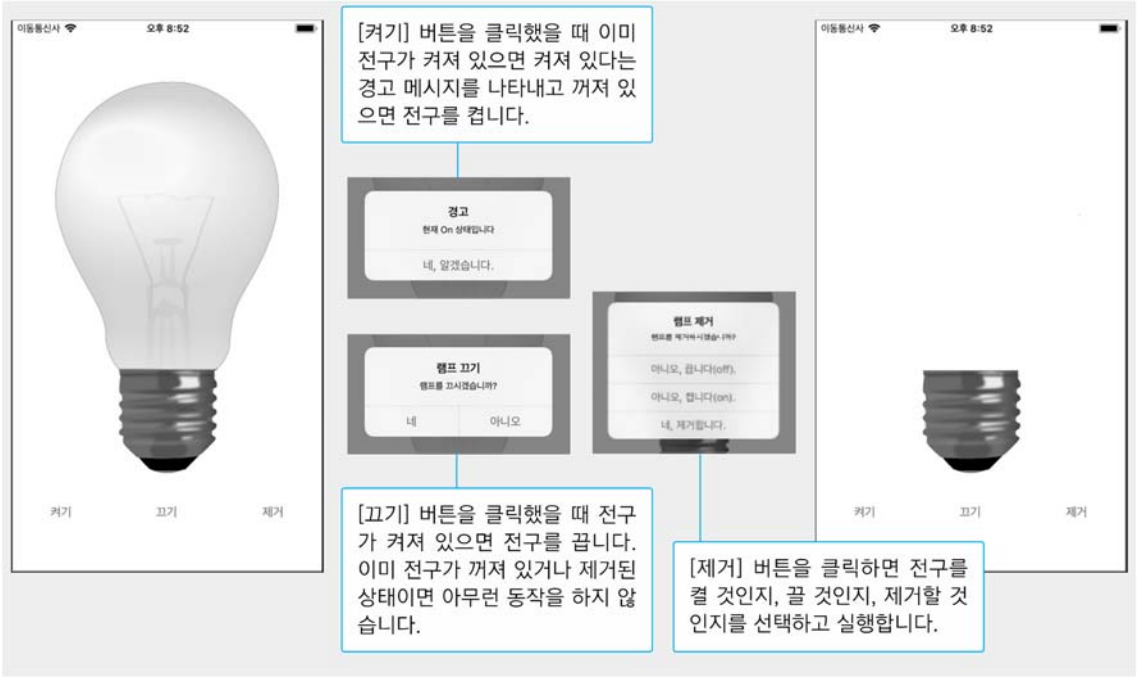
06 장.얼럿 사용해 경고 표시하기

얼럿(Alert)은 화면에 경고 메시지를 표시하는 앱이다.

단순히 경고 메시지를 나타낸 후 확인만 하게 할 수도 있지만 경고 메시지와 함께 두 가지 이상의 선택을 요구할 수도 있으며 선택에 따라 특정 작업도 수행할 수 있다.

여기서는 전구 [끄기],[켜기],[제거]버튼을 만들어 전구를 제어해 보자. 단순히 켜고 끄고만 하는것이 아니라 선택에 따라 경고 메시지를 나타내어 메시지에 따른 작업도 수행하게 할 것이다.

이반 예제를 통해 경고 메시지를 나타내고 여러 선택에 따라 특정 작업을 수행하는 방법에 대해서 알아보자.



06-1 얼럿이란

얼럿은 사용자에게 중요한 알림이나 경고 메시지를 나타내야 할 때 주로 사용합니다. 사용자 의 주의를 집중시키는 경고로 마무리할 수도 있고 후속 조치를 취할 수도 있습니다.

예를 들어 오디오 스트리밍 앱을 청취할 것인가를 물어봐야 합니다. 또한 달력 앱은 약속시간이 다가오고 있음을 사용자에게 알려 주어야 한다. 다음 그림 처럼 와이파이와 셀룰러 데이터가 모두 끊긴 상태라면 웹브라우저는 이를 사용자에게 알리고 설정을 할 수 있도록 후속 조치를 취해야 한다.

SBS 라디오 스트리밍 앱인 고릴라(Gorealra) 앱 화면. 와이파이가 끊기면 '네트워크 알림' 창이 나타납니다.

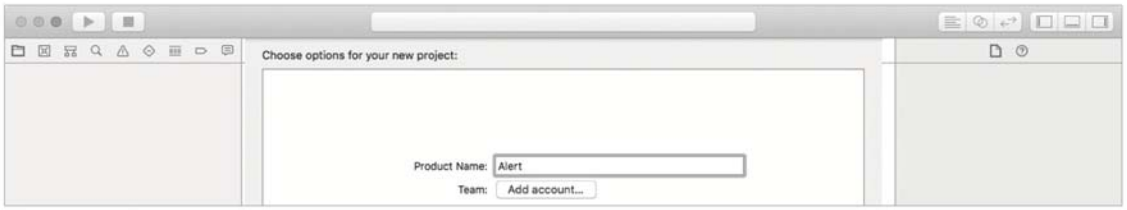
달력과 할 일을 관리해 주는 포켓 인포먼트(Pocket Informant) 앱 화면. 미팅 시간이 임박하면 미리 알려 줍니다.

iOS의 기본 브라우저인 사파리(Safari) 앱 화면. 와이파이와 셀룰러 데이터가 모두 끊긴 상태임을 알려 줍니다.

06-2 얼럿앱을위한기본환경구성하기

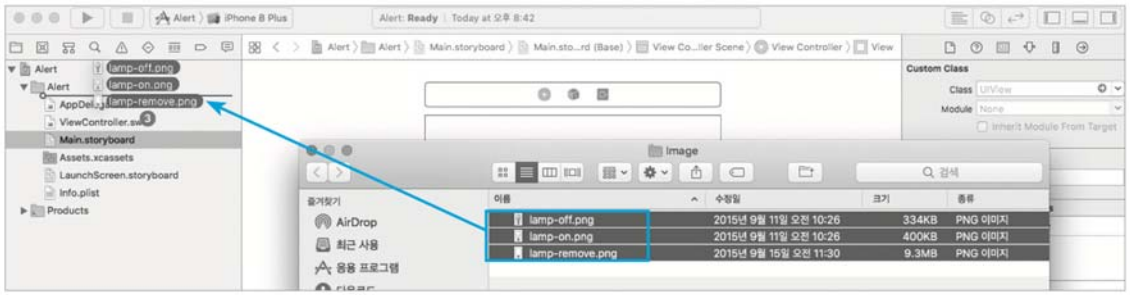
이 앱을 사용해서 구현할 수 있는 기능은 전구를 켜고,끄고,제거하는 것이다.
또한 전구의 상황에 따라 적절한 메시지를 나타내 준다. 따라서 이 앱에서는 전구의 켜져 있는 이미지, 꺼져 있는 이미지 그리고 제거된 이미지가 필요하다.

1. Xcode 를 실행한 후 'Alert'이라는 이름으로 프로젝트를 만듭니다.

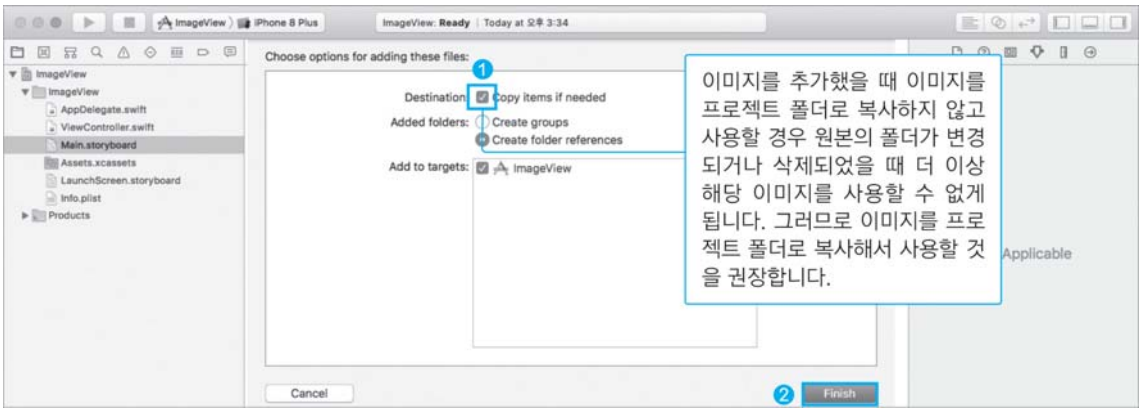


2. 이미지 추가

앱에서 사용할 이미지를 프로젝트에 추가하겠습니다.
파인더에서 원하는 이미지를 선택한 후 내비게이터 영역으로 드래그 앤 드롭하여 추가합니다.



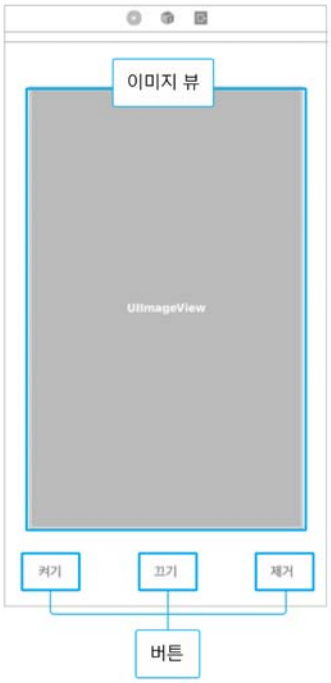
3. 파일 추가에 대한 설정 창이 나타난다. 추가한 이미지를 프로젝트 폴더로 복사하기 위하여 [Destinition: Copy items if needed]항목에 체크가 되어 있는지 확인항 후 [Finish] 버튼을 클릭한다,



06-3 스토리보드로 얼럿 앱 화면 꾸미기

이 앱에서는 켜진 전구, 꺼진 전구 그리고 제거된 잔구를 보여 주어야 하는데, 각각의 전구 이미지를 보여주기 위해 이미지 뷰를(Image View)객체를 사용한다.
그리고 전구를 켜고, 끄고, 제거하기 위해 버튼(Button)객체를 사용한다.

오른쪽 그림은 완성된 스토리보드 화면이다.



1. 이미지를 보여 줄 이미지 뷰 추가하기
오른쪽 아랫부분의 오브젝트 라이브러리에서 [이미지 뷰(Image View)]찾아 스토리보드로 끌어와 배치한다.
그리고 나서 이미지 뷰의 크기를 적당히 조절한다.

2. 전구를 제어할 버튼을 만들어 보자.
오른쪽 아랫부분의 오브젝트 라이브러리에서 [버튼(Button)]을 찾아 스토리보드로 끌어와 화면 아랫부분에 배치한다.
같은 방법으로 일직선 상에 버튼을 두개 더 추가한다.

3. 각 버튼을 마우스로 더블 클릭한 후 내용을 '켜기', '끄기', '제거'로 변경하여 화면 구상을 마친다.

06-4 아웃렛 변수와 액션 함수 추가하기

스토리보드에서 추가한 객체들을 프로그램으로 제어하기 위해 소스 코드에 변수 형태와 함수 형태로 추가한다.
객체가 선택되었을 때 어떤 동작을 수행해야 한다면 액션 함수를 추가하고, 객체의 값을 이용하거나 속성 등을 제어해야 한다면 아울렛 변수를 추가한다.

이미지 뷰의 내용, 즉 속성을 제어해야 하므로 아웃렛 변수를 추가한다.

각각의 버튼을 누르면 이미지 뷰의 내용을 변경해야 하므로 세 번튼 모두에 액션 함수를 추가한다.

1. 보조 편집기 영역 열기

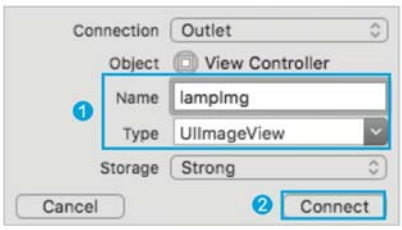
아웃렛 변수와 함수를 추가하려면 오른쪽 윗부분의 [Show the Assistant editor]버튼을 클릭하여 보조 편집기 영역을 열어야 한다.
가운데 화면의 스토리보드 부분이 둘로 나누어 지면서 왼쪽에는 스토리보드, 오른쪽에는 보조 편집기가 나타난다.

2. 이미지 뷰에 대한 아웃렛 변수 추가하기

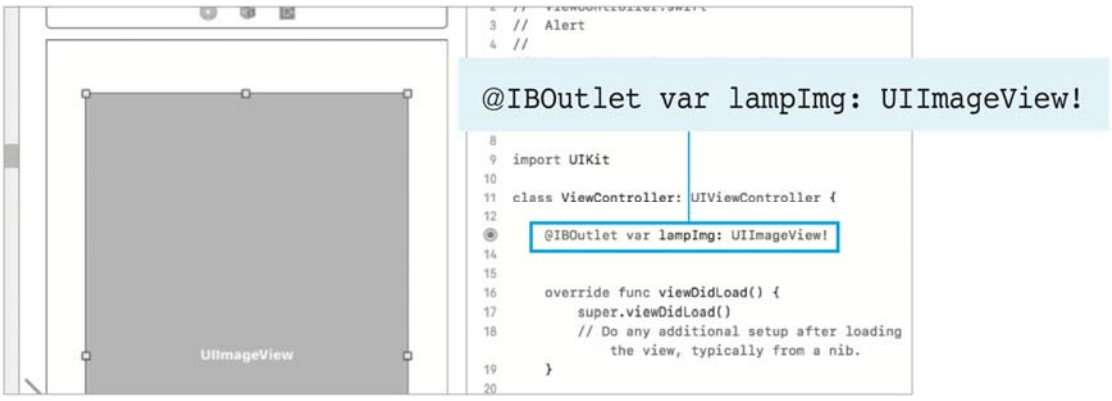
첫 번째로 이미지 뷰에 대한 아웃렛 변수를 추가한다.
이미지 뷰를 마우스 오른쪽 버튼으로 선택한 후 편집기 영역으로 드래그하면 아래 그림과 같이 연결선이 나타난다.
드래그한 연결선을 뷰 컨트롤의 클래스 선언문 바로 아래에 끌어다 놓은 후 마우스 버튼에서 손을 땐다.

[이미지 뷰(Image View)]를 마우스 오른쪽 버튼으로 선택한 후 드래그하여 보조 편집기 영역에 갖다 놓으면 다음과 같이 연결 설정 창이 나타난다.
이 설정 창에서 다음과 같이 설정 한 후 [Connect]버튼을 클릭하여 이미지 뷰와 아웃렛 변수를 연결한다.

위치	뷰 컨트롤러의 클래스 선언문 바로 아래
연결(Connection)	Outlet
이름(Name)	lampImg
유형(Type)	UIImageView



3. 이미지 뷰에 대한 아웃렛 변수가 추가된다.



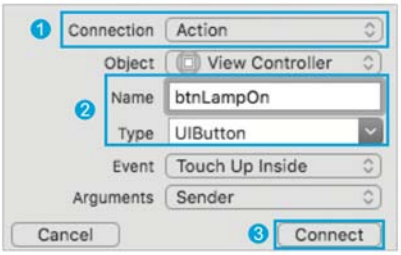
4. [켜기], [끄기], [제어] 버튼에 대한 액션 함수 추가하기.

먼저 [켜기]버튼에 대한 액션 함수를 추가 한다. 3 번 과정과 같은 방법으로 마우스 오른쪽 버튼으로 [켜기]를 선택한 후 드래그해서 오른쪽 보조 편집기 영역의 마지막 '}'괄호 바로 위에 갖다 놓는다.

연결 설정창이 나타나면 연결(Connection)을 [Action]으로 변경한다.

그리고 이름(Name)을 'btnLampOn'로 입력하고, 유형(Type)은 버튼의 액션 추가하는 것이므로[UIButton]을 선택한다. 변경을 완료한 후 [Connect] 버튼을 클릭하여 추가한다,

위치	뷰 컨트롤러 클래스의 마지막 '}' 바로 위
연결(Connection)	Action
이름(Name)	btnLampOn
유형(Type)	UIButton



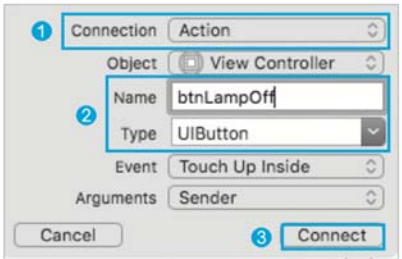
5. 같은 방법으로 [끄기] 버튼과 [제거]버튼에 대한 액션 함수를 추가하자.

우선 [끄기]의 연결선은 방금 추가한 btnLampOn 함수 아래에 놓는다. 그리고 연결 설정창에서 연결(Connection)을 [Action]으로 변경한다.

그리고 이름(Name)을 'btnLampOff'로 입력하고, 유형(Type)은 버튼의 액션을 추가하는 것이므로 [UIButton]을 선택한다.

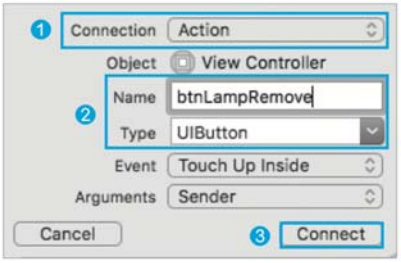
변경 완료 후 [Connect]버튼을 클릭하여 추가한다.

위치	뷰 컨트롤러 클래스의 마지막 '}' 바로 위 (btnLampOn 함수 아래)
연결(Connection)	Action
이름(Name)	btnLampOff
유형(Type)	UIButton



6. 다음으로 [제거 버튼을 방금 추가한 btnLampOff 함수 아래에 연결한다.
그리고 연결설정 창에서 연결(Connection)을 [Action]으로 변경한다. 그리고 이름(Name)을 'btnLampRemove'로 입력하고, 유형(Type)은 버튼의 액션을 추가하는 것이므로 [UIButton]을 선택한다.
변경을 완료한 후 [Connect]버튼을 클릭하여 추가한다.

위치	뷰 컨트롤러 클래스의 마지막 '}' 바로 위 (btnLampOff 함수 아래)
연결(Connection)	Action
이름(Name)	btnLampRemove
유형(Type)	UIButton



06-5 전구 켜기 구현하기

[켜기]버튼을 누르면 전구가 켜지고, 이미 전구가 켜져 있는 상태라면 현재 전구가 켜져 있다는 경고 메시지를 나타내도록 구현해 보자.

[끄기]버튼을 눌렀을 때 전국가 켜져 있는 상태라면 전구를 끌것인지를 선택할 수 있도록 하고, 이미 켜져 있는 상태라면 아무런 동작을 하지 않도록 한다. 마지막으로 [제거] 버튼을 누르면 전구의 상태에 상관없이 메시지 창을 나타내어 끌 것인지, 켤 것인지, 제거 할 것인지 선택하도록 한다. 또한 전구를 제거하는 경우에는 메시지가 한눈에 보이도록 붉은 글씨로 표시되도록 하자.

1. 스탠더드 에디터로 화면 모드 수정하기

코드를 작성하기 위해 화면 모드를 수정한다. 화면 오른쪽 윗부분에서 [Show the Standard editor]버튼을 클릭한다.

이제 왼쪽의 내비게이터 영역에서 [ViewController.swift]를 선택한다.

3. 상수 및 변수 추가하기

코딩에 필요한 상수들과 변수를 뷰 컨트롤러 클래스 선언문 바로 아래에 추가한다.

```
Class ViewController: UIViewController {

      let imgOn  = UIImage(named: "lamp-on.png")
            ❶
      let imgOff = UIImage(named: "lamp-off.png")
            ❷
      let imgRemove = UIImage(named: "lamp-remove.png")
            ❸
      var isLampOn = true
            ❹
```

1) imgOn : 켜진 전구 이미지를 가지고 있는 UIImage 타입의 상수
2) imgOff : 꺼진 전구 이미지를 가지고 있는 UIImage 타입의 상수
3) imgRemove : 제거된 전구 이미지를 가지고 있는 UIImage 타입의 상수
4) isLampOn : 전구가 켜졌는지의 여부를 나타내는 변수, 켜진 상태는 true, 꺼진 상태는 false

imgOn, imgOff, imgRemove 의 경우 앱이 종료될 때 까지 변하지 않기 때문에 상수로 처리하였다.

4. 이미지 보여주기
앱을 처음 시작할 때 전구가 켜져 있는 이미지를 보여주기 위해 viewDidLoad 함수 내의 lampImg 객체에 imgOn 을 대입한다.
그러면 앱이 실행될 때 'lamp_on.png' 이미지가 화면에 나타나게 된다.

```
21      override func viewDidLoad() {
22          super.viewDidLoad()
23          // Do any additional setup after loading the view, typically from a nib.
24          lampImg.image = imgOn
25      }
```

5. [켜기]버튼 클릭 시 동작하는 함수 코딩하기
[켜기] 버튼을 클릭했을 때 동작하는 btnLampOn 함수를 코딩하자.
[켜기] 버튼을 클릭할 경우 전구가 켜졌을 때와 그렇지 않을 때의 동작이 다르다.
따라서 조건문을 사용해야 한다.

먼저 전구가 켜져 있지 않을 때(즉, else 일 때) 전구를 켜본다. Else 문에서 lampImg 객체에 imgOn 을 대입하여 커져 있는 전구이미지를 나타내고 isLampOn 변수에 true 값을 주어 전구가 켜져 있는 상태로 바꾼다. btmLampOn 함수의 else 문에 다음 소스를 추가한다.

```
    @IBAction func btnLampOn(_ sender: UIButton) {
        if(isLampOn==true) {

        } else {
            lampImg.image = imgOn
            isLampOn=true
        }
    }
```

7. 다음으로 전구가 켜져 있는 경우 (즉, isLampOn 변수 값이 true 일 때) 얼럿이 나타나게 해보자

UIAlertController 를 매개변수로 가진 present 메서드를 실행한다.

다음 소스를 btmLampOn 함수의 if 문에 추가한다.

```
@IBAction func btnLampOn(_ sender: UIButton) {
    if(isLampOn==true) {
        let lampOnAlert = UIAlertController(title: "경고", message: "현재
            On 상태입니다", preferredStyle: UIAlertControllerStyle.alert)
        let onAction = UIAlertAction(title: "네, 알겠습니다.", style:
            UIAlertActionStyle.default, handler: nil)
        lampOnAlert.addAction(onAction)
        present(lampOnAlert, animated: true, completion: nil)
    }
    else {
        lampImg.image = imgOn
        isLampOn=true
    }
}
```

1) UIAlertController 생성한다.

2) UIAlertActuion 을 생성한다. 특별한 동작을 하지 않을 경우에는 handler 를 nil 로 한다.

3) 생성된 onAction 을 얼럿에 추가한다.

4) presentViewController 메서드를 실행한다.

06-6 전구 끄기 구현하기

이제 추가로 [끄기]버튼을 클릭했을 때 동작할 btnLampOff 함수를 코딩해 보자.
전구가 꺼져 있을 경우에는 아무런 동작을 하지 않고 전구가 켜져 있을 경우에는 끌것인지 묻도록 만들어 보겠다.

1. [끄기]버튼을 전구가 커졌을 때만 동작해야 하므로 조건문에서 if 문을 사용하겠다,
얼럿을 나타나는 순서는 앞의 방법과 같다. 대신에 ULAlertAction 을 두가지로 작성한다.
UIAlertAction 중 한가지는 아무런 동작을 하지 않아 핸들러를 nil 로 하지만 나머지 한 경우는 전구를 꺼야 하므로 핸들러에서 중괄호{,}를 추가하여 전구를 끄는 작업을 하도록 만들어야 한다.

```
@IBAction func btnLampOff(_ sender: UIButton) {
        if isLampOn {
              let lampOffAlert = UIAlertController(title: "램프 끄기",
                    message: "램프를 끄시겠습니까?",
                    preferredStyle: UIAlertControllerStyle.alert)

              let offAction = UIAlertAction(title: "네",
                    style: UIAlertActionStyle.default, handler: {
                    ACTION in self.lampImg.image = self.imgOff
                    self.isLampOn=false
              })
              let cancelAction = UIAlertAction(title: "아니오", style:
                    UIAlertActionStyle.default, handler: nil)

              lampOffAlert.addAction(offAction)
              lampOffAlert.addAction(cancelAction)

              present(lampOffAlert, animated: true, completion: nil)
        }
}
```

1) UIAlertController 를 생성한다.
2) UIAlertAction 을 생성한다. 전등을 꺼야 하므로 핸들러에 중괄호 {,}를 넣어 추가적으로 작업을 한다.
3) UIAlertAction 을 추가로 생성한다. 특별한 동작을 하지 않을 경우에는 핸들러를 nil 로 한다,
4) 생성된 offAction, cancleAction 을 얼럿에 추가한다.
5) present 메서드를 실행

06-7 전구 제거 구현하기

[제거] 버튼을 클릭할 때 동작할 btnLampRemove 함수를 코딩해 본다.
전구가 켜져 있거나 제거되었어도 동일한 동작을 하게 만들 것이다.
[제거]버튼을 클릭하면 전구가 제거할 것인지를 묻고, '켜기','끄기','제거'의 세가지 동작을 구현하도록 만들어 보자.

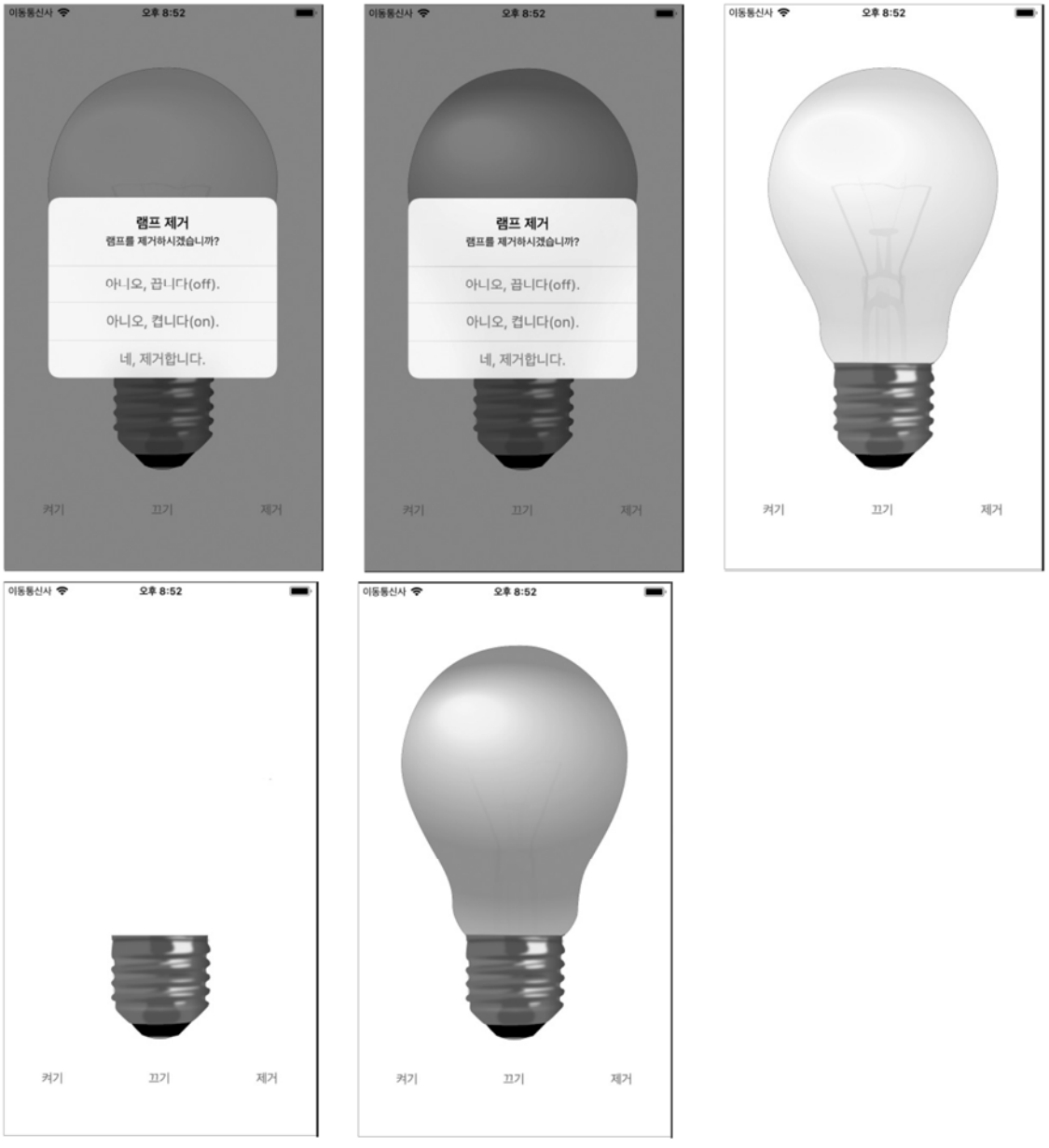
1. 얼럿을 나타내는 순서를 앞과 같다. 대신에 UIAction 을 세가지로 작성해야 한다, '켜기', '끄기', '제거'의 세가지 동작 중에서 선택하기 때문이다.
UIAlertAction 은 핸들러에서 { , }를 각각 추가하여 작업할 것이다.

```
@IBAction func btnLampRemove(_ sender: UIButton) {
    let lampRemoveAlert = UIAlertController(title: "램프 제거", message: "램프를
        제거하시겠습니까?", preferredStyle: UIAlertControllerStyle.alert) ❶

    let offAction = UIAlertAction(title: "아니오, 끕니다(off).", style:
        UIAlertActionStyle.default, handler: {
        ACTION in self.lampImg.image = self.imgOff
        self.isLampOn=false
    }) ❷
    let onAction = UIAlertAction(title: "아니오, 켭니다(on).", style:
        UIAlertActionStyle.default) {
        ACTION in self.lampImg.image = self.imgOn
        self.isLampOn=true
    } ❸
    let removeAction = UIAlertAction(title: "네, 제거합니다.", style:
        UIAlertActionStyle.destructive, handler: {
        ACTION in self.lampImg.image = self.imgRemove
        self.isLampOn=false
    }) ❹

    lampRemoveAlert.addAction(offAction)
    lampRemoveAlert.addAction(onAction)
    lampRemoveAlert.addAction(removeAction) ❺
    present(lampRemoveAlert, animated: true, completion: nil) ❻
}
```

1) UIAlertController 를 생성한다.
2) UIAlertAction 을 생성한다. 전구를 꺼야 하므로 handler 에 { , }를 넣어 추가적으로 작업한다.

3) UIAlertAction 을 추가 생성한다. 전구를 켜는 동작을 추가한다.
핸들러에 {, }를 넣어 추가 작업을 할 수 있는데, 이번에는 핸들러 매개변수를 삭제하고 뒤쪽 { , }를 넣는 방법을 이용한다. 두가지 방법 모두 에러 없이 동작을 구현한다.
4) UIAlertAction 을 추가로 생성한 후 전구를 제거하는 동작을 추가한다.
5) 생성된 offAction, onAction, removeAction 을 얼럿ㅇ에 추가 한다,
6) present 메서드를 실행한다.

2. 결과 보기
[실행] 바튼을 클릭하여 결과를 확인한다, [제거] 버튼은 전구 상태와 상관없이 동일한 경고 창을 나타낸다.



[스위프트 문법] 익명함수란?

앞에서 입력한 ❷의 소스를 보면 오른쪽 코드와 같은 형태로 정리되어 있는 것을 볼 수 있습니다. 일반적인 함수의 경우 func 키워드와 함수 이름을 선언하고 사용하지만 이처럼 효율적인 코드를 작성하기 위해 함수 이름을 선언하지 않고 바로 함수 몸체만 만들어 사용하는 일회용 함수를 익명 함수(Anonymous Functions) 혹은 클로저(Closure)라고 합니다

```
{
    ACTION in self.lampImg.image = self.imgOff
    self.isLampOn=false
}
```

익명 함수를 작성할 수 있는 구문 예는 다음과 같습니다.

```
func completeWork(finished: Bool)
-> (){
   print("complete : \(finished)")
}
```

completeWork 함수는 Bool 타입의 finished 매개변수를 받아 출력하는 함수이며 리턴 타입은 없습니다.

▼

이를 익명 함수 형태로 바꾸면 다음과 같습니다.

```
{
    (finished: Bool) -> () in
    print("complete : \(finished)")
}
```

```
{
   (매개변수) -> (반환 타입) in
   실행 구문
}
```

▼

여기서 컴파일러가 반환 타입을 미리 알고 있다면 반환 타입을 생략할 수 있습니다. 또한 매개변수의 파라미터 타입도 생략할 수 있습니다.

```
{
    (finished: Bool) in
    print("complete : \(finished)")
}
```

```
{
   (매개변수) in
   실행 구문
}
```

가장 기본적인 Swift 함수 형태

Swift 에서 함수를 만들 때에는 func 키워드를 사용 하면 된다.
다른 언어의 function 보다 간결하다.

```
func 함수이름 (){
    //함수 내용
}
```

간단하다! 그럼 실제 소스를 작성 해 보면

```
func testFunc(){
    print("함수가 실행 되었습니다")
}

testFunc() //함수가 실행 되었습니다
```

함수를 실행 하면 함수가 실행 된다.

그런데 함수는 이렇게만 사용 하는 경우는 많지 않다.
바로 함수에 인수를 입력 하여 사용 하게 되는데 기본적으로 함수에 인수를 지정 하는 방법은 아래와 같다.

```
func 함수이름 (인수:자료형, 인수:자료형...){
    //함수 내용
}
```

다른 언어와 크게 다르지 않다.
인수:자료형 형태는 변수를 선언 할 때 봐서 익숙할 것이다.

그렇다면 예를 들어 한 함수를 만들어 보자.
인수로 이름과 나이를 입력 하면 관련 문장을 출력 하는 함수는 아래와 같이 만들 수 있다.

```
func testFunc(name:String, age:UInt){
    print("₩(name)님의 나이는 ₩(age)세 입니다.")
}
```

```
testFunc(name: "푸른해커", age: 33)
```

기존 언어들에서는 함수를 호출 할 때 인수 이름을 적을 필요가 없었지만
Swift 에서는 인수 이름을 입력 해야 한다.
이는 함수 오버로딩을 하게 될 때 유용하다.
기존 언어에서 함수 오버로딩을 할 때 똑같은 인수들 목록으로는 사용 할 수 없었지만
Swift 에서는 같은 인수 목록이라고 하더라도 인수의 이름이 다르면 아래와 같이 사용이 가능하다.

```
func testFunc(name:String, age:UInt){
    print("₩(name)님의 나이는 ₩(age)세 입니다.")
}

func testFunc(group:String, memberCount:UInt){
    print("₩(group)그룹의 멤버 수는 ₩(memberCount)명입니다")
}

testFunc(name: "푸른해커", age: 33)
testFunc(group: "A", memberCount: 5)
```

두 testFunc 함수는 똑같이 String, UInt 인수가 있지만 다른 함수를 만들 수 있다.

인수명은 함수 내에서 변수로 사용이 가능 하지만
함수를 호출 할 때에는 다른 이름을 사용 하고 싶을 때가 있다.
이 때에는 인수명 앞에 사용할 이름을 적어줄 수 있다.

```
func testFunc(name:String, old age:UInt){
    print("₩(name)님의 나이는 ₩(age)세 입니다.")
}
```

```
testFunc(name: "푸른해커", old: 33)
```

함수선언 부분의 age 앞에 old 를 추가 한 것을 알 수 있다.
함수를 호출 할 때에는 age 대신에 old 를 입력해서 전달 하지만, 함수 내에서는 age 변수로 사용 되고 있다.

하지만 인수 이름을 매번 적어주기 귀찮을 경우가 있다.
이럴 경우 _ 문자를 인수 이름 앞에 넣어 주면 함수 호출 시 인수 이름을 입력 하지 않아도 함수를 호출 할 수 있게 된다.

```
func testFunc(_ name:String, old age:UInt){
    print("₩(name)님의 나이는 ₩(age)세 입니다.")
}

testFunc("푸른해커", old: 33) //푸른해커님의 나이는 33 세 입니다.
```

여러개의 인수를 받아야 할 때가 있다.
여러개의 숫자를 입력 받은 뒤 그 숫자들의 합을 내는 함수가 있다고 한다면 아래와 같이 선언 할 수 있다.

```
func sumAll(_ numbers: UInt...){
    var sum:UInt = 0
    for number in numbers{
        sum = sum + number
    }

    print("총 합은 ₩(sum) 입니다")
}

sumAll(2, 3, 4, 2, 8, 0, 5, 3) //총 합은 27 입니다
```

함수에서 값을 리턴 하기

C 계열의 언어에서는 함수 선언할 때 앞쪽에 리턴 타입을 넣게 되어 있다.

```
int sum(int a, int b){
    return a + b;
}
```

함수명 앞에 int 를 넣어서 int 형이 반환 될 것이라는 것을 명시 했다.
반면, PHP 와 같은 언어에서는 리턴 타입을 지정하지 않아도 큰 문제가 되지 않는다.

```
function sum(a, b){
    return a + b;
}
```

Swift 에서는 리턴 형이 있다면 반드시 리턴 타입을 지정 해 주어야만 한다.
리턴형을 지정 해주는 방법은
인수 목록 뒤에 -> 를 삽입하고 리턴될 데이터 형을 입력 하면 된다.

```
func testFunc(name:String, old age:UInt) -> String{
    return ("₩(name)님의 나이는 ₩(age)세 입니다.")
}

var returnedMessage = testFunc(name: "푸른해커", old: 33)
print(returnedMessage)
```

위와 같이 리턴형을 지정 해 줄 수 있다. 리턴형이 없다면, 리턴값을 지정 하지 않으면 된다.

func 함수명(파라미터명 1: 자료형, 파라터명 2: 자료형) -> 반 환 값의 자료형 {실행 코드} 의 형태

```
func today(month : String, day : String) -> String {
    return “오늘은 \(month)월 \(day)일입니다.”
}

today(month: “1”, day: “23”)
```

_를추가하면 C 언어에서와같이함 수를 호출하여 사용할 수 있음

```
func today2(_ month : String, _ day : String) -> String {
    return "오늘은 \(month)월 \(day)일입니다."
}

today2("1", "2")
```