Chapter 01. 첫 번째 ios 앱 만들기

View Controller (뷰 컨트롤러)

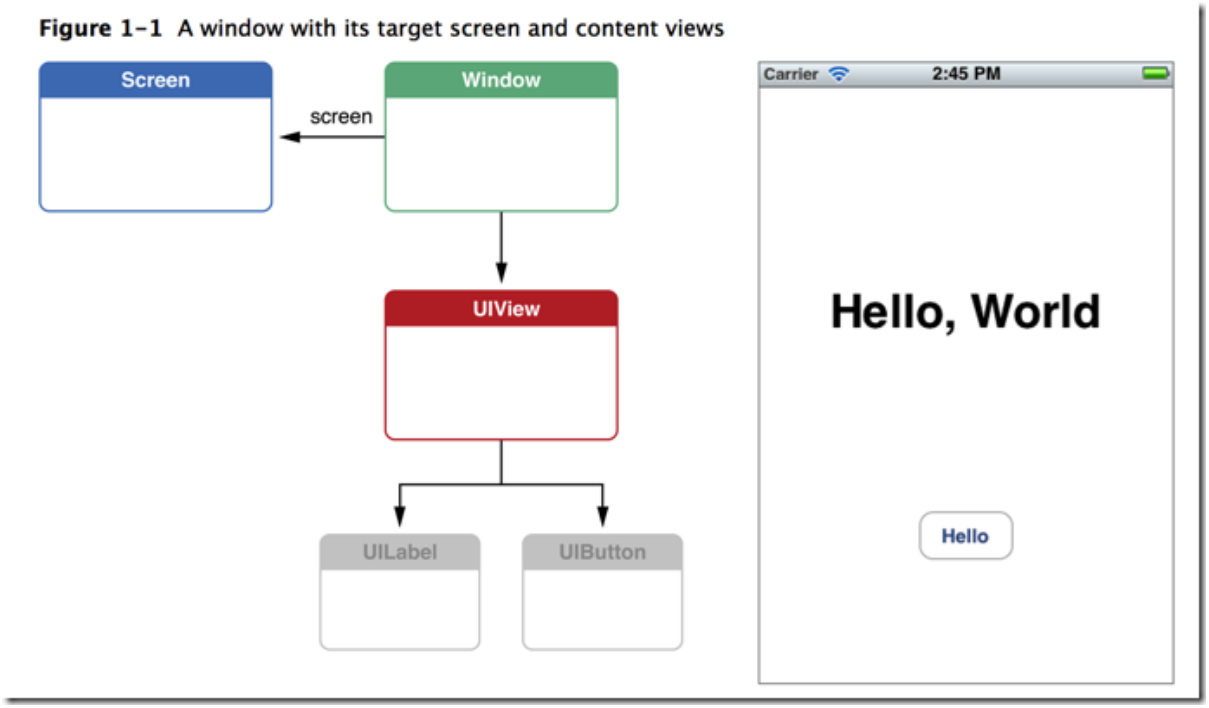
ios 기반의 기기에서 실행되는 모든 앱은 항상 정보와 콘텐츠를 사용자들에게 어떤 방식으로 전달할 것인가에 대해 창조적일 수 밖에 없다. 보여줄 내용이 많으면 필연적으로 간단하게 요약된 화면을 먼저 제공하고, 사용자의 액션에 따라 상세하고 많은 내용을 보여줄 수 있도록 UI 구성 원칙에 따라 앱을 구성하는 것이 중요하다.

View Controller는 이런 원칙에 따라 하위에 있는 콘텐츠를 관리하고, 보여주거나 숨기는 등의 구성을 조정하는 역할을 한다. 그래서 view controller는 내부적으로 view를 포함하고 있으며, view에 대한 관리와 화면 전환이 발생할 때 다른 view controller와 서로 통신하고 조정하는 일을 수행한다.

view controller가 객체간의 연결 관계까지 한꺼번에 이해하는 것은 무척 힘든 일이다. 따라서, 인터페이스 빌더를 사용하여 **스토리보드**라는 형식의 파일을 만들게 된다. 스토리보드는 앱에서 연결 관계를 더 쉽게 눈으로 직접 확인할 수 있도록 해주며, 실행 시 앱을 초기화하기 위해 필요한 여러가지 노력을 매우 간단하게 줄여준다.

화면을 구성하는 3가지 주요 객체

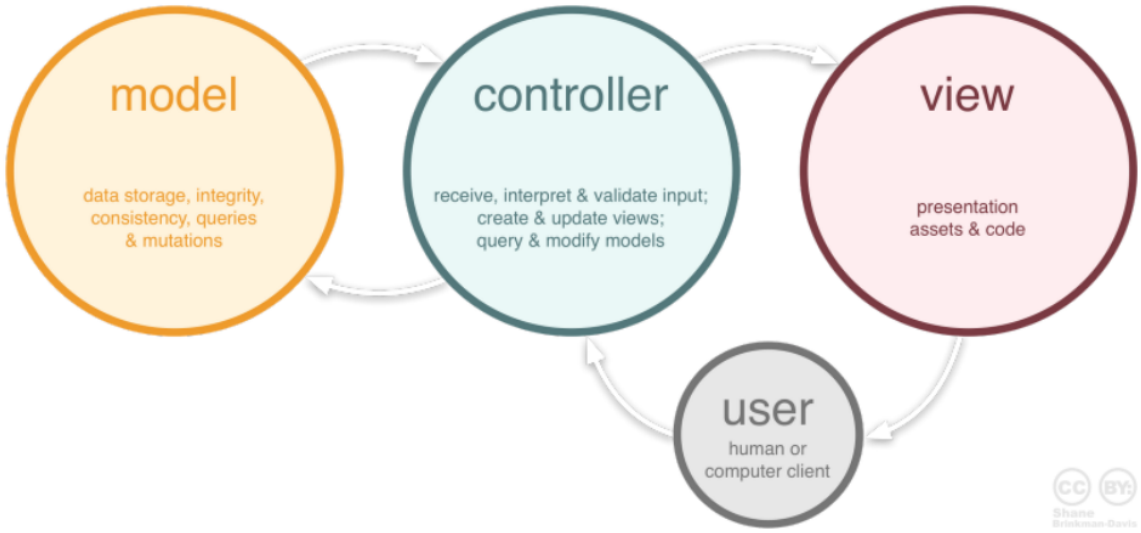
- **UIScreen** : 기기에 연결되는 물리적인 화면을 정의하는 객체
- **UIWindow** : 화면 그리기 지원 도구를 제공하는 객체
- **UIView** : 그리기를 수행할 객체 세트



window에 첨부된 UIView 객체는 이것을 기본으로 하는 많은 파생 요소들을 가지는 객체로서, 우리가 흔히 접하는 화면 구성 요소 대부분이 이에 해당한다. 수많은 UIView 객체가 모인 Window는 이들을 화면으로 구성하여 Screen 객체에 보내고, Screen 객체는 이를 물리적인 기기에 표시한다.

이때 View Controller는 화면을 그려내는데에 반드시 필요한 요소는 아니다. 앱 아키텍처에서 MVC 패턴을 도입하면서 생겨나게 된 단순한 컨트롤러 객체라고 볼 수 있다. MVC 패턴에 따라 화면을 구성할 때 Window에 View 객체를 바로 할당해서는 안 된다. 대신 Window에 view controller를 할당해서 view controller가 자동으로 자신에게 첨부된 view들을 화면에 추가하도록 해야한다.

✋ MVC 패턴이란?



모델

모델은 앱이 포함해야할 데이터가 무엇인지를 정의합니다. 데이터의 상태가 변경되면 모델을 일반적으로 뷰에게 알리며(따라서 필요한대로 화면을 변경할 수 있습니다) 가끔 컨트롤러에게 알리기도 합니다(업데이트된 뷰를 제거하기 위해 다른 로직이 필요한 경우).

다시 쇼핑 리스트 앱으로 돌아가서, 모델은 리스트 항목이 포함해야 하는 데이터 — 품목, 가격, 등. — 와 이미 존재하는 리스트 항목이 무엇인지를 지정합니다.

뷰

뷰는 앱의 데이터를 보여주는 방식을 정의합니다.

쇼핑 리스트 앱에서, 뷰는 항목이 사용자게에 보여지는 방식을 정의하며, 표시할 데이터를 모델로부터 받습니다.

컨트롤러

컨트롤러는 앱의 사용자로부터의 입력에 대한 응답으로 모델 및/또는 뷰를 업데이트하는 로직을 포함합니다.

예를 들어보면, 쇼핑 리스트는 항목을 추가하거나 제거할 수 있게 해주는 입력 폼과 버튼을 갖습니다. 이러한 액션들은 모델이 업데이트되는 것이므로 입력이 컨트롤러에게 전송되고, 모델을 적당하게 처리한다음, 업데이트된 데이터를 뷰로 전송합니다.

단순히 데이터를 다른 형태로 나타내기 위해 뷰를 업데이트하고 싶을 수도 있습니다(예를 들면, 항목을 알파벳순서로 정렬한다거나, 가격이 낮은 순서 또는 높은 순서로 정렬). 이런 경우에 컨트롤러는 모델을 업데이트할 필요 없이 바로 처리할 수 있습니다.

첫번째 앱, Hello World!

스토리보드를 사용하면 앱이 어떤 화면으로 구성되는지, 각 화면은 서로 어떻게 연관되는지, 앱의 전체 모습은 어떠한지를 한눈에 알 수 있다. view controller마다 분리된 UI파일을 만드는 대신, 하나의 스토리보드에 view controller의 연결 관계를 포함하는 전체 화면 구성을 정의함으로써 앱의 전체 구조를 쉽게 파악할 수 있게 표현해주는 것이 바로 스토리보드 방식으로 화면을 설계하는 과정이다. 또한 스토리보드를 사용하면 한 화면에서 다음 화면으로 전환하기 위해 작성해야 하는 코드를 줄일 수도 있다.

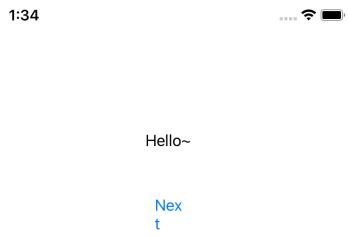
스토리보드를 사용하면 개별 UI 파일을 이용하여 화면을 구성하는 것에 비하여 다음과 같은 이점이 있다.

1. 스토리보드를 사용함으로써 앱의 화면 전체와 화면 사이 관계를 더 쉽게 파악할 수 있다. 스토리보드는 전체 설계가 파일 하나에 모두 들어있기 때문에 각각의 화면으로 분리된 UI파일보다 화면의 요소들을 추적하기가 훨씬 쉽다.
2. 스토리보드는 다양한 화면 사이의 전환을 손쉽게 처리한다. 이러한 전환 방식을 **세그웨이(segueway)**라고 부르는데, 하나의 view controlle에서 전환하고자 하는 view controller를 향해 ctrl 키를 누른 상태로 마우스를 드래그하기만 하면 세그웨이가 자동으로 만들어진다. 세그웨이 덕분에 훨씬 더 짧은 코드로 UI를 다룰 수 있다.
3. 스토리보드를 이용하면 테이블 뷰를 작업할 때 새로운 프로토타입 셀이나 정적인 셀의 외형을 만드는 것이 매우 쉽다. 인터페이스 빌더만을 사용하여 거의 완전한 수준의 테이블 뷰를 만들 수 있으며, 이렇게 작업함으로써 작성해야 하는 코드의 양이 대폭 줄어든다.

Xcode에서 Main.storyboard 파일을 클릭하여 인터페이스 빌더를 연다. 현재 인텊이스 빌더에는 한 개의 앱 UI 화면이 추가되어 있을 것이다. 이 화면을 스토리보드상의 용어로 씬(Scene)이라고 부르지만, 여기서는 view controller라고 부르겠다. 앱의 각 화면은 그에 대응되는 view controller로 이루어지기도 하거니와, 대부분의 경우에서 view controller를 추가하면 화면도 추가되기 때문이다. 따라서 지금은 **‘하나의 view controller = 하나의 화면’**이라고 생각한다.

> 🌟 스토리보드를 사용해본 결과 안드로이드 스튜디오에서 UI로 작업하는 것과 비슷하다. 하지만 안드로이드 스튜디오에서도 UI로 작업하고 세부적인 위치 지정과 디자인 변경은 xml 파일에서 하는 것이 쉬운 적이 있듯이 스토리보드의 단점이 존재하는 것 같다. 먼저 스토리보드로 기본을 익히고, 코드로 작성하는 방법도 알아보자. 🌟

먼저 간단하게, Label과 Button을 둔 화면을 구성해보았다.

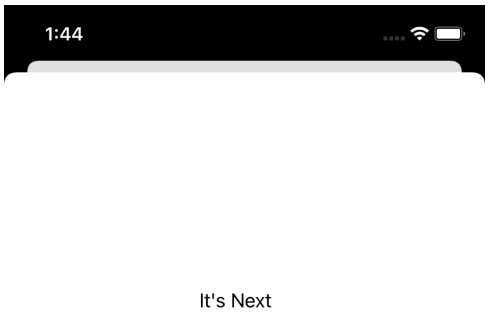


이제 화면 전환을 구현해보자.

하나의 화면은 하나의 view controller로 이루어진다 했으니, 스토리보드의 빈 영역에 view controller를 드래그한다. 이후 첫 번째 화면의 view controller에 존재하는 Next 버튼을 마

우스 오른쪽으로 클릭하여 두 번째 view controller로 드래그한다.

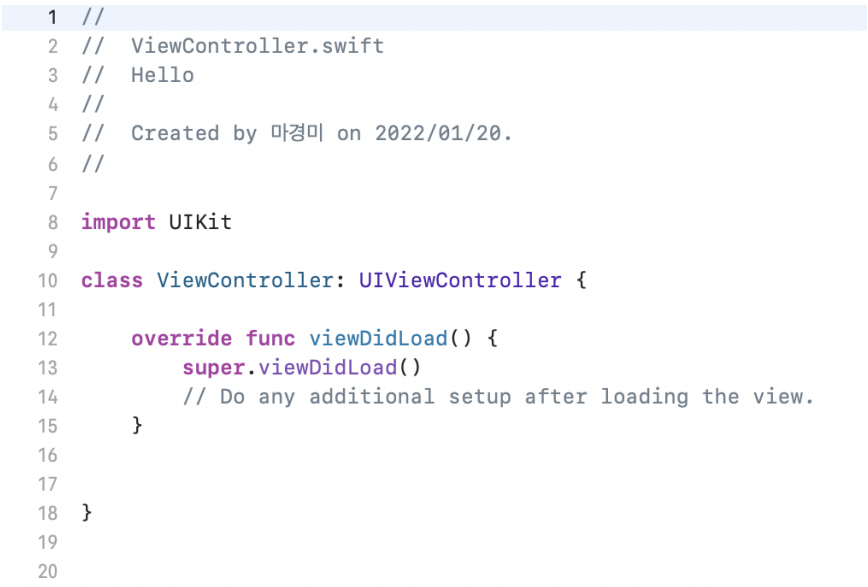
그러면 화면 전환 방식을 선택할 수 있다. 선택 후에는 팝업 메뉴가 닫히면서 두 개의 view controller 사이에 화살표가 추가된다. 나는 modal 형식을 선택했기에 아래와 같은 화면이 나왔다.



지금부터는 스위프트 코드를 작성해본다.

앱이 사용자의 동작에 반응하기 위해서는 레이블과 버튼 등을 제어하는 소스 코드를 작성해야 한다. 이때 소스 코드를 작성하는 위치는 view controller에 연결된 클래스 내부이다. UI를 표현하는 각각의 view controller에는 이를 프로그래밍적으로 제어하기 위한 클래스 객체가 배정되는데, 이 클래스의 소스 코드 내부에 swift code를 작성함으로써 UI를 마음대로 조정할 수 있다.

일반적으로 view controller를 추가하고 나면 소스 코드를 작성할 클래스를 정의하고, 이어서 view controller와 class를 서로 연결해 주어야 한다. 지금은 기본 템플릿에 의해 ViewController.swift가 만들어져있기 때문에 바로 코드를 작성해본다.



```
1  //
2  //  ViewController.swift
3  //  Hello
4  //
5  //  Created by 마경미 on 2022/01/20.
6  //
7
8  import UIKit
9
10 class ViewController: UIViewController {
11
12     override func viewDidLoad() {
13         super.viewDidLoad()
14         // Do any additional setup after loading the view.
15     }
16
17
18 }
19
20
```

```
import UIKit
```

8행은 UIKit 프레임워크를 사용하기 위해 필요한 기본 파일들을 읽어 들이는 부분이다. UIKit은 앱 화면을 구성하는데에 필요한 모든 객체들이 포함된 프레임워크이다.

```
class ViewController: UIViewController{
```

10행은 UIViewController라는 클래스를 상속받아 ViewController라는 이름의 새로운 클래스를 정의하는 것이다. UIViewController 클래스는 UIKit 프레임워크에 정의되어 있는 클래스로서, 기본 view controller를 구현하는 핵심 클래스이다.

```
override func viewDidLoad(){
  super.viewDidLoad()
}
```

12~15행은 viewDidLoad() 메소드를 정의하는 부분이다. viewDidLoad()는 부모 클래스인 UIViewController 클래스에 정의되어 있는 메소드로, view의 로딩이 완료되었을 때 시스템에 의해 자동으로 호출된다. 따라서 일반적으로 리소스를 초기화하거나 초기 화면을 구성하는 등, 처음 한 번만 실행해야 하는 초기화 코드는 대부분 이 메소드 내부에 작성하면 된다.

ios에서는 객체를 소스코드와 직접 연결하는 방법을 많이 사용한다.

label을 선택하여 마우스 오른쪽 버튼을 누른 채로 보조 에디터의 소스 창으로 드래그하면 Insert Outlet or Outlet Collection이라는 메세지가 나타난다. 이 메세지가 표시되면 마우스 드래그를 놓으면 다음과 같은 팝업 창이 나타난다.

[Name] 항목을 입력하고 ([storage] 항목을 'strong'으로 선택한다) <connect> 버튼을 클릭한다. 그러면 viewController 클래스에 속성 변수가 자동으로 추가된 것을 볼 수 있다.

```
@IBOutlet var uiTitle: UILabel!
```

이 변수는 인터페이스 빌더의 레이블을 swift class가 참조할 수 있도록 연결된 멤버 변수로, **아울렛 변수**라고 한다. 특히 @IBOutlet이라는 키워드는 인터페이스 빌더에 관련된 속성이라는 것을 알려주는 어노테이션이다.

버튼을 연결할 때는 레이블과는 달리 [Name] 항목뿐만 아니라 터치했을 때 발생하는 이벤트를 연결하기 위해 [Connection] 항목도 값을 입력한다.

```
@IBAction func sayHello(_ sender: Any) {
}
```

func 뒤에 sayHello는 우리가 설정해준 이름이다. 그리고 레이블과 달리 속성 변수가 아니라 메소드의 형태이다. 이제 사용자가 버튼을 터치했을 때 실행할 코드를 sayHello 메소드에 추가해보자.

```
@IBAction func sayHello(_ sender: Any) {
  self.uiTitle.text="Hello, World!"
}
```

이제 버튼을 클릭하고 나면 레이블의 텍스트가 변한 것을 확인할 수 있다.

> 🌟 만약 앱이 실행되지 않고 에러가 난다면? 버튼을 연결할 때 뜨는 팝업창에서 [Connection] 항목을 Outlet으로 설정했다가 해당 소스 코드를 지우고 다시 Action으로 설정하여 오류가 발생한 것이다. 이는 완벽하게 지워지지 않고 보이지 않는 설정 파일에 연결 정보가 남아있기 때문이다. 만약 이런 문제가 발생했다면, 인터페이스 빌더에서 버튼을 선택하여 활성화한 상태에서 화면 오른쪽 인스펙터 창의 탭 중 가장 마지막에 있는 [Connections Inspector] 탭을 연다. 여기서 Referencing Outlet 필드에 무엇인가 표시되어 있다면 [X]를 클릭하여 연결을 삭제한다. 🌟

시작 화면 제어하기

이전에 만들었던 화면이 처음부터 나타나는게 아니라, 어떤 화면이 잠깐 표시되었다가 잠시 후에 자동으로 우리가 만든 화면으로 전환이 된다. 이때 화면을 **Launch Screen (론치 스크린)**, Splash(스플래쉬), 시작 화면이라고 한다.

일반적으로 시작 화면은 앱을 제작하는 회사의 로고를 표시하거나 단말기에서 앱이 처음 실행될 때 주요 데이터를 초기화할 수 있는 시간을 벌어주는 역할을 하기 때문에 시작 화면이 제대로 제공되지 않는다면 부자연스러운 앱 사용 경험을 제공할 수 있다.

이 화면은 LaunchScreen.storyboard 파일이다. 프로젝트의 설정 창에서 Launch Screen File 에 LaunchScreen이 입력되어 있는 것을 볼 수 있다. 원한다면 다른 파일로도 수정이 가능하다. storyboard를 사용하는 법은 알았으니 LaunchScreen을 꾸미는 것도 넘어간다.

시작 화면 출력 시간을 조정하고싶은 경우에는 AppDelegate.swift 파일에서 application(_:didFinishLaunchingWithOptions:)이 필요하다. 이 메소드는 앱이 처음 실행될 때, 필요한 시스템적 처리를 모두 끝내고 메인 화면을 표시하기 직전에 호출된다. 다시 말해, 앱이 맨 처음 실행될 때 시작 화면이 모바일 기기의 스크린에 표시된 후 이 메소드가 호출되고, 이 메소드 내부에 작성된 내용이 모두 실행되고 나면 우리가 구현한 Main.storyboard 파일의 화면이 스크린에 표시되는 것이다. 따라서 sleep(sec)을 추가해주면 출력 시간을 조정할 수 있다.