User Interfaces: The Application Kit

On Mac OS X, the Application Kit (AppKit.framework) contains all the primary elements you’ll need to create a feature-rich graphical application. There are others, of course, but everything has its roots in the AppKit. It’s here the concepts of windows, views, controls, and input events are defined, along with the core Objective-C APIs for dealing with rendering of graphics and text.

This chapter will give you a thorough grounding in the concepts embodied within the Application Kit and its companion tool, Interface Builder. As a result, this chapter will not deal with any iOS or UIKit topics except noting contrasts between the two platforms’ respective approaches. The information here will still be useful to iOS developers, however, as iOS uses the same core concepts listed above; it just differs in its implementation. You’ll have enough knowledge of the components of graphical interfaces that you should be able to pick up iOS user interface programming with little difficulty.

I would also like to point out that this chapter is going to be predominantly based on the theory behind user interfaces. The practical part will arrive in Chapter 9 where I will guide you through the creation of a full application using the techniques you’ve learned in the book so far.

Coding Practices: Model-View-Controller

Deeply enshrined in all of Apple’s user interface code is the use of the model-view-controller programming paradigm. This defines a means by which classes that manage data are encapsulated separately from those that manage the presentation of that data, with a controller object acting as an intermediary between the two.

The idea is to enforce a strong and rigid boundary between a data’s form and its representation to the user; either ought to be able to change relatively independently of the other. Under no circumstances should any user interface code need to know about the structure or storage of data in order to present it to the user, and likewise data should not need to be aware of its
presentation context. It is the job of the controller object to deal with both of these classes: when the presentation object needs some data, it requests it from the controller. The controller knows about the data object’s format, so it can fetch it and massage it into the format required by the presentation object. Similarly, if the data object is modified and the user interface needs to be updated, it is the controller that observes the change and informs the presentation object in an appropriate manner.

An example of this abstraction can be seen in the `NSWindow` and `NSWindowController` classes. `NSWindow` defines an interface with an actual window, along with all the means of setting its attributes and content as well as receiving information about changes to that window. The window does not, however, have any intrinsic methods for interacting with any form of data whatsoever; something has to manually tell the window what it needs to do in order to represent that data. As a result, `NSWindow` is completely data-agnostic and can effectively be used for anything you might desire.

The `NSWindowController` object is a companion class that provides you with much of the structure required to load and initialize a window. It understands interface builder design documents and can create a window from the instructions therein. It also keeps track of the window for you, and it is this class that you would extend to implement the specifics of your application.

Thus it is that, while some libraries\(^1\) would have you subclass a `Window` object to create a `PreferenceWindow` object that would lay itself out and handle user input and such, when using the AppKit you’ll virtually never need to subclass the `NSWindow` object yourself. You’ll instead subclass `NSWindowController` to manage the window and any associated labels, text fields, buttons, and more. This paradigm exists all through the AppKit classes, even in data-specific views such as tables or lists. The view objects typically know about some types of atomic data (numbers, strings); how to lay out subviews particular to their use case; and how to request the actual data objects from an associated controller (or data source) object. You will see this later when you learn about table views, for example.

### Windows, Panels, and Views

On OS X, a window, as represented by the `NSWindow` class, refers to a single window onscreen. Specifically, a window is the top-level graphical element within an application, which contains various other elements. Figure 7-1 shows a window from the Finder application. All a window’s contents will move with it, but separate windows can be manipulated individually.

---

\(^1\)I’m thinking primarily of the C++ Microsoft Foundation Classes and the original Java Foundation Classes when I say this.