Defining Your Subclass

You use custom subclasses of UIViewController to present your app's content. Most custom view controllers are *content view controllers*—that is, they own all of their views and are responsible for the data in those views. By contrast, a *container view controller* does not own all of its views; some of its views are managed by other view controllers. Most of the steps for defining content and container view controllers are the same and are discussed in the sections that follow.

UIViewController을 서브 클래싱하여 여러분의 앱의 컨텐츠를 제공할 수 있습니다. 대부분의 커스텀 뷰 컨트롤러들은 컨텐츠 뷰 컨트롤러입니다. 즉, 모든 뷰 들을 소유하고 뷰가 사용할 데이터들을 책임집니다. 이와는 반대로 컨테이너 뷰 컨트롤러는 모든 뷰 들을 소유하지 않으며, 몇몇 뷰 들은 다른 뷰 컨트롤러에 의해 관리됩니다. 컨텐츠 뷰 컨트롤러와 컨테이너 뷰 컨트롤러를 정의하는 대부분의 과정은 동일하며 아래 섹션에서 설명하겠습니다.

For content view controllers, the most common parent classes are as follows:

컨텐츠 뷰 컨트롤러의 경우, 일반적인 부모 클래스들은 아래와 같습니다.

- Use `UITableViewController` specifically when your view controller's main view is a table.

  여러분의 뷰 컨트롤러의 메인 뷰가 테이블 뷰 일때 `UITableViewController`을 이용하세요.

- Use `UICollectionViewController` specifically when your view controller's main view is a collection view.

  여러분의 뷰 컨트롤러의 메인 뷰가 컬렉션 뷰 일때, `UICollectionViewController`을 이용하세요.

- Use `UIViewController` for all other view controllers.

  위 2개의 뷰 컨트롤러를 제외한 다른 모든 뷰 컨트롤러는 `UIViewController`을 이용하세요.

For container view controllers, the parent class depends on whether you are modifying an existing container class or creating your own. For existing containers, choose whichever view controller class you want to modify. For new container view controllers, you usually subclass `UIViewController`. For additional information about creating a container view controller, see Implementing a Container View Controller.

컨테이너 뷰 컨트롤러의 경우, 부모 뷰 컨트롤러 클래스는 기존의 컨테이너 뷰 컨트롤러 클래스를 수정할 것인지 새로 만들 것인지에 따라 달라집니다. 전자의 경우 기존 컨테이너 뷰 컨트롤러를 사용하고, 후자의 경우(새로운 컨테이너 뷰 컨트롤러를 만드는 경우)는 대개 `UIViewController`를 서브클래싱하여 사용하게 됩니다. 컨테이너 뷰 컨트롤러를 만드는 더 자세한 방법은 Implementing a Container View Controller을 참고해주세요.

Defining Your UI

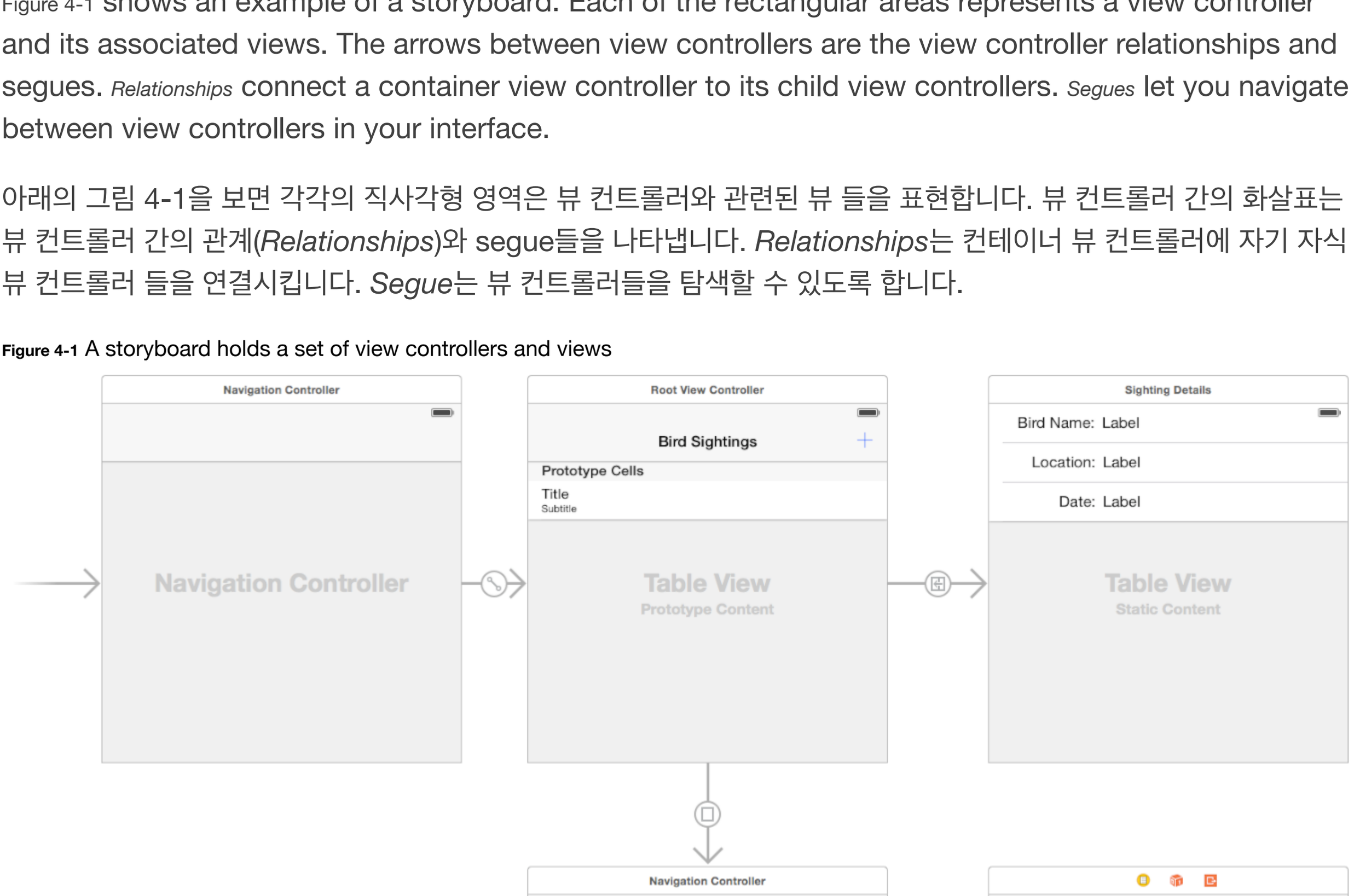
Define the UI for your view controller visually using storyboard files in Xcode. Although you can also create your UI programmatically, storyboards are an excellent way to visualize your view controller's content and customize your view hierarchy (as needed) for different environments. Building your UI visually lets you make changes quickly and lets you see the results without having to build and run your app.

스토리보드를 이용하여 뷰 컨트롤러를 정의하세요. 비록 코드로 UI를 만들 수 있으나, 스토리보드를 이용하는 것은 다양한 환경에서 필요에 따라 뷰 컨트롤러의 컨텐츠를 시각화하고 뷰 계층을 커스터마이징하는데 좋은 방법입니다. UI를 시각적으로 만들면 빠르게 변경할 수 있고 앱을 빌드하고 실행하지 않아도 그 결과를 볼 수 있습니다.

Figure 4-1 shows an example of a storyboard. Each of the rectangular areas represents a view controller and its associated views. The arrows between view controllers are the view controller relationships and segues. *Relationships* connect a container view controller to its child view controllers. *Segues* let you navigate between view controllers in your interface.

아래의 그림 4-1을 보면 각각의 직사각형 영역은 뷰 컨트롤러와 관련된 뷰 들을 표현합니다. 뷰 컨트롤러 간의 화살표는 뷰 컨트롤러 간의 관계(*Relationships*)와 segue들을 나타냅니다. *Relationships*는 컨테이너 뷰 컨트롤러에 자기 자식 뷰 컨트롤러 들을 연결시킵니다. *Segue*는 뷰 컨트롤러들을 탐색할 수 있도록 합니다.

Figure 4-1 A storyboard holds a set of view controllers and views



Each new project has a main storyboard that typically contains one or more view controllers already. You can add new view controllers to your storyboard by dragging them from the library to your canvas. New view controllers do not have an associated class initially, so you must assign one using the Identity inspector.

각 프로젝트에는 일반적으로 하나 이상의 뷰 컨트롤러가 포함된 메인 스토리보드를 가지고 있습니다. 여러분은 스토리보드의 라이브러리에서 캔버스로 드래깅하여 새로운 뷰 컨트롤러를 추가 할 수 있습니다. 새 뷰 컨트롤러를 만들면 연관된 클래스가 정의되어 있지 않으므로 Identity inspector를 이용해 클래스를 지정해주어야 합니다.

Use the storyboard editor to do the following:

스토리보드를 이용하여 아래와 같은 작업을 수행하세요.

- Add, arrange, and configure the views for a view controller.

  뷰 컨트롤러에 대한 뷰 들을 추가, 정렬, 구성하세요.

- Connect outlets and actions; see Handling User Interactions.

  아웃렛과 액션들을 연결하세요. 자세한 내용은 Handling User Interactions을 참고하세요.

- Create relationships and segues between your view controllers; see Using Segues.

  뷰 컨트롤러들 간의 관계와 segues을 만드세요. 자세한 내용은 Using Segues을 참고하세요.

- Customize your layout and views for different size classes; see Building an Adaptive Interface.

  다양한 사이즈 클래스에서 레이아웃과 뷰 들을 커스터마이징 하세요. 자세한 내용은 Building an Adaptive Interface을 참고하세요.

- Add gesture recognizers to handle user interactions with views; see *Event Handling Guide for UIKit Apps*.

  제스처 recognizer를 추가하여 뷰가 사용자의 인터랙션을 처리할 수 있게 하세요. 자세한 내용은 *Event Handling Guide for UIKit Apps*을 참고하세요.

If you are new to using storyboards to build your interface, you can find step-by-step instructions for creating a storyboard-based interface in *Start Developing iOS Apps Today*.

만약 여러분이 스토리보드를 이용하는 것이 처음이라면, "*Start Developing iOS Apps Today*" 에서 스토리보드를 기반으로 한 인터페이스를 만드는 것에 대한 방법을 단계별로 찾아 볼 수 있습니다.

Handling User Interactions

An app's responder objects process incoming events and take appropriate actions. Although view controllers are responder objects, they rarely process touch events directly. Instead, view controllers typically handle events in the following ways.

앱의 리스폰더 객체는 들어온 이벤트를 처리하고 적절한 액션을 수행하는 조치를 취합니다. 뷰 컨트롤러 자체가 리스폰더 객체이긴 하지만 터치 이벤트를 직접 처리하는 경우는 거의 없습니다. 대신에, 뷰 컨트롤러 들은 일반적으로 아래와 같은 방법들로 이벤트를 처리합니다.

- **View controllers define action methods for handling higher-level events.** Action methods respond to:

  뷰 컨트롤러는 higher-level 이벤트를 처리하기 위한 액션 메소드(Action methods)를 정의하였습니다. Action methods가 응답하는 것들로는

  - *Specific actions.* Controls and some views call an action method to report specific interactions.

    특정 액션. 컨트롤과 일부 뷰 들은 특정 인터랙션을 알려주는 액션 메소드를 호출합니다.

  - *Gesture recognizers.* Gesture recognizers call an action method to report the current status of a gesture. Use your view controller to process status changes or respond to the completed gesture.

    Gesture recognizer(제스처 인식기). Gesture recognizer는 액션 메소드를 호출하여 제스처의 현재 status를 알려줍니다. 뷰 컨트롤러를 사용하여 status의 변화를 처리하거나 완료된 제스처에 응답하게 하세요.

- **View controllers observe notifications sent by the system or other objects.** Notifications report changes and are a way for the view controller to update its state.

  뷰 컨트롤러 들은 시스템 또는 다른 객체가 보낸 notification을 받을 수 있도록 관찰합니다. Notification은 특정 변화를 알려주고 뷰 컨트롤러의 상태를 업데이트 해주는 방법중 하나로 사용됩니다.

- **View controllers act as a data source or delegate for another object.** View controllers are often used to manage the data for tables, and collection views. You can also use them as a delegate for an object such as a `CLLocationManager` object, which sends updated location values to its delegate.

  뷰 컨트롤러 들은 다른 객체의 데이터 소스 또는 델리게이트의 역할을 수행합니다. 뷰 컨트롤러 들은 종종 테이블 뷰, 컬렉션 뷰의 데이터를 관리하는데 사용됩니다. 여러분은 위치를 업데이트한 내용을 해당 델리게이트에 보내는 `CLLocationManager` 객체의 델리게이트(위임자, 대리자)로 사용할 수 있습니다.

Responding to events often involves updating the content of views, which requires having references to those views. Your view controller is a good place to define outlets for any views that you need to modify later. Declare your outlets as properties using the syntax shown in Listing 4-1. The custom class in the listing defines two outlets (designated by the `IBOutlet` keyword) and a single action method (designated by the `IBAction` return type). The outlets store references to a button and a text field in the storyboard, while the action method responds to taps in the button.

이벤트에 응답하는 것은 뷰의 컨텐츠를 업데이트 해야하고 뷰에 대한 참조를 가지고 있어야 한다는 것을 의미합니다. 뷰 컨트롤러는 나중에 수정할 필요가 있는 뷰에 대한 outlets을 정의하는데 매우 좋습니다. 아래의 Listing 4-1에 표시된 구문을 이용하여 아웃렛을 정의하세요. 아래의 커스텀 클래스는 두 개의 아웃렛(`IBOutlet`키워드로 지정된)과 하나의 액션 메소드(`IBAction`리턴 타입을 지정된)를 정의하였습니다. 아웃렛은 스토리보드의 버튼과 텍스트필드에 대한 참조를 가지고 있는 반면 액션 메소드는 버튼의 탭이벤트에 반응합니다.

Listing 4-1 Defining outlets and actions in a view controller class

OBJECTIVE-C

```objc
@interface MyViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIButton *myButton;
@property (weak, nonatomic) IBOutlet UITextField *myTextField;
- (IBAction)myButtonAction:(id)sender;
@end
```

SWIFT

```swift
class MyViewController: UIViewController {
    @IBOutlet weak var myButton : UIButton!
    @IBOutlet weak var myTextField : UITextField!
    @IBAction func myButtonAction(sender: id)
}
```

In your storyboard, remember to connect your view controller's outlets and actions to the corresponding views. Connecting outlets and actions in your storyboard file ensures that they are configured when the views are loaded. For information about how to create outlet and action connections in Interface Builder, see Interface Builder Connections Help. For information about how to handle events in your app, see *Event Handling Guide for UIKit Apps*.

스토리보드에서 뷰 컨트롤러의 아웃렛과 액션에 적절한 뷰와 연결해야 하는 것을 잊지마세요. 스토리보드에서 아웃렛과 액션을 연결하는 것은 뷰가 로드될 때 적절히 구성될 수 있게합니다. Interface Builder에서 아웃렛과 액션을 생성하는 방법에 대한 정보는 Interface Builder Connections 도움말을 참조해주세요. 앱에서 이벤트를 처리하는 방법에 대한 설명은 *Event Handling Guide for UIKit Apps*을 참고해주세요.

Displaying Your Views at Runtime

Storyboards make the process of loading and displaying your view controller's views very simple. UIKit automatically loads views from your storyboard file when they are needed. As part of the loading process, UIKit performs the following sequence of tasks:

스토리보드는 뷰 컨트롤러의 뷰 들을 로드하고 나타내는 것을 매우 간단하게 만들어 줍니다. UIKit은 필요할 때 자동적으로 스토리보드에서 뷰를 로드합니다. 로딩 프로세스의 일부로 UIKit은 아래와 같은 작업을 수행합니다.

1. Instantiates views using the information in your storyboard file.

   스토리보드파일의 정보를 이용하여 뷰 들을 인스턴스화 합니다.

2. Connects all outlets and actions.

   모든 아웃렛과 액션을 연결합니다.

3. Assigns the root view to the view controller's `view` property.

   뷰 컨트롤러의 `view`프로퍼티에 루트 뷰 를 할당합니다.

4. Calls the view controller's `awakeFromNib` method.

   뷰 컨트롤러의 `awakeFromNib` 메소드를 호출합니다.

   When this method is called, the view controller's trait collection is empty and views may not be in their final positions.

   이 메소드가 호출될 때, 뷰 컨트롤러의 trait collection이 비어있고 뷰 들의 위치가 제대로 잡혀있지 않을 수 있습니다.

5. Calls the view controller's `viewDidLoad` method.

   뷰 컨트롤러의 `viewDidLoad`메소드를 호출합니다.

   Use this method to add or remove views, modify layout constraints, and load data for your views.

   이 메소드를 이용하여 뷰를 추가 또는 삭제하고 레이아웃 조건을 조정하여 뷰의 데이터를 로드하세요.

Before displaying a view controller's views onscreen, UIKit gives you some additional chances to prepare those views before and after they are onscreen. Specifically, UIKit performs the following sequence of tasks:

뷰 컨트롤러의 뷰 들이 스크린에 표시되기 전에 UIKit은 화면에 표시된 전, 후를 대비하기 위한 기회를 제공합니다. 특히, UIKit은 아래와 같은 작업들을 수행합니다.

1. Calls the view controller's `viewWillAppear:` method to let it know that its views are about to appear onscreen.

   뷰 컨트롤러의 `viewWillAppear:`메소드를 호출하여 뷰 들이 곧 화면에 나타날 것임을 알려줍니다.

2. Updates the layout of the views.

   뷰 들의 레이아웃을 업데이트합니다.

3. Displays the views onscreen.

   뷰 들을 스크린에 표시합니다.

4. Calls the `viewDidAppear:` method when the views are onscreen.

   뷰 들이 화면에 표시되면 뷰 컨트롤러의 `viewDidAppear:`메소드를 호출합니다.

When you add, remove, or modify the size or position of views, remember to add and remove any constraints that apply to those views. Making layout-related changes to your view hierarchy causes UIKit to mark the layout as dirty. During the next update cycle, the layout engine computes the size and position of views using the current layout constraints and applies those changes to the view hierarchy.

뷰들을 추가, 삭제 또는 크기나 위치를 조정할 때, 뷰에게 적용된 constraint를 추가 및 제거하는 것을 기억하세요. 뷰 계층에 대한 레이아웃과 관련된 변화는 UIKit이 레이아웃을 dirty로 표시하게 합니다. 다음 업데이트 사이클동안, 레이아웃 엔진은 현재 레이아웃 constraint를 이용하여 뷰의 크기와 위치를 계산하고 해당 변경사항을 뷰 계층에 적용합니다.

For information about how to create views without using storyboards, see the view management information in *UIViewController Class Reference*.

스토리보드 없이 뷰 들을 생성하는 방법에 대한 설명은 *UIViewController Class Reference*을 참고하세요.

Managing View Layout

When the size and position of views changes, UIKit updates the layout information for your view hierarchy. For views configured using Auto Layout, UIKit engages the Auto Layout engine and uses it to update the layout according to the current constraints. UIKit also lets other interested objects, such as the active presentation controller, know abut the layout changes so that they can respond accordingly.

뷰의 크기와 위치가 변경되면, UIKit은 뷰 계층의 레이아웃 정보를 업데이트 합니다. 오토레이아웃을 이용하여 뷰를 구성한 경우, UIKit은 오토 레이아웃을 이용하여 현재 constraint에 따라 레이아웃을 업데이트합니다. UIKit은 또한 active 프리젠테이션 컨트롤러 같이 다른 객체가 레이아웃 변경을 알려주어 적절히 응답할 수 있게끔 합니다.

During the layout process, UIKit notifies you at several points so that you can perform additional layout-related tasks. Use these notifications to modify your layout constraints or to make final tweaks to the layout after the layout constraints have been applied. During the layout process, UIKit does the following for each affected view controller:

레이아웃 프로세스 동안 UIKit은 여러분이 추가적인 레이아웃과 관련된 일을 수행할 수 있도록 몇가지 포인트를 여러분에게 알려줍니다. 이 notification을 이용하여 레이아웃 constraint가 적용된 후에 레이아웃 constraint를 수정하세요. 레이아웃 프로세스동안, UIKit은 영향을 받은 각 뷰 컨트롤러에 대해 아래와 같은 작업을 수행합니다.

1. Updates the trait collections of the view controller and its views, as needed; see When Do Trait and Size Changes Happen?

   필요에 따라 뷰 컨트롤러와 뷰의 trait collection을 업데이트합니다. 자세한 내용은 When Do Trait and Size Changes Happen을 참고하세요.

2. Calls the view controller's `viewWillLayoutSubviews` method.

   뷰 컨트롤러의 `viewWillLayoutSubviews`메소드를 호출합니다.

3. Calls the `containerViewWillLayoutSubviews` method of the current `UIPresentationController` object.

   현재의 `UIPresentationController`객체의 `containerViewWillLayoutSubviews`메소드를 호출합니다.

4. Calls the `layoutSubviews` method of view controller's root view.

   뷰 컨트롤러의 루트뷰의 `layoutSubviews`메소드를 호출합니다.

   The default implementation of this method computes the new layout information using the available constraints. The method then traverses the view hierarchy and calls `layoutSubviews` for each subview.

   이 메소드의 기본 동작은 사용가능한 constraint를 이용하여 새로운 레이아웃 정보를 계산하는 것입니다. 그런 다음 뷰 계층을 계속 탐색하며 각 서브뷰의 `layoutSubviews`메소드를 호출합니다.

5. Applies the computed layout information to the views.

   계산된 레이아웃 정보를 뷰에게 적용합니다.

6. Calls the view controller's `viewDidLayoutSubviews` method.

   뷰 컨트롤러의 `viewDidLayoutSubviews`메소드를 호출합니다.

7. Calls the `containerViewDidLayoutSubviews` method of the current `UIPresentationController` object.

   현재 `UIPresentationController`객체의 `containerViewDidLayoutSubviews`메소드를 호출합니다.

View controllers can use the `viewWillLayoutSubviews` and `viewDidLayoutSubviews` methods to perform additional updates that might impact the layout process. Before layout, you might add or remove views, update the size or position of views, update constraints, or update other view-related properties. After layout, you might reload table data, update the content of other views, or make final adjustments to the size and position of views.

뷰 컨트롤러는 `viewWillLayoutSubviews` 그리고 `viewDidLayoutSubviews`메소드를 이용하여 레이아웃 프로세스에 영향을 미치는 추가적인 업데이트를 수행할 수 있습니다. 레이아웃을 수행하기 전에 뷰 들을 추가 또는 제거하고, 위치 또는 크기와 constraint를 업데이트하거나 다른 뷰와 관련된 프로퍼티를 업데이트 할 수 있습니다. 레이아웃을 수행하고 난 후, 테이블 데이터를 리로드하고, 다른 뷰 들의 컨텐츠를 업데이트 하거나 뷰 들의 크기와 위치에 대한 최종수정을 할 수 있습니다.

Here are some tips for managing your layout effectively:

아래에 레이아웃을 효과적으로 관리하기 위한 몇가지 팁 들이 있습니다.

- **Use Auto Layout.** The constraints you create using Auto Layout are a flexible and easy way to position your content on different screen sizes.

  오토레이아웃을 사용하세요. 오토레이아웃을 이용하여 만든 constraint는 유연하며 컨텐츠를 각기 다른 스크린 사이즈에 위치할 수 있게하는 쉬운방법을 제공합니다.

- **Take advantage of the top and bottom layout guides.** Laying out content to these guides ensures that your content is always visible. The position of the top layout guide factors in the height of the status bar and navigation bar. Similarly, the position of the bottom layout guide factors in the height of a tab bar or toolbar.

  top 그리고 bottom 레이아웃 가이드의 이점을 활용하세요. 이 가이드에 컨텐츠를 배치하면 여러분의 컨텐츠가 항상 보일 수 있게 합니다. top 레이아웃 가이드의 위치는 스테이터스바 그리고 네비게이션 바의 높이에 따라 결정됩니다. 이와 비슷하게, bottom 레이아웃 가이드의 위치는 탭 바 또는 툴바의 위치에 따라 결정됩니다.

- **Remember to update constraints when adding or removing views.** If you add or remove views dynamically, remember to update the corresponding constraints.

  뷰를 추가 또는 삭제할 때 constraint를 업데이트 해야함을 기억하세요. 만약 동적으로 뷰를 추가 또는 삭제하는 경우 해당 constraint를 업데이트 해야합니다.

- **Remove constraints temporarily while animating your view controller's views.** When animating views using UIKit Core Animation, remove your constraints for the duration of the animations and add them back when the animations finish. Remember to update your constraints if the position or size of your views changed during the animation.

  뷰 컨트롤러의 뷰가 애니메이션되는동안 일시적으로 constraint를 제거하세요. UIKit Core Animation을 사용하여 뷰에 애니메이션을 적용할 때, 애니메이션이 실행되는 동안 constraint를 제거하고 애니메이션이 끝나면 다시 추가하세요. 만약 애니메이션 중에 뷰의 위치 또는 크기가 변경되었다면 constraint를 업데이트 해야합니다.

For information about presentation controllers and the role they play in the view controller architecture, see The Presentation and Transition Process.

뷰 컨트롤러 아키텍처에서의 프리젠테이션 컨트롤러의 역할에 대해선 The Presentation and Transition Process을 참고하세요.

Managing Memory Efficiently

Although most aspects of memory allocation are for you to decide, Table 4-1 lists the methods of UIViewController where you are most likely to allocate or deallocate memory. Most deallocations involve removing strong references to objects. To remove a strong reference to an object, set properties and variables pointing to that object to `nil`.

메모리를 할당하는 것은 대부분 여러분이 결정에 달렸습니다. 아래 Table 4-1에 메모리를 할당 또는 해제하는 코드를 넣어야 할 UIViewController의 메소드를 나열하였습니다. 대부분의 할당 해제는 객체의 강한 참조를 제거하는 것과 연관이 있습니다. 객체에 대한 강한 참조를 제거하려면 프로퍼티와 객체를 가리키는 변수에 nil을 설정해야 합니다.

Table 4-1 Places to allocate and deallocate memory

Task	Methods	Discussion
Allocate critical data structures required by your view controller. 뷰 컨트롤러에 필요한 데이터 구조를 할당하세요.	Initialization methods	Your custom initialization method (whether it is named `init` or something else) is always responsible for putting your view controller object into a known good state. Use these methods to allocate whatever data structures are needed to ensure proper operation. 여러분의 커스텀 initialization메소드(init 또는 다른 것으로 불리는)는 항상 뷰 컨트롤러 객체를 양호한 상태로 설정해야할 책임이 있습니다. 이 메소드를 사용하여 적절한 동작을 하는 데 필요한 모든 데이터 구조를 할당하세요.
Allocate or load data to be displayed in your view. 뷰에 표시할 데이터를 할당 또는 로드하세요.	`viewDidLoad`	Use the `viewDidLoad` method to load any data objects you intend to display. By the time this method is called, your view objects are guaranteed to exist and to be in a known good state. `viewDidLoad`메소드를 이용하여 화면에 나타내고 싶은 데이터 객체를 로드하세요. 이 메소드를 호출할 때, 뷰 객체들은 반드시 존재할 것이고 양호한 상태로 설정되어 있을 것 입니다.
Respond to low-memory notifications. low-memory notification에 응답하세요.	`didReceiveMemoryWarning`	Use this method to deallocate all noncritical objects associated with your view controller. Deallocate as much memory as you can. 이 메소드를 이용하여 뷰 컨트롤러와 관련된 중요하지 않은 객체들을 할당 해제하세요. 최대한 많은 메모리를 할당 해제하세요.
Release critical data structures required by your view controller. 뷰 컨트롤러에 필요한 중요한 데이터 구조들을 해제하세요.	`dealloc`	Override this method only to perform any last-minute cleanup of your view controller class. The system automatically releases objects stored in instance variables and properties of your class, so you do not need to release those explicitly. 오직 뷰 컨트롤러 클래스의 마지막 정리를 위해서만 이 메소드를 재정의 하세요. 시스템은 자동적으로 인스턴스 변수에 저장된 객체와 클래스의 프로퍼티를 해제하므로 명시적으로 해제할 필요가 없습니다.