The Role of View Controllers

View controllers are the foundation of your app's internal structure. Every app has at least one view controller, and most apps have several. Each view controller manages a portion of your app's user interface as well as the interactions between that interface and the underlying data. View controllers also facilitate transitions between different parts of your user interface.

뷰 컨트롤러는 앱 구조의 기초를 구성합니다. 모든 앱에는 적어도 한개 이상의 뷰 컨트롤러를 가지게 되며, 대부분의 앱은 여러개를 가지게 될 것입니다. 각 뷰 컨트롤러는 여러분의 앱 인터페이스의 일부분을 담당할 뿐만이 아니라 인터페이스와 기본 데이터간의 상호작용을 관리합니다. 뷰 컨트롤러는 또한 사용자 인터페이스 간의 전환을 손 쉽게 합니다.

Because they play such an important role in your app, view controllers are at the center of almost everything you do. The `UIViewController` class defines the methods and properties for managing your views, handling events, transitioning from one view controller to another, and coordinating with other parts of your app. You subclass `UIViewController` (or one of its subclasses) and add the custom code you need to implement your app's behavior.

뷰 컨트롤러는 앱에서 중요한 역할을 수행합니다. `UIViewController` 클래스는 뷰 관리, 이벤트 처리, 다른 뷰 컨트롤러로의 전환, 앱의 다른 부분과의 조정을 위한 메소드들과 프로퍼티들을 정의합니다. 여러분은 클래스(또는 그 하위 클래스)를 서브클래싱하고 앱의 특정기능 구현을 위한 코드를 작성하게 될 것입니다.

There are two types of view controllers:

아래에 뷰 컨트롤러의 두 가지 타입에 대한 설명이 있습니다.

Content view controllers manage a discrete piece of your app's content and are the main type of view controller that you create.
컨텐트 뷰 컨트롤러는 앱의 여러분의 앱 컨텐츠의 개별적인 부분을 관리하며 여러분이 만든 뷰 컨트롤러의 기본 타입입니다.

Container view controllers collect information from other view controllers (known as *child view controllers*) and present it in a way that facilitates navigation or presents the content of those view controllers differently.

컨테이너 뷰 컨트롤러는 다른 뷰 컨트롤러의 자식 뷰 컨트롤러라고 부릅니다) 의 정보를 수집하고 뷰 컨트롤러의 탐색을 쉽게하거나 컨텐츠를 다르게 표시하는 방식을 제공합니다.

Most apps are a mixture of both types of view controllers.

대부분의 앱은 두 가지 타입의 뷰 컨트롤러가 혼합되어 구성됩니다.

View Management

The most important role of a view controller is to manage a hierarchy of views. Every view controller has a single root view that encloses all of the view controller's content. To that root view, you add the views you need to display your content. Figure 1-1 illustrates the built-in relationship between the view controller and its views. The view controller always has a reference to its root view and each view has strong references to its subviews.

뷰 컨트롤러의 가장 중요한 역할은 뷰의 계층을 관리하는 것입니다. 모든 뷰 컨트롤러는 모든 컨텐츠가 들어있는 하나의 루트 뷰를 가지고 있습니다. 여러분은 화면에 나타내기 위한 뷰 들을 이 루트 뷰에 추가하게 될 것입니다. 아래의 그림 1-1에 뷰 컨트롤러의 뷰 컨트롤러가 가진 뷰 들간의 관계를 설명하였습니다. 뷰 컨트롤러는 항상 루트 뷰에 대한 참조를 가지고 있으며 각 뷰는 하위 뷰들에 대한 강한 참조를 가지고 있습니다.

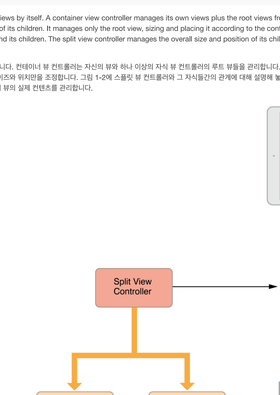

Figure 1-1 Relationship between a view controller and its views

> NOTE
> It is common practice to use outlets to access other views in your view controller's view hierarchy. Because a view controller manages the content of all its views, outlets let you store references to the views that you need. The outlets themselves are connected to the actual view objects automatically when the views are loaded from the storyboard.
> outlets 를 사용하여 여러분의 뷰 컨트롤러의 계층에 다른 뷰 들을 액세스하는 것이 일반적인 방법입니다. 뷰 컨트롤러는 계층속의 모든 뷰들의 컨텐츠를 관리하기 때문에 outlets 를 통해 여러분이 필요한 뷰에 대한 참조를 저장할 수 있습니다. 뷰들이 스토리보드에서 로드될 때 아웃렛은 자동으로 실제 뷰 객체들과 연결되게 됩니다.

A content view controller manages all of its views by itself. A container view controller manages its own views plus the root views from one or more of its child view controllers. The container does not manage the content of its children. It manages only the root view, sizing and placing it according to the container's design. Figure 1-2 illustrates the relationship between a split view controller and its children. The split view controller provides the views and size and position of its child views, but the child view controllers manage the actual contents of those views.

컨테이너 뷰 컨트롤러는 모든 뷰들을 단독으로 관리합니다. 컨테이너 뷰 컨트롤러는 자식의 뷰와 하나 이상의 자식 뷰 컨트롤러의 루트 뷰들을 관리합니다. 컨테이너 뷰 컨트롤러는 자식의 컨텐츠를 관리하지는 않고 오직 루트 뷰 만관리하고 디자인에 따라 사이즈와 위치만을 조정합니다. 그림 1-2에 스플릿 뷰 컨트롤러와 그 자식들간의 관계에 대해 설명해 놓았습니다. 스플릿 뷰 컨트롤러는 자식 뷰의 전체인 크기와 위치를 관리하지만, 자식 뷰 컨트롤러는 자기 뷰의 실제 컨텐츠를 관리합니다.

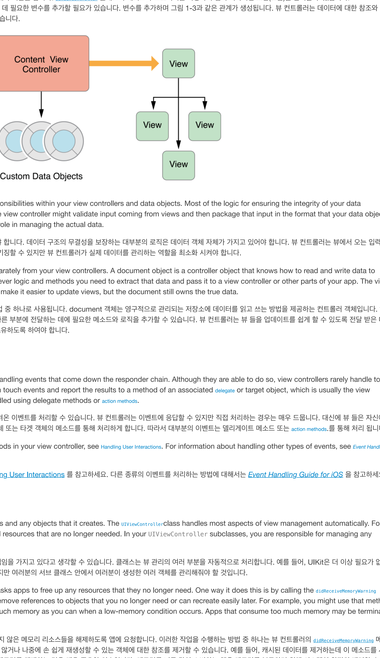

Figure 1-2 View controllers can manage content from other view controllers

For information about managing your view controller's views, see Managing View Layout.

뷰 컨트롤러의 관리에 대한 자세한 정보는 Managing View Layout 를 참고하세요

Data Marshaling

A view controller acts as an intermediary between the views it manages and the data of your app. The methods and properties of the `UIViewController` class let you manage the visual presentation of your app. When you subclass `UIViewController`, you add any variables you need to manage your data in your subclass. Adding custom variables creates a relationship like the one in Figure 1-3, where the view controller has references to your data and to the views used to present that data. Moving data back and forth between the two is your responsibility.

뷰 컨트롤러는 앱의 데이터와 관리되어지고 있는 뷰 들간의 중개자 역할을 합니다. `UIViewController` 클래스의 메서드와 프로퍼티를 사용하여 앱의 시각적인 표현(모습)을 관리할 수 있습니다. `UIViewController` 를 서브 클래싱할 때, 데이터를 관리하는 데 필요한 변수들 추가할 필요가 있습니다. 변수를 추가하여 그림 1-3과 같은 관계가 생성됩니다. 뷰 컨트롤러는 데이터에 대한 참조와 데이터를 표시하는데 필요한 뷰들을 가지고 있는 것을 확인할 수 있습니다.

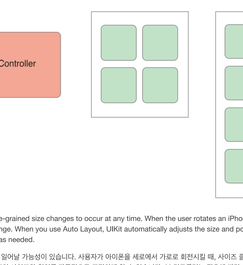

Figure 1-3 A view controller mediates between data objects and views

You should always maintain a clean separation of responsibilities within your view controllers and data objects. Most of the logic for ensuring the integrity of your data structures belongs in the data objects themselves. The view controller might validate input coming from views and then package that input in the format that your data objects require, but you should minimize the view controller's role in managing the actual data.

항상 뷰 컨트롤러와 데이터 객체간의 책임을 완전히 분리시켜야 합니다. 데이터 구조의 무결성을 보장하는 대부분의 로직은 데이터 객체 자체가 가지고 있어야 합니다. 뷰 컨트롤러는 뷰에서 오는 입력을 확인한 다음 데이터 객체에 필요한 형식으로 받아온 입력을 패키징할 수 있지만 뷰 컨트롤러가 실제 데이터를 관리하는 역할을 최소화 시켜야 합니다.

A `UIDocument` object is one way to manage your data separately from your view controllers. A document object is a controller object that knows how to read and write data to persistent storage. When you subclass, you add whatever logic and methods you need to extract that data and pass it to a view controller or other parts of your app. The view controller might store a copy of any data it receives to make it easier to update views, but the document still owns the true data.

`UIDocument` 객체는 뷰 컨트롤러와 별도로 데이터를 관리하는 방법 중 하나로 사용됩니다. document 객체는 영구적으로 관리되는 저장소에 데이터를 읽고 쓰는 방법을 제공하는 컨트롤러 객체입니다. 하위 클래스를 만들 때, 데이터를 추출하고 뷰 컨트롤러 또는 앱의 다른 부분에 전달하는 데에 필요한 메소드와 로직을 추가할 수 있습니다. 뷰 컨트롤러는 뷰 들을 업데이트를 쉽게 할 수 있도록 전달 받은 데이터의 사본을 저장할 수 있지만, 실제 데이터는 document가 소유하도록 하여야 합니다.

User Interactions

View controllers are responder objects and are capable of handling events that come down the responder chain. Although they are able to do so, view controllers rarely handle touch events directly. Instead, views usually handle their own touch events and report the results to a method of an associated delegate or target object, which is usually the view controller. So most events in a view controller are handled using delegate methods or action methods.

뷰 컨트롤러는 responder 객체이고 리스폰더 체인을 통해 내려온 이벤트를 처리할 수 있습니다. 뷰 컨트롤러는 이벤트에 응답할 수 있지만 직접 처리하는 경우는 매우 드뭅니다. 대신에 뷰 들은 자신에게 온 터치 이벤트를 처리하고 그 결과를 델리게이트와 연결된 객체 또는 타겟 객체의 메소드를 통해 처리하게 합니다. 따라서 대부분의 이벤트는 델리게이트 메소드 또는 action methods 를 통해 처리 됩니다.

For more information about implementing action methods in your view controller, see Handling User Interactions. For information about handling other types of events, see *Event Handling Guide for iOS*.

뷰 컨트롤러의 액션 메소드 구현에 대한 자세한 정보는 Handling User Interactions 를 참고하세요. 다른 종류의 이벤트를 처리하는 방법에 대해서는 *Event Handling Guide for iOS* 을 참고하세요.

Resource Management

A view controller assumes all responsibility for its views and any objects that it creates. The `UIViewController` class handles most aspects of view management automatically. For example, UIKit automatically releases any view-related resources that are no longer needed. In your `UIViewController` subclasses, you are responsible for managing any objects you create explicitly.

뷰 컨트롤러는 자신이 생성한 뷰 들과 다른 객체에 대한 모든 책임을 가지고 있다고 생각할 수 있습니다. 클래스는 뷰 관리의 여러 부분을 자동적으로 처리합니다. 예를 들어, UIKit은 더 이상 필요가 없어진 뷰와 관련된 리소스 자원들을 자동적으로 해제시켜 줍니다. 하지만 여러분이 서브 클래스 안에서 여러분이 생성한 여러 객체를 관리해야야 할 것입니다.

When the available free memory is running low, UIKit asks apps to free up any resources that they no longer need. One way it does this is by calling the `didReceiveMemoryWarning` method of your view controllers. Use that method to remove references to objects that you no longer need or can recreate easily later. For example, you might use that method to remove cached data. It is important to release as much memory as you can when a low-memory condition occurs. Apps that consume too much memory may be terminated outright by the system to recover memory.

사용 가능한 메모리 영역이 부족할 때, UIKit은 더 이상 필요하지 않은 메모리 리소스들을 해제하도록 앱에 요청합니다. 이러한 작업을 수행하는 방법 중 하나는 뷰 컨트롤러의 `didReceiveMemoryWarning` 메소드를 호출하는 것입니다. 이 메소드를 사용하여 더 이상 필요하지 않거나 나중에 손 쉽게 재생성할 수 있는 객체에 대한 참조를 제거할 수 있습니다. 예를 들어, 캐시된 데이터를 제거하는데 이 메소드를 사용할 수 있을 것입니다. 메모리 가용량이 부족할 때 최대한 많은 메모리를 해제하는 것은 매우 중요한 이슈입니다. 메모리를 너무 많이 소비하는 앱은 메모리를 복구하기 위해 시스템이 완전히 제거할 수 있습니다.

Adaptivity

View controllers are responsible for the presentation of their views and for adapting that presentation to match the underlying environment. Every iOS app should be able to run on iPad and on several different sizes of iPhone. Rather than provide different view controllers and view hierarchies for each device, it is simpler to use a single view controller that adapts its views to the changing space requirements.

뷰 컨트롤러는 뷰 들의 프리젠테이션을 담당하고 기본 환경과 일치하도록 조정합니다. 모든 iOS 앱은 iPad 그리고 여러 사이즈를 가진 iPhone에서 사용할 수 있습니다. 각 디바이스에 각기 다른 뷰 컨트롤러, 뷰 계층을 제공하는 것보단 변화되는 크기에 맞춰 뷰 들을 조정하는 하나의 뷰 컨트롤러를 사용하는 것이 더 간단할 것입니다.

In iOS, view controllers need to handle coarse-grained changes and fine-grained changes. Coarse-grained changes happen when a view controller's traits change. Traits are attributes that describe the overall environment, such as the display scale. Two of the most important traits are the view controller's horizontal and vertical size classes, which indicate how much space the view controller has in the given dimension. You can use size classes changes to change the way you lay out your views, as shown in Figure 1-4. When the horizontal size class is *regular*, the view controller takes advantage of the extra horizontal space to arrange its content. When the horizontal size class is *compact*, the view controller arranges its content vertically.

iOS에서 뷰 컨트롤러는 coarse-grained(많은 변경)과 fine-grained(미세한 변경) 모두를 처리해야 합니다. Coarse-grained 변경은 뷰 컨트롤러의 trait이 변경될 때 일어난다. Traits란 디스플레이 배율과 같은 전반적인 환경을 설명한 속성을 의미합니다. 중요한 특징 중 두 가지는 뷰 컨트롤러의 가로(horizontal)와 세로(vertical)사이즈 클래스로 뷰 컨트롤러가 주어진 차원에 얼마만큼의 공간을 차지하는지 나타냅니다. 그림 1-4에서처럼 사이즈 클래스를 사용하여 뷰의 레이아웃이 변경되는 방식을 조정할 수 있습니다. 가로 사이즈 클래스가 *regular*일 때, 뷰 컨트롤러는 더 넓은 가로 공간을 사용하여 컨텐츠들을 정렬할 수 있습니다. 가로 사이즈 클래스가 compact일 때, 뷰 컨트롤러는 컨텐츠를 세로로 정렬합니다.

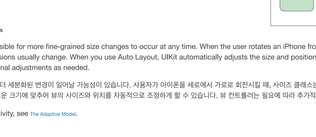

Figure 1-4 Adapting views to size class changes

Within a given size class, it is possible for more fine-grained size changes to occur at any time. When the user rotates an iPhone from portrait to landscape, the size class might not change but the screen dimensions usually change. When you use Auto Layout, UIKit automatically adjusts the size and position of views to match the new dimensions. View controllers can make additional adjustments as needed.

주어진 사이즈 클래스내에서 언제든지 더 세분화된 변경이 일어날 가능성이 있습니다. 사용자가 아이폰을 세로에서 가로로 회전시킬 때, 사이즈 클래스는 변경되지 않지만 화면 크기는 변경될 것입니다. 오토레이아웃을 사용한다면 UIKit은 새로운 크기에 맞추어 뷰의 사이즈와 위치를 자동적으로 조정하게 할 수 있습니다. 뷰 컨트롤러는 필요에 따라 추가적인 조정을 할 수 있습니다.

For more information about adaptivity, see The Adaptive Model.