

Swift : beyond the basics

Beyond and beyond !

Plan

- Classes et structures
- Énumérations
- Gestion des erreurs
- Initialisation
- Désinitialisation
- Gestion de la mémoire : ARC
- La documentation
- Du playground à l'application

Classes et structures

Généralités

- En Swift, les classes et les structures sont plus proches que dans d'autres langages
- Les classes ont quelques capacités supplémentaires
- Les instances de classes sont gérées par références
- Les instances de structures sont gérées par valeur

Classes et structures

	Classe	Structure
Définir des propriétés	✓	✓
Définir des méthodes	✓	✓
Définir des initialiseurs	✓	✓
Possibilité d'extension	✓	✓
Héritage	✓	
"Désinitialiseurs"	✓	
Passage par référence	✓	
Initialiseur des propriétés auto-généré		✓

Définitions des classes et structures

- Structure :
 - préfixe struct
 - nom de la structure
 - propriétés et méthodes

- Classe :
 - préfixe class
 - nom de la classe
 - nom de la classe parente
 - facultatif
 - propriétés et méthodes

Définitions des classes et structures

```
struct Engine {  
    var name = "Engine"  
    var horsePower = 0  
    var nbOfCylinder = 0  
    var fuel = "Diesel"  
}
```

Classes et structures

```
struct Engine {  
  
    var name = "Engine"  
    var horsePower = 0  
    var nbOfCylinder = 0  
    var fuel = "Diesel"  
}  
  
class Vehicule {  
  
    var name = "Not Branded"  
    var speed = 0  
    var engine = Engine()  
}
```

Classes et structures

```
var nbCylindres = 0
var fuel = "Diesel"
}

class Vehicule {

    var name = "Not Branded"
    var speed = 0
    var engine = Engine()

}

class Car: Vehicule {

    var convertible = false
    var automatic = false

}
```

Propriétés

- Propriétés stockées
 - Variable ou constante associée avec l'instance
 - Initialisées au moment de la création de l'instance par défaut
 - Initialisée au moment de l'utilisation si définie avec le préfixe **lazy** (**var** uniquement)
- Propriétés calculées
 - Ne stockent pas de valeurs
 - Fournissent des accesseurs (**get** et **set**) pour récupérer ou modifier une ou plusieurs propriétés stockées

Classes et structures

```
extension Engine {  
  
    var computedName: String {  
  
        get {  
            return "Moteur \$(fuel) \$(horsePower) ch"  
        }  
  
        set(newComputedName){  
            print(newComputedName)  
        }  
    }  
}
```

Classes et structures

```
extension Engine {  
  
    var computedName: String {  
  
        get {  
            return "Moteur \$(fuel) \$(horsePower) ch"  
        }  
  
        set{  
            print(newValue)  
        }  
    }  
  
}
```

Propriétés calculées (lecture seule)

```
extension Engine {  
  
    var computedName: String {  
  
        get {  
            return "Moteur \$(fuel) \$(horsePower)ch"  
        }  
    }  
  
}
```

Propriétés calculées (lecture seule)

```
extension Engine {  
  
    var computedName: String {  
        return "Moteur \(fuel) \((horsePower))ch"  
    }  
  
}
```

Propriétés de type

- Une propriété de type est utile pour partager des informations entre toutes les instances
 - Préfixe **static** devant la déclaration de la propriété
 - Propriété stockée ou calculée
- Pour une classe vous pourrez aussi voir **class** au lieu de **static**.
static correspond à **final class**

Propriétés de type

```
class MaClass {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
struct MaStruct {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}
```

Classes et structures

```
class MaClass {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
struct MaStruct {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
print(MaStruct.aStoredTypeProperty)  
print(MaStruct.aComputedTypeProperty)  
print(MaClass.aComputedTypeProperty)  
print(MaClass.aStoredTypeProperty)
```

Observateurs

- Il est possible d'observer une propriété pour réagir à ses changements de valeurs
- Deux observateurs possibles
 - Avant le changement de la valeur (**willSet**)
 - Permet d'accéder à la nouvelle valeur (**newValue**)
 - Après le changement de la valeur (**didSet**)
 - Permet d'accéder à l'ancienne valeur (**oldValue**)

Observateurs

```
var observedProperty: String = "Initial Value" {  
    willSet{  
        print("Will set \(newValue) instead of \(observedProperty)")  
    }  
    didSet{  
        print("Did set \(observedProperty) instead of \(oldValue)")  
    }  
}
```

Méthodes

- Les méthodes sont des fonctions associées à des types particuliers
- Classes et structures peuvent définir :
 - Des méthodes d'instances qui seront utilisées avec leurs instances
 - Des méthodes de types qui seront utilisées avec le type directement

Méthodes

- Une méthode est une fonction.
- La syntaxe pour déclarer une méthode est la même que pour déclarer une fonction.
- En cas de surcharge d'une méthode, il faut préfixer la déclaration par `override`

Méthodes de type

- Comme pour les propriétés de type, une méthode de type est une méthode associée à un type en général, et pas à une instance
- On utilise le préfixe **static**

Contrôle d'accès

External module : Subclass / Override

External module : Access only

Module : Access / Subclass / Override

File : Access / Subclass / Override

Scope :
Access / Subclass / Override

private

fileprivate

internal

public

open

Cas des types valeur

- Une structure est un type valeur
- Par défaut, les propriétés d'un type valeur ne peuvent pas être modifiées depuis une méthode d'instance
- En cas de besoin de ce type de comportement, il faut déclarer la méthode en **mutating**.
- La méthode pourra ensuite modifier les propriétés de la structure

Cas des types valeur

```
extension Engine {  
    ! func add(newCylinders numberOfNewCylinders: Int) {  
        nbOfCylinder += numberOfNewCylinders  
    }  
}
```

Cas des types valeur

```
extension Engine {  
    mutating func add(newCylinders numberOfNewCylinders: Int) {  
        nbOfCylinder += numberOfNewCylinders  
    }  
  
    defaultEngine.addCylinders(2)
```

self

- Dans une méthode d'instance, la propriété implicite **self** symbolise l'instance actuelle
- Dans une méthode de type, la propriété implicite **self** symbolise le type en lui-même
- L'utilisation de **self** n'est pas obligatoire, sauf dans des cas où une confusion est possible (nom interne d'argument identique à un nom de propriété, capture dans une clôture, etc.)

Classes et structures

self

```
extension Vehicule {  
    func replaceName(by name: String) {  
        name = name  
    }  
}
```

Classes et structures

self

```
extension Vehicule {  
    func replaceName(by name: String) {  
        ! name = name  
    }  
}
```

Classes et structures

self

```
extension Vehicule {  
    func replaceName(by name: String) {  
        self.name = name  
    }  
}
```

Transtypage

- Swift ne réalise pas de transtypage implicite
- Possibilité de réaliser des transtypage explicites entre des types liés par héritage
 - Transtypage "vers le haut" sans risque (le compilateur peut vérifier)
 - Transtypage "vers le bas" avec risque (le compilateur ne peut pas vérifier)
- Possibilité de vérifier les types à l'exécution

Transtypage vers le haut

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

Transtypage vers le haut

```
let aCar: Car
```

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

Transtypage vers le haut

```
let aCar: Car
```

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarConsideredAsVehicule: Vehicule
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as? Car
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as? Car
```

```
let aCarIsACar: Car?
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as! Car
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as! Car
```

```
let aCarIsACar: Car
```

Vérification dynamique du type

```
if aCarConsideredAsVehicule is Car {  
    // do something  
}
```

Vérification dynamique du type

```
let aCarConsideredAsVehicule: Vehicule  
if aCarConsideredAsVehicule is Car {  
    // do something  
}
```

Énumérations

Généralités

- Définition d'un type pour un groupe de valeurs liées entre elles
- Plus flexibles qu'en C
- Possibilité de fournir :
 - Soit une valeur brute (rawValue) : **String**, **Character**, **Int**, **Float** ou **Double**
 - Soit une valeur associée de n'importe quel type
- Peuvent avoir des méthodes, initialiseurs, propriétés calculées
- Type valeur !

Généralités

```
enum TransportStatus {  
    case onTime  
    case delayed  
    case cancelled  
    case unknown  
}
```

Généralités

```
enum TransportStatus {  
    case onTime  
    case delayed  
    case cancelled  
    case unknown  
}
```

```
var status = TransportStatus.onTime
```

Généralités

```
enum TransportStatus {  
    case onTime  
    case delayed  
    case cancelled  
    case unknown  
}
```

```
var status = TransportStatus.onTime  
status = .delayed
```

Valeurs associées

```
enum TransportStatus {  
    case onTime  
    case delayed (delay: Int, reason: String)  
    case cancelled (reason: String)  
    case unknown  
}
```

Valeurs associées

```
enum TransportStatus {  
    case onTime  
    case delayed (delay: Int, reason: String)  
    case cancelled (reason: String)  
    case unknown  
}
```

```
var status = TransportStatus.onTime
```

Valeurs associées

```
enum TransportStatus {  
    case onTime  
    case delayed (delay: Int, reason: String)  
    case cancelled (reason: String)  
    case unknown  
}
```

```
var status = TransportStatus.onTime  
status = .delayed(delay: 15, reason: "Driver is late")
```

```
switch status {  
    case .onTime:  
        print("Everything is OK")  
  
    case .delayed (let delay, let message) where delay < 15:  
        print("A small delay is expected : \(message)")  
  
    case .delayed (let delay, let message) where delay >= 15:  
        print("A delay is expected : \(message). Please be patient.")  
  
    case .cancelled (let reason):  
        print("Your transport have been cancelled. Reason : \(reason)")  
  
    case .unknown:  
        print("We don't have any status info at this time")  
  
    default:  
        break  
}
```



Switch exhaustif ou
doit inclure default !

Valeurs brutes

```
enum TransportType: String {  
    case plane = "plane"  
    case train = "train"  
    case bus = "bus"  
    case carSharing = "carSharing"  
}
```

Valeurs brutes

```
enum TransportType: String {
    case plane = "plane"
    case train = "train"
    case bus = "bus"
    case carSharing = "carSharing"
}

if let transport = TransportType(rawValue: "plane") {
    print(transport.rawValue)
}
```

plane

Valeurs brutes

```
enum TransportType: String {
    case plane
    case train
    case bus
    case carSharing
}

if let transport = TransportType(rawValue: "plane") {
    print(transport.rawValue)
}
```

plane

Gestion des erreurs

Généralités

- Les méthodes et fonctions peuvent retourner des erreurs si "quelque chose" se passe mal
- Les optionnels peuvent servir à représenter l'échec, ou l'absence de valeur, mais il est parfois utile de savoir quel en est la cause pour adopter le bon comportement
- La gestion des erreurs permet de traiter et de récupérer ces erreurs

Créer une erreur

- Une erreur est représentée par une valeur d'un type qui est conforme au protocole `Error`
- Les énumérations se prêtent bien à la représentation d'une erreur
- Lancer une erreur permet d'indiquer que quelque chose ne s'est pas passé comme prévu

```
enum FoodDispenserError: Error {  
    case OutOfStock  
    case InsufficientCoins(missing: Int)  
    case UnavailableChoice  
}  
  
throw FoodDispenserError.InsufficientCoins(missing: 5)
```

Gérer une erreur

- Lorsque une erreur est lancée, le code environnant doit gérer cette erreur
 - Propager l'erreur
 - Traiter l'erreur avec un bloc do-catch
 - Traiter l'erreur comme un optionnel
 - Supposer qu'elle ne se produira jamais
- Pour identifier les méthodes qui peuvent lancer une erreur, on les préfixe par `try`

Propager une erreur

- Une méthode peut indiquer qu'elle propage une erreur en utilisant le mot clé `throws`
- Seule les méthodes déclarées avec `throws` peuvent propager une erreur.
- Sinon, l'erreur doit être gérée à l'intérieur de la méthode

```
func vend(foodName: String) throws { ... }
```

Propager une erreur

- Une méthode peut indiquer qu'elle propage une erreur en utilisant le mot clé `throws`
- Seule les méthodes déclarées avec `throws` peuvent propager une erreur.
- Sinon, l'erreur doit être gérée à l'intérieur de la méthode

```
func vend(foodName: String) throws -> Food { ... }
```

Traiter une erreur : do-catch

- Une erreur survenant dans le bloc do peut être traitée dans le bloc catch
- Un bloc catch peut avoir un motif indiquant l'erreur qu'il traite
- Plusieurs blocs catch peuvent se succéder
- Si un bloc catch ne précise pas de motif, il traitera toutes les erreurs
- Tous les cas peuvent ne pas être traités
 - Dans ce cas l'erreur est propagée

Traiter une erreur : do-catch

```
do {  
  
    try machine.vend(foodName: "Cake")  
  
} catch FoodDispenserError.InsufficientCoins(let missingCoins) {  
  
    print("Missing \u2028missingCoins\u2029 coins")  
  
} catch FoodDispenserError.OutOfStock {  
  
    print("Item is out of stock")  
  
} catch {  
  
    print(error)  
}
```

Traiter une erreur : optionnel

- Une méthode qui peut lancer une erreur peut être traitée comme un optionnel
- Utilisation de `try?`
- Permet d'utiliser la syntaxe `if-let` pour traiter simplement une erreur

```
if let food = try? machine.vend(foodName: "Cake") {  
    print("Everything was OK")  
}  
  
else { ... }
```

Traiter une erreur : supposer que non

- Lorsqu'on est certain qu'une erreur ne peut pas se produire, on peut ne pas la traiter et empêcher sa propagation
- Dans ce cas, on utilise `try` !
 - À utiliser avec précaution
 - Dans le cas où une erreur survient : crash

Traiter une erreur : supposer que non

- Lorsqu'on est certain qu'une erreur ne peut pas se produire, on peut ne pas la traiter et empêcher sa propagation
- Dans ce cas, on utilise `try` !
 - À utiliser avec précaution
 - Dans le cas où une erreur survient : crash

Initialisation

Généralités

- L'initialisation est le processus qui permet de préparer une instance à être utilisée
- Lors de l'initialisation, toutes les propriétés doivent se voir affecter une valeur
- L'initialisation peut soit passer par des valeurs par défaut aux propriétés, ou par l'implémentation de méthodes `init`

Généralités

```
class Human {  
  
    var age: Int  
    let name: String  
    var size: Float  
    var gender: String  
    var childrens: [Human]?  
}
```

Initialisation

```
class Human {  
  
    var age: Int  
    let name: String  
    var size: Float  
    var gender: String  
    var childrens: [Human]?  
  
    init() {  
  
        self.age = 0  
        self.name = "John Doe"  
        self.size = 175  
        self.gender = "Male"  
    }  
}
```

```
var childrens: [Human]?

init() {

    self.age = 0
    self.name = "John Doe"
    self.size = 175
    self.gender = "Male"
}

init(name: String, age: Int, size: Float, gender: String) {

    self.name = name
    self.age = age
    self.size = size
    self.gender = gender
}

}
```

```
var gender: String
var childrens: [Human]?

convenience init() {

    self.init(name: "John Doe", age: 0, size: 175, gender:
"Male")
}

init(name: String, age: Int, size: Float, gender: String) {

    self.name = name
    self.age = age
    self.size = size
    self.gender = gender
}
}
```

Convenience initialiser

- Un convenience initialiser est un initialiseur qui fait appel à un autre initialiseur au sein du même type, avec `self.init(...)`
- Un convenience initialiser dans une classe doit être préfixé par convenience
- Un convenience initialiser ne peut pas être appelé par une sous-classe

Cas de l'héritage

```
class Student: Human {  
    var school = "Not enrolled"  
    var serious = false  
    var grade = 0  
}
```

Initialisation

```
class Student: Human {  
  
    var school: String  
    var serious: Bool  
    var grade: Int  
  
    init(name: String, age: Int, size: Float, gender: String,  
school: String, isSerious: Bool, grade: Int) {  
  
        self.school = school  
        self.serious = isSerious  
        self.grade = grade  
  
        super.init(name: name, age: age, size: size, gender:  
gender)  
    }  
}
```

Initialisation

```
self.serious = isSerious  
self.grade = grade
```

```
super.init(name: name, age: age, size: size, gender:  
gender)  
}
```

```
convenience init() {
```

```
    self.init(name: "Student Jane Doe", age: 18, size:  
180, gender: "Female", school: "Not enrolled", isSerious:  
true, grade: 10)  
}
```

```
}
```

Initialisation

```
convenience init() {  
    self.init(name: "Student Jane Doe", age: 18, size:  
180, gender: "Female", school: "Not enrolled", isSerious:  
true, grade: 10)  
}  
  
convenience override init(name: String, age: Int, size:  
Float, gender: String) {  
    self.init(name: name, age: age, size: size, gender:  
gender, school: "Not enrolled", isSerious: true, grade: 10)  
}  
}
```

Cas de l'héritage

- Si une classe hérite d'une autre, elle doit s'assurer que les propriétés issues de cette classes ont également des valeurs
- Dans le cas de l'héritage, on doit faire appel à l'initialiseur de la classe parente avec `super.init(...)`
- Si vous surchargez un initialiseur de la classe parente, vous devez rajouter `override` devant le `init`

Failable initialiser

- Un failable initialiser est un initialiseur qui peut échouer.
- Dans certains cas, il se peut qu'un initialiser ne puisse pas créer un objet (argument invalide par exemple)
- Dans ce cas, à l'issue de l'initialisation, on aura un optionnel au lieu de l'instance

```
init?() {  
    // If something blocks init, simply return nil  
}
```

Désinitialisation

Généralités

- La désinitialisation est le processus qui permet de détruire une instance et de libérer les ressources associées
- La désinitialisation est gérée automatiquement par le langage, et les ressources libérées grâce à ARC (Automatic Reference Counting)
- Dans certains cas particuliers, il peut être utile de pouvoir faire des actions lors de cette désinitialisation, pour cela on implémente la méthode `deinit`
- Seules les instances de classes peuvent avoir un désinitialiseur

Généralités

- Seules les instances de classes peuvent avoir un désinitialiseur
- On n'appelle jamais `deinit` nous même, cette méthode est appelée automatiquement au moment opportun

```
init() {  
    self.age = 0  
    self.name = "John Doe"  
    self.size = 175  
    self.gender = "Male"  
  
    self.childrens = nil  
  
    print("\(self.name) is being created")  
}  
  
deinit{  
    print("\(self.name) is being destroyed")  
}
```

```
        print("\(self.name) is being created")  
    }  
  
deinit{  
    print("\(self.name) is being destroyed")  
}  
  
var aHuman: Human? = Human()
```

```
        print("\(self.name) is being created")
    }

deinit{
    print("\(self.name) is being destroyed")

}

var aHuman: Human? = Human()
```

John Doe is being created

```
        print("\(self.name) is being created")
    }

deinit{
    print("\(self.name) is being destroyed")

}

var aHuman: Human? = Human()

aHuman = nil
```

John Doe is being created

John Doe is being destroyed

Gestion de la mémoire : ARC

Généralités

- Plateforme mobile = ressources limitées
- Gestion de la mémoire efficace indispensable !
- Donc pas de Garbage Collection

Principe

- Langage de haut niveau = gestion de haut niveau
 - Ne pas se soucier de quand il faut libérer l'instance
 - Simplicité pour le développeur
 - Mais avoir une instance libérée dès qu'elle n'est plus utilisée
 - Utilisation efficace de la mémoire

Principe

- Historiquement, la gestion était manuelle
 - On indiquait quand on utilisait un objet, et quand on avait fini de s'en servir
 - Cela influait sur un compteur de référence (nombre de référence pointant sur un même objet)
- Depuis iOS 5, il existe un système de gestion automatique (ARC).
 - Le développeur n'a plus à gérer le compteur de référence
 - Les nouveaux projets sont en ARC

Automatic Reference Counting

- Fonctionnalité au niveau compilateur
- Compteur de référence géré par le compilateur
- Insère le code de gestion de la mémoire à la compilation
- "Meilleur des deux mondes"

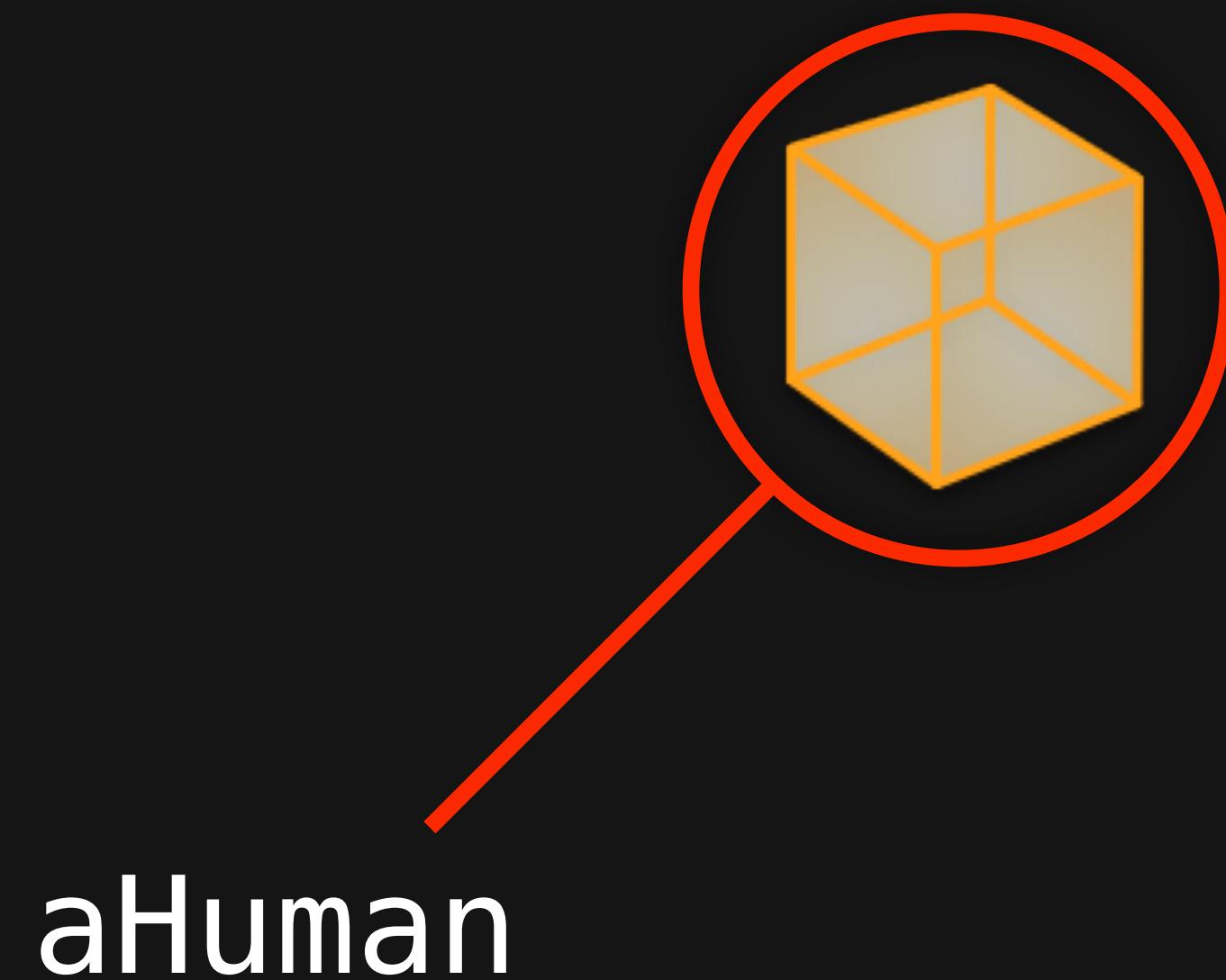
Automatic Reference Counting

- Lors de la création d'un instance de classe ARC crée une référence **strong** entre la propriété et l'instance
- Tant qu'une référence **strong** existe pour une instance, celle-ci reste allouée en mémoire
- Il existe d'autre type de références qui ne suffisent pas à maintenir une instance en mémoire

Automatic Reference Counting

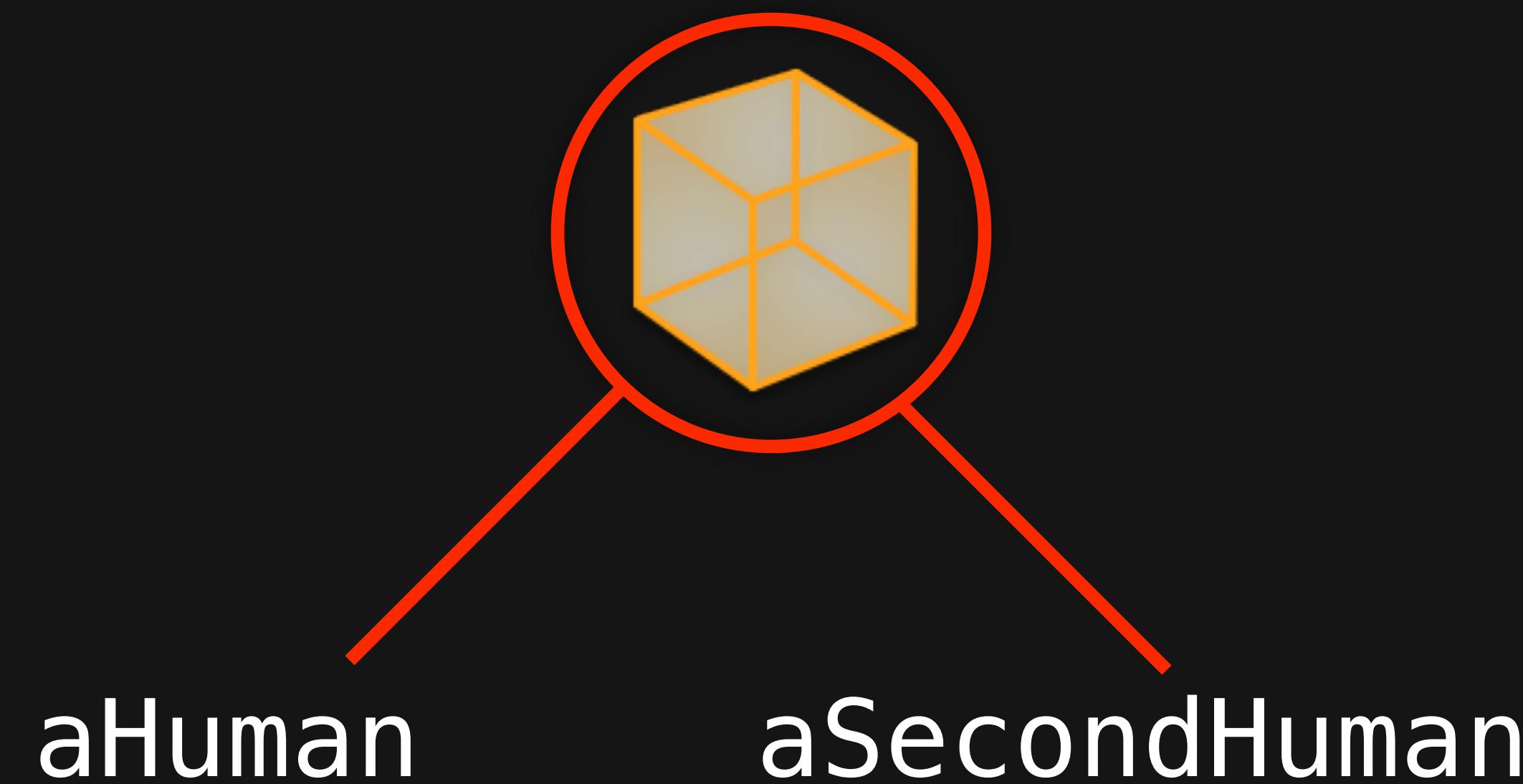
Automatic Reference Counting

```
var aHuman: Human? = Human()
```



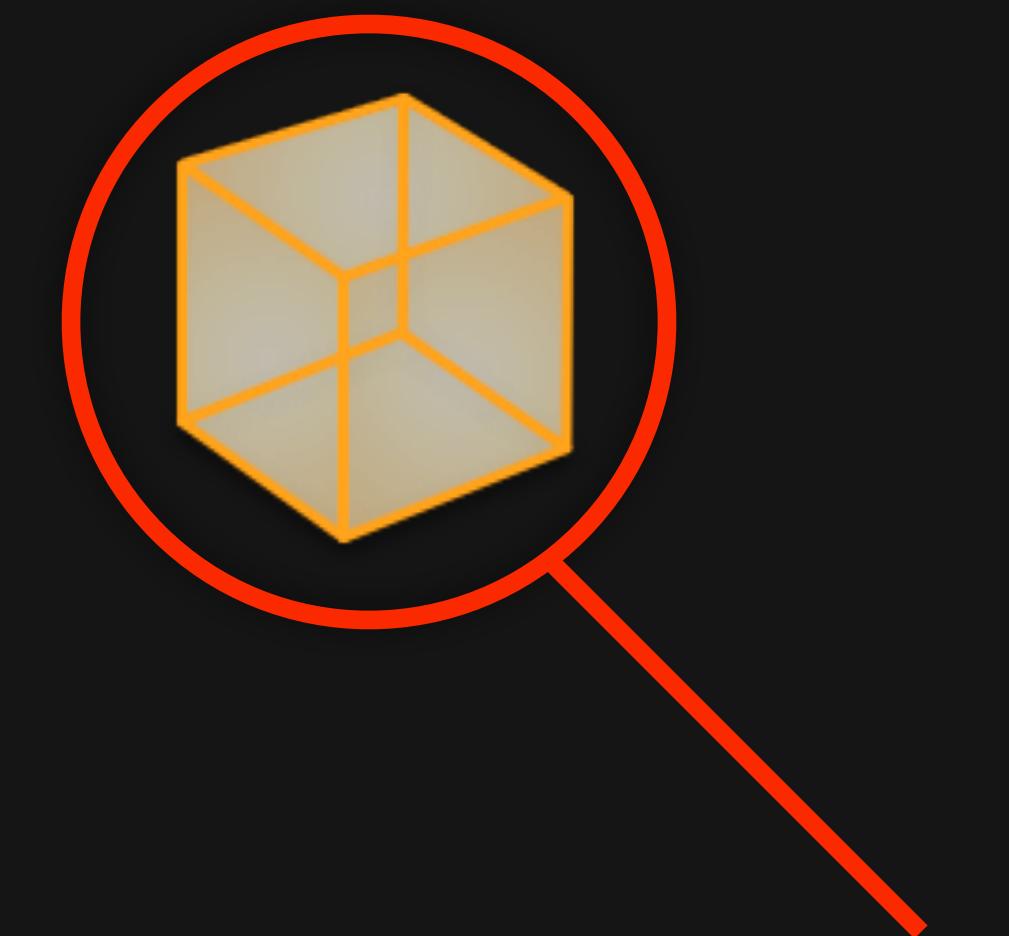
Automatic Reference Counting

```
var aSecondHuman = aHuman
```



Automatic Reference Counting

aHuman = nil



aSecondHuman

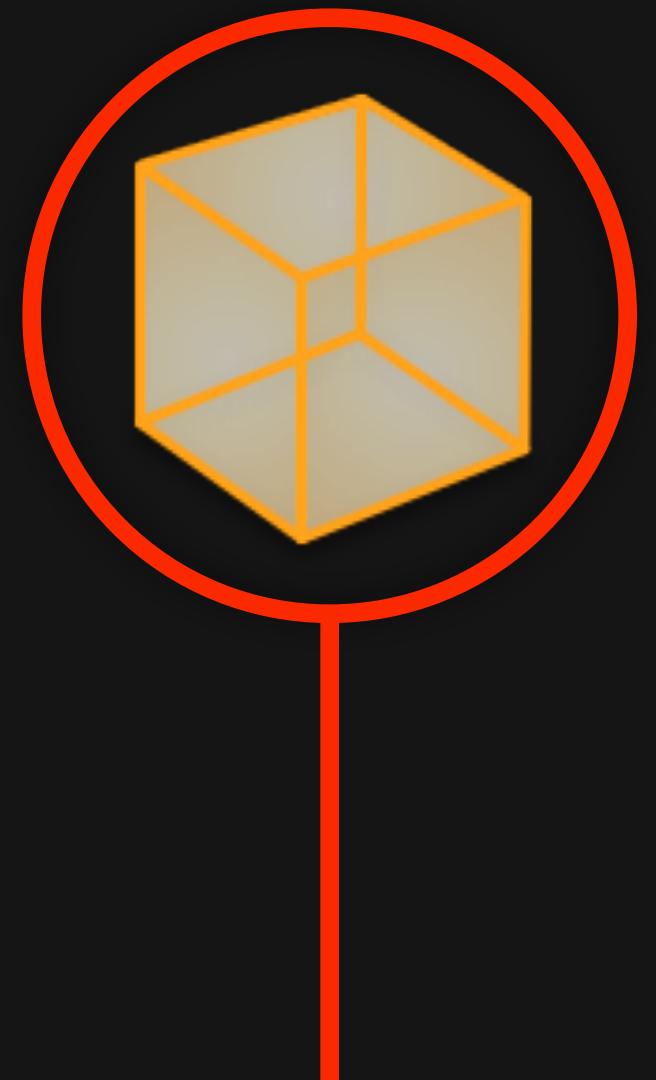
Automatic Reference Counting

aSecondHuman = nil

Cycle de références

Cycle de références

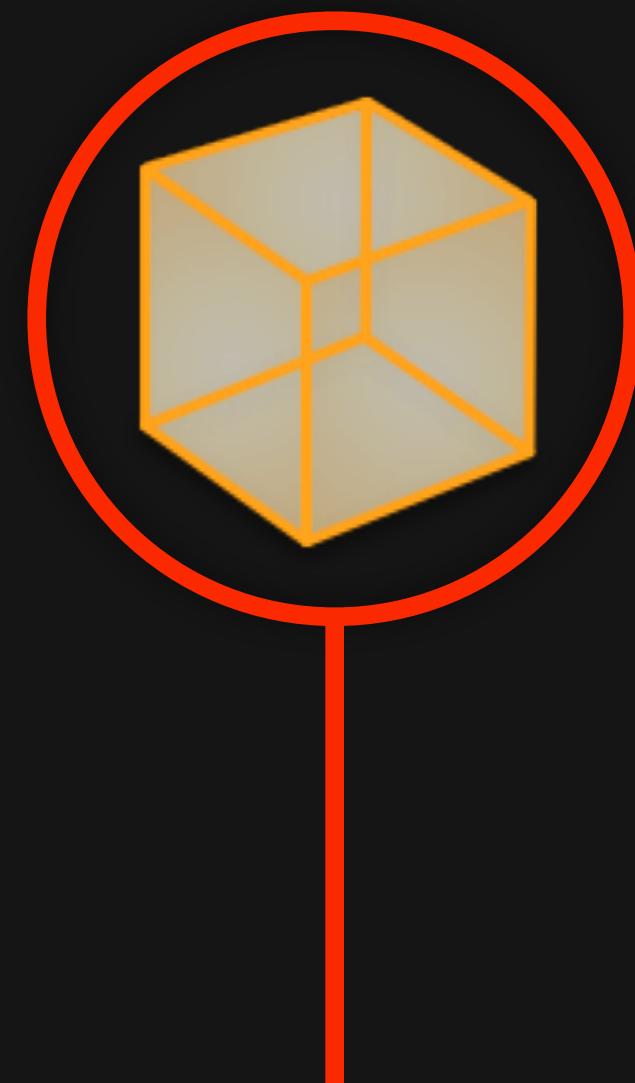
```
var aHuman: Human? = Human()
```



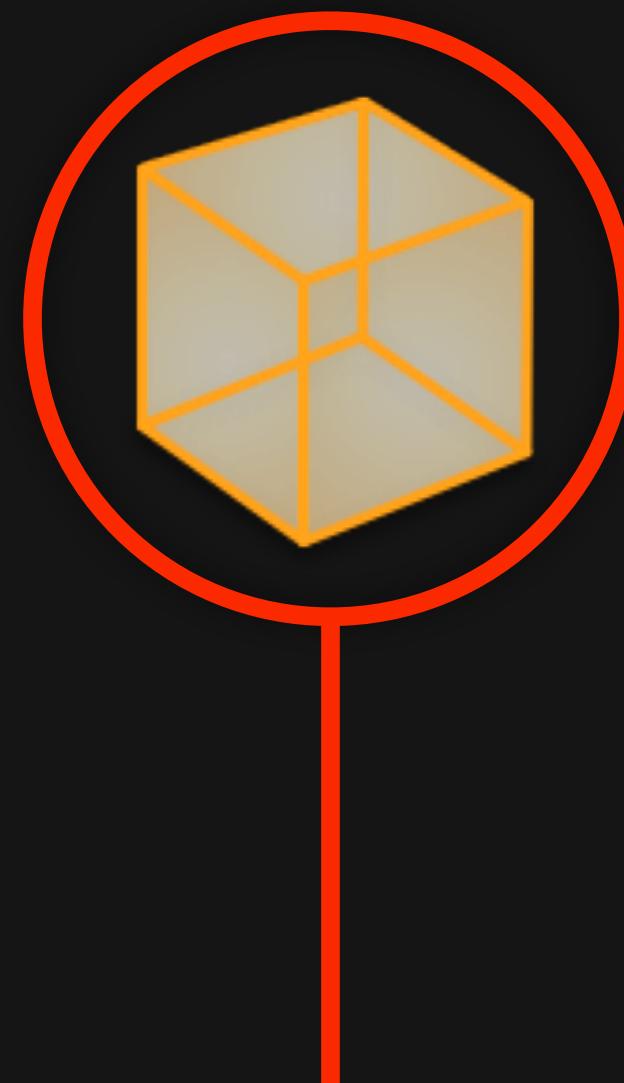
aHuman

Cycle de références

```
var aSecondHuman: Human? = Human()
```



aHuman



aSecondHuman

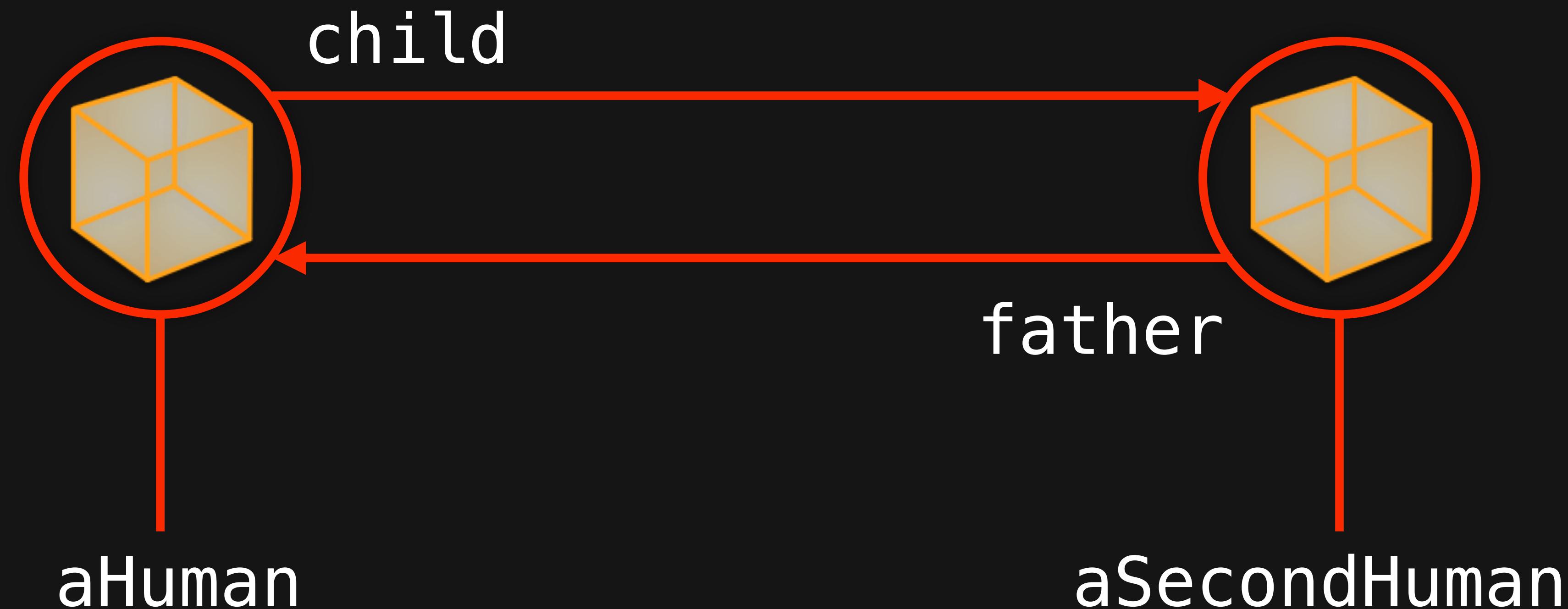
Cycle de références

`aHuman.child = aSecondHuman`



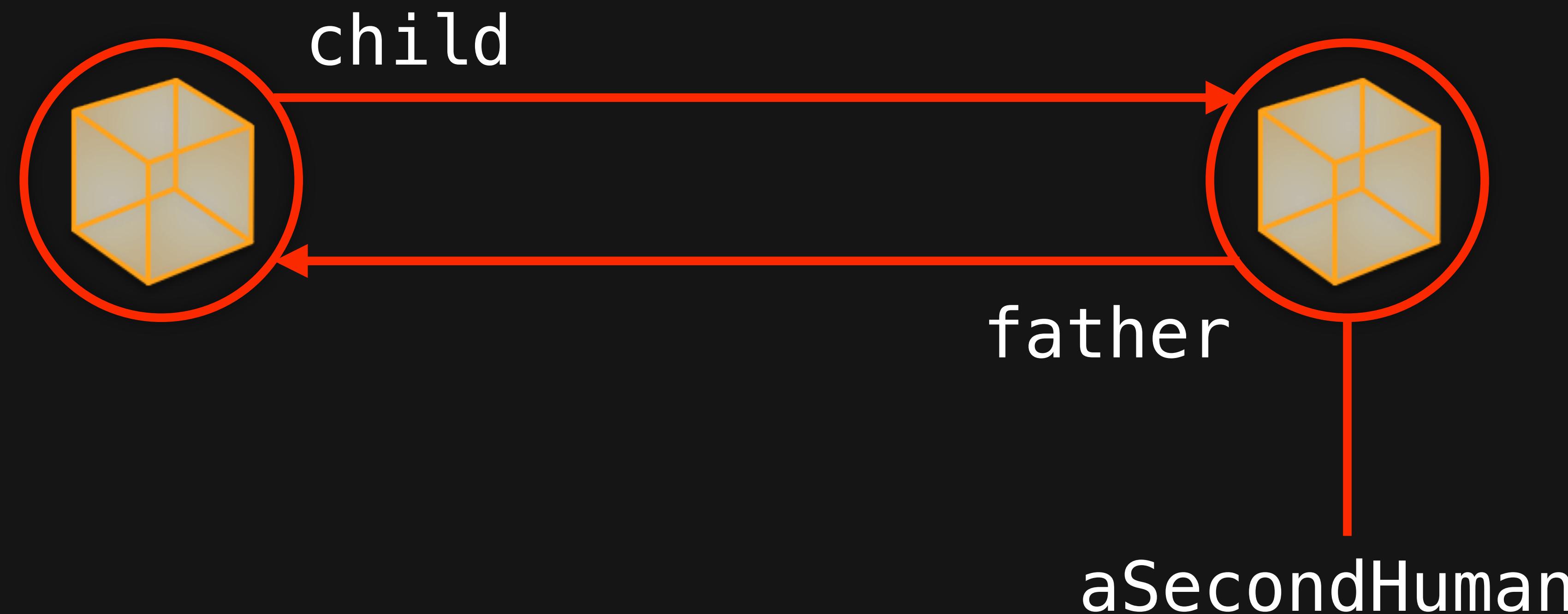
Cycle de références

`aSecondHuman.father = aHuman`



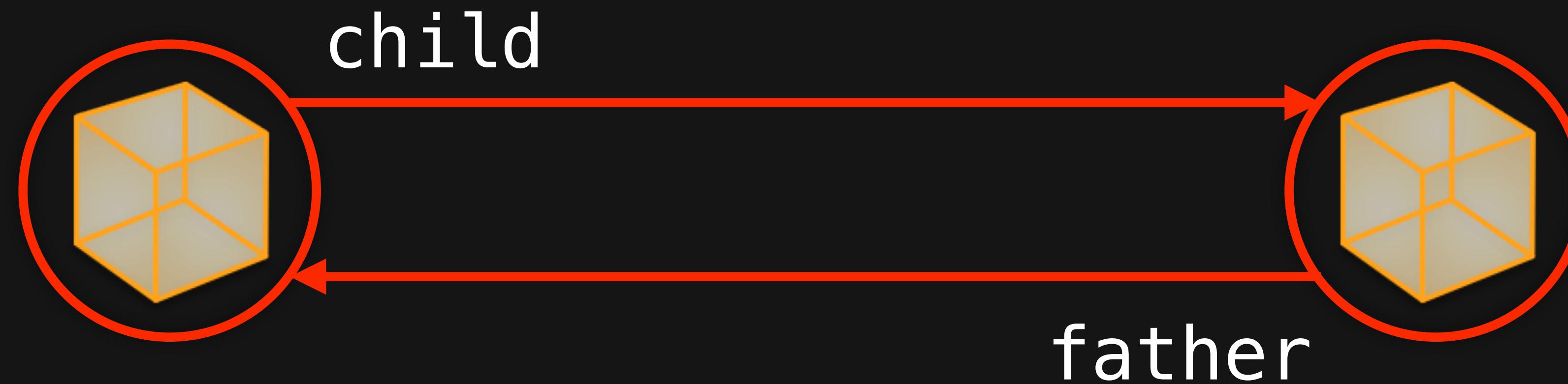
Cycle de références

aHuman = nil



Cycle de références

aSecondHuman = nil



Cycle de références



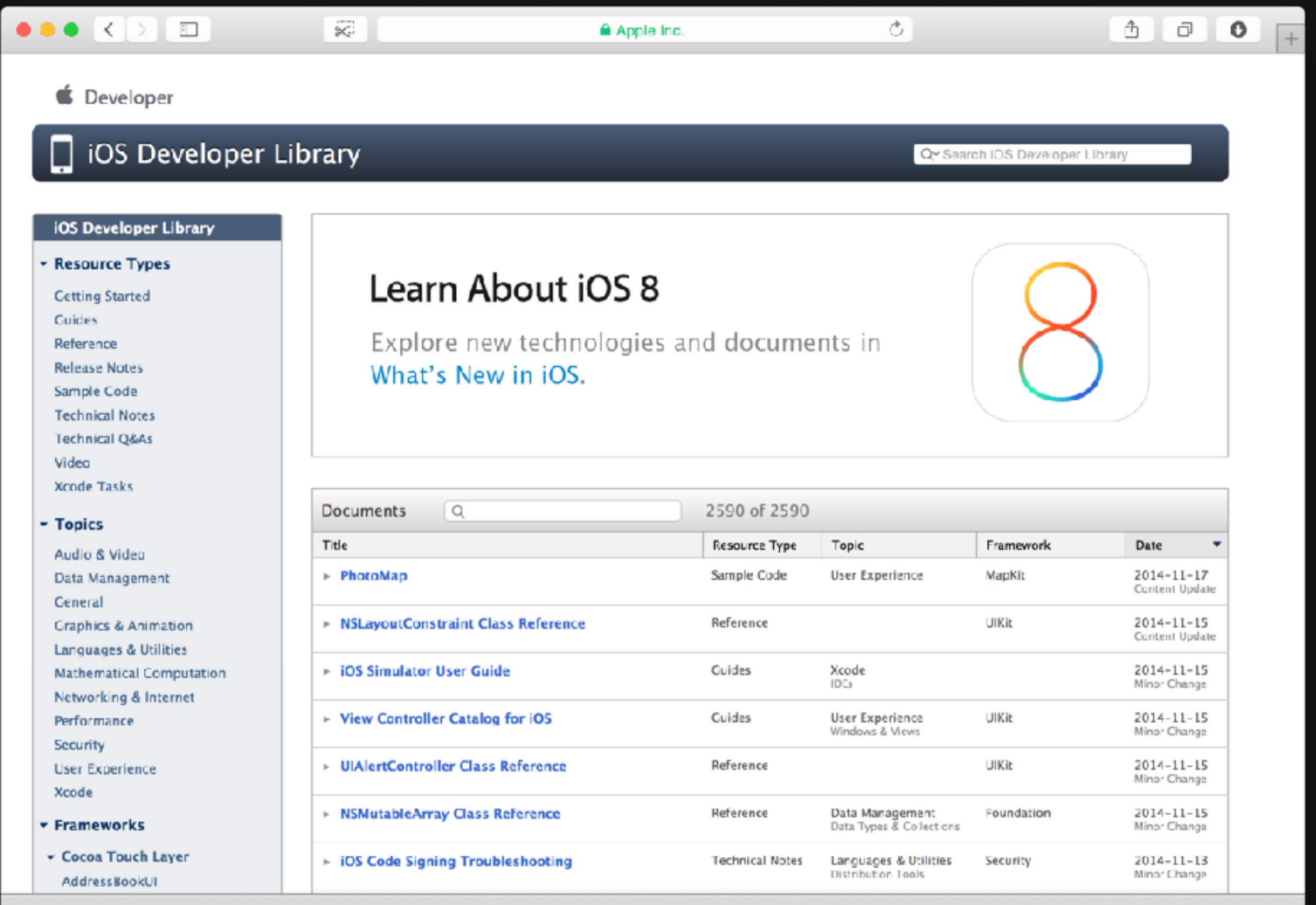
Cycle de références

Automatic Reference Counting

- **strong** : l'objet doit rester vivant tant qu'on pointe vers lui (par défaut)
- **weak** : l'objet restera vivant tant qu'un pointeur **strong** pointera vers lui
 - La référence est mise à **nil** lorsque l'instance est détruite
 - Une propriété **weak** doit être un optionnel
- **unowned** : l'objet restera vivant tant qu'un pointeur **strong** pointera vers lui
 - La référence n'est pas mise à **nil**. Un appel à la référence après destruction de l'instance engendrera un crash
 - **unowned** est utilisé dans les cas où la propriété est supposée toujours avoir une valeur pendant la vie d'une instance

La documentation

La documentation

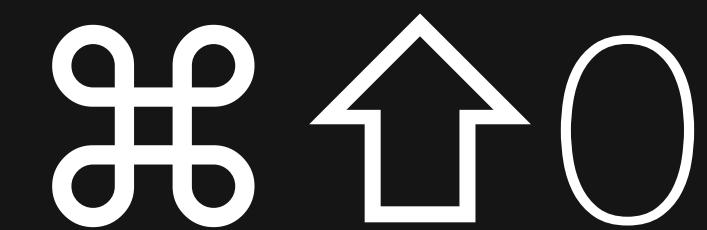


developer.apple.com/library/ios

XCODE



La documentation



developer.apple.com/library/ios

XCODE

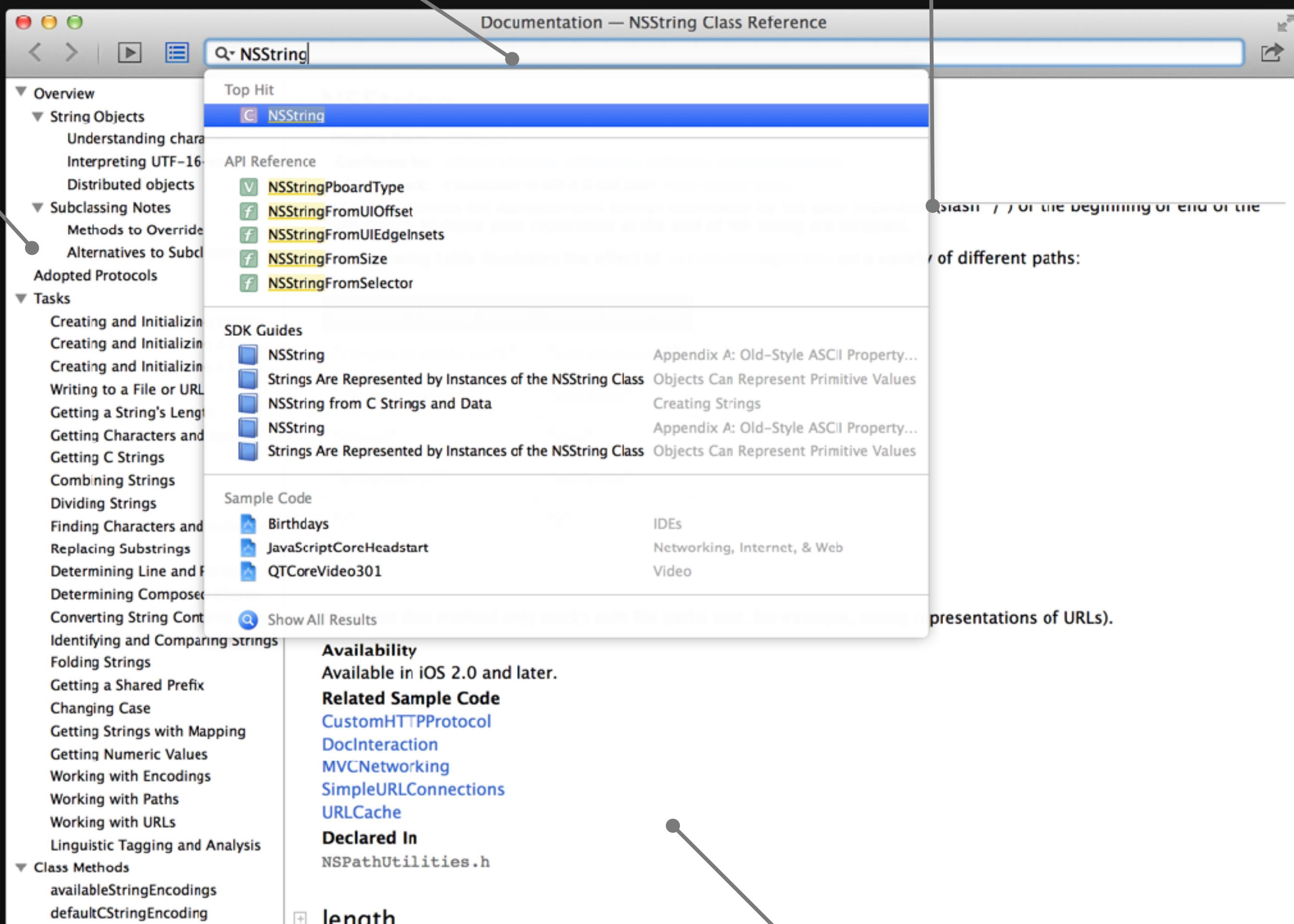


La documentation

Recherche

Résultats

Table des matières



Documentation

Description The [UILabel](#) class implements a read-only text view. You can use this class to draw one or multiple lines of static text, such as those you might use to identify other parts of your user interface. The base [UILabel](#) class provides support for both simple and complex styling of the label text. You can also control over aspects of appearance, such as whether the label uses a shadow or draws with a highlight. If needed, you can customize the appearance of your text further by subclassing.

Availability [iOS \(2.0 and later\)](#)

Declared In [UILabel.h](#)

Reference [UILabel Class Reference](#)

XCODE QUICK HELP
⌘ clic

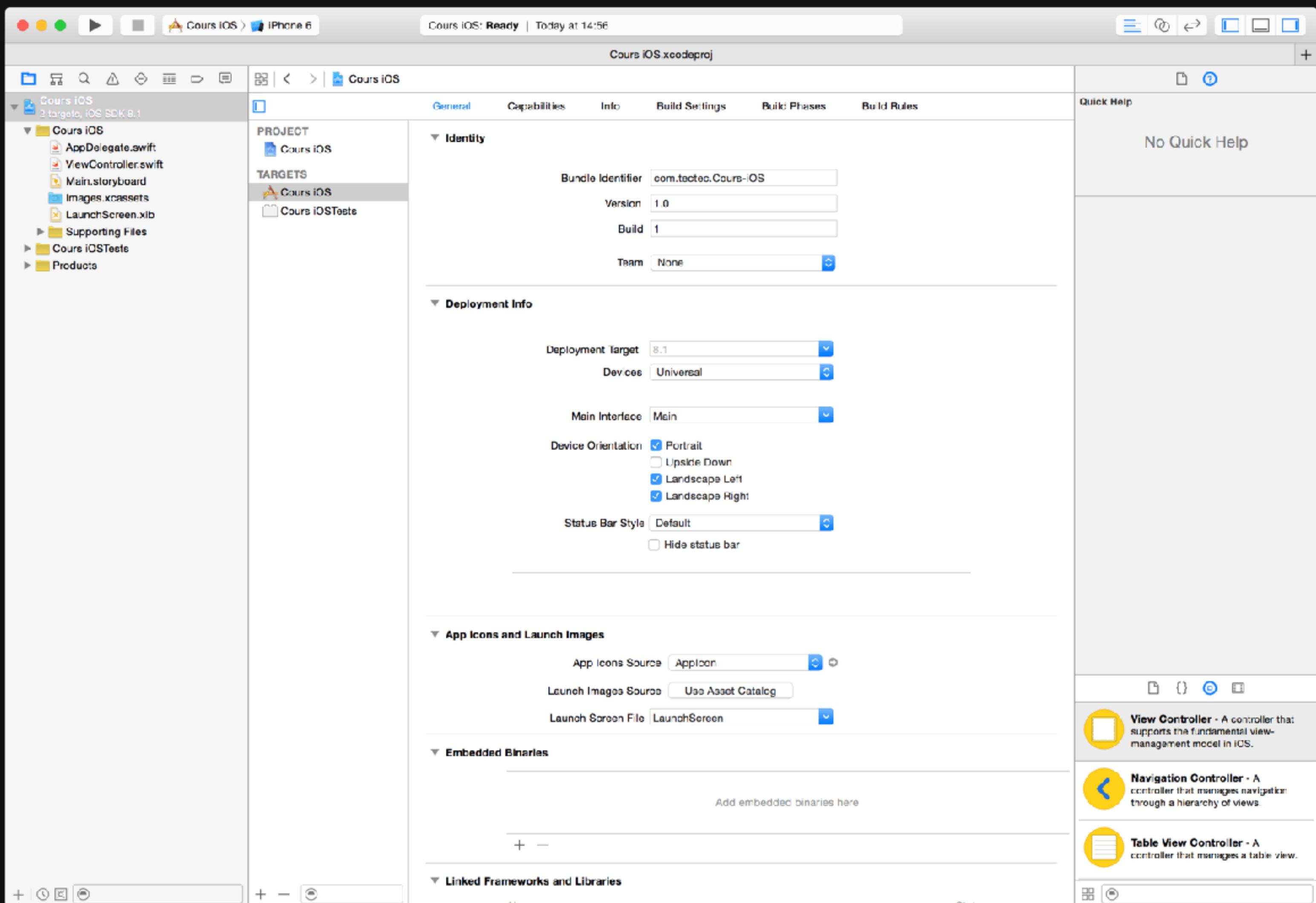
Du playground à l'application

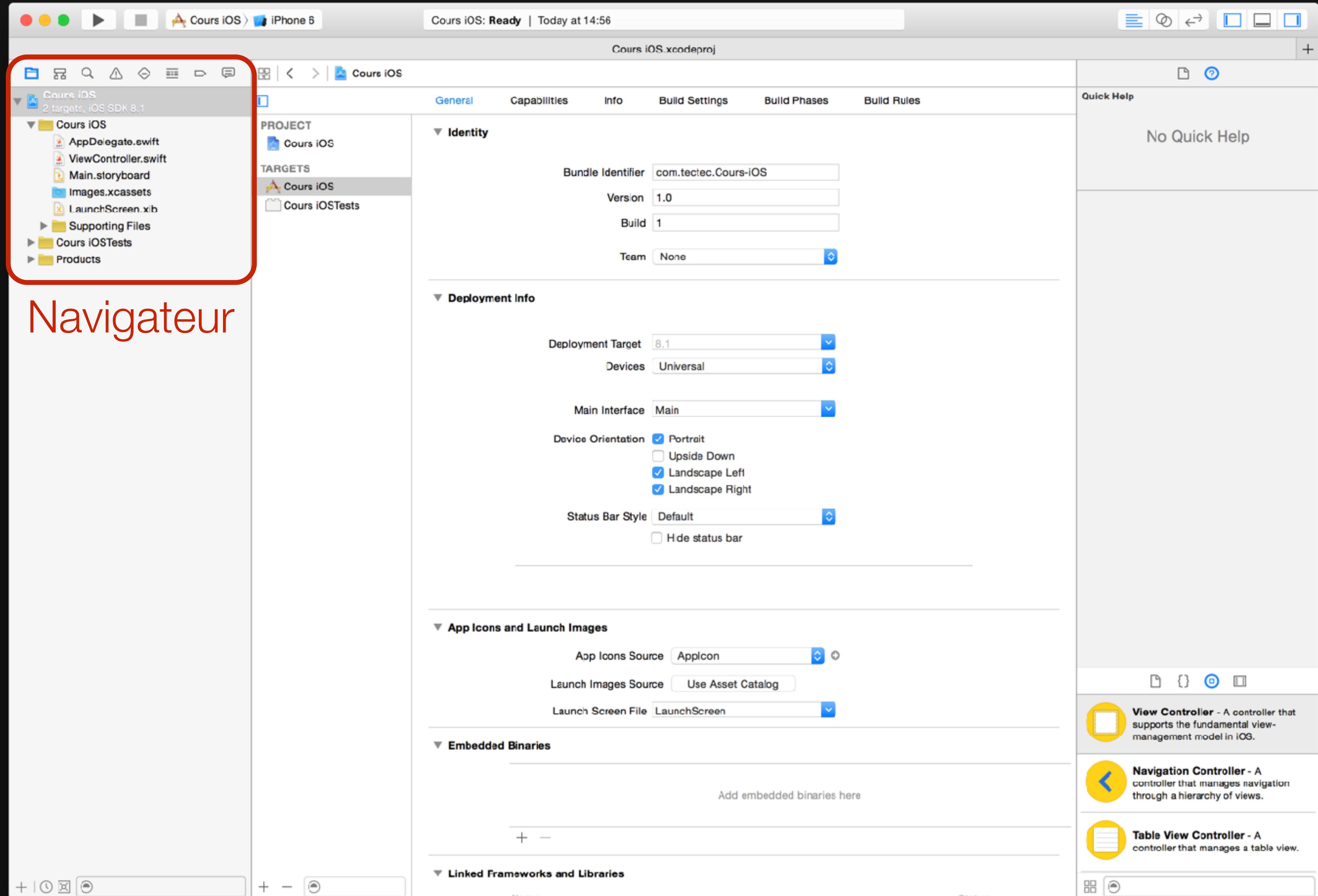
Généralités

- Les playgrounds sont utile pour apprendre le langage, peaufiner des algorithmes et faire des tests rapides
- Mais on voudra rapidement aller plus loin : créer une application iOS complète
- Pour cela, il va falloir découvrir comme lier notre code et notre interface graphique

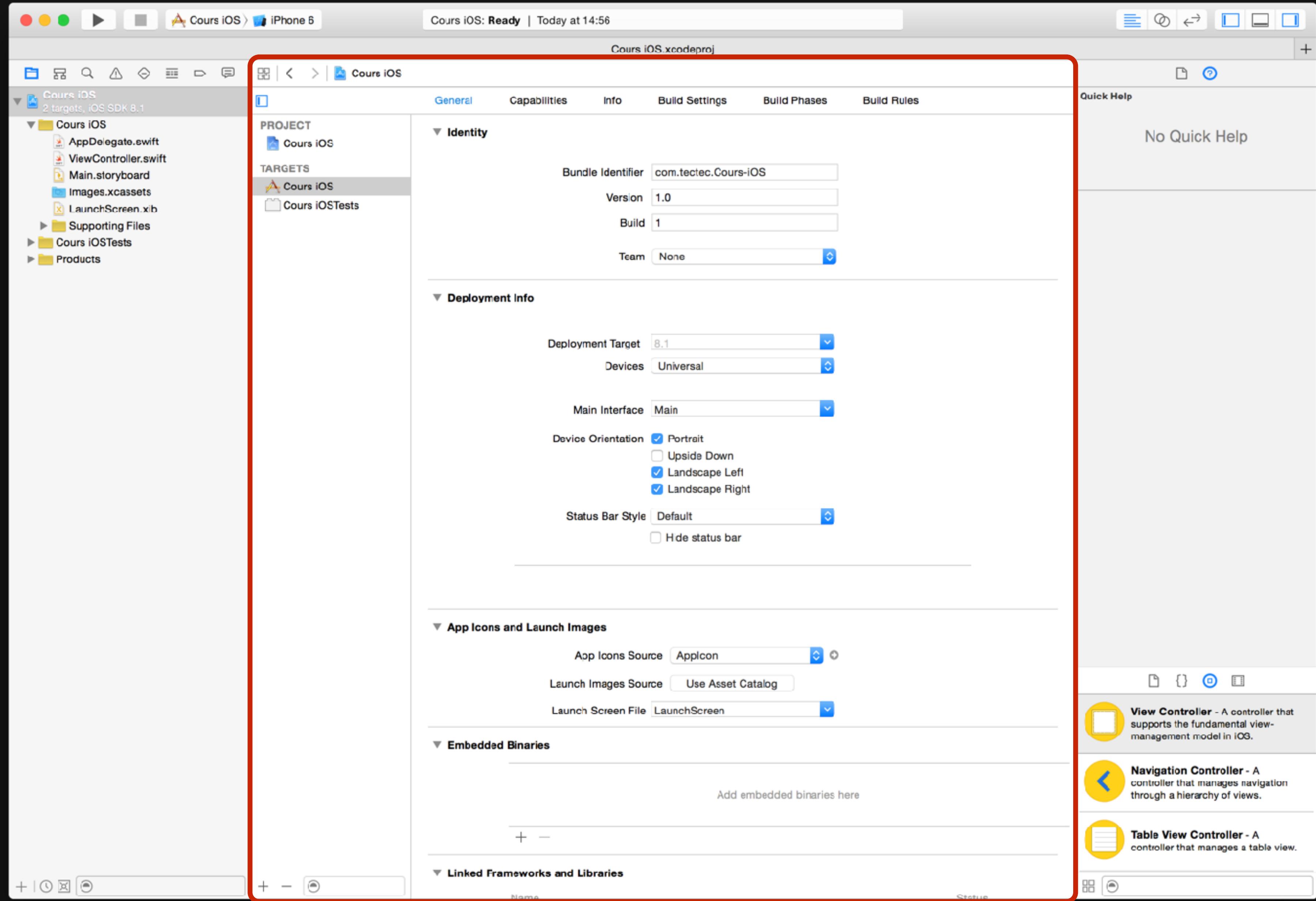
Du playground à l'application

Architecture d'un projet Xcode

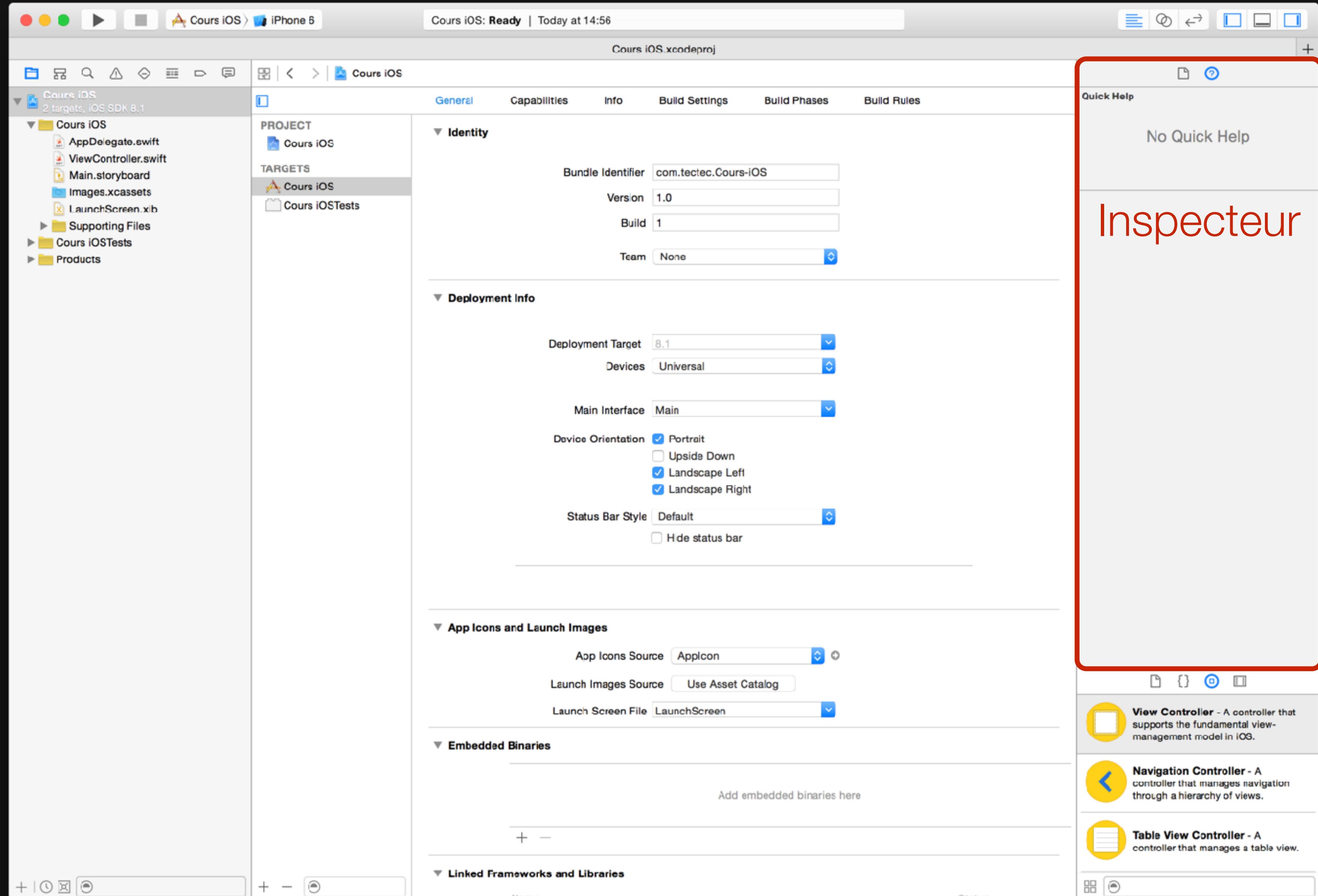


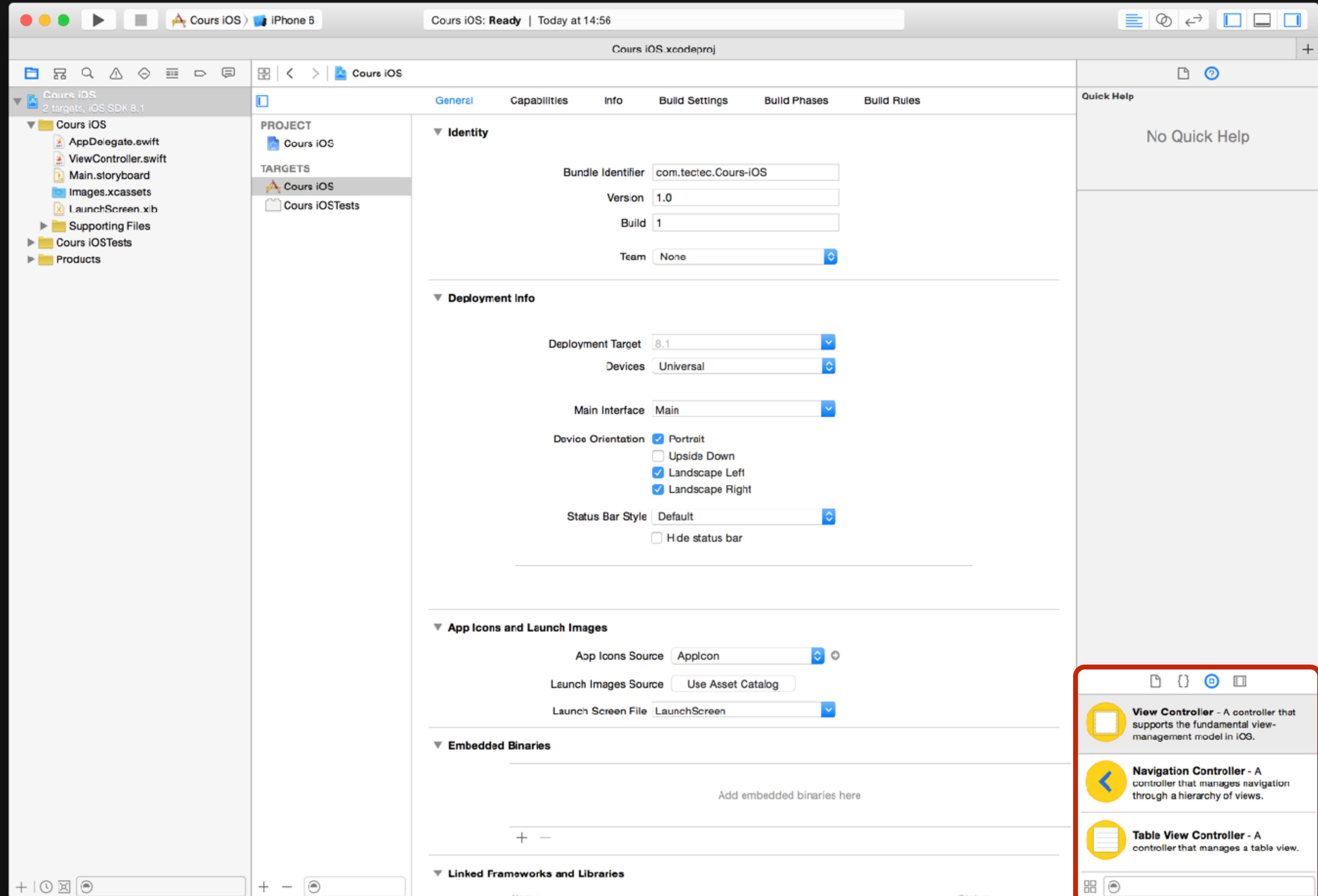


Navigateur



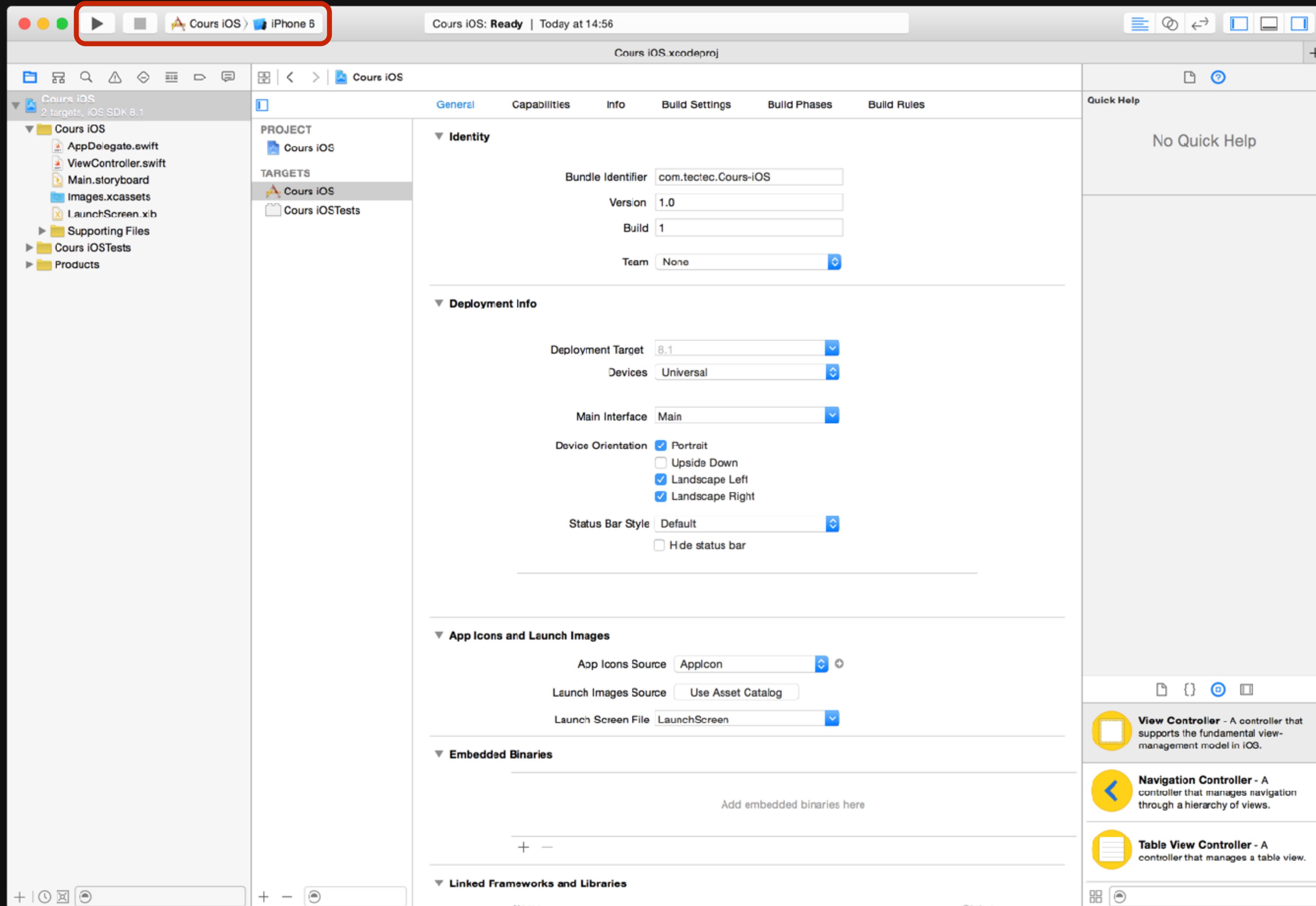
Vue centrale



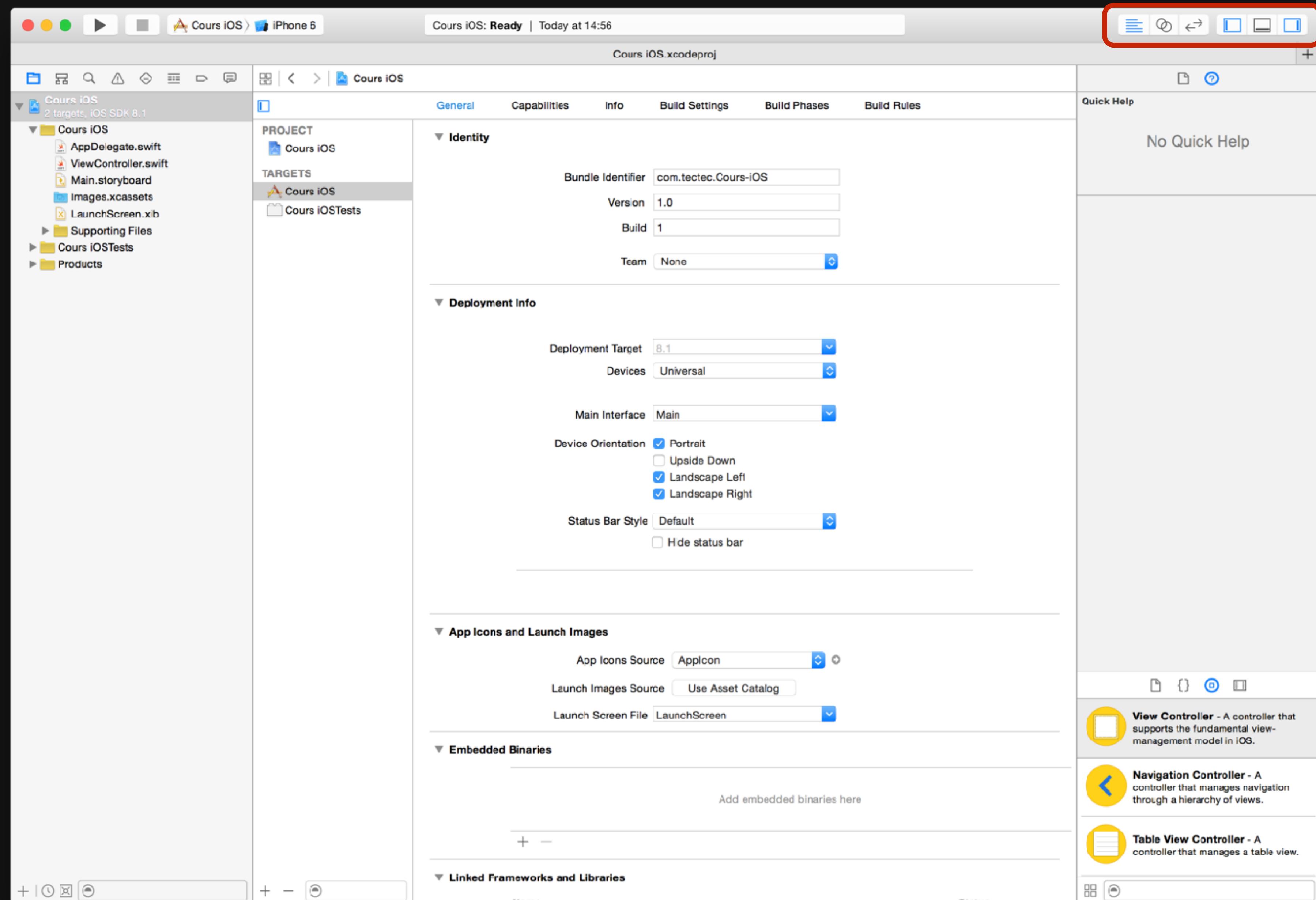


Bibliothèque

Contrôle de l'application



Contrôle des vues



Architecture d'un projet Xcode

- Fichiers .swift
 - Code
 - AppDelegate permet de réagir aux changements d'états de l'application
- Fichiers .storyboard
 - Interface graphique
- Fichiers .xcassets
 - Bibliothèque de ressources graphiques

Lien avec l'interface

- Action : L'interface graphique appelle une méthode sur une instance
- Outlet : Propriété faisant référence à un élément de l'interface graphique

Lien avec l'interface

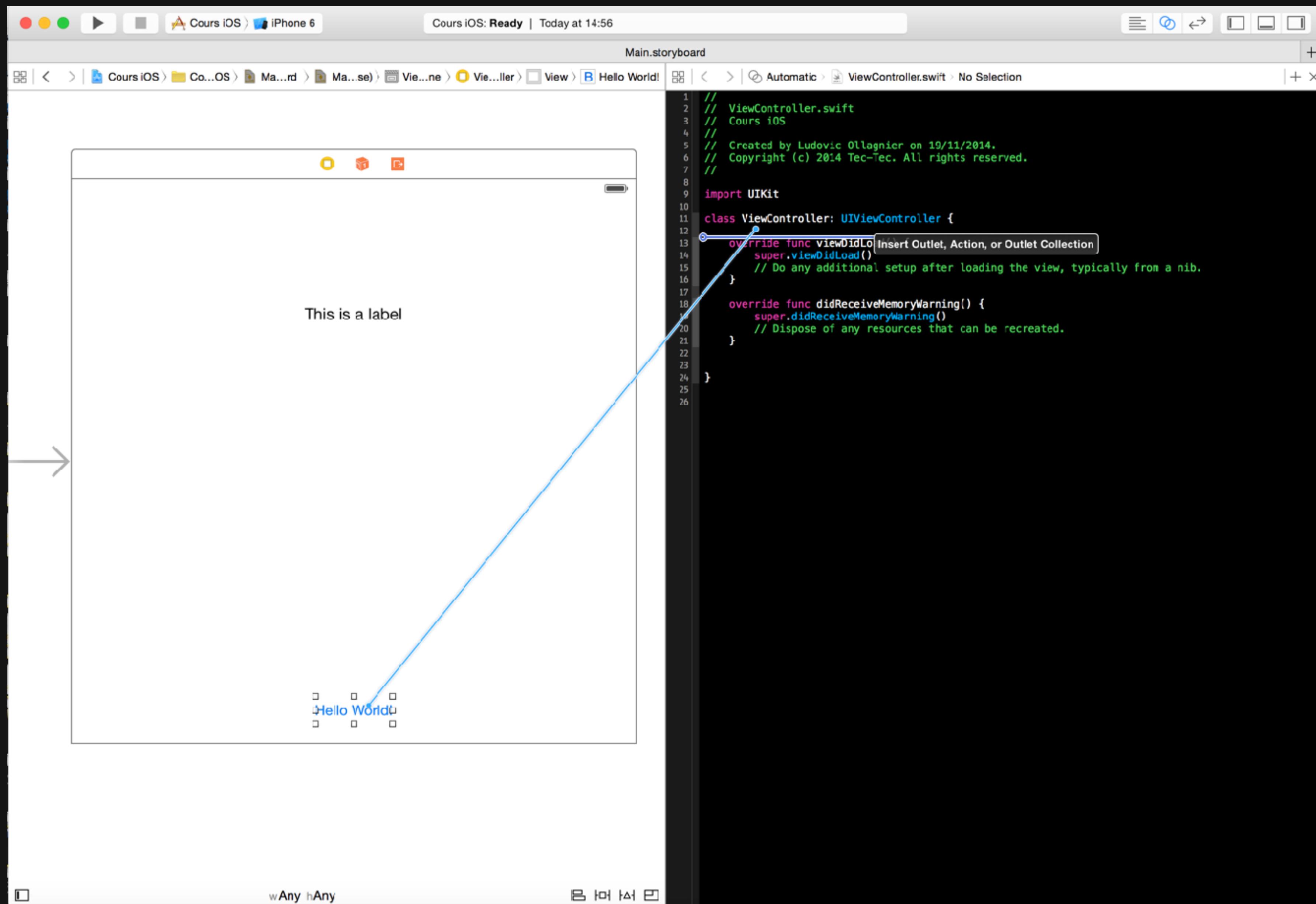
- Déclaration d'une action

```
@IBAction func actionName(_ sender: AnyObject) {  
}
```

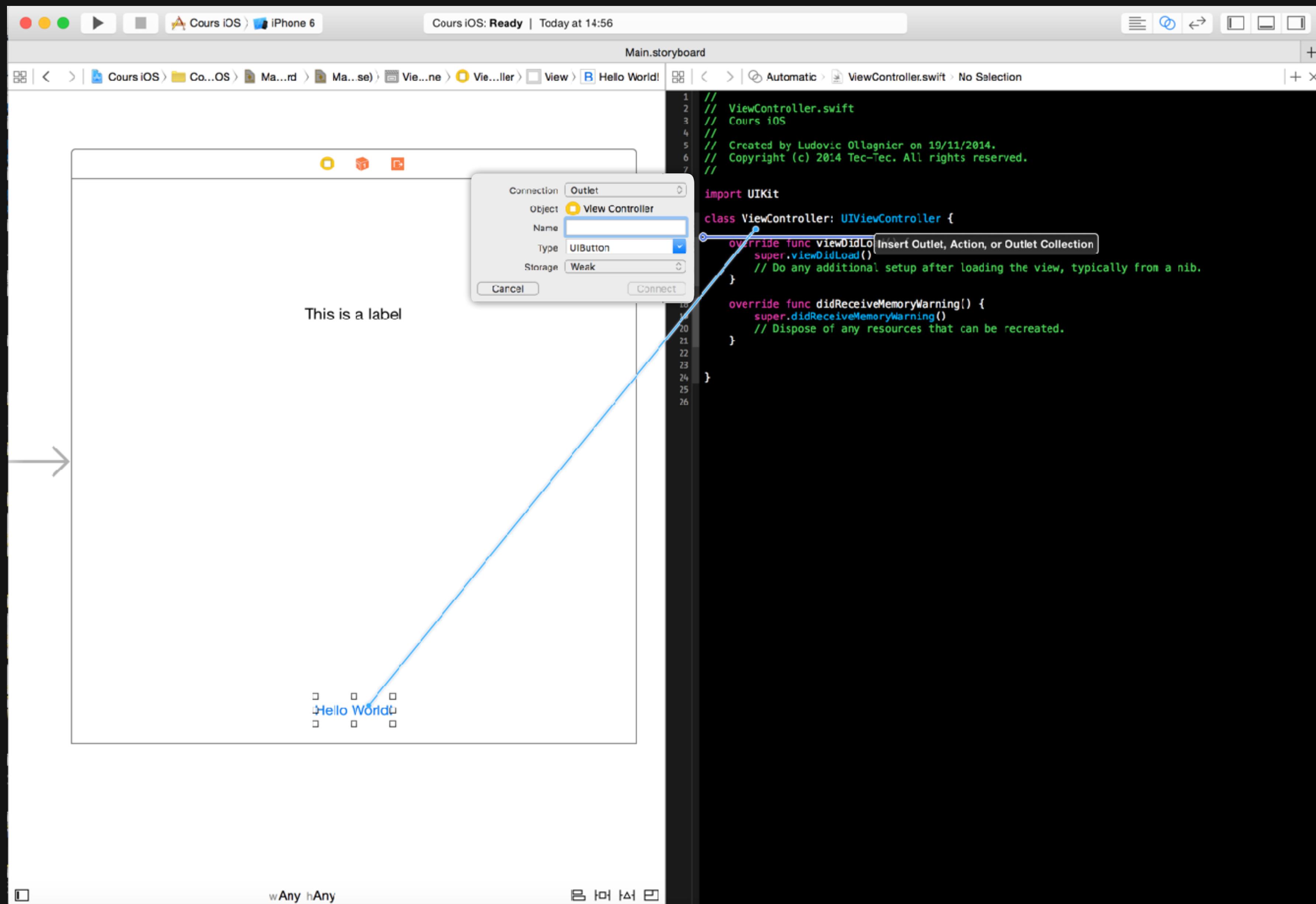
- Déclaration d'un outlet

```
@IBOutlet weak var outletName: OutletType!
```

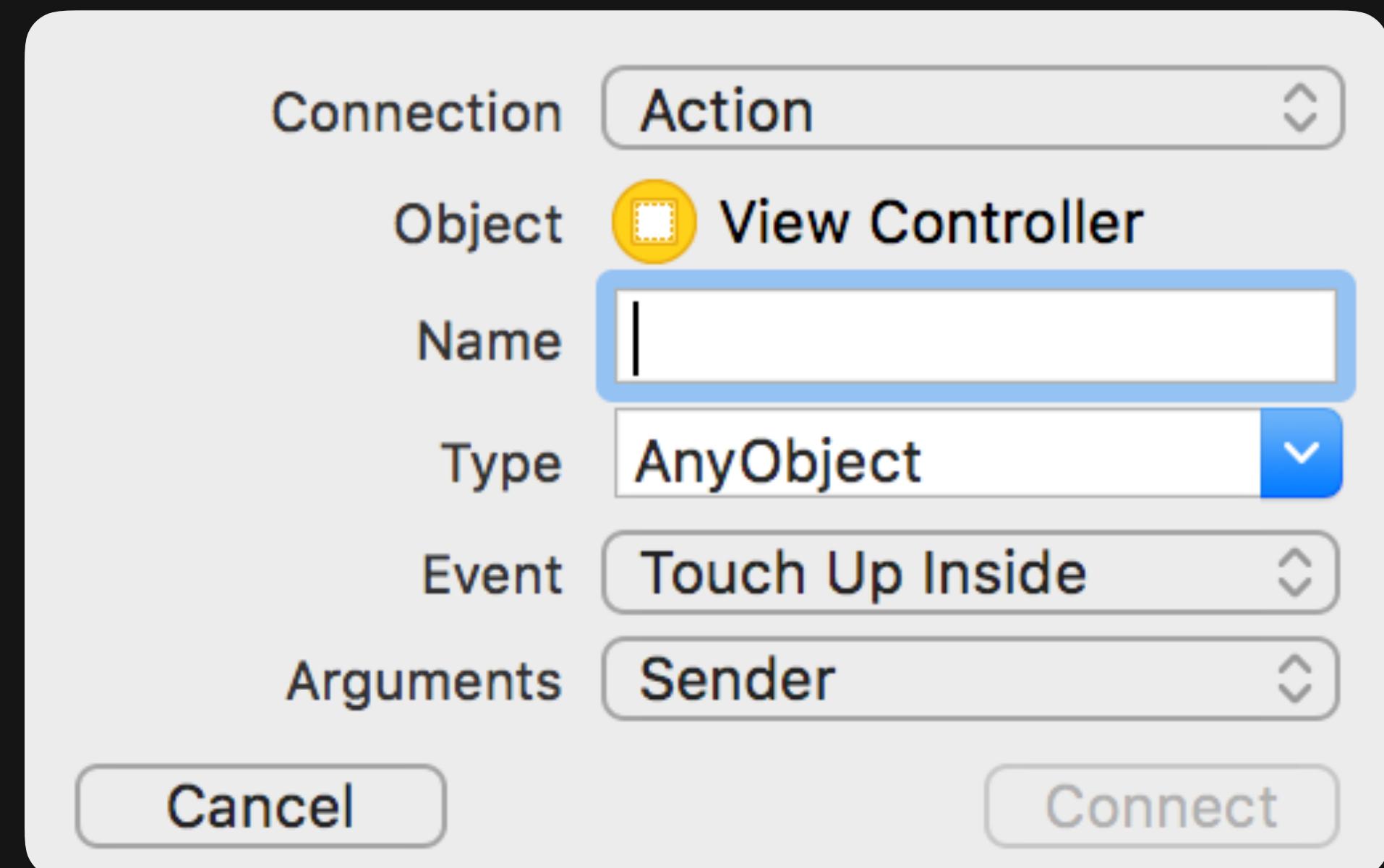
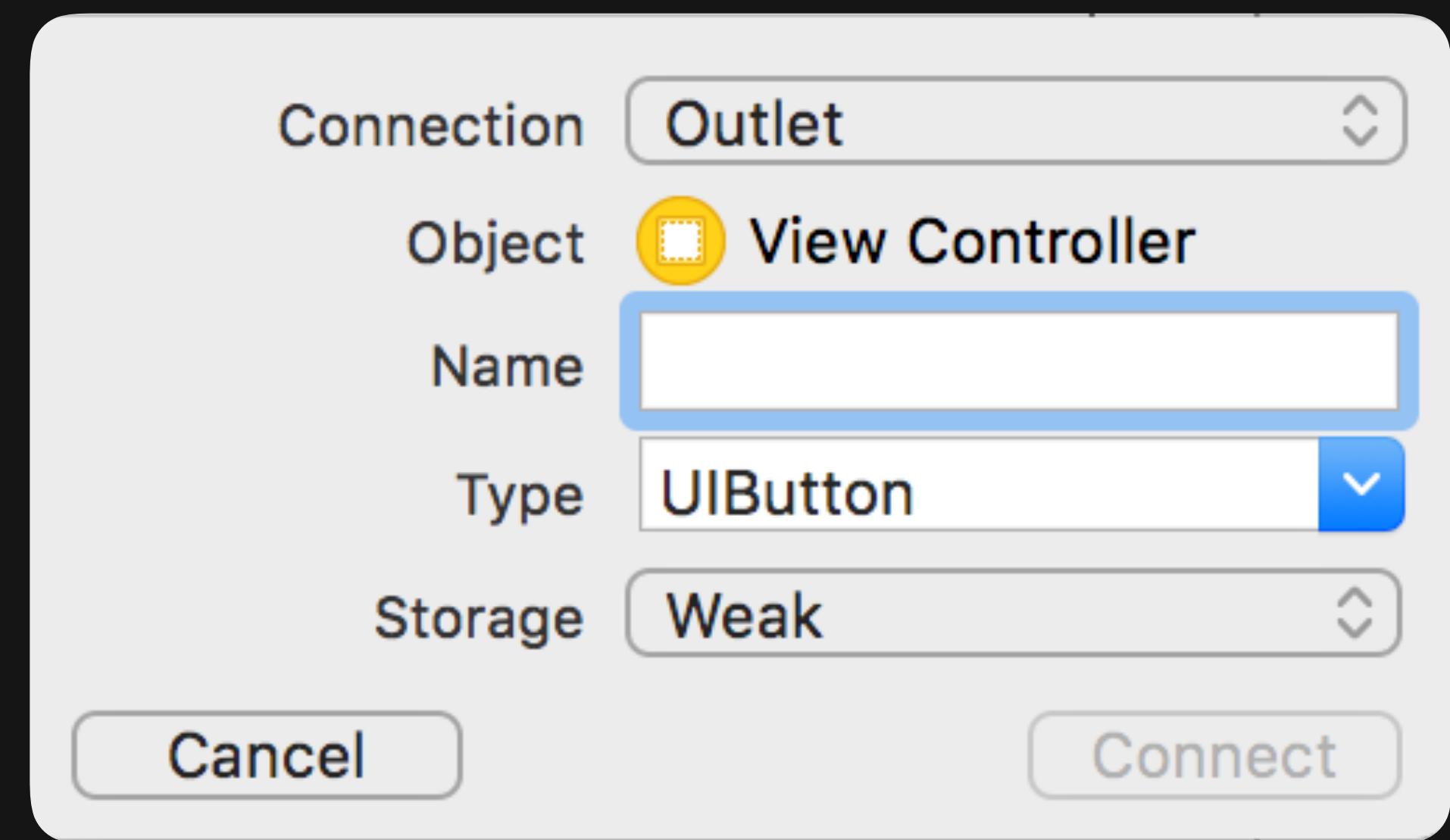
Du playground à l'application



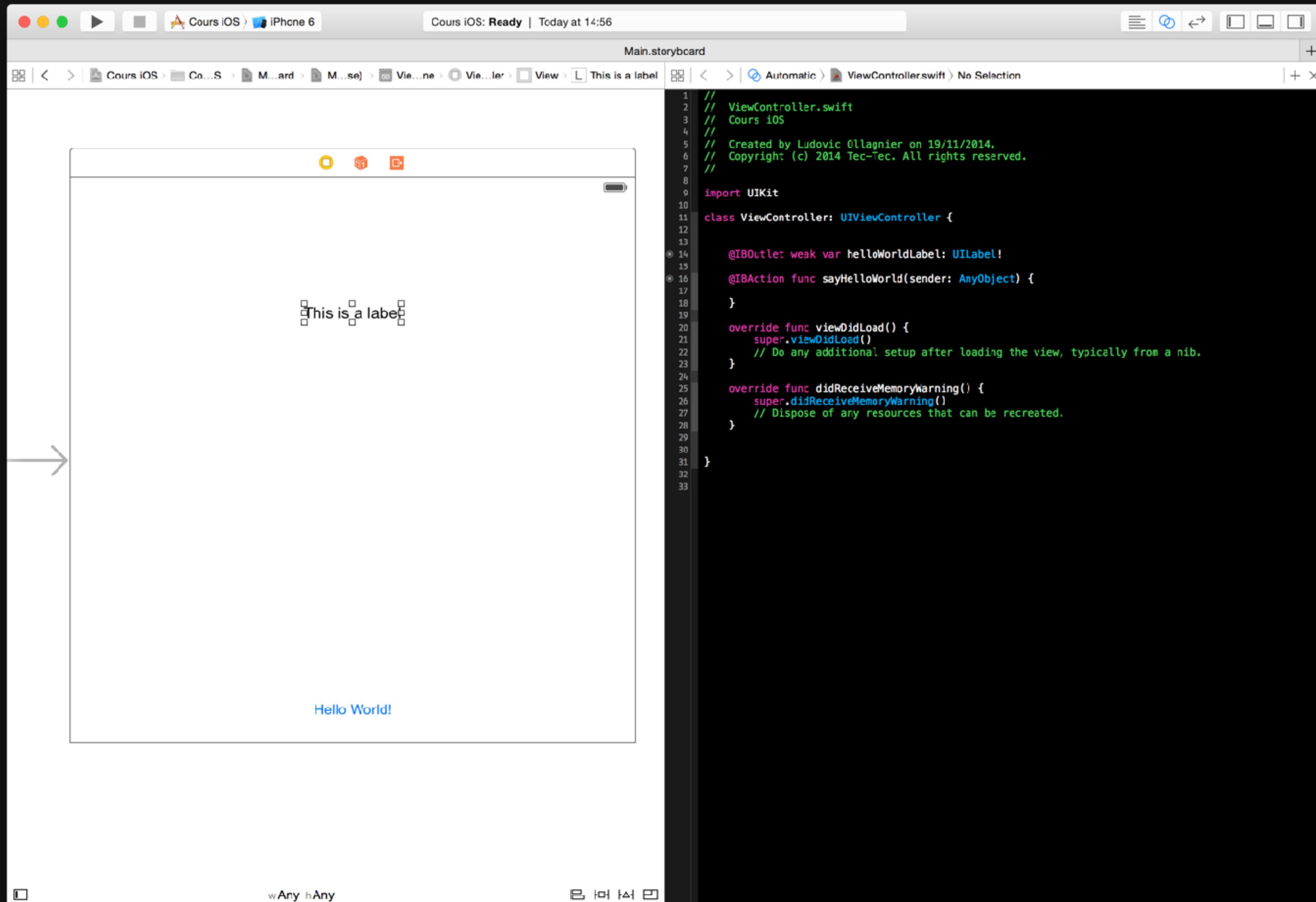
Du playground à l'application



Du playground à l'application



Du playground à l'application



Du playground à l'application

The screenshot shows the Xcode interface with two main windows open:

- Main.storyboard:** Displays a single view controller containing a label with the text "This is a label".
- ViewController.swift:** Shows the corresponding Swift code:

```
// ViewController.swift
// Cours iOS
//
// Created by Ludovic Ollagnier on 19/11/2014.
// Copyright (c) 2014 Tec-Tec. All rights reserved.

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var helloWorldLabel: UILabel!

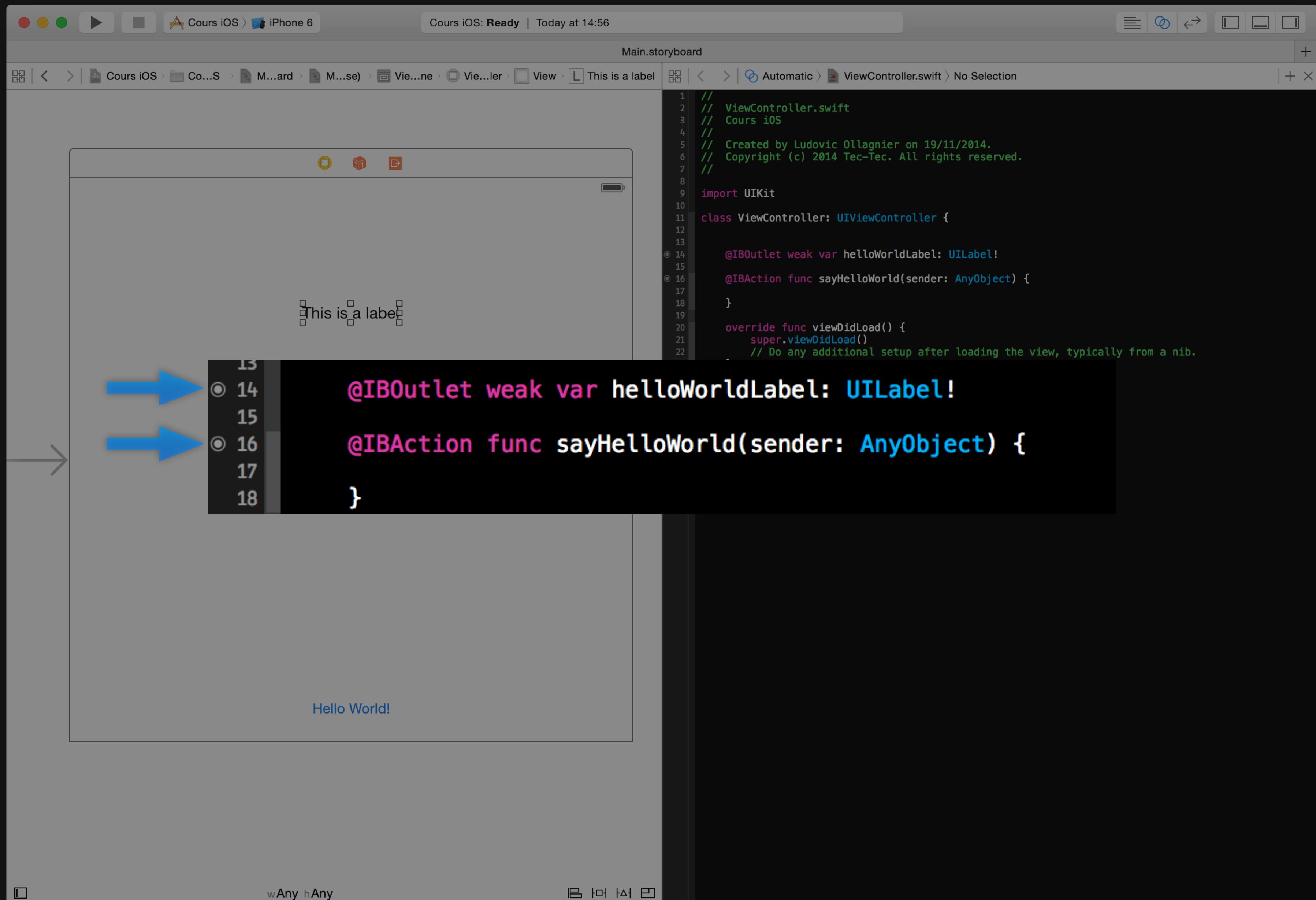
    @IBAction func sayHelloWorld(sender: AnyObject) {

    }

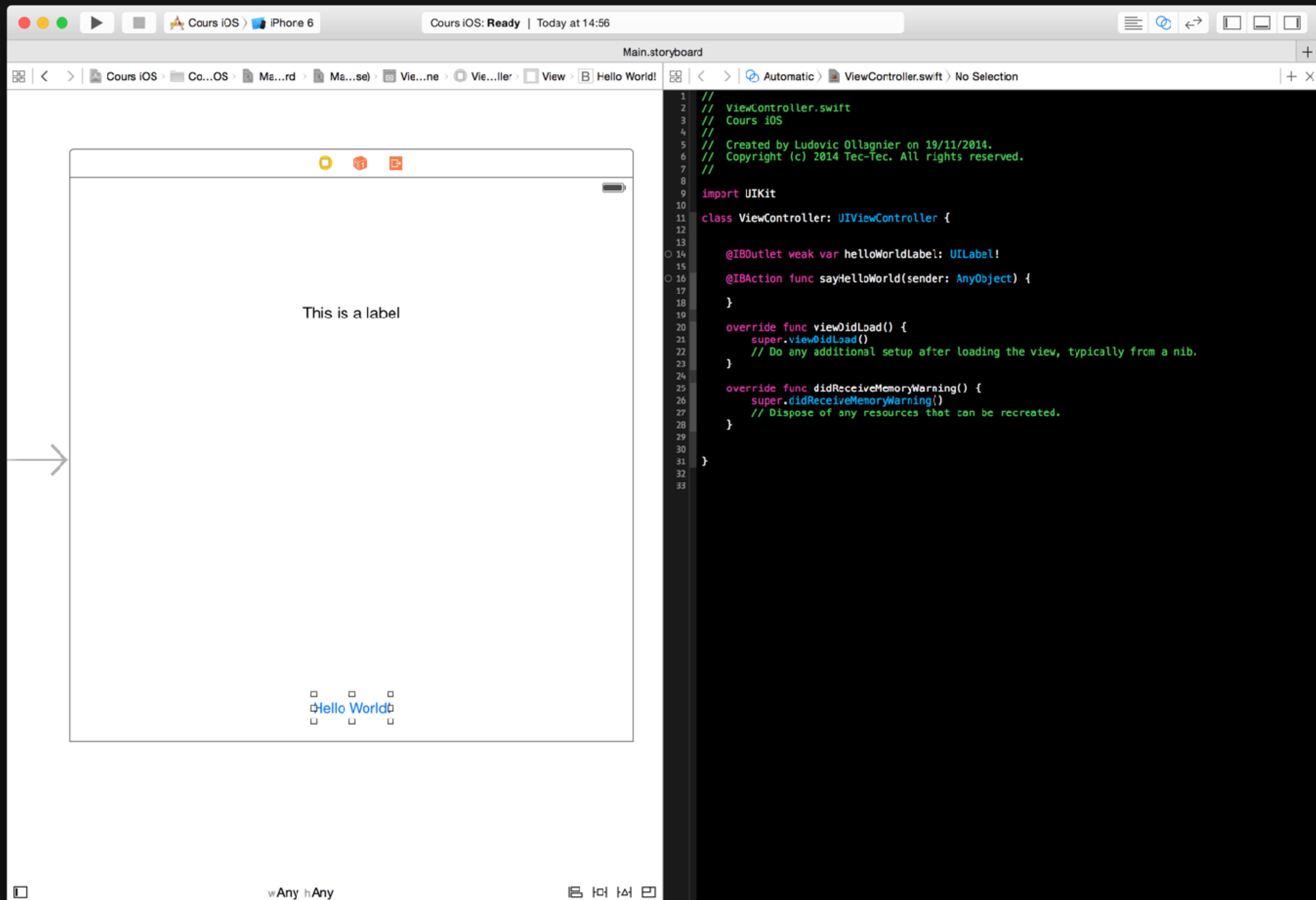
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }
}
```

A callout arrow points from the storyboard label to the `@IBOutlet` declaration in the code. The storyboard also displays the text "Hello World!" at the bottom.

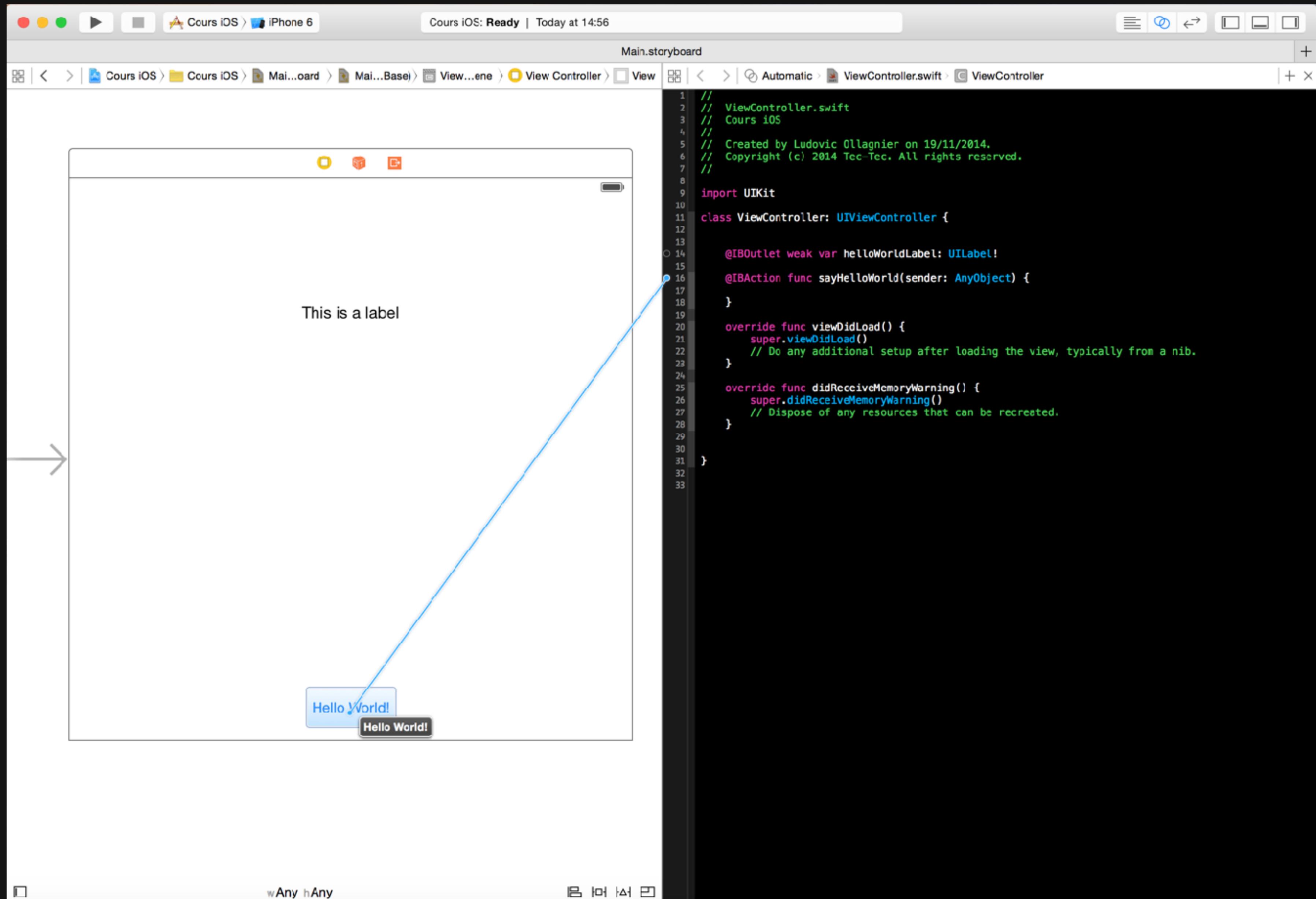
Du playground à l'application



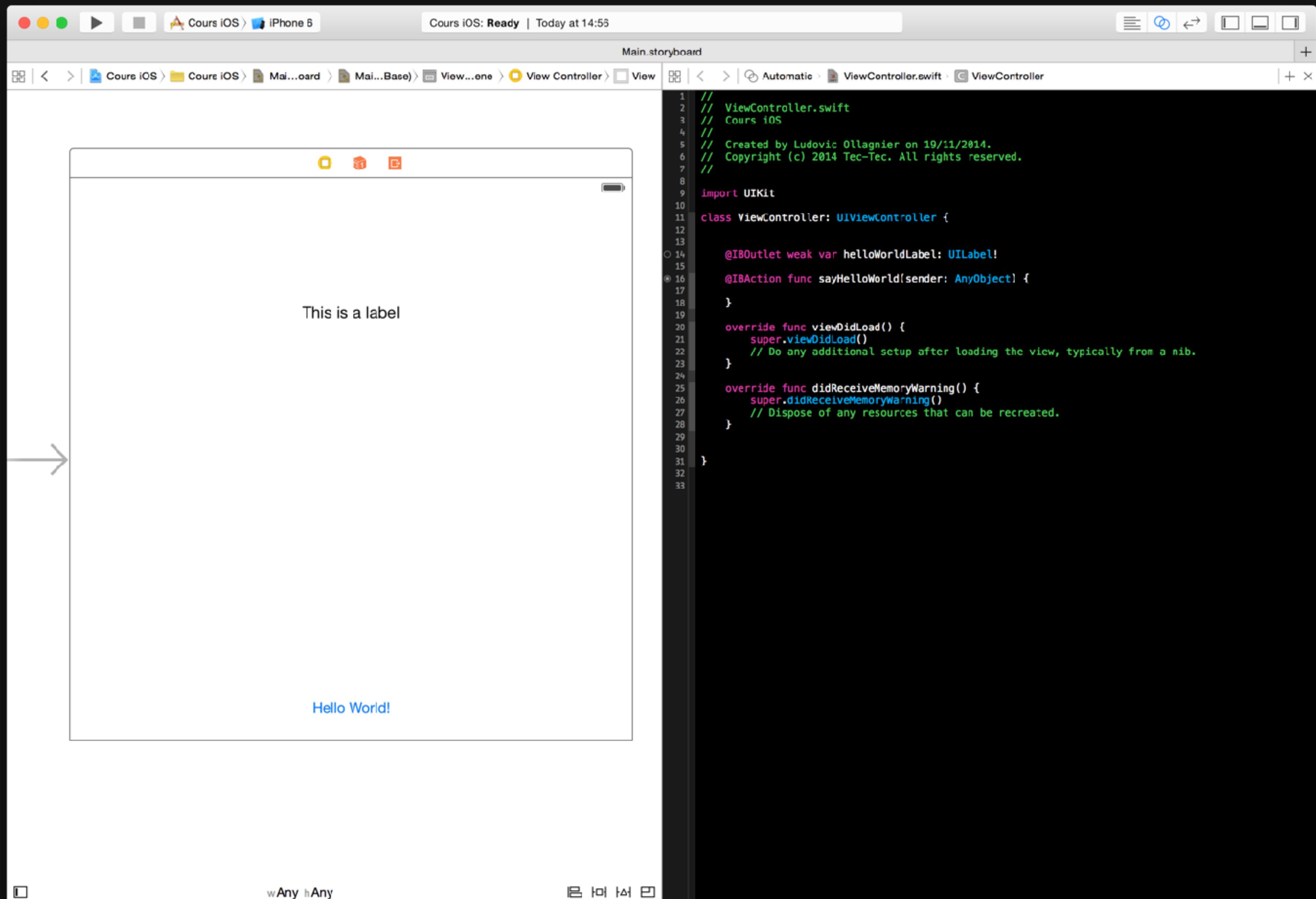
Du playground à l'application



Du playground à l'application



Du playground à l'application



Pour aller plus loin . . .



- The Swift Programming Language : Classes and structures
- The Swift Programming Language : Initialization
- The Swift Programming Language : Deinitialization
- The Swift Programming Language : ARC