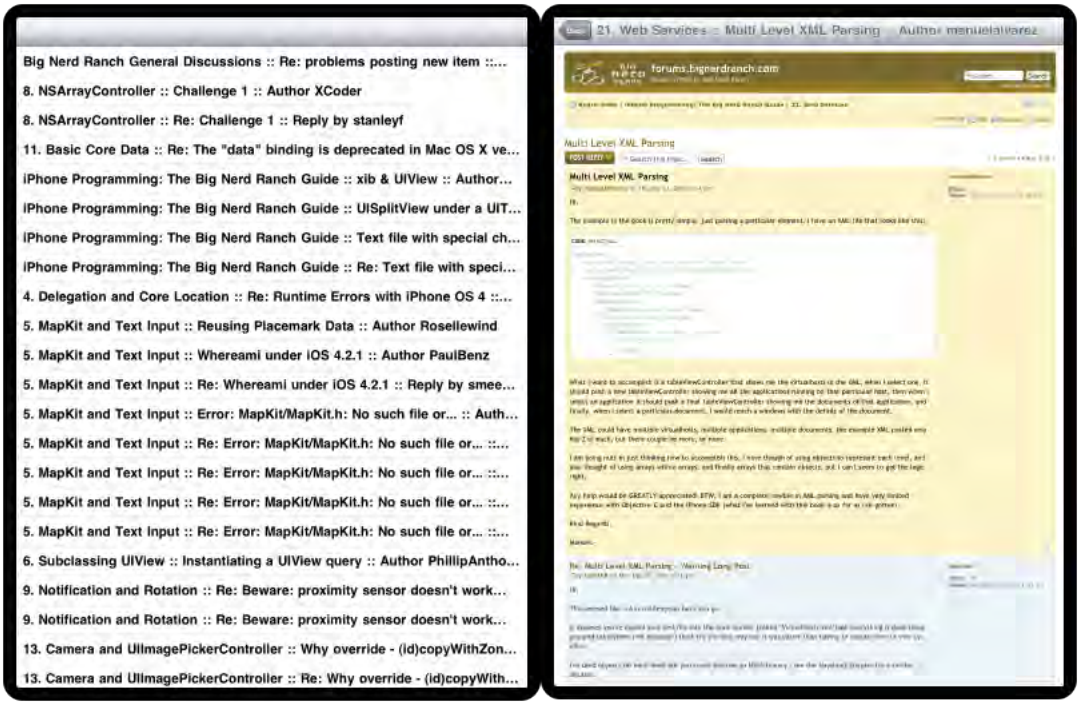
Joe Conway & Aaron Hillegass

Big nerd ranch

25
Web Services and UIWebView

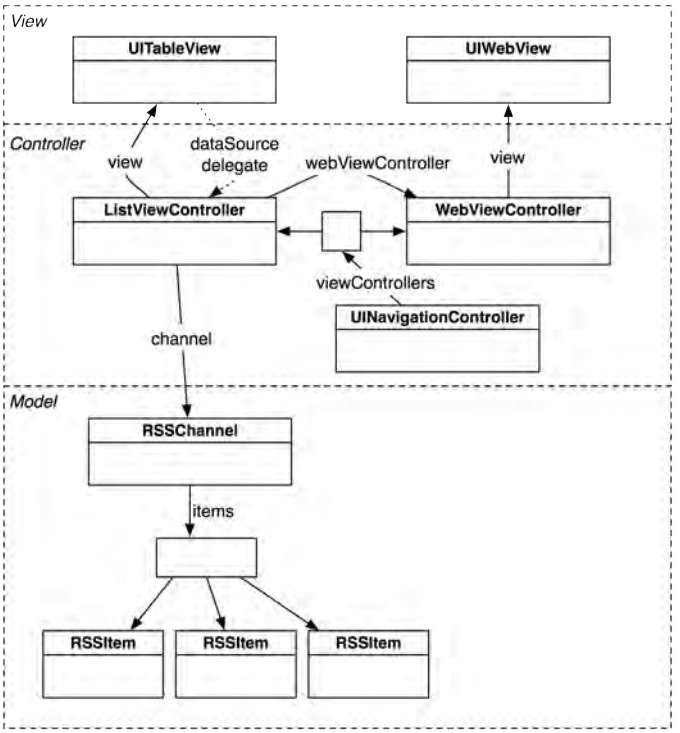
In this chapter, you will lay the foundation of an application that reads the RSS feed from the Big Nerd Ranch Forums (Figure 25.1). Forum posts will be listed in a table view, and selecting a post from the table will display it from the site. Figure 25.1 shows the Nerdfeed application at the end of this chapter.

Figure 25.1 Nerdfeed



We will divide the work into two parts. The first is connecting to and collecting data from a web service and using that data to create model objects. The second part is using the **`UIWebView`** class to display web content. Figure 25.2 shows an object diagram for Nerdfeed.

Figure 25.2 Nerdfeed object diagram



Web Services

Your handy web browser uses the HTTP protocol to communicate with a web server. In the simplest interaction, the browser sends a request to the server specifying a URL. The server responds by sending back the requested page (typically HTML and images), which the browser formats and displays.

In more complex interactions, browser requests include other parameters, like form data. The server processes these parameters and returns a customized, or dynamic, web page.

Web browsers are widely used and have been around for a long time. So the technologies surrounding HTTP are stable and well-developed: HTTP traffic passes neatly through most firewalls, web servers are very secure and have great performance, and web application development tools have become easy to use.

You can write a client application for iOS that leverages the HTTP infrastructure to talk to a web-enabled server. The server side of this application is a *web service*. Your client application and the web service can exchange requests and responses via HTTP.

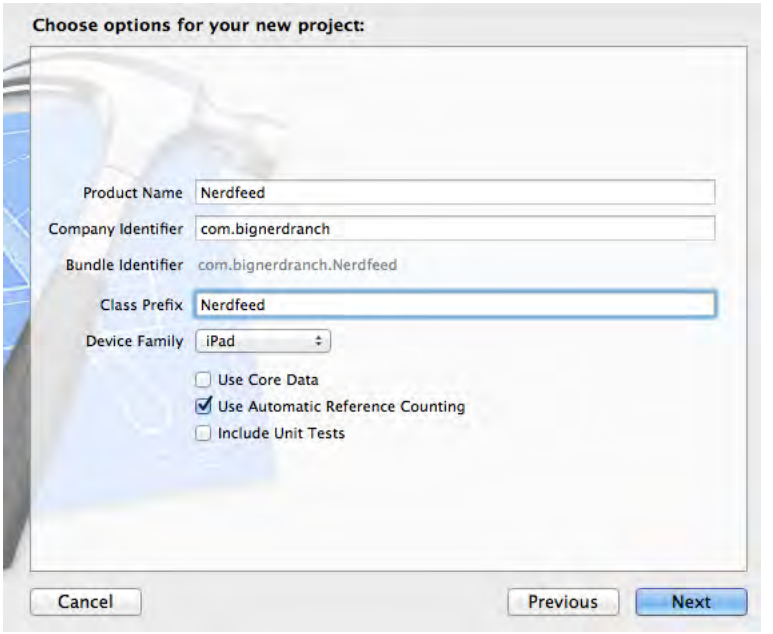
Because the HTTP protocol doesn't care what data it transports, these exchanges can contain complex data. This data is typically in XML or JSON (JavaScript Object Notation) format. If you control the web server as well as the client, you can use any format you like; if not, you have to build your application to use whatever the server supports.

In this chapter, you will create a client application that will make a request to the `smartfeed` web service hosted at `http://forums.bignerdranch.com`. You will pass a number of arguments to this service that determine the format of the data that is returned. This data will be XML that describes the most recent posts at our developer forums.

Starting the Nerdfeed application

Create a new Empty Application for the iPad Device Family. Name this application Nerdfeed, as shown in Figure 25.3. (If you don't have an iPad to deploy to, use the iPad simulator.)

Figure 25.3 Creating an iPad Empty Application



Let's knock out the basic UI before focusing on web services. Create a new **`NSObject`** subclass and name it **`ListViewController`**. In `ListViewController.h`, change the superclass to **`UITableViewController`**.

```
~~@interface ListViewController : NSObject~~
@interface ListViewController : UITableViewController
```

In `ListViewController.m`, write stubs for the required data source methods so that we can build and run as we go through this exercise.

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 0;
}
```

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;
}
```

In `NerdfeedAppDelegate.m`, create an instance of **`ListViewController`** and set it as the root view controller of a navigation controller. Make that navigation controller the root view controller of the window.

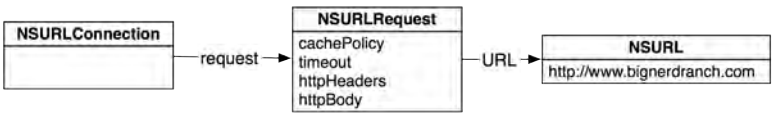
```
#import "NerdfeedAppDelegate.h"
#import "ListViewController.h"

@implementation NerdfeedAppDelegate
@synthesize window;

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


    ListViewController *lvc =
        [[ListViewController alloc] initWithStyle:UITableViewStylePlain];

    UINavigationController *masterNav =
        [[UINavigationController alloc] initWithRootViewController:lvc];

    [[self window] setRootViewController:masterNav];

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

Build and run the application. You should see an empty **`UITableView`** and a navigation bar.

NSURL, NSURLRequest, and NSURLConnection

The Nerdfeed application will fetch data from a web server using three handy classes: **`NSURL`**, **`NSURLRequest`**, and **`NSURLConnection`** (Figure 25.4).

Figure 25.4 Relationship of web service classes



Each of these classes has an important role in communicating with a web server:

- An **`NSURL`** instance contains the location of a web application in URL format. For many web services, the URL will be composed of the base address, the web application you are communicating with, and any arguments that are being passed.

- An **NSURLRequest** instance holds all the data necessary to communicate with a web server. This includes an **NSURL** object, as well as a caching policy, a limit on how long you will give the web server to respond, and additional data passed through the HTTP protocol. (**NSMutableURLRequest** is the mutable subclass of **NSURLRequest**.)
- An **NSURLConnection** instance is responsible for actually making the connection to a web server, sending the information in its **NSURLRequest**, and gathering the response from the server.

Formatting URLs and requests

The form of a web service request varies depending on who implements the web service; there are no set-in-stone rules when it comes to web services. You will need to find the documentation for the web service to know how to format a request. As long as a client application sends the server what it wants, you have a working exchange.

The Big Nerd Ranch Forum's RSS feed wants a URL that looks like this:

```
http://forums.bignerdranch.com/smartfeed.php?limit=1_DAY&sort_by=standard
&feed_type=RSS2.0&feed_style=COMPACT
```

You can see that the base URL is `forums.bignerdranch.com`, the web application is `smartfeed`, and there are five arguments. These arguments are required by the smartfeed web application.

This is a pretty common form for a web service request. Generally, a request URL looks like this:

```
http://baseURL.com/serviceName?argumentX=valueX&argumentY=valueY
```

At times, you will need to make a string "URL-safe." For example, space characters and quotes are not allowed in URLs; They must be replaced with escape-sequences. Here is how that is done.

```
NSString *search = @"Play some \"Abba\"";
NSString *escaped =
      [search stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

// escaped is now "Play%20some%20%22Abba%22"
```

When the request to the Big Nerd Ranch forums is processed, the server will return XML data that contains the last 20 posts. The **ListViewController**, who made the request, will populate its table view with the titles of the posts.

In `ListViewController.h`, add an instance variable for the connection and one for the data that is returned from that connection. Also add a new method declaration.

```
@interface ListViewController : UITableViewController
{
    NSURLConnection *connection;
    NSMutableData *xmlData;
}
- (void)fetchEntries;
@end
```

Working with NSURLConnection

An **NSURLConnection** instance can communicate with a web server either synchronously or asynchronously. Because passing data to and from a remote server can take some time, synchronous connections are generally frowned upon because they stall your application until the connection

completes. This chapter will teach you how to perform an asynchronous connection with **NSURLConnection**.

When an instance of **NSURLConnection** is created, it needs to know the location of the web application and the data to pass to that web server. It also needs a delegate. When told to start communicating with the web server, **NSURLConnection** will initiate a connection to the location, begin passing it data, and possibly receive data back. It will update its delegate each step of the way with useful information.

In ListViewController.m, implement the **fetchEntries** method to create an **NSURLRequest** that connects to http://forums.bignerdranch.com and asks for the last 20 posts in RSS 2.0 format. Then, create an **NSURLConnection** that transfers this request to the server.

```
- (void)fetchEntries
{
    // Create a new data container for the stuff that comes back from the service
    xmlData = [[NSMutableData alloc] init];

    // Construct a URL that will ask the service for what you want -
    // note we can concatenate literal strings together on multiple
    // lines in this way - this results in a single NSString instance
    NSURL *url = [NSURL URLWithString:
        @"http://forums.bignerdranch.com/smartfeed.php?"
        @"limit=1_DAY&sort_by=standard&feed_type=RSS2.0&feed_style=COMPACT"];


    // For Apple's Hot News feed, replace the line above with
    // NSURL *url = [NSURL URLWithString:@"http://www.apple.com/pr/feeds/pr.rss"];

    // Put that URL into an NSURLRequest
    NSURLRequest *req = [NSURLRequest requestWithURL:url];

    // Create a connection that will exchange this request for data from the URL
    connection = [[NSURLConnection alloc] initWithRequest:req
                                                 delegate:self
                                         startImmediately:YES];
}
```

Kick off the exchange whenever the **ListViewController** is created. In ListViewController.m, override **initWithStyle:**.

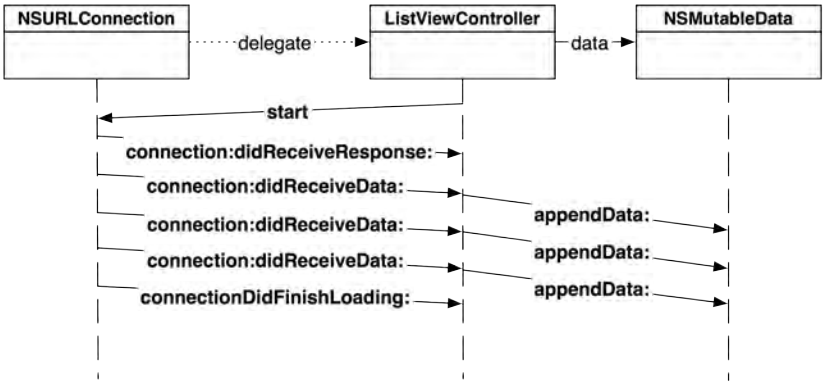
```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];

    if (self) {
        [self fetchEntries];
    }

    return self;
}
```

Build the application to make sure there are no syntax errors.

Collecting XML data

This code, as it stands, will make the connection to the web service and retrieve the last 20 posts. However, there is one problem: you don't see those posts anywhere. You need to implement delegate methods for **NSURLConnection** to collect the XML data returned from this request.

Figure 25.5 NSURLConnection flow chart



The delegate of an **`NSURLConnection`** is responsible for overseeing the connection and for collecting the data returned from the request. (This data is typically an XML or JSON document; for this web service, it is XML.) However, the data returned usually comes back in pieces, and it is the delegate's job to collect the pieces and put them together.

In `ListViewController.m`, implement **`connection:didReceiveData:`** to put all of the data received by the connection into the instance variable `xmlData`.

```
// This method will be called several times as the data arrives
- (void)connection:(NSURLConnection *)conn didReceiveData:(NSData *)data
{
    // Add the incoming chunk of data to the container we are keeping
    // The data always comes in the correct order
    [xmlData appendData:data];
}
```

When a connection has finished retrieving all of the data from a web service, it sends the message **`connectionDidFinishLoading:`** to its delegate. In this method, you are guaranteed to have the complete response from the web service request and can start working with that data. For now, implement **`connectionDidFinishLoading:`** in `ListViewController.m` to print out the string representation of that data to the console to make sure good stuff is coming back.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)conn
{
    // We are just checking to make sure we are getting the XML
    NSString *xmlCheck = [[NSString alloc] initWithData:xmlData
                                               encoding:NSUTF8StringEncoding];
    NSLog(@"xmlCheck = %@", xmlCheck);
}
```

There is a possibility that a connection will fail. If an instance of **`NSURLConnection`** cannot make a connection to a web service, it sends its delegate the message **`connection:didFailWithError:`**. Note that this message gets sent for a *connection* failure, like having no Internet connectivity or if the server doesn't exist. For other types of errors, such as data sent to a web service in the wrong format, the error information is returned in **`connection:didReceiveData:`**.

In `ListViewController.m`, implement **`connection:didFailWithError:`** to inform your application of a connection failure.

```
- (void)connection:(NSURLConnection *)conn
  didFailWithError:(NSError *)error
{
    // Release the connection object, we're done with it
    connection = nil;

    // Release the xmlData object, we're done with it
    xmlData = nil;

    // Grab the description of the error object passed to us
    NSString *errorString = [NSString stringWithFormat:@"Fetch failed: %@",
                             [error localizedDescription]];

    // Create and show an alert view with this error displayed
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:@"Error"
                                                 message:errorString
                                                delegate:nil
                                       cancelButtonTitle:@"OK"
                                       otherButtonTitles:nil];
    [av show];
}
```

Try building and running your application. You should see the XML results in the console shortly after you launch the application. If you put your device in Airplane Mode (or if it is not connected to a network), you should see a friendly error message when you try to fetch again. (For now, you will have to restart the application from Xcode in order to refetch the data after you've received the error.)

The XML that comes back from the server looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
  <channel>
    <title>forums.bignerdranch.com</title>
    <description>Books written by Big Nerd Ranch</description>
        ...
    <item>
      <title>Big Nerd Ranch General Discussions :: Big Nerd Ranch!</title>
      <link>http://forums.bignerdranch.com/viewtopic.php?f=4&t=532</link>
      <author>no_email@example.com (bignerd)</author>
      <category>Big Nerd Ranch General Discussions</category>
      <comments>http://forums.bignerdranch.com/posting.php?mode=reply</comments>
      <pubDate>Mon, 27 Dec 2010 11:27:01 GMT</pubDate>
    </item>
    ...
  </channel>
</rss>
```

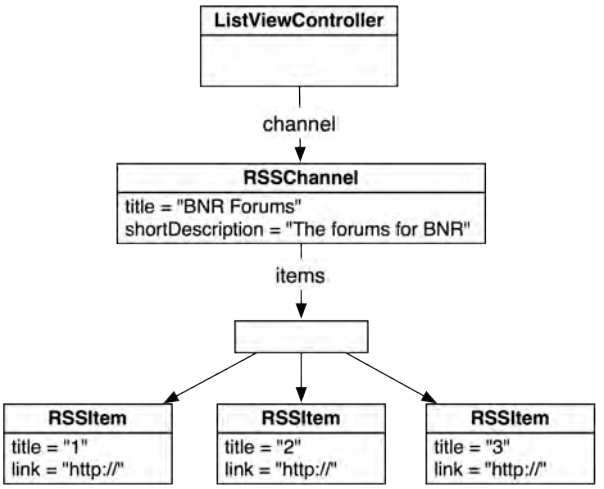
(If you aren't seeing anything like this in your console, verify that you typed the URL correctly.)

Let's break down the XML the server returned. The top-level element in this document is an `rss` element. It contains a `channel` element. That `channel` element has some metadata that describes it (a title and a description). Then, there is a series of `item` elements. Each `item` has a title, link, author, etc. and represents a single post on the forum.

In a moment, you will create two new classes, **`RSSChannel`** and **`RSSItem`**, to represent the `channel` and `item` elements. The **`ListViewController`** will have an instance variable for the **`RSSChannel`**. The

RSSChannel will hold an array of **RSSItem**s. Each **RSSItem** will be displayed as a row in the table view. Both **RSSChannel** and **RSSItem** will retain some of their metadata as instance variables, as shown in Figure 25.6.

Figure 25.6 Model object graph



Parsing XML with NSXMLParser

To parse the XML, you will use the class **NSXMLParser**. An **NSXMLParser** instance takes a chunk of XML data and reads it line by line. As it finds interesting information, it sends messages to its delegate, like, "I found a new element tag," or "I found a string inside of an element." The delegate object is responsible for interpreting what these messages mean in the context of the application.

In `ListViewController.m`, delete the code you wrote in **connectionDidFinishLoading:** to log the XML. Replace it with code to kick off the parsing and set the parser's delegate to point at the instance of **ListViewController**.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)conn
{
    NSString *xmlCheck = [[NSString alloc] initWithData:xmlData
                                               encoding:NSUTF8StringEncoding];
    NSLog(@"xmlCheck = %@", xmlCheck);


    // Create the parser object with the data received from the web service
    NSXMLParser *parser = [[NSXMLParser alloc] initWithData:xmlData];

    // Give it a delegate - ignore the warning here for now
    [parser setDelegate:self];

    // Tell it to start parsing - the document will be parsed and
    // the delegate of NSXMLParser will get all of its delegate messages
    // sent to it before this line finishes execution - it is blocking
    [parser parse];
```

```
    // Get rid of the XML data as we no longer need it
    xmlData = nil;

    // Get rid of the connection, no longer need it
    connection = nil;

    // Reload the table.. for now, the table will be empty.
    [[self tableView] reloadData];
}
```

The delegate of the parser, **ListViewController**, will receive a message when the parser finds a new element, another message when it finds a string within an element, and another when an element is closed.

For example, if a parser saw this XML:

```
<title>Big Nerd Ranch</title>.
```

it would send its delegate three consecutive messages: "I found a new element: 'title'," then, "I found a string: 'Big Nerd Ranch'," and finally, "I found the end of an element: 'title'." These messages are found in the `NSXMLParserDelegate` protocol:

```
// The "I found a new element" message
  - (void)parser:(NSXMLParser *)parser            // Parser that is sending message
 didStartElement:(NSString *)elementName          // Name of the element found
    namespaceURI:(NSString *)namespaceURI
   qualifiedName:(NSString *)qualifiedName
      attributes:(NSDictionary *)attributeDict;

// The "I found a string" message
  - (void)parser:(NSXMLParser *)parser            // Parser that is sending message
 foundCharacters:(NSString *)string;              // Contents of element (string)

// The "I found the end of an element" message
- (void)parser:(NSXMLParser *)parser             // Parser that is sending message
 didEndElement:(NSString *)elementName           // Name of the element found
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName;
```

The `namespaceURI`, `qualifiedName`, and `attributes` arguments are for more complex XML, and we'll return to them at the end of the chapter.

Constructing the tree of model objects

It is up to the **ListViewController** to make sense of that series of messages, and it does this by constructing an object tree that represents the XML feed. In this case, after the XML is parsed, there will be an instance of **RSSChannel** that contains a number of **RSSItem** instances. Here are the steps to constructing the tree:

- When the parser reports it found the start of the `channel` element, create an instance of **RSSChannel**.
- When the parser finds a `title` or `description` element and it is currently inside a `channel` element, set the appropriate property of the **RSSChannel** instance.
- When the parser finds an `item` element, create an instance of **RSSItem** and add it to the `items` array of the **RSSChannel**.

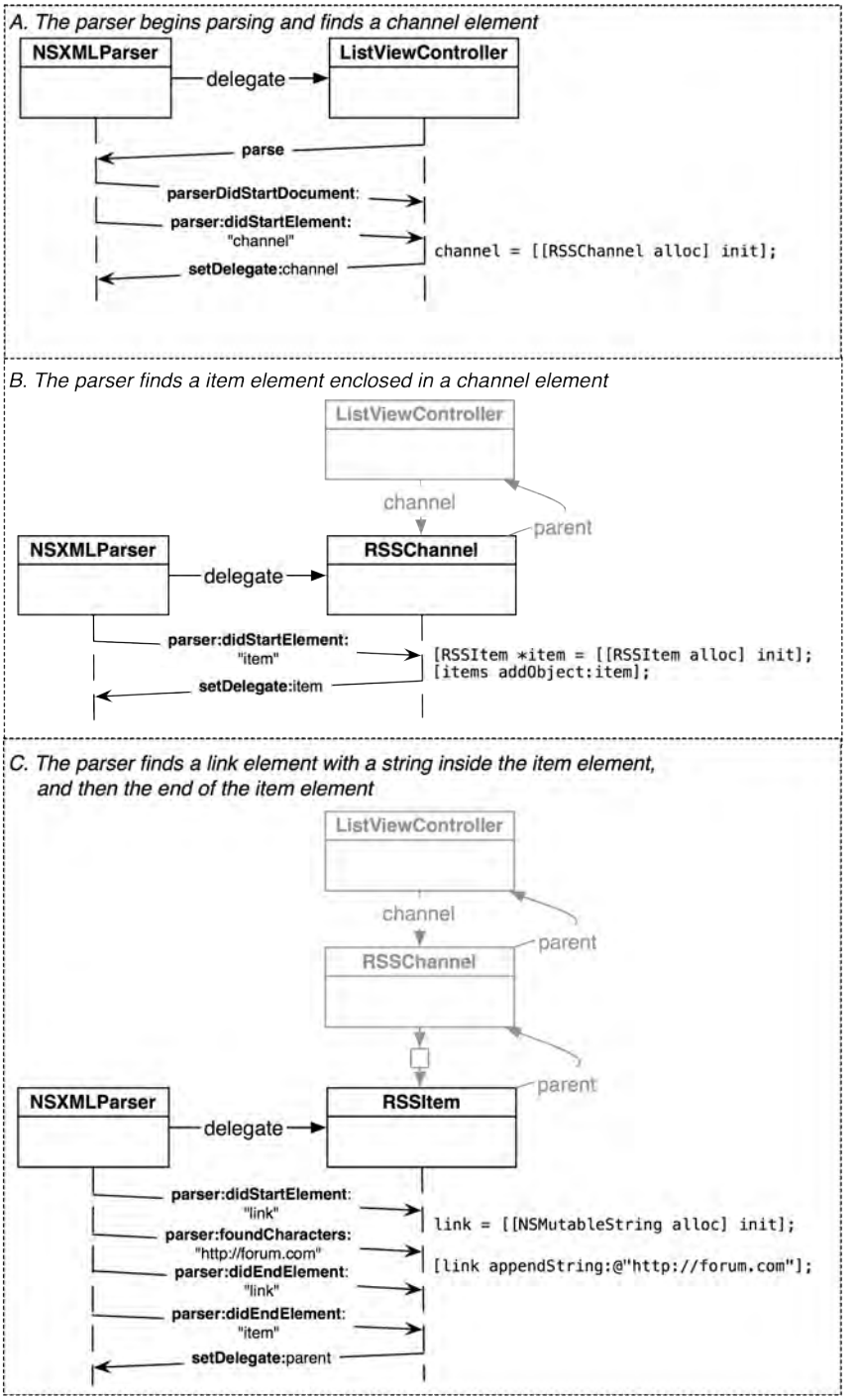
- When the parser finds a `title` or `link` element and it is currently inside a `item` element, set the appropriate property of the **`RSSItem`** instance.

This list doesn't seem too daunting. However, there is one issue that makes it difficult: the parser doesn't remember anything about what it has parsed. A parser may report, "I found a title element." Its next report is "Now I've found the string inside an element." At this point, if you asked the parser which element that string was inside, it couldn't tell you. It only knows about the string it just found. This leaves the burden of tracking state on the parser's delegate, and maintaining the state for an entire tree of objects in a single object is cumbersome.

Instead, you will spread out the logic for handling messages from the parser among the classes involved. If the last found element is a `channel`, then that instance of **`RSSChannel`** will be responsible for handling what the parser spits out next. The same goes for **`RSSItem`**; it will be responsible for grabbing its own `title` and `link` strings.

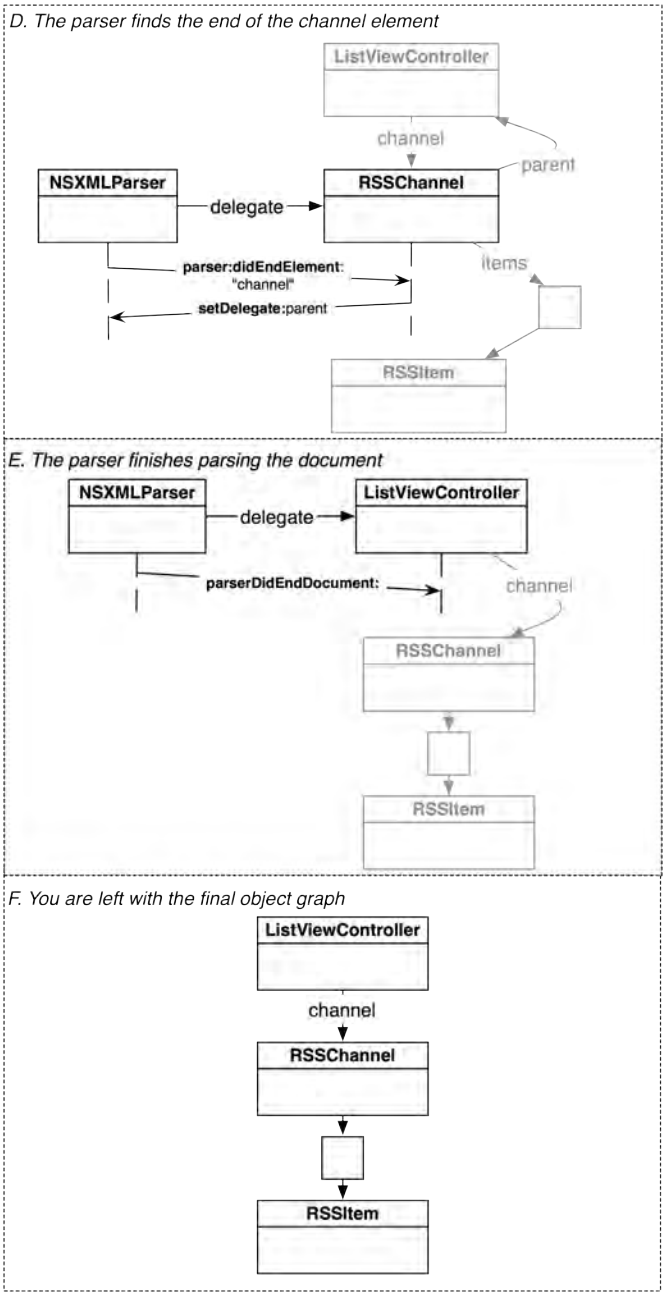
"But the parser can only have one delegate," you say. And you're right; it can only have one delegate *at a time*. We can change the delegate of an **`NSXMLParser`** whenever we please, and the parser will keep chugging through the XML and sending messages to its current delegate. The flow of the parser and the related objects is shown in Figure 25.7.

Figure 25.7 Flow diagram of XML being parsed into a tree, creating the tree



When the parser finds the end of an element, it tells its delegate. If the delegate is the object that represents that element, that object returns control to the previous delegate (Figure 25.8).

Figure 25.8 Flow diagram of XML being parsed into a tree, back up the tree



Now that we have a plan, let's get to work. Create a new **NSObject** subclass named **RSSChannel**. A channel object needs to hold some metadata, an array of **RSSItem** instances, and a pointer back to the previous parser delegate. In RSSChannel.h, add these properties:

```
@interface RSSChannel : NSObject

@property (nonatomic, weak) id parentParserDelegate;

@property (nonatomic, strong) NSString *title;
@property (nonatomic, strong) NSString *infoString;
@property (nonatomic, readonly, strong) NSMutableArray *items;

@end
```

In RSSChannel.m, synthesize the properties and override **init**.

```
@implementation RSSChannel
@synthesize items, title, infoString, parentParserDelegate;

- (id)init
{
    self = [super init];

    if (self) {
        // Create the container for the RSSItems this channel has;
        // we'll create the RSSItem class shortly.
        items = [[NSMutableArray alloc] init];
    }

    return self;
}

@end
```

Back in ListViewController.h, add an instance variable to hold an **RSSChannel** object and have the class conform to the NSXMLParserDelegate protocol.

```
// a forward declaration; we'll import the header in the .m
@class RSSChannel;

@interface ListViewController : UITableViewController <NSXMLParserDelegate>
{
    NSURLConnection *connection;
    NSMutableData *xmlData;

    RSSChannel *channel;
```

In ListViewController.m, implement an NSXMLParserDelegate method to catch the start of a channel element. Also, at the top of the file, import the header for **RSSChannel**.

```
#import "RSSChannel.h"

@implementation ListViewController

- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
       namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qualifiedName
         attributes:(NSDictionary *)attributeDict
```

```
{
    NSLog(@"%@ found a %@ element", self, elementName);
    if ([elementName isEqual:@"channel"]) {

        // If the parser saw a channel, create new instance, store in our ivar
        channel = [[RSSChannel alloc] init];

        // Give the channel object a pointer back to ourselves for later
        [channel setParentParserDelegate:self];

        // Set the parser's delegate to the channel object
        // There will be a warning here, ignore it warning for now
        [parser setDelegate:channel];
    }
}
```

Build and run the application. You should see a log message that the channel was found. If you don't see this message, double-check that the URL you typed in **`fetchEntries`** is correct.

Now that the channel is sometimes the parser's delegate, it needs to implement `NSXMLParserDelegate` methods to handle the XML. The **`RSSChannel`** instance will catch the metadata it cares about along with any `item` elements.

The channel is interested in the `title` and `description` metadata elements, and you will store those strings that the parser finds in the appropriate instance variables. When the start of one of these elements is found, an **`NSMutableString`** instance will be created. When the parser finds a string, that string will be concatenated to the mutable string.

In `RSSChannel.h`, declare that the class conforms to `NSXMLParserDelegate` and add an instance variable for the mutable string.

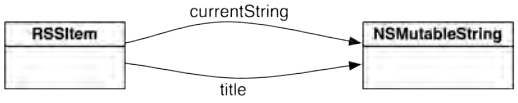
```
@interface RSSChannel : NSObject <NSXMLParserDelegate>
{
    NSMutableString *currentString;
}
```

In `RSSChannel.m`, implement one of the `NSXMLParserDelegate` methods to catch the metadata.

```
- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
       namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qualifiedName
         attributes:(NSDictionary *)attributeDict
{
    NSLog(@"\t%@ found a %@ element", self, elementName);

    if ([elementName isEqual:@"title"]) {
        currentString = [[NSMutableString alloc] init];
        [self setTitle:currentString];
    }
    else if ([elementName isEqual:@"description"]) {
        currentString = [[NSMutableString alloc] init];
        [self setInfoString:currentString];
    }
}
```

Note that `currentString` points at the same object as the appropriate instance variable – either `title` or `infoString` (Figure 25.9).

Figure 25.9 Two variables pointing at the same object



This means that when you append characters to the `currentString`, you are also appending them to the `title` or to the `infoString`.

In `RSSChannel.m`, implement the **`parser:foundCharacters:`** method.

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)str
{
    [currentString appendString:str];
}
```

When the parser finds the end of the `channel` element, the channel object will return control of the parser to the **`ListViewController`**. Implement this method in `RSSChannel.m`.

```
- (void)parser:(NSXMLParser *)parser
 didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
{
    // If we were in an element that we were collecting the string for,
    // this appropriately releases our hold on it and the permanent ivar keeps
    // ownership of it. If we weren't parsing such an element, currentString
    // is nil already.
    currentString = nil;

    // If the element that ended was the channel, give up control to
    // who gave us control in the first place
    if ([elementName isEqual:@"channel"])
        [parser setDelegate:parentParserDelegate];
}
```

Let's double-check our work so far. In `ListViewController.m`, add the following log statement to **`connectionDidFinishLoading:`**.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)conn
{
    NSXMLParser *parser = [[NSXMLParser alloc] initWithData:xmlData];
    [parser setDelegate:self];
    [parser parse];

    xmlData = nil;

    connection = nil;

    [[self tableView] reloadData];
    NSLog(@"%@\n %@\n %@\n", channel, [channel title], [channel infoString]);
}
```

Build and run the application. At the end of the console, you should see the log statement with valid values for the three strings. The data isn't correct yet, but there should still be three blocks of text separated by a new line.

Now we will need to write the code for the leaves of the object tree represented by the XML – the **RSSItem** instances. Create a new **NSObject** subclass. Name it **RSSItem**. In RSSItem.h, give the item instance variables for its metadata and for parsing.

```
@interface RSSItem : NSObject <NSXMLParserDelegate>
{
    NSMutableString *currentString;
}
@property (nonatomic, weak) id parentParserDelegate;

@property (nonatomic, strong) NSString *title;
@property (nonatomic, strong) NSString *link;


@end
```

In RSSItem.m, synthesize these properties and set up the parsing code similar to what you did for **RSSChannel**.

```
@implementation RSSItem

@synthesize title, link, parentParserDelegate;

- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
       namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qualifiedName
         attributes:(NSDictionary *)attributeDict
{
    NSLog(@"\t\t%@ found a %@ element", self, elementName);

    if ([elementName isEqual:@"title"]) {
        currentString = [[NSMutableString alloc] init];
        [self setTitle:currentString];
    }
    else if ([elementName isEqual:@"link"]) {
        currentString = [[NSMutableString alloc] init];
        [self setLink:currentString];
    }
}

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)str
{
    [currentString appendString:str];
}

- (void)parser:(NSXMLParser *)parser
 didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
{
    currentString = nil;

    if ([elementName isEqual:@"item"])
        [parser setDelegate:parentParserDelegate];
}

@end
```

Build the application to check for syntax errors.

In `RSSChannel.m`, put **`RSSItem`** into the object tree. At the top of this file, make sure to import the header for **`RSSItem`**.

```
#import "RSSItem.h"

@implementation RSSChannel

- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
       namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qualifiedName
         attributes:(NSDictionary *)attributeDict
{
    if ([elementName isEqual:@"title"]) {
        currentString = [[NSMutableString alloc] init];
        [self setTitle:currentString];
    }
    else if ([elementName isEqual:@"description"]) {
        currentString = [[NSMutableString alloc] init];
        [self setInfoString:currentString];
    }
    else if ([elementName isEqual:@"item"]) {
        // When we find an item, create an instance of RSSItem
        RSSItem *entry = [[RSSItem alloc] init];

        // Set up its parent as ourselves so we can regain control of the parser
        [entry setParentParserDelegate:self];

        // Turn the parser to the RSSItem
        [parser setDelegate:entry];

        // Add the item to our array and release our hold on it
        [items addObject:entry];
    }
}
```

Build and run the application. You should see log statements in the console that indicate the tree is being built. The last log statement in the console should have the correct data for the channel object, which looks something like this:

```
<RSSChannel: 0x4e18f80>
forums.bignerdranch.com
Books written by Big Nerd Ranch
```

Finally, we need to connect the channel and its items to the table view. In `ListViewController.m`, import the header file for **`RSSItem`** and fill out the two data source methods you temporarily implemented earlier.

```
#import "RSSItem.h"

@implementation ListViewController

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 0;
```

```
    return [[channel items] count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;

    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:@"UITableViewCell"];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                      reuseIdentifier:@"UITableViewCell"];
    }
    RSSItem *item = [[channel items] objectAtIndex:[indexPath row]];
    [[cell textLabel] setText:[item title]];

    return cell;
}
```

Build and run the application. You should now see the titles of the last 20 posts in a table view. Also, take a good look at the console to see the flow of the parser and how the delegate role is passed around.

A quick tip on logging

In this application, you log a lot of data to the console. It would be easy to miss an important log statement. One way to catch important statements is to prefix the most important ones with an easily searchable token (like `xxx`), but that's a quick-and-dirty fix.

A more elegant and useful option is to define a preprocessor macro that you can use to categorize your log statements. For example, in Nerdfeed, you can generate a ton of log statements for checking the input and output of your web service requests. You can also generate a ton of log statements for checking the logic in the rest of the application. When you are debugging Nerdfeed, it would be helpful to separate the web service-related statements from the others so that you can turn them on or off as needed.

While there are many ways to do this, here is the simplest one:

```
#define WSLog(...) NSLog(__VA_ARGS__)
```

This statement tells the compiler, "When you come across **`WSLog`**, see **`NSLog`**." Save this statement in its own `.h` file and import it into your precompiled header (`Nerdfeed_Prefix.pch`). Then, when you want to log a web service-related statement in your code, use **`WSLog`** instead of **`NSLog`**, passing the exact same arguments. For example, in `ListViewController.m`, you could change the log statement in **`connectionDidFinishLoading:`** to the following:

```
WSLog(@"%@\n %@\n %@\n", channel, [channel title], [channel infoString]);
```

As long as **`WSLog`** is defined to **`NSLog`**, nothing will change. You will still see all of your log statements in the console. When you want to turn off the web service-related statements to concentrate on other areas, simply re-define **`WSLog`** to the following in its header file:

```
#define WSLog(...) do {} while(0)
```

Now any **`WSLog`** calls will be invisible to the compiler, so they will not appear in the console to distract you from your non-web service debugging. (Defining `WSLog` in this way means it will be optimized out by the compiler.)

UIWebView

In addition to its title, an **RSSItem** also keeps a link that points to the web page where the post lives. It would be neat if Nerdfeed could open up Safari and navigate to that page. It would be even neater if Nerdfeed could render that web page without having to leave Nerdfeed to open Safari. Good news it can using the class **UIWebView**.

Instances of **UIWebView** render web content. In fact, the Safari application on your device uses a **UIWebView** to render its web content. In this part of the chapter, you will create a view controller whose view is an instance of **UIWebView**. When one of the items is selected from the table view of **RSSItem**s, you will push the web view's controller onto the navigation stack and have it load the link stored in the **RSSItem**.

Create a new **NSObject** subclass and name it **WebViewController**. In `WebViewController.h`, add a property (but not an instance variable) and change the superclass to **UIViewController**:

```
@interface WebViewController : NSObject
@interface WebViewController : UIViewController

@property (nonatomic, readonly) UIWebView *webView;

@end
```

In `WebViewController.m`, override **loadView** to create an instance of **UIWebView** as the view of this view controller. Also, implement the method **webView** to return that view.

```
@implementation WebViewController

- (void)loadView
{
    // Create an instance of UIWebView as large as the screen
    CGRect screenFrame = [[UIScreen mainScreen] applicationFrame];
    UIWebView *wv = [[UIWebView alloc] initWithFrame:screenFrame];
    // Tell web view to scale web content to fit within bounds of webview
    [wv setScalesPageToFit:YES];

    [self setView:wv];
}

- (UIWebView *)webView
{
    return (UIWebView *)[self view];
}
```

In `ListViewController.h`, add a new property to **ListViewController**.

```
@class WebViewController;

@interface ListViewController : UITableViewController <NSXMLParserDelegate>
{
    NSURLConnection *connection;
    NSMutableData *xmlData;

    RSSChannel *channel;
}
@property (nonatomic, strong) WebViewController *webViewController;
```

```
- (void)fetchEntries;
@end
```

In `ListViewController.m`, import the header file and synthesize the property.

```
#import "WebViewController.h"

@implementation ListViewController
@synthesize webViewController;
```

In `NerdfeedAppDelegate.m`, import the header for **`WebViewController`**, create an instance of **`WebViewController`**, and set it as the `webViewController` of the **`ListViewController`**.

```
#import "WebViewController.h"

@implementation NerdfeedAppDelegate

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

    UINavigationController *masterNav =
        [[UINavigationController alloc] initWithRootViewController:lvc];


    WebViewController *wvc = [[WebViewController alloc] init];
    [lvc setWebViewController:wvc];

    [[self window] setRootViewController:masterNav];

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

(Note that we are instantiating the **`WebViewController`** in the application delegate in preparation for the next chapter where we will use a **`UISplitViewController`** to present view controllers on the iPad.)

When the user taps on a row in the table view, we want the **`WebViewController`** to be pushed onto the navigation stack and the link for the selected **`RSSItem`** to be loaded in its web view. To have a web view load a web page, you send it the message **`loadRequest:`**. The argument is an instance of **`NSURLRequest`** that contains the URL you wish to navigate to. In `ListViewController.m`, implement the following table view delegate method:

```
- (void)tableView:(UITableView *)tableView
                didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Push the web view controller onto the navigation stack - this implicitly
    // creates the web view controller's view the first time through
    [[self navigationController] pushViewController:webViewController animated:YES];

    // Grab the selected item
    RSSItem *entry = [[channel items] objectAtIndex:[indexPath row]];
```

```
    // Construct a URL with the link string of the item
    NSURL *url = [NSURL URLWithString:[entry link]];

    // Construct a requst object with that URL
    NSURLRequest *req = [NSURLRequest requestWithURL:url];

    // Load the request into the web view
    [[webViewController webView] loadRequest:req];

    // Set the title of the web view controller's navigation item
    [[webViewController navigationItem] setTitle:[entry title]];
}
```

Build and run the application. You should be able to select one of the posts, and it should take you to a new view controller that displays the web page for that post.

For the More Curious: NSXMLParser

`NSXMLParser` is the built-in XML parser in the iOS SDK. While there are plenty of parsers you can pick up on the Internet, adding a third party dependency is sometimes difficult. Many developers, seeing that **`NSXMLParser`** is not a tree-based parser (it doesn't create an object graph out of the box), go searching for an alternative parser. However, in this chapter, you've learned how to make **`NSXMLParser`** into a tree-based parser.

To parse simple XML, all you need are the three delegate methods used in this chapter. More complex XML has element attributes, namespaces, CDATA, and a slew of other items that need to be handled. **`NSXMLParser`** can handle these, too. The `NSXMLParserDelegate` protocol includes many more methods that handle nearly anything XML can throw at you. There are also arguments to the methods you have already used that can handle more complex XML. For example, in **`parser:didStartElement:namespaceURI:qualifiedName:attributes:`**, we only used the first two arguments. For the other arguments, consider the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<container version="2.0" xmlns:foo="BNR">
    <foo:item attribute1="one" attribute2="two"></item>
</container>
```

When the `foo:item` element is encountered by the parser, the values for the parameters to the delegate method are as follows:

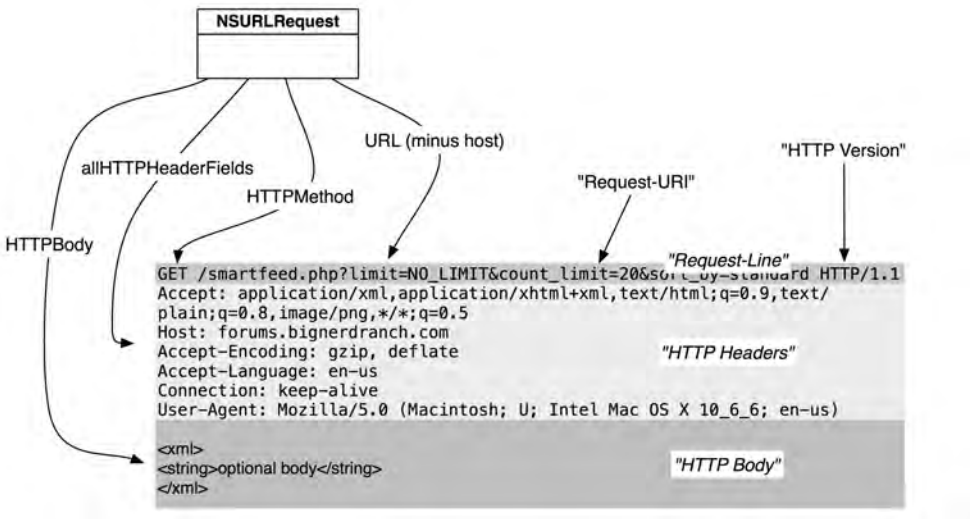
- The `element` is "item." The namespace is ignored, and the name of the element is kept.
- The `namespaceURI` is "BNR." The element's name is `item`, and it is in the `foo` namespace, which has a value of "BNR."
- The `qualifiedName` is "foo:item."
- `attributes` is a dictionary that contains two keys, "attribute1" and "attribute2." Their values are "one" and "two," respectively.

One thing **`NSXMLParser`** can't do is resolve XPaths. You have to use another library to handle this. (For more information, check out the Tree-Based XML Programming Guide in the Apple documentation.)

For the More Curious: The Request Body

When **`NSURLConnection`** talks to a web server, it uses the HTTP protocol. This protocol says that any data you send or receive must follow the HTTP specification. The actual data transferred to the server in this chapter is shown in Figure 25.10.

Figure 25.10 HTTP Request Format



`NSURLRequest` has a number of methods that allow you to specify a piece of the request and then properly format it for you.

Any service request has three parts: a request-line, the HTTP headers, and the HTTP body, which is optional. The request-line (which Apple calls a status line) is the first line of the request and tells the server what the client is trying to do. In this request, the client is trying to `GET` the resource at `smartfeed.php?limit=1_DAY&etc`. (It also specifies the HTTP specification version that the data is in.)

The command `GET` is an HTTP method. While there are a number of supported HTTP methods, you typically only see `GET` and `POST`. The default of **`NSURLRequest`**, `GET`, indicates that the client wants something *from* the server. The thing that it wants is called the Request-URI (`smartfeed.php?limit=1_DAY&etc`).

In the early days of the web, the Request-URI would be the path of a file on the server. For example, the request `http://www.website.com/index.html` would return the file `index.html`, and your browser would render that file in a window. Today, we also use the Request-URI to specify a service that the server implements. For example, in this chapter, you accessed the `smartfeed.php` service, supplied parameters to it, and were returned an XML document. You are still `GET`ting something, but the server is more clever in interpreting what you are asking for.

In addition to getting things from a server, you can send it information. For example, many web servers allow you to upload photos. A client application would pass the image data to the server through a service request. In this situation, you use the HTTP method `POST`, which indicates to the server that you are including the optional HTTP body. The body of a request is data you can include with the request – typically XML, JSON, or Base-64 encoded data.

When the request has a body, it must also have the `Content-Length` header. Handily enough, **`NSURLRequest`** will compute the size of the body and add this header for you.

```
NSURL *someURL = [NSURL URLWithString:@"http://www.photos.com/upload"];
UIImage *image = [self profilePicture];
NSData *data = UIImagePNGRepresentation(image);

NSMutableURLRequest *req =
    [NSMutableURLRequest requestWithURL:someURL
                            cachePolicy:NSURLRequestReloadIgnoringCacheData
                        timeoutInterval:90];

// This adds the HTTP body data and automatically sets the Content-Length header
[req setHTTPBody:data];

// This changes the HTTP Method in the request-line
[req setHTTPMethod:@"POST"];

// If you wanted to set the Content-Length programmatically...
[req setValue:[NSString stringWithFormat:@"%d", [data length]]
    forHTTPHeaderField:@"Content-Length"];
```

For the More Curious: Credentials

When you try to access a web service, it will sometimes respond with an *authentication challenge*, which means "Who the heck are you?" You then need to send a username and password (a *credential*) before the server will send its genuine response.

There are objects that represent these ideas. When the challenge is received, your connection delegate is sent a message that includes an instance of **`NSURLAuthenticationChallenge`**. The sender of that challenge conforms to the `NSURLAuthenticationChallengeSender` protocol. If you want to continue to get the data, you send back an instance of **`NSURLCredential`**, which typically looks something like this:

```
- (void)connection:(NSURLConnection *)conn
 didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{
    // Have I already failed at least once?
    if ([challenge previousFailureCount] > 0) {

        // Why did I fail?
        NSError *failure = [challenge error];
        NSLog(@"Can't authenticate: %@", [error localizedDescription]);

        // Give up
        [[challenge sender] cancelAuthenticationChallenge:challenge];
        return;
    }

    // Create a credential
    NSURLCredential *newCred =
            [NSURLCredential credentialWithUser:@"sid"
                                       password:@"MomIsCool"
                                    persistence:NSURLCredentialPersistenceNone];
```

```
    // Supply the credential to the sender of the challenge
    [[challenge sender] useCredential:newCred
           forAuthenticationChallenge:challenge];
}
```

If you are dealing with a more secure and sophisticated web service, it may want a certificate (or certificates) to confirm your identity. Most, however, will just want a username and a password.

Credentials can have persistence. There are three possibilities:

- `NSURLCredentialPersistenceNone` says to the URL loading system, "Forget this credential as soon as you use it."
- `NSURLCredentialPersistenceForSession` says to the URL loading system, "Forget this credential when this application terminates."
- `NSURLCredentialPersistencePermanent` says to the URL loading system, "Put this credential in my keychain so that other applications can use it."

Bronze Challenge: More Data

Create a **`UITableViewCell`** subclass that has three labels. Parse the author and category elements into the **`RSSItem`** and display the title, author, and category for each row.

Silver Challenge: More UIWebView

A **`UIWebView`** keeps its own history. You can send the messages **`goBack`** and **`goForward`** to a web view, and it will traverse through that history. Create a **`UIToolbar`** instance and add it to the **`WebViewController`**'s view hierarchy. This toolbar should have back and forward buttons that will let the web view move through its history. Bonus: use two other properties of **`UIWebView`** to enable and disable the toolbar items.

CHAPTER 2
Your First Program

OBJECTIVE-C PROGRAMMING: THE BIG NERD RANCH GUIDE, by Aaron Hillegass

TABLE OF CONTENTS

AVAILABLE FORMATS

- 9780321706287 Book
- 9780132983150 eBook
- 9780321706522 Safari Books Online

Objective-C Programming

The Big Nerd Ranch Guide

Aaron Hillegass

Big nerd ranch

2

Your First Program

Now that we know how this book is organized, it's time to see how programming for the Mac and for iPhone and iPad works. To do that, you will

- install Apple's Developer Tools
- create a simple project using those tools
- explore how these tools are used to make sure our project works

At the end of this chapter, you will have successfully written your first program for the Mac.

Installing Apple's developer tools

To write applications for Mac OS X (the Macintosh) or iOS (the iPhone and iPad), you will be using Apple's developer tools. You can download these tools from `http://developer.apple.com/` or purchase them from the Mac App Store.

After you've installed the tools, find the `/Developer` folder at the root level of your hard drive. This folder contains what you need to develop applications for Mac OS X desktops and iOS mobile devices.

Our work in this book is going to be conducted almost entirely with one application – Xcode, which is found in the `/Developer/Applications` folder. (It is a good idea to drag the Xcode icon over to the dock; you'll be using it an awful lot.)

Getting started with Xcode

Xcode is Apple's *Integrated Development Environment*. That means that everything you need to write, build, and run new applications is in Xcode.

A note on terminology: anything that is executable on a computer we call a *program*. Some programs have graphical user interfaces; we will call these *applications*.

Some programs have no graphical user interface and run for days in the background; we call these *daemons*. Daemons sound scary, but they aren't. You probably have about 60 daemons running on your Mac right now. They are waiting around, hoping to be useful. For example, one of the daemons running on your system is called `pboard`. When you do a copy and paste, the `pboard` daemon holds onto the data that you are copying.

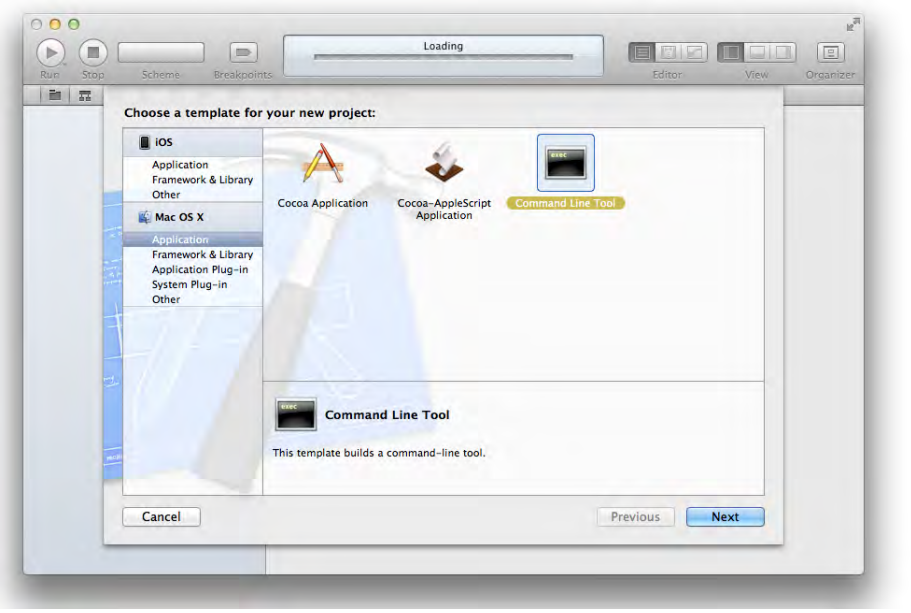
Some programs have no graphical user interface and run for a short time in the terminal; we call these *command-line tools*. In this book, you will be writing mostly command-line tools to focus on programming essentials without the distraction of creating and managing a user interface.

Now we're going to create a simple command-line tool using Xcode so you can see how it all works.

When you write a program, you create and edit a set of files. Xcode keeps track of those files in a *project*. Launch Xcode. From the File menu, choose New and then New Project….

To help you get started, Xcode suggests a number of possible project templates. You choose a template depending on what sort of program you want to write. In the lefthand column, select Application from the Mac OS X section. Then choose Command Line Tool from the choices that appear to the right.

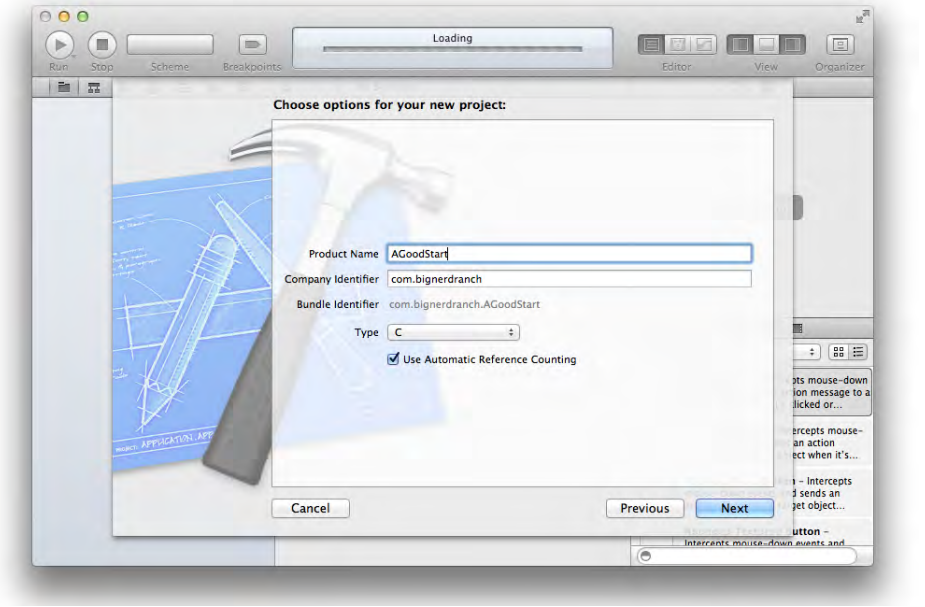
Figure 2.1 Choosing a template



Press the Next button.

Name your new project AGoodStart. The company identifier won't matter for our exercises in this book, but you have to enter one here to continue. You can use BigNerdRanch or another name. From the Type pop-up menu, choose C because you will write this program in C. Finally, make sure the checkbox labeled Use Automatic Reference Counting is checked.

Figure 2.2 Choose options



Press the Next button.

Now choose the folder in which your project directory will be created. You won't need a repository for version control, so you can uncheck that box. Finally, click the Create button.

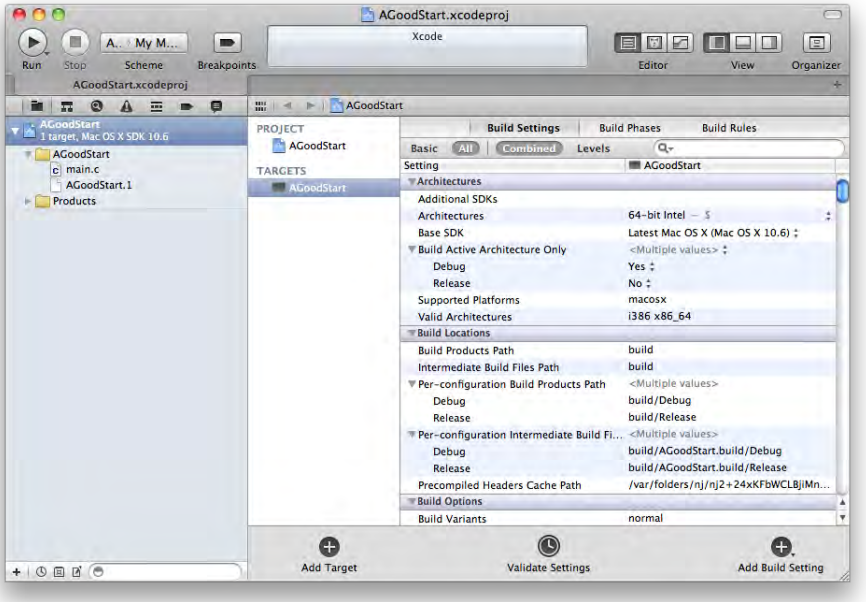
You'll be creating this same type of project for the next several chapters. In the future, I'll just say, "Create a new C Command Line Tool named *program-name-here*" to get you to follow this same sequence.

(Why C? Remember, Objective-C is built on top of the C programming language. You'll need to have an understanding of parts of C before we can get to the particulars of Objective-C.)

Where do I start writing code?

After creating your project, you'll be greeted by a window that shows how AGoodStart will be produced.

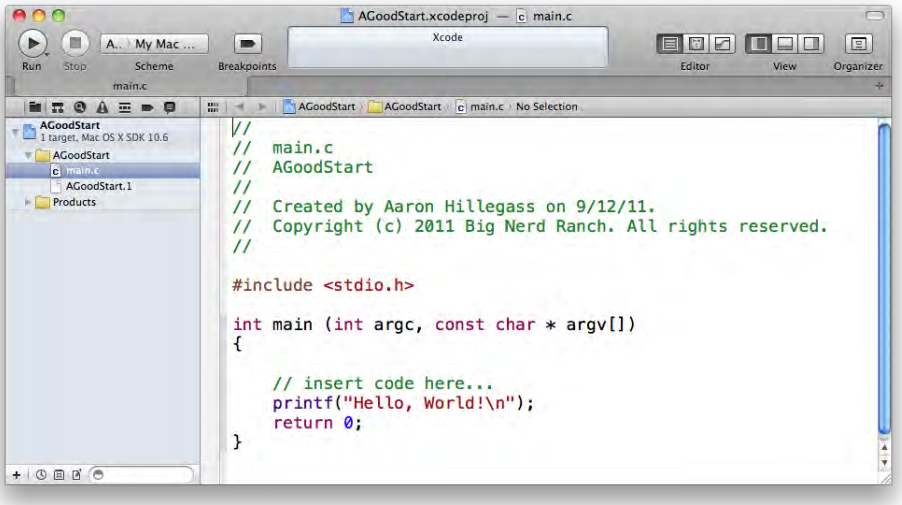
Figure 2.3 First view of the AGoodStart project



This window includes details like which versions of Mac OS X can run your application, the configurations to use when compiling the code that you write, and any localizations that have been applied to your project. But let's ignore those details for now and find a simple starting point to get to work.

Near the top of the lefthand panel, find a file called `main.c` and click on it. (If you don't see `main.c`, click the triangle next to the folder labeled AGoodStart to reveal its contents.)

Figure 2.4 Finding main.c in the AGoodStart group



Notice that our original view with the production details changes to show the contents of `main.c`. The `main.c` file contains a function called **`main`**.

A *function* is a list of instructions for the computer to execute, and every function has a name. In a C or Objective-C program, **`main`** is the function that is called when a program first starts.

```
#include <stdio.h>

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

    // insert code here...
    printf("Hello, World!\n");
    return 0;
}
```

In this function, you'll find the two kinds of information you write in a program: code and comments.

- Code is the set of instructions that tell the computer to do something.
- Comments are ignored by the computer, but we programmers use them to document code we've written. The more difficult the programming problem you are trying to solve, the more comments will help document how you solved the problem. That becomes especially important when you return to your work months later, look at code you forgot to comment, and think, "I'm sure this solution is brilliant, but I have absolutely no memory of how it works."

In C and Objective-C, there are two ways to distinguish comments from code:

- If you put `//` at the beginning of a line of code, everything from those forward slashes to the end of that line is considered a comment. You can see this used in Apple's "insert code here..." comment.
- If you have more extensive remarks in mind, you can use `/*` and `*/` to mark the beginning and end of comments that span more than one line.

These rules for marking comments are part of the *syntax* of C. Syntax is the set of rules that governs how code must be written in a given programming language. These rules are extremely specific, and if you fail to follow them, your program won't work.

While the syntax regarding comments is fairly simple, the syntax of code can vary widely depending on what the code does and how it does it. But there's one feature that remains consistent: every *statement* ends in a semicolon. (We'll see examples of code statements in just a moment.) If you forget a semicolon, you will have made a syntax error, and your program won't work.

Fortunately, Xcode has ways to warn you of these kinds of errors. In fact, one of the first challenges you will face as a programmer is interpreting what Xcode tells you when something goes wrong and then fixing your errors. You'll get to see some of Xcode's responses to common syntax errors as we go through the book.

Let's make some changes to `main.c`. First, we need to make some space. Find the curly braces (`{` and `}`) that mark the beginning and the end of the **`main`** function. Then delete everything in between them.

Now update `main.c` to look like the code below. You'll add a comment, two lines of code, and another comment to the **`main`** function. For now, don't worry if you don't understand what you are typing. The idea is to get started. You have an entire book ahead to learn what it all means.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Print the beginning of the novel
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");
    /* Is that actually any good?
       Maybe it needs a rewrite. */

    return 0;
}
```

(Notice that the new code you need to type in is shown in a bold font. The code that isn't bold is code that is already in place. That's a convention we'll use for the rest of the book.)

As you type, you may notice that Xcode tries to make helpful suggestions. This feature is called *code completion*, and it is very handy. You may want to ignore it right now and focus on typing things in all yourself. But as you continue through the book, start playing with code completion and how it can help you write code more conveniently and more accurately. You can see and set the different options for code completion in Xcode's preferences, which are accessible from the Xcode menu.

In addition, keep an eye on the font color. Xcode uses different font colors to make it easy to identify comments and different parts of your code. (For example, comments are green.) This comes in handy, too: after a while of working with Xcode, you begin to instinctively notice when the colors don't look right. Often, this is a clue that there is a syntax error in what you've written (like a forgotten semicolon). And the sooner you know that you've made a syntax error, the easier it is to find and fix it.

These color differences are just one way in which Xcode lets you know when you (may) have done something wrong.

How do I run my program?

When the contents of your `main.c` file match what you see above, it's time to run your program and see what it does. This is a two-step process. Xcode *builds* your program and then *runs* it. When building your program, Xcode prepares your code to run. This includes checking for syntax and other kinds of errors.

Look again at the lefthand area of the Xcode window. This area is called the navigator area. At the top of the navigator area is a series of buttons. You are currently viewing the *project navigator*, which shows you the files in your project. The project navigator's icon is .

Now find and click the button to reveal the *log navigator*. The *log* is Xcode's way of communicating with you when it is building and running your program.

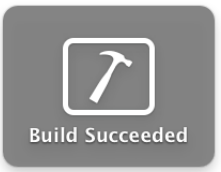
You can also use the log for your own purposes. For instance, the line in your code that reads

```
printf("It was the best of times.\n");
```

is an instruction to display the words "It was the best of times." in the log.

Since you haven't built and run your program yet, there isn't anything in the log navigator. Let's fix that. In the upper lefthand corner of the project window, find the button that looks suspiciously like the Play button in iTunes or on a DVD player. If you leave your cursor over that button, you'll see a tool tip that says Build and then run the current scheme. That is Xcode-speak for "Press this button, and I will build and run your program."

If all goes well, you'll be rewarded with the following:



If not, you'll get this:

What do you do then? Carefully compare your code with the code in the book. Look for typos and missing semicolons. Xcode will highlight the lines it thinks are problematic. After you find the problem, click the Run button again. Repeat until you have a successful build.

(Don't get disheartened when you have failed builds with this code or with any code you write in the future. Making and fixing mistakes helps you understand what you're doing. In fact, it's actually better than lucking out and getting it right the first time.)

Once your build has succeeded, find the item at the top of the log navigator labeled Debug AGoodStart. Click this item to display the log from the most recent run of your program.

The log can be quite verbose. The important part is the Dickens quote at the end. That's your code being executed!

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Tue Jul  5 07:36:45 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
[Switching to process 2723 thread 0x0]
It was the best of times.
It was the worst of times.
```

(As I'm writing this, Apple is working on a new debugger called `LLDB`. Eventually it will replace `GDB`, the current debugger. If you aren't seeing all the `GDB` information, it means that `LLDB` is now Xcode's standard debugger. The future must be a terrific place; I envy you.)

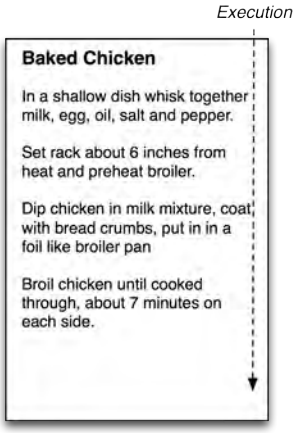
So what is a program?

Now that you've built and run your program, let's take a look inside. A program is a collection of functions. A function is a list of operations for the processor to execute. Every function has a name, and the function that you just wrote is named **`main`**. There was also another function – **`printf`**. You didn't write this function, but you did use it. (We'll find out where **`printf`** comes from in Chapter 5.)

To a programmer, writing a function is a lot like writing a recipe: "Stir a quart of water slowly until it boils. Then mix in a cup of flour. Serve while hot."

In the mid-1970's, Betty Crocker started selling a box containing a set of recipe cards. A recipe card is a pretty good metaphor for a function. Like a function, each card has a name and a set of instructions. The difference is that you execute a recipe, and the computer executes a function.

Figure 2.5 A recipe card named Baked Chicken



Betty Crocker’s cooking instructions are in English. In the first part of this book, your functions will be written in the C programming language. However, a computer processor expects its instructions in machine code. How do we get there?

When you write a program in C (which is relatively pleasant for you), the *compiler* converts your program’s functions into machine code (which is pleasant and efficient for the processor). The compiler is itself a program that is run by Xcode when you press the Run button. Compiling a program is the same as building a program, and we’ll use these terms interchangeably.

When you run a program, the compiled functions are copied from the hard drive into memory, and the function called **`main`** is executed by the processor. The **`main`** function usually calls other functions. For example, your **`main`** function called the **`printf`** function. (We’ll see more about how functions interact in Chapter 5.)

Don’t stop

At this point, you’ve probably dealt with several frustrations: installation problems, typos, and lots of new vocabulary. And maybe nothing you’ve done so far makes any sense. That is completely normal.

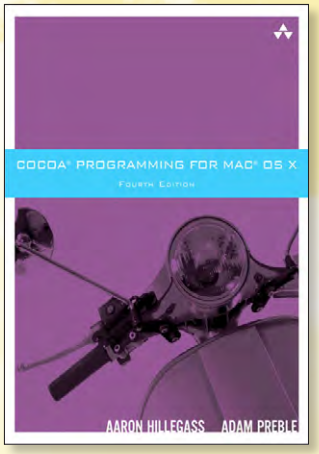
As I write this, my son Otto is six. Otto is baffled several times a day. He is constantly trying to absorb knowledge that doesn’t fit into his existing mental scaffolding. Bafflement happens so frequently, that it doesn’t really bother him. He never stops to wonder, “Why is this so confusing? Should I throw this book away?”

As we get older, we are baffled much less often – not because we know everything, but because we tend to steer away from things that leave us bewildered. For example, reading a book on history is quite pleasant because we get nuggets of knowledge that we can hang from our existing mental scaffolding. This is easy learning.

Learning a new language is an example of difficult learning. You know that there are millions of people who work in that language effortlessly, but it seems incredibly strange and awkward in your mouth. And when people speak it to you, you are often flummoxed.

Learning to program a computer is also difficult learning. You will be baffled from time to time – especially here at the beginning. This is fine. In fact, it’s kind of cool. It is a little like being six again.

Stick with this book; I promise that the bewilderment will cease before you get to the final page.



CHAPTER 29
Blocks

COCOA PROGRAMMING FOR MAC OS X, FOURTH EDITION, by Aaron Hillegass and Adam Preble

TABLE OF CONTENTS

AVAILABLE FORMATS

- 9780321774088 Book
- 9780132902212 eBook
- 9780132902199 Safari Books Online

Cocoa Programming for Mac OS X

4th Edition

AARON HILLEGASS & ADAM PREBLE

29
Blocks

Let's pretend that we're writing a zombie game. Specifically, we're working on the zombie AI code. We want a method on our **`Zombie`** object to find nearby brains. So we start with this:

```
@implementation Zombie

- (NSArray *)brainsForFlags:(NSInteger)flags
{
    return [[self game] allBrains];
}

@end
```

It's a good start, but it would be a lot more useful it if returned the brains in order of proximity to the zombie, that is, sorted by the distance between the zombie and the brain. The zombie is hungry, after all. **`NSArray`**'s **`sortedArrayUsingSelector:`** is usually a great first choice for sorting. It calls the given selector on the objects in the array in order to compare them with their neighbors. For example, **`NSString`** provides a **`compare:`** method. Thus, we can use it to sort an array of strings:

```
NSArray *sortedStrings =
      [theStrings sortedArrayUsingSelector:@selector(compare:)];
```

We might entertain adding a **`compareByDistanceToZombie:`** method to the Brain class. But how would it know which zombie it's comparing the distance to? The method **`sortedArrayUsingSelector:`** doesn't provide any way to pass contextual information to the sorting process.

`NSArray`'s **`sortedArrayUsingFunction:context:`** seems like a better choice. We can write a C function and tell **`sortedArrayUsingFunction:context:`** what function to use to compare the brains. Using it would look something like this:

```
NSInteger CmpBrainsByZombieDist(id a, id b, void *context)
{
     Brain *brainA = a;
     Brain *brainB = b;
     Zombie *zombie = (__bridge Zombie *)context;
     float distA = [zombie distanceToBrain:brainA];
     float distB = [zombie distanceToBrain:brainB];
     if (distA == distB) return NSOrderedSame;
     else if (distA < distB) return NSOrderedAscending;
     else return NSOrderedDescending;
}

- (NSArray *)brainsForFlags:(NSInteger)flags
{
```

```
    NSArray *allBrains = [[self game] allBrains];
    return [brains sortedArrayUsingFunction:CmpBrainsByZombieDist
                                    context:(__bridge void *)self];
}
```

The void pointer context argument is used to provide additional data to the comparison function; we use this to pass the pointer to the **`Zombie`** instance (`self`). Note that the `__bridge` casting is necessary to convert an object reference into a type out of ARC's control.

We've got a workable solution now. After some playtesting, however, we decide that we want our zombies to have a more varied palette. If `frenzy` mode is on, the zombies should seek out the brain with the highest IQ, no matter where it is in the game world. Now we need to supply multiple parameters to **`CmpBrainsByZombieDist`**, but it all needs to be passed in through a single void pointer argument.

Perhaps you are starting to see that this approach has a number of downsides. Maintaining C functions for custom sorting forces our input parameters to be awkwardly funneled through a void pointer. An **`NSDictionary`** or custom C struct will get the job done, but they add complexity. Additionally, the C function must be separate from the code that calls it, making it more challenging to efficiently maintain the code.

There is, however, an elegant solution to this problem: blocks. You can think of blocks as functions that can be passed around just like an object. Consider the following solution. The caret (^) in the following code is the start of the block:

```
- (NSArray *)brainsForFlags:(NSInteger)flags
{
    NSArray *allBrains = [[self game] allBrains];
    return [brains sortedArrayUsingComparator:^(id a, id b) {
        Brain *brainA = a;
        Brain *brainB = b;
        float criteriaA, criteriaB;
        if (flags & FrenzyMode)
        {
            criteriaA = [brainA iq];
            criteriaB = [brainB iq];
        }
        else
        {
            criteriaA = [self distanceToBrain:brainA];
            criteriaB = [self distanceToBrain:brainB];
        }
        if (criteriaA == criteriaB) return NSOrderedSame;
        else if (criteriaA < criteriaB) return NSOrderedAscending;
        else return NSOrderedDescending;
    }];
}
```

We'll get into the particulars of blocks syntax in the next section. Until then, let's look at some of the more interesting parts of this method. The **`sortedArrayUsingComparator:`** method takes a block as its only parameter. You'll notice that blocks look quite a bit like C functions. They have arguments and a body. Where they differ from C functions is that they do not have to be named (they are anonymous) and can be treated just like an expression. In fact, they are objects.

This particular block takes two arguments (`a` and `b`) and refers to variables that are defined outside the block (`self` and `flags`). This is one of the more useful aspects of blocks: They capture the value of

variables from the scope outside the block. There's generally no need to package up your variables to squeeze into the argument list: You can simply use the variables that are in scope.

Blocks provide an elegant way to address such problems as nontrivial sort criteria, as well as much more sophisticated problems. Next, we'll talk about the particulars of using blocks.

Block Syntax

Blocks enable the developer to create objects that encapsulate instructions, inline with the rest of their code, which capture the values of variables that are within scope. The resulting object can then be passed about and even copied, just like any other object.

The block syntax can be a little off-putting at first (it is not dissimilar from C function pointer syntax), but the benefits far outweigh the time you will spend getting used to it. Let's define a simple block:

```
int captured = 1; // Local variable 'captured'

int (^offsetter)(int) = ^(int x) { return x + captured; };
```

On the first line, we create a local variable: `captured`. Next, we declare a variable named `offsetter`, which is a block. Whenever we are creating or defining a block, we use the `^` operator. This block returns an integer and takes an integer as its only argument.

On the right side of the equal sign, we define the block (note the `^`, again). This part looks a lot like a C function definition. We specify that the integer parameter will be called `x`, and then we provide the body of the block in braces. An annotated version is shown in Figure 29.1

Figure 29.1 Anatomy of a Block

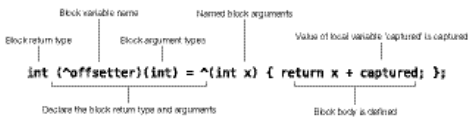


Figure 29.1 Anatomy of a Block

Aside from the magic of creating a block of code on the stack like this, two interesting things are happening here: First, our block definition does not specify the return type of the block. The compiler is smart enough to figure it out from the return statement. Second, we refer to the variable `captured` inside the block. A blocks programmer would say that we are capturing the value of `captured` in the block. This is a very powerful feature of blocks.

How do we call a block? As it turns out, it looks quite a bit like a C function call:

```
int answer = offsetter(2);
```

Note that `answer` is now 3 (2 + 1 = 3). What if we change the value of `captured` and call the block again?

```
captured = 64;
answer = offsetter(2);
```

The result is the same; `answer` is still 3 because the *value* of `captured` was captured when we defined the block. We cannot change `captured` from outside the block, and as it happens, we can't change it from inside the block either. To do that, we need the `__block` type specifier.

By default, captured values are considered `const` within the block. If you need to modify a captured value from inside a block, you can add the `__block` type specifier:

```
__block BOOL modifiable = YES;
```

However, when a variable is marked with `__block`, the compiler treats it as what is essentially a global variable. Any block that was created with it in scope can change its value. Because it does come with some performance overhead, `__block` is not the default. Typically, it is more useful to capture only the value of a variable.

Memory and Objects within Blocks

When a block is defined, it is created on the stack. When the method or function it was defined in exits, the block is removed from memory along with all the other local stack variables. Sometimes, this is fine; we may wish to use the block only for the lifetime of that particular method call, as in our earlier brain-sorting example. The block is not used after the method returns.

Other times, however, we want the block to live on well after the method returns. Because the block is created on the stack, we must copy it in order to make sure that it is not deallocated with the current stack frame. For this reason, we recommend that you copy blocks when assigning them to instance variables:

```
@property (nonatomic, copy) int (^arithmeticOperationBlock)(int);
```

Just like we can capture scalar values within blocks, we can also capture pointers to objects. When a pointer to an Objective-C object is captured by a block, it is retained (a strong reference is formed). Any objects retained by the block are released when the block goes out of scope or is deallocated:

```
NSMutableArray *array;
array = [NSMutableArray array]; // retain count of 1, autoreleased

void (^simpleBlock)() = ^{
    [array addObject:@"Q"];     // array pointer captured, retained
};

simpleBlock();

return; // simpleBlock is popped from stack, releases array
```

Note that variables with the `__block` specifier will not be retained by the block. This can be useful in preventing strong reference cycles. Note that this is still the case under ARC; however, ARC still considers a `__block` pointer a strong reference and thus retains it, unless you mark the variable `__weak`.

Consider the following code:

```
controller = [[MyController alloc] init];
controller.block = ^{
    [controller doSomething];
};
```

This code creates a strong reference cycle. Do you see it?

`MyController` holds a strong reference to `block`. The block, however, holds a strong reference to the instance of **`MyController`**! The simplest approach is to use a temporary weak reference variable, since child objects (the block) should have only weak references to their parents (the controller).

```
controller = [[MyController alloc] init];
__weak MyController *weakController = controller;
controller.block = ^{
    [weakController doSomething];
};
```

This resolves the strong reference cycle.

Availability of Blocks

Blocks are available beginning with Mac OS X 10.6 and iOS 4.0 and are an extension to the C language. Thus, you don't need to be using Objective-C to take advantage of blocks, but you do need a compiler that understands blocks, as well as runtime support. If you are targeting Mac OS X 10.5 or iOS 2.2, PLBlocks from Plausible Labs provides a solution well worth looking into.

The first high-profile API to make use of blocks was Grand Central Dispatch, a Mac OS X concurrency library. As such, many people think of blocks as being useful only in multithreaded programming. We believe that blocks are extremely handy in a very broad range of programming settings. As you solve problems in your own projects, you may find blocks to be a great fit in some unexpected places.

RanchForecast: Going Asynchronous

Note: Readers create the RanchForecast application in the previous chapter. This app fetches XML data representing the BNR class schedule from a web service and then displays that data. However, RanchForecast fetches data synchronously, which can make the UI unresponsive while network requests are taking place. In this section, readers use blocks to address this problem.

Our RanchForecast application works great in ideal circumstances. That is, with a speedy Internet connection. However, what if our Internet connection is poor? Or what if we were loading a much larger XML document that might take several seconds (or worse) to download?

Our customers will complain, and rightly so, that the application looks as if it has frozen until the request completes and the table updates. The reason is that we are running a synchronous request in the main thread, the very same thread that handles UI events. If we use **`NSURLConnection`** in the asynchronous style, however, we can avoid blocking the main thread, and the UI will be nice and responsive. Let's update our **`ScheduleFetcher`** class to do things the Right Way.

Recall that using **`NSURLConnection`** asynchronously means that we will create the connection and specify a delegate, using **`initWithRequest:delegate:`**. This call will return immediately, and the delegate will be called to handle various events during the connection's lifetime. **`NSURLConnection`** works with the run loop to make this possible.

In our application, **`ScheduleFetcher`** will act as the connection's delegate and implement the three delegate methods needed to handle receiving data and the successful and unsuccessful completion of the request:

```
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data;

- (void)connectionDidFinishLoading:(NSURLConnection *)connection;
```

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error;
```

Receiving the Asynchronous Response

If the response loading is taking place asynchronously and **ScheduleFetcher** will not block, how will **RanchForecastAppDelegate** know when the class schedule has been loaded or when an error has occurred?

We can accomplish this in several ways. The most obvious approach would be to add a pointer to **RanchForecastAppDelegate** in the **ScheduleFetcher**. Once the schedule has been fetched, the fetcher would call a method on the app delegate (**updateWithClasses:**, perhaps). The downside of this approach, however, is that we would have just made **ScheduleFetcher** dependent on **RanchForecastAppDelegate**. If we wanted to use **ScheduleFetcher** in another project later on (and we will), we would need to edit its code, which then leaves us with multiple versions of **ScheduleFetcher**.

Another approach is to use the delegate pattern. It works great for **NSURLConnection**; **ScheduleFetcher** could use it as well. We would define a `ScheduleFetcherDelegate` protocol, and **RanchForecastAppDelegate** would conform to the protocol and set itself as the delegate. This approach is very reasonable; it decouples the classes, keeping **ScheduleFetcher** reusable, but it feels somewhat heavy-handed for such a simple Web service response.

Yet another approach is to use our knowledge of blocks to apply the completion block design pattern. In this pattern, our fetch method on **ScheduleFetcher** would take a block as its only parameter. **ScheduleFetcher** then calls the block later when it is ready to deliver the results or report an error. Completion blocks are very compact; no additional methods are needed and they have the advantage of allowing us to keep the response-handling code close to the place where the Web service call is initiated.

Let's modify **ScheduleFetcher** to perform the request asynchronously and report the results using a completion block.

In `ScheduleFetcher.h`, define the `ScheduleFetchResultBlock` type. Because the block syntax involves a good bit of punctuation, it is often helpful to typedef block types so that they can be used more gracefully in the future.

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

typedef void (^ScheduleFetcherResultBlock)(NSArray *classes,
                                           NSError *error);
Now add three new instance variables and the replacement for fetchClassesWithError: to ScheduleFe
@interface ScheduleFetcher : NSObject <NSXMLParserDelegate> {
@private
    NSMutableArray *classes;
    NSMutableString *currentString;
    NSMutableDictionary *currentFields;
    NSDateFormatter *dateFormatter;

    ScheduleFetcherResultBlock resultBlock;
    NSMutableData *responseData;
    NSURLConnection *connection;
}
- (void)fetchClassesWithBlock:(ScheduleFetcherResultBlock)theBlock;
```

```
@end
```

Now, in `ScheduleFetcher.m`, remove **`fetchClassesWithError:`** and implement **`fetchClassesWithBlock:`**:

```
- (void)fetchClassesWithBlock:(ScheduleFetcherResultBlock)theBlock
{
    // Copy the block to ensure that it is not kept on the stack:
    resultBlock = [theBlock copy];

    NSURL *xmlURL = [NSURL URLWithString:
                        @"http://bignerdranch.com/xml/schedule"];

    NSURLRequest *req = [NSURLRequest requestWithURL:xmlURL
                    cachePolicy:NSURLRequestReturnCacheDataElseLoad
                timeoutInterval:30];

    connection = [[NSURLConnection alloc] initWithRequest:req
                                                 delegate:self];
    if (connection)
    {
        responseData = [[NSMutableData alloc] init];
    }
}
```

Note that `theBlock` is copied and the resulting pointer stored in `resultBlock`. We copy the block because it may still be on the stack of the calling method. If so, the block will be deallocated when that method exits. Because we are starting an asynchronous request and the calling method is guaranteed to return before the request completes, we need to be sure that the block's memory will be valid until we call it with the results. If `theBlock` were going to be used only within this method and not after we return, copying it would not be necessary.

The `resultBlock`, `connection`, and `responseData` objects are created when the fetch is initiated. It's a good idea to clean them up when the request completes. To reduce repetition, add a new method called **`cleanup`**:

```
- (void)cleanup
{
    responseData = nil;
    connection = nil;
    resultBlock = nil;
}
```

Still in `ScheduleFetcher.m`, implement the **`NSURLConnection`** delegate methods:

```
#pragma mark -
#pragma mark NSURLConnection Delegate

- (void)connection:(NSURLConnection *)theConnection
    didReceiveData:(NSData *)data
{
    [responseData appendData:data];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)theConnection
{
    [classes removeAllObjects];
```

```
    NSXMLParser *parser = [[NSXMLParser alloc]
                           initWithData:responseData];
    [parser setDelegate:self];

    BOOL success = [parser parse];
    if (!success)
    {
        resultBlock(nil, [parser parserError]);
    }
    else
    {
        NSArray *output = [classes copy];
        resultBlock(output, nil);
    }

    [self cleanup];
}

- (void)connection:(NSURLConnection *)theConnection
  didFailWithError:(NSError *)error
{
    resultBlock(nil, error);

    [self cleanup];
}
```

Response data is collected in **`connection:didReceiveData:`** and then parsed in **`connectionDidFinishLoading:`**. We then call `resultBlock` with the results or error condition.

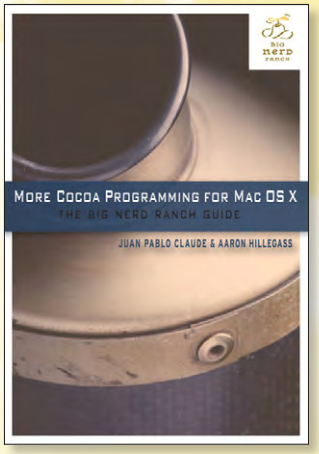
Now it's time to update **`RanchForecastAppDelegate`** to work with the new interface to **`ScheduleFetcher`**. In `RanchForecastAppDelegate.m`, update **`applicationDidFinishLaunching:`**:

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    [tableView setTarget:self];
    [tableView setDoubleAction:@selector(openClass:)];

    ScheduleFetcher *fetcher = [[ScheduleFetcher alloc] init];

    [fetcher fetchClassesWithBlock:^(NSArray *theClasses,
                                     NSError *error)     {
        if (theClasses)
        {
            classes = theClasses;
            [tableView reloadData];
        }
        else
        {
            NSAlert *alert = [[NSAlert alloc] init];
            [alert setAlertStyle:NSCriticalAlertStyle];
            [alert setMessageText:@"Error loading schedule."];
            [alert setInformativeText:[error localizedDescription]];
            [alert addButtonWithTitle:@"OK"];
            [alert beginSheetModalForWindow:self.window
                                modalDelegate:nil
                               didEndSelector:nil
                                  contextInfo:nil];
        }
    }];
}
```

Build and run the application. Notice that the application appears more quickly now because the request is performed asynchronously.

Challenge: Design a Delegate

Earlier in this chapter, we discussed how we might use the delegate design pattern as a means for passing Web service data back to the interested parties. Create a copy of this project and refactor it to use this pattern instead of blocks. Remember that delegate properties should be weak references to prevent a strong reference cycle.



CHAPTER 23: Status Items

DRAFT MANUSCRIPT
PUBLISHING OCT 2012

MORE COCOA PROGRAMMING FOR MAC OS X: THE BIG NERD RANCH GUIDE,

by Juan Pablo Claude and Aaron Hillegass

TABLE OF CONTENTS

AVAILABLE FORMATS

- 9780321706263 Book
- 9780321706577 eBook
- 9780321706607 Safari Books Online

More Cocoa Programming

the big nerd ranch guide

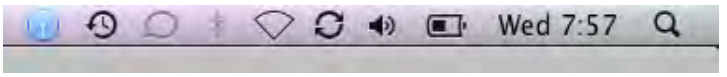
JUAN PABLO CLAUDE & AARON HILLEGASS

BiG
nerD
ranch

25
Status Items

Status items are the little icons on the right side of the OS X menu bar as seen in Figure 25.1. A status item has a menu associated with it, This menu gives the user information and settings options for a process that would otherwise be faceless, i.e. have no user graphical user interface.

Figure 25.1 Status Items on the menu bar



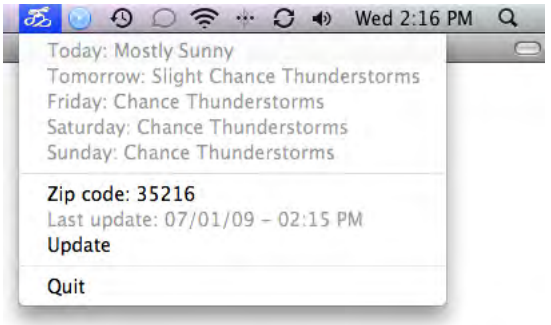
Status items are instances of the **`NSStatusItem`** class. Status items must be placed inside a status bar, which is an instance of **`NSStatusBar`**. Currently, OS X provides only one system-wide status bar that is displayed on the right side of the menu bar.

Because menu bar space is limited, Apple recommends that status items be used sparingly and only when there is no other reasonable way to give the user a means to interact with a program.

In Figure 25.1, however, the only **`NSStatusItem`** is the leftmost icon. The others are instances of **`NSMenuExtra`** instead. These items use a private API that Apple reserves for its own status items. Menu extras have additional functionality and can be rearranged and removed from the menu bar by Command-dragging them.

In this chapter, you will create an application called WeatherStatus that has a status item. This status item will display the five-day weather forecast for a given zip code and update itself periodically, as seen in Figure 25.2.

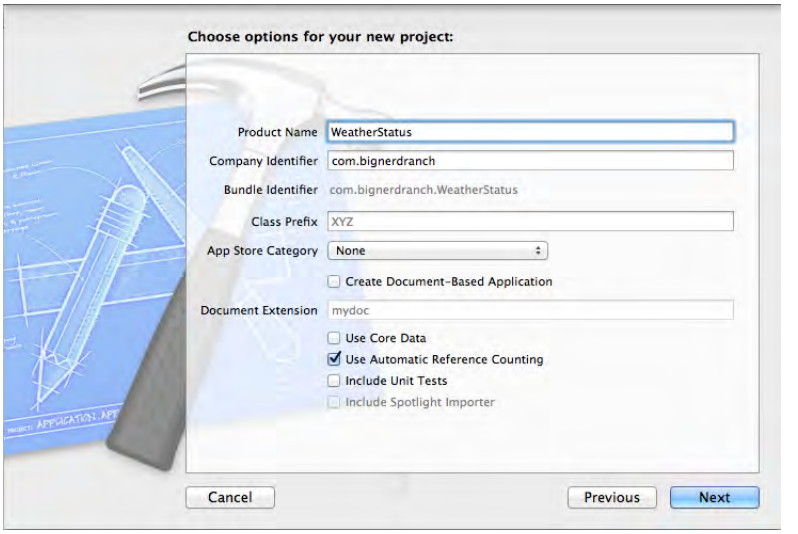
Figure 25.2 WeatherStatus application status item menu



Starting the Status Item Project

Launch Xcode, start a new Cocoa Application project and name it WeatherStatus. Make sure the bundle identifier is `com.bignerdranch.WeatherStatus`, as seen in Figure 25.3.

Figure 25.3 Start the WeatherStatus Project



After creating the project, open the `WeatherStatus-info.plist` file and add the following key-value pair:

```
<key>LSUIElement</key>
<string>1</string>
```

The `LSUIElement` entry marks the application as a faceless agent, and if you build and run the project, you will see that an empty window appears on the screen, but no dock icon or application menu is displayed. Stop the application from Xcode. Then open `MainMenu.xib` and delete the unneeded window. If you want, you can also delete the Font Manager and Main Menu objects from the NIB file.

The next step is to flesh out the automatically created **`AppDelegate`** object. If you are using an older version of Xcode, you may need to create this object by dragging an **`NSObject`** from the IB library and changing its class to **`AppDelegate`**. Make it the application delegate by Control-dragging from the **`Application`** icon onto it and selecting the delegate outlet from the HUD panel. Finally, create the class files and add them to the WeatherStatus project.

Back in Xcode, open `AppDelegate.h` and add the code listed below.

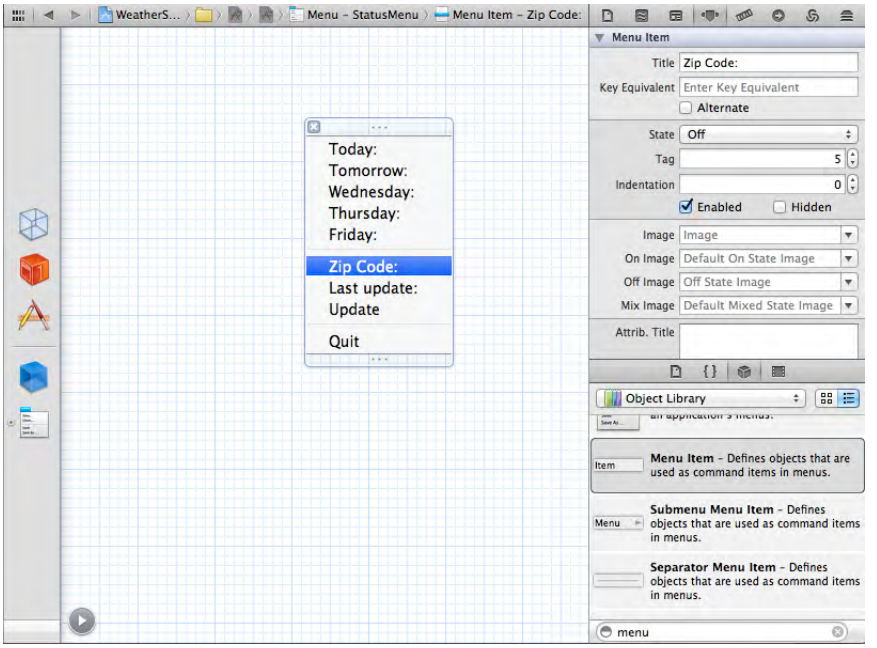
```
#import <Cocoa/Cocoa.h>

@interface AppDelegate : NSObject <NSApplicationDelegate>
{
    __weak IBOutlet NSMenu *statusMenu;
```

```
    NSStatusItem *theItem;
    NSUInteger zipCode;
    double latitude, longitude;
    NSTimer *timer;
    NSUInteger updateInterval;    // In minutes
}

// Properties:
@property (assign) IBOutlet NSWindow *window;
@property (weak) NSMenu *statusMenu;
@property (assign) double latitude;
@property (assign) double longitude;

// Accessors:
- (NSUInteger)zipCode;
- (BOOL)setZipCode:(NSUInteger)code;
- (NSUInteger)updateInterval;
- (void)setUpdateInterval:(NSUInteger)interval;

// Actions:
- (IBAction)fetchWeather:(id)sender;
- (IBAction)changeZipCode:(id)sender;

// Other:
- (NSMutableDictionary *)fetchCoordinatesForZipCode:(NSInteger)code
                                              error:(NSError **)error;
- (NSData *)sendRequestWithURLString:(NSString *)urlString
                               error:(NSError **)error;
- (NSArray *)parseXMLNodesWithPath:(NSString *)path
                       fromXMLData:(NSData *)xmlData
                             error:(NSError **)error;
- (NSArray *)dayStringsFromDate:(NSDate *)date;
- (void)launchTimer;
- (void)stopTimer;
- (void)setTitle:(NSString *)title forMenuItemWithTag:(NSUInteger)tag;

@end
```

The only outlet declared in `AppDelegate.h` is `statusMenu`, and it needs to point to the status item menu shown in Figure 25.2. Make sure the header file is saved and go back to Interface Builder to create the menu and connect it to **`AppDelegate`**. Begin by deleting the auto-generated menu, window, and font manager objects. Then drag an **`NSMenu`** object from the object library into the `MainMenu.xib` object bar and name it StatusMenu. Connect the new menu to the **`AppDelegate`** object by Control-dragging and selecting the `statusMenu` outlet from the HUD panel.

Double-click the **`StatusMenu`** icon in the object bar on the left to display the menu. Add menu items and separators and label the items as shown in Figure 25.4. In the attributes panel, give tags to the menu items from 0 to 8, top to bottom.

Figure 25.4 Create a menu for WeatherStatus



Make the following connections between menu items and action methods:

- Zip code: (tag 5) menu item to the **`-changeZipCode:`** action of **`AppDelegate`**
- Update (tag 7) to **`-fetchWeather:`** of **`AppDelegate`**
- Quit (tag 8) to **`-terminate:`** of **`Application`**

The rest of the menu items are informational and do not need to be connected to any actions. Save your changes and return to Xcode and the WeatherStatus project.

Data Persistence

Because the WeatherStatus application is an agent designed to only display information, it will not provide a direct means for the user to input the zip code and update interval. Instead, you will make these parameters settable in the next chapter, where you will create a preference pane for this application.

Considering this design, a simple means to provide persistence for the application's data is to store the two parameters (`zipCode` and `updateInterval`) in the application's preferences file.

To make the management of the preferences easier, create and add the files `PreferenceStrings.h` and `PreferenceStrings.m` to the project. Open these files and edit them to look like the listings below.

`PreferenceStrings.h`:

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

```
extern NSString * const BNRZipCode;
extern NSString * const BNRUpdateInterval;
```

PreferenceStrings.m:

```
#import "PreferenceStrings.h"

NSString * const BNRZipCode = @"ZipCode";
NSString * const BNRUpdateInterval = @"UpdateInterval";
```

With this preliminary work behind you, you can start editing `AppDelegate.m` and working on the bulk of the application. Open the file and start by synthesizing properties and registering the default preferences in the **`+initialize`** class method.

```
#import "AppDelegate.h"
#import "PreferenceStrings.h"

@implementation AppDelegate

@synthesize window = _window;
@synthesize statusMenu;
@synthesize latitude;
@synthesize longitude;


+ (void)initialize
{
    // Register the default preferences for the application:
    NSMutableDictionary *defaultPreferences = [NSMutableDictionary dictionary];

    NSNumber *zipNumber = [NSNumber numberWithUnsignedInteger:35216];
    NSNumber *updateNumber = [NSNumber numberWithUnsignedInteger:15];

    [defaultPreferences setObject:zipNumber forKey:BNRZipCode];
    [defaultPreferences setObject:updateNumber forKey:BNRUpdateInterval];

    [[NSUserDefaults standardUserDefaults] registerDefaults:defaultPreferences];
    // Synchronize the preferences to make sure they are immediately written to disk
    // and are accessible to the preference pane plug-in:
    [[NSUserDefaults standardUserDefaults] synchronize];
}

@end
```

At the end of the **`+initialize`** method, you synchronize the user defaults to save them to disk immediately. Typically, this is not an operation you want to do very frequently because it can harm the application's performance. However, in this case, the user defaults need to be accessible from a different process (a preference pane) as soon as possible, and you are only saving a small amount of data.

At this point, you can also implement the **`-init`** method, which just reads the user defaults.

```
- (id)init
{
    self = [super init];
    if (self) {
```

```
        // Read defaults:
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        updateInterval = [defaults integerForKey:BNRUpdateInterval];
        zipCode = [defaults integerForKey:BNRZipCode];
    }

    return self;
}
```

Using NOAA Web Services

To obtain the weather forecast for a given zip code, you will use two web services provided by the National Oceanic and Atmospheric Administration (NOAA).

The first web service is located at `www.weather.gov/forecasts/xml/sample_products/browser_interface/ndfdXMLclient.php` and accepts a list of zip codes as a GET argument called `listZipCodeList`. The service returns a list of coordinates (latitude, longitude) for all the zip codes in the list. A sample of the XML returned by the service for `listZipCodeList=35216` is shown below.

```
<?xml version='1.0' ?>
<dwml version='1.0' xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
     "http://www.nws.noaa.gov/mdl/survey/pgb_survey/dev/DWMLgen/schema/DWML.xsd">
    <latLonList>33.5277,-86.7992&lt;/latLonList>
</dwml>
```

The second web service is located at `www.weather.gov/forecasts/xml/sample_products/browser_interface/ndfdBrowserClientByDay.php`. This service takes several GET parameters, including latitude, longitude, format, start date, and number of days for the forecast. A partial sample of the XML results obtained with the parameters `lat=33.5277&lon=-86.7992&format=24+hourly&startDate=2009-7-1&numDays=5` is listed below. The interesting information for WeatherStatus is contained in the XPath `dwml/data/parameters/weather/weather-conditions/@weather-summary`.

```
<?xml version="1.0"?>
<dwml version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation=
        "http://www.nws.noaa.gov/forecasts/xml/DWMLgen/schema/DWML.xsd">
  <head>

    <!-- Omitted header information -->

  </head>
  <data>
    <location>
      <location-key>point1</location-key>
      <point latitude="33.53" longitude="-86.80"/>
    </location>

    <!-- Omitted data -->
```

```
    <parameters applicable-location="point1">

     <!-- Omitted data -->

      <weather time-layout="k-p24h-n5-1">
        <name>Weather Type, Coverage, and Intensity</name>
        <weather-conditions weather-summary="Sunny"/>
        <weather-conditions weather-summary="Slight Chance Thunderstorms">
          <value coverage="slight chance" intensity="none"
                 weather-type="thunderstorms" qualifier="none"/>
          <value coverage="slight chance" intensity="moderate"
                 additive="and" weather-type="rain showers" qualifier="none"/>
        </weather-conditions>
        <weather-conditions weather-summary="Chance Thunderstorms">
          <value coverage="chance" intensity="none"
                 weather-type="thunderstorms" qualifier="none"/>
          <value coverage="chance" intensity="moderate"
                 additive="and" weather-type="rain showers" qualifier="none"/>
        </weather-conditions>
        <weather-conditions weather-summary="Chance Thunderstorms">
          <value coverage="chance" intensity="none"
                 weather-type="thunderstorms" qualifier="none"/>
          <value coverage="chance" intensity="moderate" additive="and"
                 weather-type="rain showers" qualifier="none"/>
        </weather-conditions>
        <weather-conditions weather-summary="Chance Thunderstorms">
          <value coverage="chance" intensity="none"
                 weather-type="thunderstorms" qualifier="none"/>
          <value coverage="chance" intensity="moderate" additive="and"
                 weather-type="rain showers" qualifier="none"/>
        </weather-conditions>
      </weather>

      <!-- Omitted data -->

    </parameters>
  </data>
</dwml>
```

Your next task is to implement the **`AppDelegate`** methods that invoke the web services just described and parse the returned data.

The **`-sendRequestWithURLString:error:`** method uses the standard Cocoa facilities to send an HTTP request with a timeout of 30 seconds and override locally cached data If the request fails, `nil` is returned.

```
- (NSData *)sendRequestWithURLString:(NSString *)urlString
                               error:(NSError **)error
{
    NSURL *url = [NSURL URLWithString:urlString];
    NSURLRequest *urlRequest = [NSURLRequest
                                    requestWithURL:url
                                       cachePolicy:NSURLRequestReloadIgnoringCacheData
                                   timeoutInterval:30];

    // Fetch the XML response:
```

```
    NSData *xmlData = [NSURLConnection sendSynchronousRequest:urlRequest
                                            returningResponse:NULL
                                                        error:error];
    if (!xmlData) {
        NSLog(@"Web service error - no data");
        return nil;
    }
    return xmlData;
}
```

The next method parses XML data returned by a web service and returns an array of nodes for a given Xpath. Just as in the method above, `nil` is returned if there is a failure.

```
- (NSArray *)parseXMLNodesWithPath:(NSString *)path
                       fromXMLData:(NSData *)xmlData
                             error:(NSError **)error
{
    NSXMLDocument *doc = [[NSXMLDocument alloc] initWithData:xmlData
                                                     options:0
                                                       error:error];
    NSArray *itemNodes = [doc nodesForXPath:path error:error];
    if (!itemNodes) {
        NSLog(@"XML parsing error - no nodes");
    }
    return itemNodes;
}
```

Now you can implement a method for retrieving the coordinates for a given zip code.

```
- (NSMutableDictionary *)fetchCoordinatesForZipCode:(NSInteger)code
                                              error:(NSError **)error
{
    // Prepare the web service URL:
    NSString *urlString = [NSString stringWithFormat:
        @"http://www.weather.gov/forecasts/xml/sample_products/browser_interface/"
        @"ndfdXMLclient.php?listZipCodeList=%d", code];

    // Send the web service request and process:
    NSData *xmlData = [self sendRequestWithURLString:urlString error:error];
    if (xmlData) {
        // Parse the XML response:
        NSArray *itemNodes = [self parseXMLNodesWithPath:@"dwml/latLonList"
                                             fromXMLData:xmlData
                                                   error:error];
        if (itemNodes) {
            // Expect only one node:
            if ([itemNodes count] == 1) {
                NSString *latLonStr = [[itemNodes objectAtIndex:0] stringValue];
                NSArray *latLonValues = [latLonStr componentsSeparatedByString:@","];
                NSMutableDictionary *result = [NSMutableDictionary dictionary];
                [result setObject:[latLonValues objectAtIndex:0] forKey:@"latitude"];
                [result setObject:[latLonValues objectAtIndex:1] forKey:@"longitude"];
                // Return with success:
                return result;
            } else {
                // Web service returned XML, but no coordinates.
                // Is the caller interested in the error?
                if (error) {
```

```
                    NSDictionary *errorData =
                    [NSDictionary
                        dictionaryWithObject:@"Could not get coordinates for ZIP code"
                                      forKey:NSLocalizedDescriptionKey];
                    *error = [NSError errorWithDomain:@"ApplicationDomain"
                                                 code:1
                                             userInfo:errorData];
                }
            }
        }
    }
    // If we are here, the operation failed. Return with failure:
    return nil;
}
```

The **-fetchCoordinatesForZipCode:error:** method converts a given zip code using the appropriate NOAA web service. It expects only one data node in the response data, and any other condition is considered an error. The data is returned in a dictionary with the `@"latitude"` and `@"longitude"` keys.

Finally, implement the action for fetching the weather forecast. The **-fetchWeather:** action method retrieves the five-day forecast for a given date and set of coordinates. Note that the web service may return fewer than five forecasts if not all are available. That condition is not considered an error and the missing forecasts are simply marked for the user. Also notice that the weather information is displayed as menu items using the method **dayStringsFromDate:**, which we will implement next.

```
- (IBAction)fetchWeather:(id)sender
{
    // We need the current year, month, and date:
    NSDate *now = [NSDate date];
    NSCalendar *cal = [NSCalendar currentCalendar];
    NSDateComponents *components = [cal components:
        (NSYearCalendarUnit | NSMonthCalendarUnit | NSDayCalendarUnit)
                                          fromDate:now];

    // Prepare the web service URL:
    NSString *urlString = [NSString stringWithFormat:
        @"http://www.weather.gov/forecasts/xml/sample_products/browser_interface/"
        @"ndfdBrowserClientByDay.php?"
        @"lat=%f&lon=%f&format=24+hourly&startDate=%d-%d-%d&numDays=5",
        [self latitude], [self longitude],
        [components year], [components month], [components day]];

    // Send the web service request and process:
    NSError *error = nil;
    NSData *xmlData = [self sendRequestWithURLString:urlString error:&error];
    if (xmlData) {
        NSArray *itemNodes = [self parseXMLNodesWithPath:
        @"dwml/data/parameters/weather/weather-conditions/@weather-summary"
                                           fromXMLData:xmlData
                                                 error:&error];
        if (itemNodes) {
            NSArray *dayStrings = [self dayStringsFromDate:now];
            NSInteger i;
            NSInteger itemCount = [itemNodes count];
            for (i=0; i<itemCount; i++) {
                NSString *day = [dayStrings objectAtIndex:i];
                NSXMLNode *node = [itemNodes objectAtIndex:i];
                [self setTitle:[NSString stringWithFormat:
```

```
                @"%@: %@", day, [node stringValue]] forMenuItemWithTag:i];
            }
            if (itemCount < 5) {
                for (i=itemCount; i<5; i++) {
                    NSString *day = [dayStrings objectAtIndex:i];
                    [self setTitle:[NSString stringWithFormat:
                    @"%@: No forecast available", day] forMenuItemWithTag:i];
                }
            }
            // Set the update time:
            [self setTitle:[NSString stringWithFormat:@"Last update: %@",
                [now dateWithCalendarFormat:@"%m/%d/%y - %I:%M %p" timeZone:nil]]
                forMenuItemWithTag:6];
            return;
        }
    }
    // If we got here, the update failed:
    [self setTitle:@"Last update: Failed!" forMenuItemWithTag:6];
}
```

Implementing Other Utility Methods

In this section, you will implement four utility methods that are used elsewhere in the program.

The first method returns an array of strings with the names for the next five days for a given date. This method is needed to update the status item menu.

```
- (NSArray *)dayStringsFromDate:(NSDate *)date
{
    NSMutableArray *days = [NSMutableArray arrayWithCapacity:5];
    [days addObject:@"Today"];
    [days addObject:@"Tomorrow"];

    NSInteger i;
    for (i=2; i<5; i++) {
        // There are 86400 seconds in a day:
        NSDate *nextDate = [date dateByAddingTimeInterval:86400.0 * i];
        [days addObject:[nextDate dateWithCalendarFormat:@"%A" timeZone:nil]];
    }
    return days;
}
```

Next is a method for changing the title of a menu item given its tag number.

```
- (void)setTitle:(NSString *)title forMenuItemWithTag:(NSUInteger)tag
{
    NSMenuItem *menuItem = [statusMenu itemWithTag:tag];
    [menuItem setTitle:title];
}
```

Finally, we have two methods for managing the **`NSTimer`** that periodically updates the weather forecast. The **`-launchTimer`** method uses an invocation to the **`-fetchWeather:`** action and calls it every `updateInterval * 60` seconds.

```
- (void)launchTimer
```

```
{
    SEL selector = @selector(fetchWeather:);
    NSMethodSignature *signature =
    [AppDelegate instanceMethodSignatureForSelector:selector];
    NSInvocation *invocation = [NSInvocation invocationWithMethodSignature:signature];
    [invocation setSelector:selector];
    [invocation setTarget:self];
    [invocation setArgument:(__bridge void *)self atIndex:2];
    timer = [NSTimer scheduledTimerWithTimeInterval:[self updateInterval] * 60
                                             invocation:invocation
                                                repeats:YES];
}

- (void)stopTimer
{
    [timer invalidate];
    timer = nil;
}
```

Implementing Accessors

At this point, you have everything you need to implement the missing accessor methods. Note that the setter methods do not save preference data; that will be done by the preference pane. Also notice that **`-setZipCode:`** returns a BOOL rather than void because if a given zip code is invalid, the operation will fail.

```
- (void)setUpdateInterval:(NSUInteger)interval
{
    updateInterval = interval;
    [self stopTimer];
    [self launchTimer];
}

- (NSUInteger)updateInterval
{
    return updateInterval;
}

- (BOOL)setZipCode:(NSUInteger)code
{
    NSError *error = nil;
    NSMutableDictionary *coordinates = [self fetchCoordinatesForZipCode:code error:&error];
    if (!error) {
        // Save data:
        zipCode = code;
        [self setLatitude:[[coordinates objectForKey:@"latitude"] doubleValue]];
        [self setLongitude:[[coordinates objectForKey:@"longitude"] doubleValue]];

        // Update menu item:
        [self setTitle:[NSString stringWithFormat:@"Zip code: %d", code]
        forMenuItemWithTag:5];

        // Return with success:
        return YES;
    } else {
        // Change nothing and return with failure:
        return NO;
```

```
    }
}

- (NSUInteger)zipCode
{
    return zipCode;
}
```

Even though it is an action rather than an accessor, you will also implement **-changeZipCode:** here. However, leave the method empty for now because the zip code change will be done via the preference pane.

```
- (IBAction)changeZipCode:(id)sender
{
    // Pass for now
}
```

Finishing the Program

Only two methods remain to complete the WeatherStatus program. The first one is **-awakeFromNib**, where the **NSStatusItem** is actually created. In this method, the initial weather update is also performed, and the possibility of an invalid zip code is handled. Notice how the **NSStatusItem** is actually created by a method of the system status bar: **-statusItemWithLength:**. This method is called with the `NSVariableStatusItemLength` argument to guarantee the smallest possible status item size to conserve space.

```
- (void)awakeFromNib
{
    // Create the NSStatusItem:
    NSStatusBar *bar = [NSStatusBar systemStatusBar];
    theItem = [bar statusItemWithLength:NSVariableStatusItemLength];

    // Get image resources:
    NSString *pathToIcon = [[NSBundle mainBundle] pathForResource:@"tinyhat"
                                                           ofType:@"png"];
    NSImage *iconImage = [[NSImage alloc] initWithContentsOfFile:pathToIcon];
    NSString *pathToNegativeIcon = [[NSBundle mainBundle] pathForResource:@"tinyhatneg"
                                                                   ofType:@"png"];
    NSImage *negativeIconImage =
    [[NSImage alloc] initWithContentsOfFile:pathToNegativeIcon];

    // Set the images for the NSStatusItem:
    [theItem setImage:iconImage];
    [theItem setAlternateImage:negativeIconImage];
    [theItem setHighlightMode:YES];
    [theItem setMenu:statusMenu];

    // Set the zip code and fetch the weather:
    BOOL success = [self setZipCode:zipCode];
    if (success) {
        [self fetchWeather:self];
    } else {
        NSArray *dayStrings = [self dayStringsFromDate:[NSDate date]];
        int i;
        for (i=0; i<5; i++) {
```

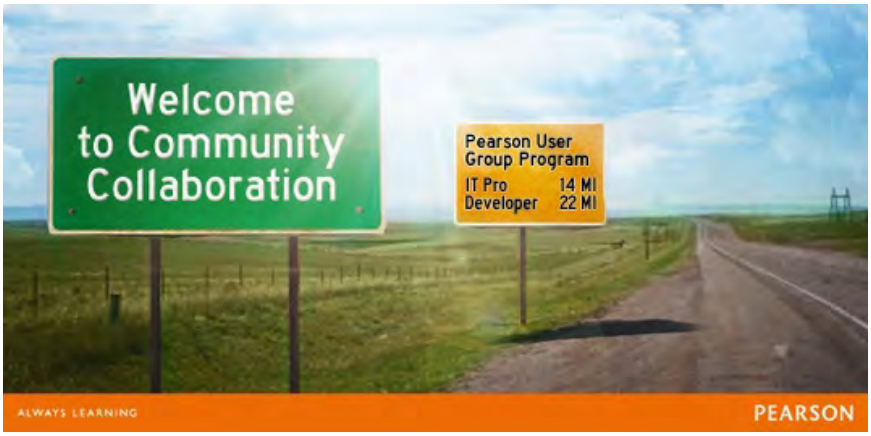
```
            NSString *day = [dayStrings objectAtIndex:i];
            [self setTitle:[NSString stringWithFormat:@"%@: No forecast available", day]
        forMenuItemWithTag:i];
        }
        [self setTitle:@"Zip code: Invalid!" forMenuItemWithTag:5];
        [self setTitle:@"Last update: Failed!" forMenuItemWithTag:6];
    }

    // Set a timer to periodically update:
    [self launchTimer];
}
```

At this point, you need to add the image files `tinyhat.png` and `tinyhatneg.png` to the project's resources. Download the files from `http://www.bignerdranch.com/solutions/MoreCocoa.zip` and add them to your project. Make sure the image files are copied to the target bundle.

After all this work, you can finally build and enjoy your WeatherStatus status item. Go for it!

Challenge

The WeatherStatus program currently displays a failure to connect to the weather service when the zip code is invalid. Fix this situation to give the user more accurate feedback.



PART OF A USER GROUP?

Are you a member of a group that meets to discuss IT-related topics? Your group may be eligible for the Pearson User Group Program!

BENEFITS INCLUDE:

- Member Discount – SAVE 35% on print titles, SAVE 45% on eBooks
- FREE review copies
- Advanced ACCESS to content, product review opportunities and more!

LEARN MORE!

To learn if your group qualifies please visit **informit.com/usergroups**.

Addison Wesley | Cisco Press | IBM Press | que | PEARSON IT CERTIFICATION | PRENTICE HALL | SAMS | vmware PRESS | informIT