PART I
Mac OS X Developer Resources

1

The Mac OS X Environment

WHAT YOU WILL LEARN IN THIS CHAPTER:

- How the Mac OS X operating system is structured, including what the major areas of the system are and how they work together
- How to use Mac OS X's command-line interface
- How applications take advantage of the operating system services on Mac OS X
- How Apple encourages a common look and feel for Mac OS X applications

Welcome to the wonderful world of Mac OS X, the next-generation operating system from Apple Computer!

The Mac OS X operating system powers modern Macintosh computers. After many long years and a few scrapped attempts to modernize the older Mac OS operating system, Apple released Mac OS X in April 2001. Since then, Apple has released a steady stream of upgrades and system updates. This book was written around Mac OS X v10.6 Snow Leopard, the latest version.

To write software for Mac OS X, you need to know your way around the system. By now you may already be familiar with Mac OS X's applications and user interface style. Those things all rest on top of a number of subsystems and services that make up the Mac OS X operating system.

INTRODUCING THE MAC OS X

What comes to mind when you think of Mac OS X? Is it the applications you use? Perhaps you recall Mac OS X's distinctive user interface? Or maybe you think of Mac OS X's stability? In truth, Mac OS X embodies all these things.

The Mac OS X operating system is often described as a collection of layers, as seen in Figure 1-1.

You are probably already familiar with the topmost layer: the applications that run on Mac OS X (such as Mail, iTunes, Safari, and so on). These applications are all written against a collection of application *frameworks*. These frameworks are special libraries that provide the code and all the other resources (icons, translated strings, and so on) to perform common tasks. For example, the Cocoa framework contains a number of resources necessary to make a Cocoa application.

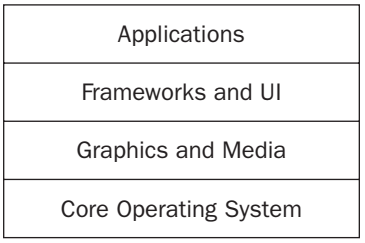

FIGURE 1-1

All Mac OS X applications use graphics to some extent, ranging from simply presenting its user interface to processing graphical data such as QuickTime movies. The system provides several specialized libraries for working with graphics and graphics files.

These layers rest on the broad shoulders of the core operating system, which at the lowest level is responsible for making your Macintosh run. For example, the core OS handles reading from and writing to your hard drive and random access memory (RAM), it manages your network connections, it powers down the computer when it falls to "sleep," and so on. In fact, any program that talks to your hardware in any way ultimately goes through the core OS.

Throughout this book you examine Mac OS X in detail through Slide Master, an application that builds and displays photo slideshows. You will build Slide Master bit-by-bit as you learn more about how the elements of Mac OS X come together. The Slide Master application and its source code can be downloaded from Wiley's web site; so you can check your work against our complete solution as you go.

This is a good time to take a quick tour of Slide Master. You can download Slide Master from Wiley's web site, make a slideshow, and view your handiwork. In doing so, you touch on all the major areas of the Mac OS X operating system.

TRY IT OUT Slide Master

1. Download the files for this chapter from `www.wrox.com`. Refer to the Introduction for instructions on finding the files you need from the Wrox web site. You can search for the book by its ISBN number: 978-0-470-57752-3. You are looking for a file named `MacOSXProg Chapter01.zip`.
2. Uncompress the `MacOSXProg Chapter01.zip` archive using your favorite decompression tool. (Mac OS X supports uncompressing `.zip` files directly in the Finder.) Inside you will find the Slide Master application, a folder of pictures called `Images`, and a folder of source code.

3. Run the Slide Master application by double-clicking it in Finder. The application opens an untitled document window.

4. Add the pictures in the Images folder to Slide Master by choosing Slide Show ➪ Add Slide. You can select all the files at once from the open panel. The images appear in a drawer to the side of the document window and the main window displays the selected image, as shown in Figure 1-2. You can use the arrow keys to change the selection.

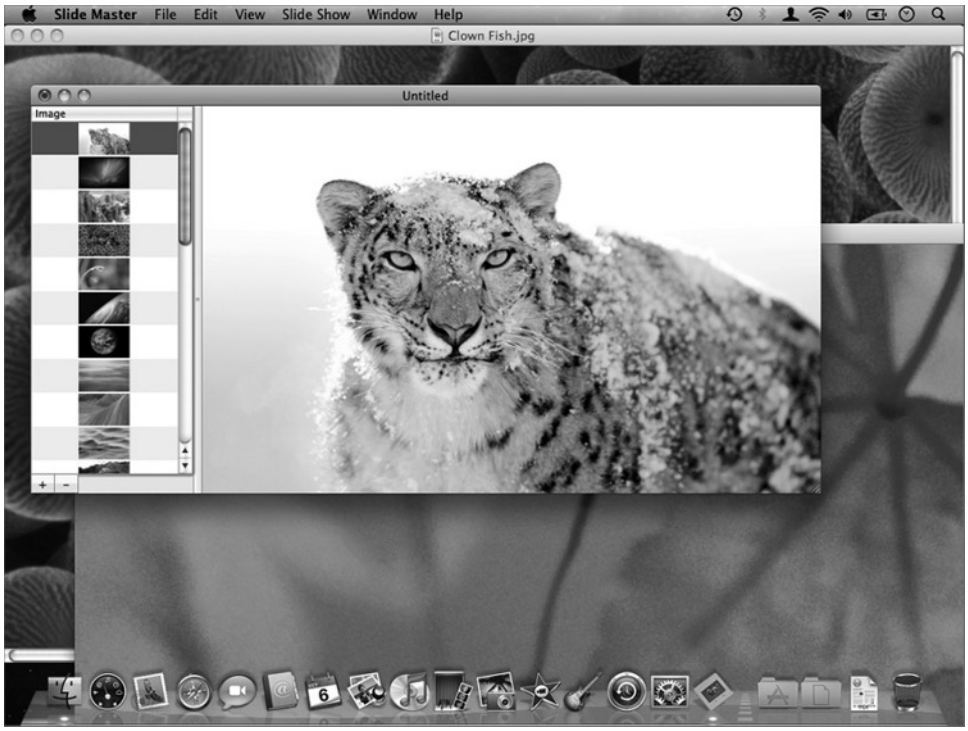

FIGURE 1-2

5. Export a slideshow as a QuickTime movie by choosing File ➪ Export. Slide Master writes out a QuickTime movie and opens it with QuickTime Player.

6. Save your document by choosing File ➪ Save.

How It Works

Slide Master is a document-based application, which means that it provides a user interface for individual documents. In this case, documents are collections of slides that you can sift through and export as QuickTime movies. Slide Master documents can be opened, saved, and closed using the File menu. Other document-based applications also support printing, although Slide Master does not.

Much of the functionality you see here comes from Slide Master's application framework: Cocoa. The Cocoa application framework provides the implementation for the things you see on the screen: windows, pictures, menus, buttons, and so on. Cocoa also provides support for managing the document: reading and writing document files, closing the document when its window is closed, and routing menu commands to the selected document. Finally, Cocoa provides tools for storing application data, including working with user preferences and storing lists of items in memory.

Of course Slide Master uses QuickTime to generate movie files. You are probably already familiar with QuickTime, both through QuickTime Player and through web browsers that support the display of QuickTime movies. But QuickTime also makes most, if not all, of its functionality available to applications through its framework interface.

When you save a Slide Master document, the document file contains a list of image files that are part of your slideshow, not the actual images themselves. As a result, these documents can be relatively small. Behind the scenes, Slide Master uses aliases to track these image files so that they can be found if the files are moved around on your disk. These aliases are the same aliases you can create in the Finder, although they are embedded in your document rather than saved separately to disk.

You learn more about Cocoa, QuickTime, and other technologies later in this chapter, and as you proceed through this book.

THE CORE OPERATING SYSTEM

The heart of Mac OS X is based on the Unix operating system. Unix was developed by AT&T in the early 1970s. In those days, computers were large and expensive, and Unix was intended as a way to share computing resources between multiple users at once. It was likely that an organization at that time could afford only one computer for all its members, and Unix provided a way for people to use that computer simultaneously without getting in each other's way.

Over the years, Unix development has split off into many distinct "flavors" of Unix, all headed up by different groups of people, all with somewhat different goals. BSD and Linux are two such examples. Each version of Unix shares some portion of the original vision and typically implements a common set of libraries and commands.

Unix is regarded as a robust operating system whose scalability and innate networking capability make it ideal for use as a server. In fact, most of the modern-day Internet is powered by Unix servers of one version or another. It turns out that these features are also desirable in modern desktop operating systems. So it is no surprise that when Apple was seeking to modernize the original Macintosh operating system, it turned to Unix.

Mac OS X's core operating system is a Unix flavor called *Darwin*. As with most Unix flavors, Darwin's source code is freely available, allowing interested parties to see exactly how the core operating system works. Apple maintains several resources for programmers interested in Darwin, including a way for people-at-large to contribute changes and bug fixes back to Apple.

Although Mac OS X tries to hide Darwin from the average user, there are some places where the Unix command line pokes through. The most obvious example is the Terminal application, found in `/Application/Utilities`. You can use Terminal to work directly with Darwin's command-line tools. A more subtle example includes the way you describe file locations on Mac OS X: by using a *file path*. A file path is a string of text that describes a file's location.

The original Mac OS operating system abhorred file paths and tried its best to avoid them; but even so, it devised a convention for describing a path to a file. Mac OS file paths are composed of a disk volume name followed by several folder names and possibly a file, all separated by colons, as in `Macintosh HD:Applications:Utilities:Terminal.app`.

PROGRAM, PROCESS, APPLICATION — WHAT'S THE DIFFERENCE?

Much of the time you can use the terms *program* and *process* interchangeably to refer to something that's *executable*. But these terms do have distinct definitions. The word *program* refers to a file on disk containing a series of computer instructions. When this file is executed (or run, launched, and so on), the computer starts processing the instructions in the file. *Process* describes the act of executing the file. To borrow an example from the kitchen, it may help to think of a program as a recipe for baking a cake, and the process as the act of baking that cake.

Ultimately, an *application* is just a program. On Mac OS X, however, programs can take many forms: simple tools typed in a command-line interface, a program you can double-click in the Finder, a plug-in file loaded by other programs, and so on. To avoid some confusion, we use the term *application* in this book to refer specifically to programs that appear in the Finder; we use the term *program* when no distinction is necessary.

Although there are places where this old convention still exists, Mac OS X mostly uses Unix's method of describing file paths: a series of directories from the *root* directory all separated by slashes, as in `/Applications/Utilities/Terminal.app`. The root directory contains all the files and directories on a Mac OS X system and is referred to simply as `/`. The path `/Applications` refers to a file or directory named `Applications` in the root directory. A path that begins with the root slash is called an *absolute* (or *full*) *path* because it describes a precise file location. If the root slash is not included, the path is called a *relative path* because it is relative to your current location.

NOTE *If you look in* `/Applications/Utilities` *in the Finder, you might notice that there is no* `Terminal.app`*; instead there's just a program called Terminal. By default, Finder and other applications hide file extensions such as .app and .txt from you. So the application at* `/Applications/Utilities/Terminal.app` *appears simply as Terminal. The Core OS makes no attempt to hide extensions from you; if you browse the file system using Mac OS X's command-line interface, you can see all these extensions. You learn more about Mac OS X's command-line interface later in this chapter.*

Darwin is composed of several parts, including a kernel, a system library, and numerous commands, as illustrated in Figure 1-3.

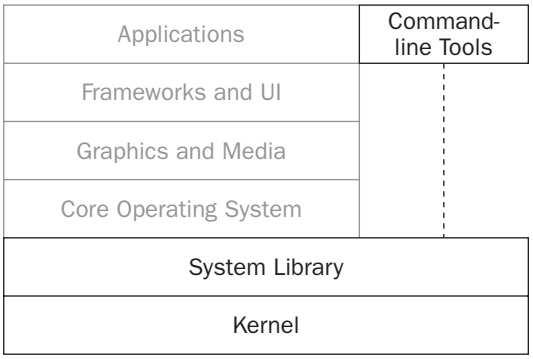

FIGURE 1-3

The Kernel

The heart of a Unix operating system is its *kernel*. The kernel is the program that loads when the computer is first turned on and is responsible for managing all the hardware resources available to the computer. The kernel is also responsible for running the other programs on the system, scheduling process execution so that they can share the central processing unit (CPU) and other resources, and preventing one process from seeing what another process is doing. These last two responsibilities are more commonly known as *preemptive multitasking* and *protected memory*, respectively.

Because Unix prevents programs from accessing the computer hardware or other programs directly, it protects against the most common forms of system crashes. If a process misbehaves in one way or another, the system simply terminates the process and continues on its way. In other words, the misbehaving process crashes. In some operating systems, a misbehaving process can stomp all over other applications, or even break the operating system itself, before the system is able to terminate the process. As a result, poorly written programs can cause the entire computer to freeze or crash. Not so on Unix; because a process cannot modify other processes, including the kernel, there is virtually no risk of a bad process bringing down the entire operating system.

Although the kernel is responsible for accessing hardware, much of the knowledge of specific hardware details is delegated to *device drivers*. Device drivers are small programs that are loaded directly into the kernel. Whereas the kernel might know how to talk to hard disks, a specific device driver generally knows how to talk to specific makes and models of hard disks. This provides a way for third parties to add support for new devices without having to build it into Apple's kernel. Mac OS X includes default drivers for talking to a wide variety of devices, so much of the time you won't need to install separate drivers when you install new third-party hardware.

The System Library

The kernel is responsible for critical functions such as memory management and device access, so programs must ask the kernel to perform work on its behalf. Programs communicate with the kernel through an application program interface (API) provided by a special library. This library defines some common data structures for describing system operations, provides functions to request these operations, and handles shuttling data back and forth between the kernel and other programs. This library is simply called the *system library*.

As you might imagine, every program on Mac OS X links against this library, either directly or indirectly. Without it, a program would be unable to allocate memory, access the file system, and perform other simple tasks.

> **WHAT IS AN API?**
>
> All libraries and frameworks provide a collection of functions and data structures that programs can use to perform a task. For example, the system library provides functions for reading from files, and QuickTime provides functions for playing back QuickTime movies. These functions and data structures are collectively known as the library's *application program interface,* or API.

The system library takes the form of a dynamic library installed as `/usr/lib/libSystem.B.dylib`. Mac OS X also includes a framework called `System.framework` in `/System/Library/Frameworks` that refers to this library. The files that define the Darwin interface live in the `/usr/include` directory. By the way, neither of these directories is visible from Finder; Mac OS X actively hides much of the complexity of Darwin from the average Mac user.

Unix Commands

Unix users interact with their systems using command-line tools. These tools typically perform very specialized functions, such as listing files in a directory or displaying files on-screen. The advantage of supplying many specialized tools lies in the way commands can be combined to form more sophisticated commands. For example, a command that lists the contents of a directory can be combined with a program that lists text in "pages" for easy reading.

As you have learned, you use the Terminal application to gain access to Darwin's command-line tools.

The following Try It Out looks at Darwin's command-line interface. You start by browsing files using the command line, looking up command information in Darwin's online help system, and running a command that displays its own arguments.

TRY IT OUT Experiencing Darwin's Command-Line Interface

1. In the Finder, go to Applications ➪ Utilities and launch the Terminal application. You will see a few status lines of text ending in a command-line prompt (your lines may look slightly different from what is shown here):

```
Last login: Sat May 15 23:28:46 on ttys000
Macintosh:~ sample $
```

2. When you're using Terminal, there are commands that let you navigate the file system. The Terminal application always keeps track of where you are, maintaining the notion of your *current directory*. You can display the contents of the current directory using the `ls` (list) command that follows. As a matter of fact, the Terminal window is currently "in" your home directory. Your results may vary from what's printed here, but they will match what you see in the Finder when you browse your home directory. (Throughout this book, any text you are asked to type on the command line is indicated in **bold**.)

```
Macintosh:~ sample $ ls
Desktop         Downloads       Movies          Pictures        Sites
Documents       Library         Music           Public
```

3. You can display more information about the files in your home directory by passing additional arguments, called *flags*, into `ls`. By using `ls -l`, you can build what is often called a *long list*. Again, your results may differ from what is printed here:

```
Macintosh:~ sample$ ls -l
total 0
drwx------+  4 sample  staff   136 Jul 16 01:49 Desktop
drwx------+ 10 sample  staff   340 Jul 22 00:10 Documents
drwx------+  6 sample  staff   204 Jul 21 10:22 Downloads
drwx------+ 31 sample  staff  1054 Jul 16 00:05 Library
drwx------+  3 sample  staff   102 Jul 15 09:19 Movies
drwx------+  4 sample  staff   136 Jul 18 23:34 Music
drwx------+  4 sample  staff   136 Jul 15 09:19 Pictures
drwxr-xr-x+  5 sample  staff   170 Jul 15 09:19 Public
drwxr-xr-x+  5 sample  staff   170 Jul 15 09:19 Sites
```

4. You can view the contents of a specific directory by specifying its name as the argument to `ls`. Note that this argument can co-exist with other flags you might want to use:

```
Macintosh:~ sample$ ls -l Library
total 0
drwx------+ 11 sample  staff   374 Jul 18 23:37 Application Support
drwx------+  2 sample  staff    68 Jul 15 09:19 Assistants
drwx------+  5 sample  staff   170 Jul 15 09:19 Audio
drwx------   4 sample  staff   136 Jul 22 00:12 Autosave Information
drwx------  23 sample  staff   782 Jul 20 23:39 Caches
drwxr-xr-x   6 sample  staff   204 Jul 15 15:44 Calendars
drwx------+  2 sample  staff    68 Jul 15 09:19 ColorPickers
drwx------+  3 sample  staff   102 Jul 15 09:19 Compositions
drwxr-xr-x   3 sample  staff   102 Jul 21 10:35 Cookies
drwx------+  3 sample  staff   102 Jul 15 09:19 Favorites
```

```
drwx------+  9 sample  staff   306 Jul 18 23:37 FontCollections
drwx------+  2 sample  staff    68 Jul 15 09:19 Fonts
drwxr-xr-x   2 sample  staff    68 Jul 16 00:05 Fonts Disabled
drwx------+  3 sample  staff   102 Jul 15 09:19 Input Methods
drwx------+  2 sample  staff    68 Jul 15 09:19 Internet Plug-Ins
drwx------+  2 sample  staff    68 Jul 15 09:19 Keyboard Layouts
drwxr-xr-x   4 sample  staff   136 Jul 18 23:35 Keychains
drwx------   3 sample  staff   102 Jul 15 12:29 Logs
. . .
```

5. Two new questions immediately come to mind: exactly what is `ls -l` telling you, and what other flags can you pass into `ls`? The answer to both of these questions resides in Darwin's online help system, which is better known as the Unix Manual. You can consult the manual by using the `man` command and including the name of another command as the argument:

```
Macintosh:~ sample$ man ls
LS(1)                    BSD General Commands Manual                    LS(1)
NAME
     ls - list directory contents
SYNOPSIS
     ls [-ABCFGHLPRTWZabcdfghiklmnopqrstuwx1] [file ...]
DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.
     If no operands are given, the contents of the current directory are dis-
     played.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted sepa-
     rately and in lexicographical order.
     The following options are available:
     -A      List all entries except for . and ...  Always set for the super-
:
```

6. The arguments you are allowed to pass to a Unix command depend entirely on the command. As you have seen, the `ls` command accepts filenames, and the `man` command accepts the names of other Unix commands. The `echo` command accepts arbitrary arguments and simply repeats them on the screen. It turns out that this command is especially useful when writing shell scripts, as you see in Chapter 11.

```
Macintosh:~ sample$ echo hello, my name is sample
hello, my name is sample
```

How It Works

In spite of appearances, Terminal doesn't understand any of the commands you just entered. In fact, Terminal's only job is to read input from your keyboard and display text coming from a special program called a *shell*. Terminal starts your shell for you when its window appears. The shell is a special program that provides a command-line prompt, parses instructions into command names and lists of arguments, runs the requested commands, and passes back the resulting text.

When the shell has decided which command to launch, the shell starts that command and passes the remaining flags and arguments into the command for further evaluation. That's why `ls`, `man`, and `echo` all interpret their arguments in different ways. Flags are also interpreted by individual commands, so it's not uncommon to use a particular flag in more than one Unix command, although the flag might have different meanings.

One thing to watch out for: Unix shells historically are *case-sensitive*, meaning that the command `LS` is not the same as `ls`, the directory `library` is not the same as the directory `Library`, and so on. Mac OS X's default file system, HFS+, is case-insensitive, and much of the time the shell can figure out what you mean. But if you had some trouble with the commands in the preceding Try It Out, make sure you entered the text exactly as it appears here.

You have only just scratched the surface of what the shell can do. You will continue to learn more about the shell as you continue through the book.

GRAPHICS AND MEDIA LAYERS

Much of the user experience on Mac OS X is built around graphics. All the elements you see on the screen — windows, menus, buttons, and text — are graphics. It comes as no surprise that Mac OS X has several subsystems dedicated to graphics, as shown in Figure 1-4.

Mac OS X provides a rich graphics library for doing two-dimensional drawings, called Quartz 2D. The Quartz 2D library is specific to Mac OS X, although it uses industry-standard graphic formats, such as PDF. Mac OS X also includes OpenGL for those interested in three-dimensional drawings. Although popularized by cross-platform video games, Mac OS X itself uses OpenGL for certain operations. Finally, QuickTime is built into Mac OS X, providing support for what Apple occasionally calls four-dimensional drawing. QuickTime is also available for Microsoft Windows operating systems, and for older versions of Mac OS. All these programming libraries rely on the Quartz Compositor for actually drawing their content.

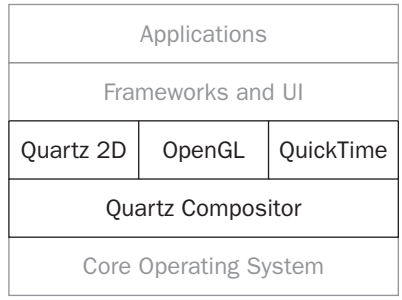

FIGURE 1-4

The following sections look at these subsystems in more detail.

The Quartz Compositor

The Quartz Compositor is a private system service that oversees all graphics operations on Mac OS X. Apple does not provide a means for developers to interact with the Quartz Compositor directly, so we won't look at it in detail here. The Quartz Compositor plays such an important role in Mac OS X's graphic strategy, however, that it pays to understand what it does.

Among its many duties, the Quartz Compositor handles these tasks:

Manages all the windows on your screen — Although the actual look of the window may come from an application or an application framework such as Cocoa or Carbon, the Quartz Compositor provides most of the window's guts: where the window sits on the screen, how the window casts its drop shadow, and so on.

Ensures that graphics are drawn appropriately, regardless of which library or libraries an application may be using — In fact, an application may use commands from Quartz 2D, OpenGL, and QuickTime when drawing a given window. The Quartz Compositor ensures that the drawing reaches the screen correctly.

Collects user events from the core operating system and dispatches them to the Application Frameworks layer — User events such as keystrokes and mouse movements are collected from drivers in the core operating system and sent to the Quartz Compositor. Some of these events are passed along where they may be interpreted by the application. The Quartz Compositor will also send its own special events to the application for responding to special conditions, such as when the user brings the application to the foreground or when a window needs to be updated.

The Quartz Compositor was designed with modern best practices for graphics in mind. For example, the drawing coordinate space uses floating-point values, allowing for sub-pixel precision and image smoothing. Compositing operations can take advantage of available hardware. Transparency is supported natively and naturally in all drawing operations.

Apple has been able to capitalize on this architecture to provide a number of exciting features, such as Quartz Extreme and Exposé. Quartz Extreme allows graphic operations to take full advantage of the graphics processing unit (GPU) found on modern video cards to provide hardware-accelerated drawing. This has two benefits. The GPU is specially optimized for common drawing operations, so drawing is much faster than when using the computer's CPU. Second, by offloading drawing onto the GPU in the video card, Quartz Extreme frees up the CPU for other tasks. Although in the past, developers needed to use OpenGL to do hardware-accelerated drawing, Quartz Extreme provides this support to Quartz 2D as well, and ultimately to QuickTime. Exposé allows the user to quickly view all windows at once. It is a very handy way to find a specific window that might be buried underneath a number of other windows, as shown in Figure 1-5.

The Quartz Compositor is one of the most fundamental parts of Mac OS X. Although you will not be working with it directly in this book, you will feel its influence in almost everything you do.

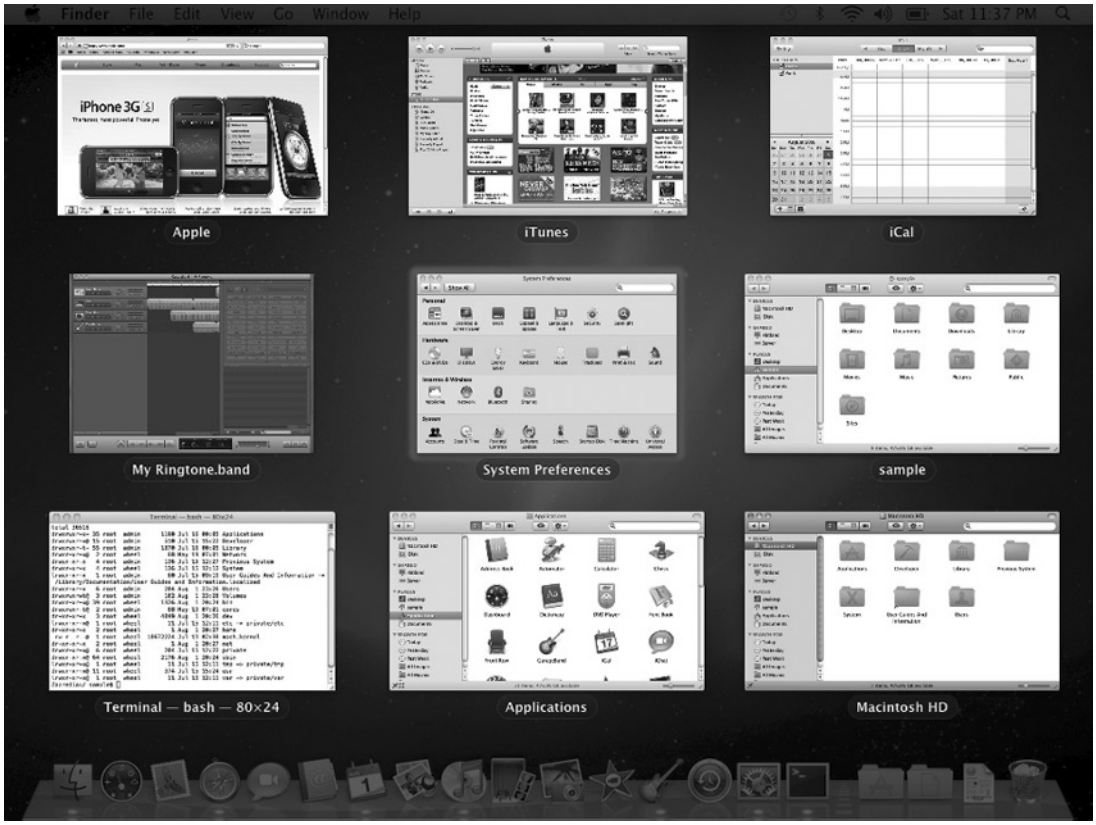

FIGURE 1-5

WHAT ARE PDF FILES?

PDF stands for Portable Document Format. The PDF standard was invented by Adobe as a means for describing documents that can be displayed or printed virtually anywhere. The file specification itself is *open*, meaning the public-at-large can view the format and write their own tools for reading and generating PDF documents. Adobe continues to own and develop the standard.

Mac OS X reads and writes PDF documents as its preferred native image file format. You can save any document in PDF format simply by "printing" it and clicking Save as PDF in Mac OS X's print panel. PDF files can be displayed in Mac OS X's Preview application.

Quartz 2D

The Quartz 2D graphics library is Mac OS X's native graphics library. It is responsible for all the two-dimensional drawing performed by Mac OS X. As you might imagine, Quartz 2D provides an interface for drawing two-dimensional shapes, such as lines and rectangles, and compositing images. It is also capable of drawing sophisticated curves, arbitrary shapes expressed as paths or vectors, and color gradients. Quartz 2D also includes support for generating and displaying PDF files.

The Quartz 2D programming interface is provided by CoreGraphics, which is part of the ApplicationServices framework: `/System/Library/Frameworks/ApplicationServices.framework`. The Quartz 2D API is very powerful and is best approached by an experienced programmer. In this book, you focus more on the drawing API in the Application Frameworks layer, which is a little easier to use.

OpenGL

OpenGL is a powerful, cross-platform graphics library for doing 2D and 3D graphics. Although OpenGL is owned by SGI, the OpenGL specification is governed by an independent consortium called the OpenGL Architecture Review Board — ARB for short. As a voting member of the ARB, Apple contributes to the OpenGL community as a whole, in addition to improving the state of OpenGL on Mac OS X.

One of OpenGL's most compelling features is its tight integration with video card technology. Many OpenGL commands, such as image and shape drawing, blending, and texture-mapping, can be performed directly by the video card's GPU. Recall that the GPU is optimized to perform these operations very quickly, and after graphic operations have been unloaded onto the video card, the CPU is free to perform other computational functions. The net result of this tight integration is very fast drawing.

Performance combined with its cross-platform nature makes OpenGL uniquely suited for certain kinds of situations, including scientific research, professional video editing, and games. If you have played a 3D video game on Mac OS X, you've seen OpenGL in action. For that matter, if you have used one of Mac OS X's built-in screen saver modules, you've seen OpenGL.

OpenGL's programming interface is spread across two frameworks: core OpenGL functionality lives in the OpenGL framework (`/System/Library/Frameworks/OpenGL.framework`), and a basic cross-platform Application Framework called GLUT resides at `/System/Library/Frameworks /GLUT .framework`. As with Quartz 2D, the OpenGL API is fairly advanced and better suited for more experienced programmers.

QuickTime

Apple Computer invented QuickTime back in 1991 as a way to describe, author, and play back video on Macintosh computers running System 6 and System 7. Since then, QuickTime has exploded into a cross-platform library encompassing a variety of multimedia file formats and algorithms. QuickTime provides tools for working with digital video, panoramic images, digital sound, MIDI, and more. It has spawned entire genres of software, including CD-ROM adventure games, digital audio/video editing suites, and desktop video conferencing.

Mac OS X increased Apple's commitment to QuickTime by building it directly into the operating system. Though versions of QuickTime shipped with Mac OS releases since the earliest days of QuickTime, Mac OS X actually relies on QuickTime in ways earlier OS versions did not. For example, Finder uses QuickTime to allow you to preview video and audio files directly in the Finder when using column view. Mac OS X's Internet connectivity apps, including iChat and Safari, make substantial use of QuickTime.

Mac OS X Snow Leopard introduces QuickTime X, integrating QuickTime more tightly into the Mac OS X architecture than before. Although QuickTime has taken advantage of available video hardware resources for years, QuickTime X has been redesigned around the multiple CPUs and powerful programmable GPUs found in current Macintosh computers. QuickTime X also reintroduces some simple editing features into QuickTime Player, so you can make and edit videos without additional software.

The QuickTime X API is supplied by the QTKit framework: `/System/Library/Frameworks/QTKit.framework`. The QuickTime programming interface has undergone nearly 20 years of evolution, and many of its concepts are quite advanced.

Core Animation

Animations can make tasks more appealing or more understandable. When you activate Exposé, all your windows reorganize on your screen with a sweeping animation. When you minimize a document, it flows into the Dock. When using Cover Flow in the Finder or in iTunes, files flip smoothly to and fro.

Mac OS X v10.5 Leopard introduced a new technology called Core Animation to manage common animations. Core Animation takes flat, two-dimensional images called "layers" and basically pushes them around. Layers are drawn using the GPU, freeing up the CPU to manage the actual business of animation. You can use Core Animation to animate a number of individual parameters, such as the layer's position, its angle of rotation (in three-dimensional space), its size, how transparent it is, and so on.

The Core Animation API is part of the QuartzCore framework: `/System/Library/Frameworks/QuartzCore.framework`.

APPLICATION FRAMEWORKS AND UI

All applications rely on common interface elements to communicate with the user. By packaging these elements in a library, an operating system can make sure all applications look and behave the same way. And the more functionality the operating system provides "for free," the less work application developers need to do themselves.

Toward that end, Mac OS X provides a number of application frameworks, as shown in Figure 1-6, upon which programmers can build their applications: Cocoa, Carbon, and the Java JDK. These frameworks, described in more detail in the following sections, all provide the basic concepts essential for application design: how events are processed by the application, how window contents are organized and drawn, how controls are presented to the user, and so on.

It is important that all applications present their user interface (UI) in a consistent manner, regardless of which application framework the program uses. In other words, all windows, menus, buttons, text fields, and so on should look and behave the same way on Mac OS X. These UI elements together on Mac OS X form a distinctive user experience that Apple calls the Aqua user interface. Consistency among apps is so important that Apple has published guidelines enumerating the proper way to use Aqua user interface elements; these guidelines are called the Apple Human Interface Guidelines.

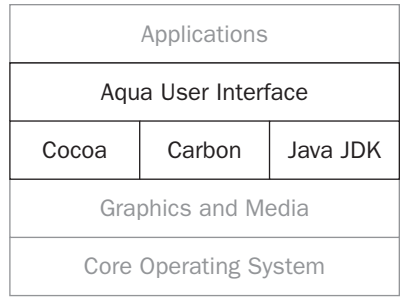

FIGURE 1-6

Each of these application frameworks is appropriate in different situations. In addition, these application frameworks are not mutually exclusive. An application may draw on features from all three frameworks.

Cocoa

The Cocoa application framework provides programmers with a means of building feature-rich Mac OS X applications quickly. The roots of Cocoa lie in NeXTSTEP, the operating system that powered NeXT computers in the early 1990s. When Apple announced Mac OS X in 1998, the API was re-christened Cocoa, and introduced alongside Carbon as Mac OS X's application development strategy.

Cocoa is an object-oriented API written in Objective-C, an object-oriented language descended from ANSI C and Smalltalk. Programmers work with Cocoa by creating objects and hooking them together in various ways. Objects provide a convenient way for programmers to extend basic application functionality without having to design the entire application from the ground up. Put another way, Cocoa allows you to focus on writing the code that makes your application unique, rather than forcing you to write the code that all applications must share.

The Cocoa API is divided between two frameworks:

- **The AppKit framework** (`/System/Library/Frameworks/AppKit.framework`): Provides high-level objects and services for writing applications, including Aqua UI elements.
- **The Foundation framework** (`/System/Library/Frameworks/Foundation.framework`): Provides objects and services useful for all programs, such as collection data types, Unicode string support, and so on.

These features are divided into two separate frameworks so programs can use Foundation's utility classes without having to bring in a full graphical user interface (GUI). For example, a command-line tool written in Objective-C might simply use Foundation.

Carbon

What we know as Carbon today started out as the programmatic interface to the original Macintosh operating system. Although sufficient for writing Macintosh applications, the API had some problems that made transitioning to a new core operating system impossible. In 1998,

Apple set out to revise the traditional Mac OS API and eliminate these problems, which would give existing Macintosh developers an easy path for migrating their code to Mac OS X. This revised API was called Carbon.

It used to be the case that you needed to work with Carbon to do a number of useful things in Mac OS X. For example, programmers interested in working with aliases, customized menus, or QuickTime all needed to use Carbon, even if they were writing a Cocoa application. Many of these things are no longer true in Mac OS X Snow Leopard. Cocoa programmers can now access things either through Cocoa or through specialized frameworks, such as QTKit.

If you are interested in porting a traditional Mac OS application to Mac OS X, Carbon is a good place to start. However, Apple has begun encouraging programmers to move away from Carbon altogether. Many Carbon technologies simply don't play well with modern hardware such as accelerated GPUs or modern software such as Core Animation. Apple has chosen to stop investing in Carbon to spend time on newer, more interesting technology.

The Carbon API is built around a collection of C interfaces, spread across several frameworks, including the Carbon framework (`/System/Library/Frameworks/Carbon.framework`), the Core Services framework (`/System/Library/Frameworks/CoreServices.framework`), and the ApplicationServices framework (`/System/Library/Frameworks/ApplicationServices.framework`). The Carbon framework includes a number of interfaces for working with high-level concepts, such as UI elements, online help, and speech recognition. CoreServices provides interfaces for working with lower-level Carbon data structures and services. ApplicationServices fits somewhere between the other two, building on CoreServices to provide important infrastructure supporting the high-level interfaces in the Carbon framework, such as Apple events, font and type services, and speech synthesis.

Java JDK

Mac OS X comes with built-in support for Java applications. Java is an object-oriented programming language created by Sun Microsystems for developing solid applications that can deploy on a wide variety of machines. Java itself is best thought of as three separate technologies: an object-oriented programming language, a collection of application frameworks, and a runtime environment, as described in the following list:

Java the programming language — Designed to make writing programs as safe as possible. Toward that end, Java shields the programmer from certain concepts that often are a source of trouble. For example, because programmers often make mistakes when accessing memory directly, Java doesn't allow programmers to access memory in that way.

Java the application framework — Provides a number of ways to develop applications using the Java programming language. Java and Cocoa are similar in many ways; for example, many of the objects and concepts in Cocoa also appear in Java.

Java the virtual machine — Provides the runtime environment, called a virtual machine, in which all Java programs live. This virtual machine protects Java programs from subtle differences one encounters when trying to deploy programs on a variety of systems. For

example, different systems may have widely divergent hardware characteristics, supply different kinds of operating system services, and so on. Java Virtual Machine levels the playing field for all Java apps, so that Java programmers do not need to worry about these issues themselves.

Java's greatest strength is that it enables you to easily write applications that are deployable on a wide variety of computers and devices. In this respect, Java has no equal. On the other hand, for the purposes of writing a Mac OS X–specific application, the Java application frameworks have some serious drawbacks. Because Java must deploy on several different computers, Java's approach to application design tends to focus on commonly available technologies and concepts. It is difficult to gain access to features unique to Mac OS X, such as the power of CoreGraphics, through Java's application frameworks, because those features are not available on all Java systems. Because this book focuses on technologies specific to Mac OS X, we will not examine Java in further detail.

APPLE HUMAN INTERFACE GUIDELINES

All Mac OS X programs share a specific look and feel that makes them instantly recognizable as Mac OS X programs. This creates the illusion that all the applications on your system were designed to work together — even though your applications may have been designed by different people, all with different interests. After you learn how to use one application, you have a pretty good idea of how to use all applications.

Apple provides a document, called the Apple Human Interface Guidelines, which spells out how Mac OS X applications should look and behave. Applications written against one of Mac OS X's application frameworks start with a bit of an advantage: all the UI elements provided by these frameworks meet the specifications in the Apple Human Interface Guidelines. All the controls in Figure 1-7 are drawn using the Cocoa application framework; notice that they all look like Mac OS X controls.

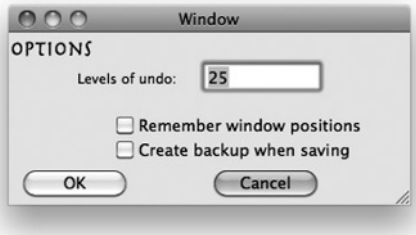

FIGURE 1-7

Unfortunately, simply using the right controls isn't enough to make an Aqua-compliant interface. A large part of UI design is in collecting and organizing controls so they make sense. The Apple Human Interface Guidelines provide metrics for how far apart related controls and groups of controls should be and where certain kinds of controls should go. The Aqua guidelines specify specific fonts and font sizes for UI elements. It also specifies when certain features are appropriate, such as default buttons, hierarchical menu items, and so on. Figure 1-8 illustrates the same controls from Figure 1-7, laid out in compliance with the Apple Human Interface Guidelines — note that it looks much cleaner.

The information in the Apple Human Interface Guidelines is quite extensive. It covers all the user interface elements available within Mac OS X, such as windows, menus, controls, separators, text labels, and icons. All Mac OS X programmers should be familiar with the Apple Human Interface Guidelines to know what correct Aqua user interfaces are supposed to look like, and how they're supposed to behave.

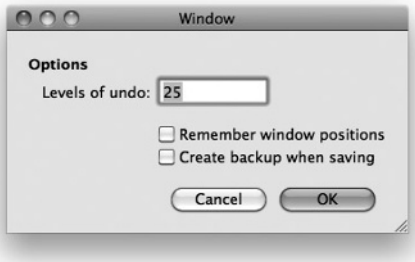

FIGURE 1-8

SUMMARY

You have seen how the major elements of Mac OS X come together on your computer. The applications you use every day are but one element. These applications are built on application frameworks, system services, and ultimately Mac OS X's core operating system; all these pieces contribute to your application experience. The high-level picture might look similar to Figure 1-9.

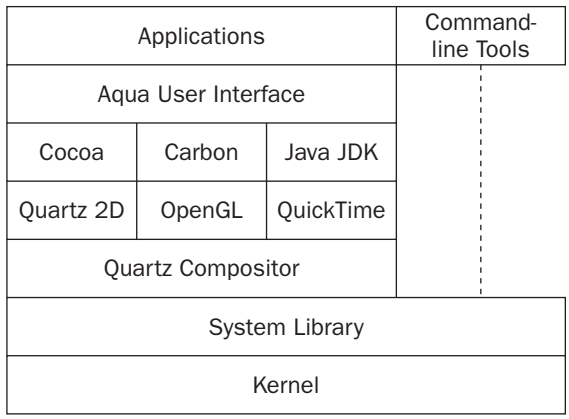

FIGURE 1-9

In the next chapter, you learn about the developer resources bundled with Mac OS X. These include tools used during the development process, as well as online documentation and other resources. Before proceeding, you can use the exercises that follow to practice some of the things you learned in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. The `apropos` command returns a list of manual pages that match one or more keywords. Try entering the following commands into Terminal:

- **a.** apropos copy
- **b.** apropos copy file
- **c.** apropos "copy file"

Which of these commands provides the best result?

2. You have seen how you can use `man` to read the online help for a specific command. Type `man man` into Terminal and read about what `man` is capable of. For example, what does `man -k "copy file"` do?

▶ WHAT YOU LEARNED IN THIS CHAPTER

Kernel	the heart of the Core OS, responsible for talking to hardware and running programs
System Library	API for "talking to" the Core OS
Quartz Compositor	the process responsible for all application drawing and event handling
Quartz 2D	API for drawing rich 2D graphics
OpenGL	an open, standard API for drawing hardware accelerated 2D and 3D graphics
QuickTime	a framework for reading and creating multimedia files
CoreAnimation	a framework for animating user interfaces and other application content
Cocoa	a collection of frameworks used for writing Mac OS X applications using the Objective-C programming language
Carbon	a collection of frameworks used for older Mac OS and Mac OS X applications