

Les bonnes pratiques  
Let's do things the right way...

# Les bonnes pratiques

- ✦ Syntaxe
- ✦ Extensions
- ✦ Protocoles
- ✦ Design pattern
  - ✦ MVC
  - ✦ Objets assistants
  - ✦ Singleton

# Syntaxe

- ✦ Les noms de classes, structures (les types) prennent une majuscule
  - ✦ MaClasse
- ✦ Les noms d'instances prennent une minuscule
  - ✦ monInstance
- ✦ La bonne écriture pour préciser un type est `var maVar: String`
- ✦ Respecter ces conventions, c'est avoir un code plus lisible !

# Extensions

- ✦ Sous classer n'est pas toujours nécessaire
- ✦ Possibilité de modifier, d'étendre une classe existante directement
- ✦ Porte sur tous les objets de ce type de notre application

- ✦ Ajout de possibilité sur des objets récupérés
- ✦ Sert également à organiser son code

# Exemple

```
extension UIColor {  
    static var rouse: UIColor {  
        return UIColor(red: 1.0, green: 0.078, blue: 0.576, alpha: 1)  
    }  
}
```



- ✦ Fonctionne pour les méthodes de type et d'instances
- ✦ Ne peut pas surcharger une méthode implémentée au même niveau
- ✦ Ne peut pas rajouter de propriétés stockées

# Protocoles

# Généralités

- ✦ Un protocole définit le cadre à *respecter* en terme de méthodes ou propriétés pour réaliser une tâche spécifique.
- ✦ Le protocole peut ensuite être *adopté* par des classes, des structures ou des énumérations pour fournir la fonctionnalité.
- ✦ On dit d'un type qui satisfait aux prérequis d'un protocole, qu'il s'y *conforme*.
- ✦ Le protocole ne fait que poser des déclarations. C'est lors de l'adoption que l'on se charge d'implémenter les méthodes ou propriétés.

# Définir un protocole

```
protocol AProtocol {  
    var aStringPropertyThatNeedsToBeSettable: String { get set }  
    var anIntPropertyThatShouldNotBeSettable: Int { get }  
  
    static var aTypeProperty: Float { get }  
  
    func aRequiredMethodThatTakesAnInt(anInt: Int)  
    static func aRequiredTypeMethodThatReturnsADouble() -> Double  
}
```

Si une méthode doit pouvoir modifier le type, il faut penser à préciser *mutating* pour que les types valeurs puissent l'implémenter

# Définir un protocole

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

# Se conformer à un protocole

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

```
struct Human: FullyNamed {  
    let firstName: String  
    let lastName: String  
}
```

! Type 'Human' does not conform to protocol 'FullyNamed'

# Se conformer à un protocole

```
protocol FullyNamed {  
    var fullName: String { get }  
}  
  
struct Human: FullyNamed {  
    let firstName: String  
    let lastName: String  
  
    var fullName: String {  
        return firstName + " " + lastName  
    }  
}
```

# Se conformer à un protocole

- ✦ La conformité à un protocole se déclare comme la déclaration de l'héritage
  - ✦ Si la classe hérite d'une super-classe, celle-ci est toujours déclarée en premier
- ✦ Depuis Swift 2.0, il est également possible de créer une extension d'un protocole pour lui fournir une implémentation par défaut.



# Design pattern

# Design Pattern

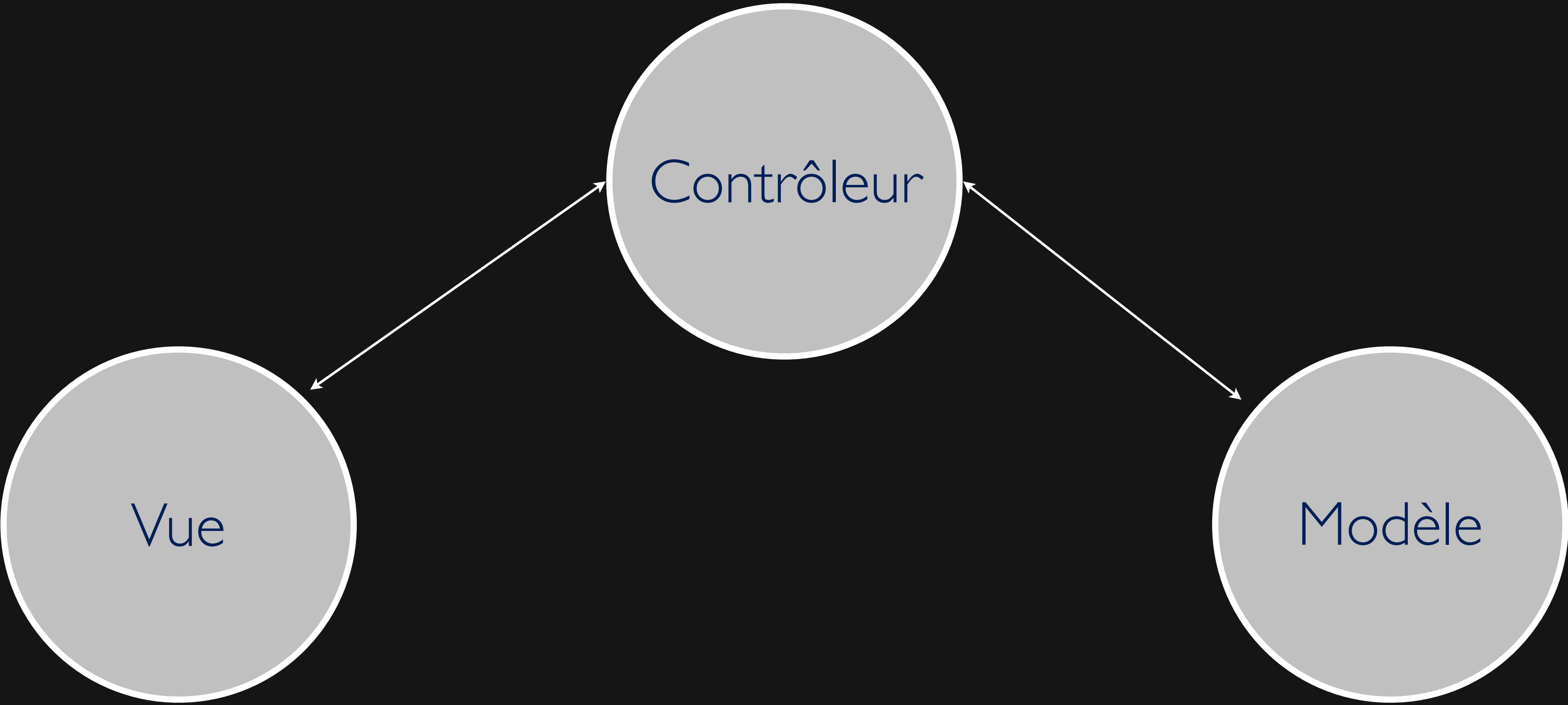
[...]Un patron de conception est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

# Design Pattern

Il décrit une solution standard, utilisable dans la conception de différents logiciels.

*Wikipedia*

MVC



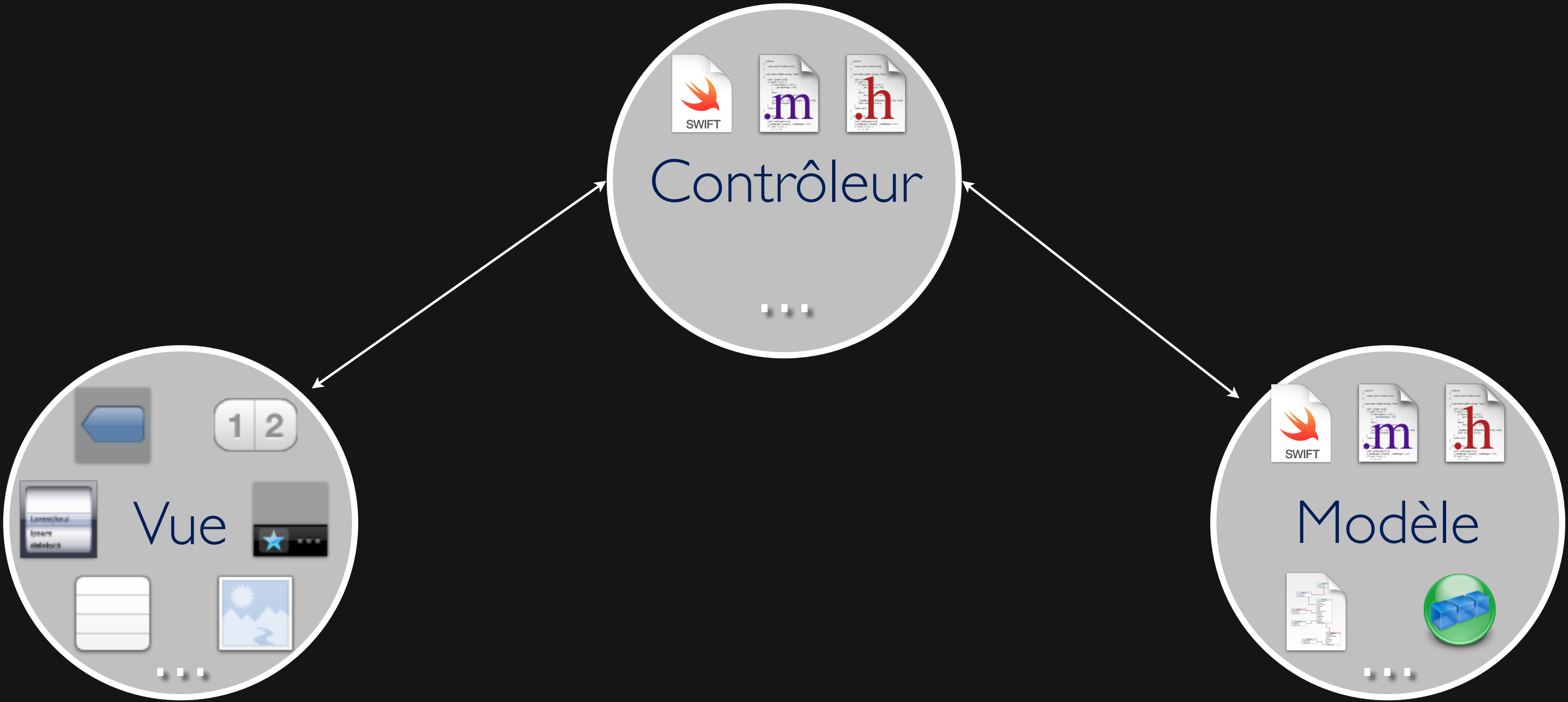
- ✦ Modèle
  - ✦ Gestion et manipulation des données
  - ✦ Contient la base de données éventuelle
  - ✦ Gère le cache
  - ✦ Définit la représentation des données
  - ✦ Réutilisable

- ✦ Vue
  - ✦ Élément en interaction avec l'utilisateur
  - ✦ Affiche les résultats à l'utilisateur
  - ✦ Récupère les actions de l'utilisateur
  - ✦ Réutilisable

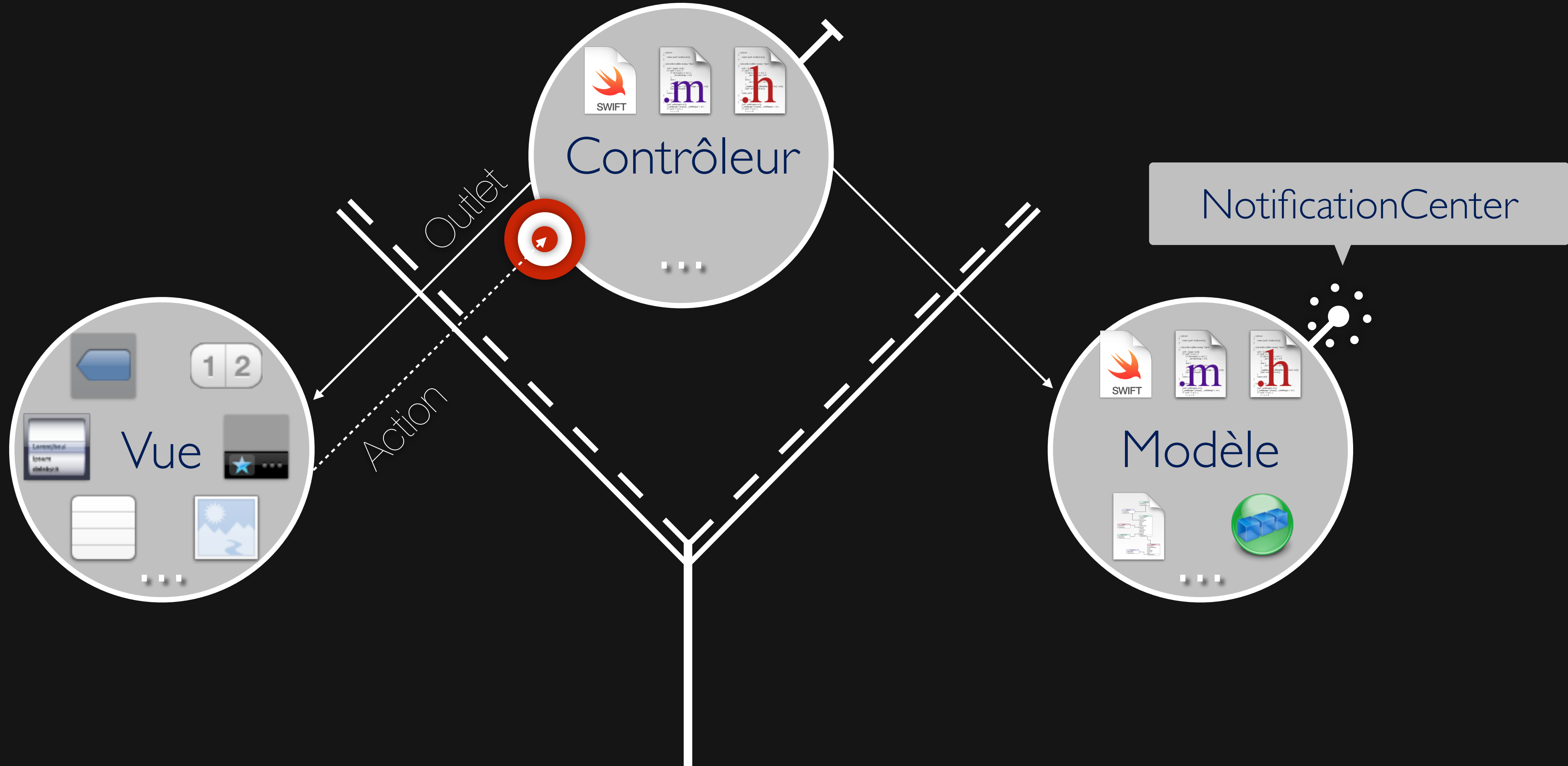
- ✦ Contrôleur
  - ✦ Fait le lien entre la vue et le modèle
  - ✦ Adapte les données à la vue
  - ✦ Interprète les actions sur la vue
  - ✦ «glue-code»
  - ✦ Rarement réutilisable



# En pratique



# En pratique



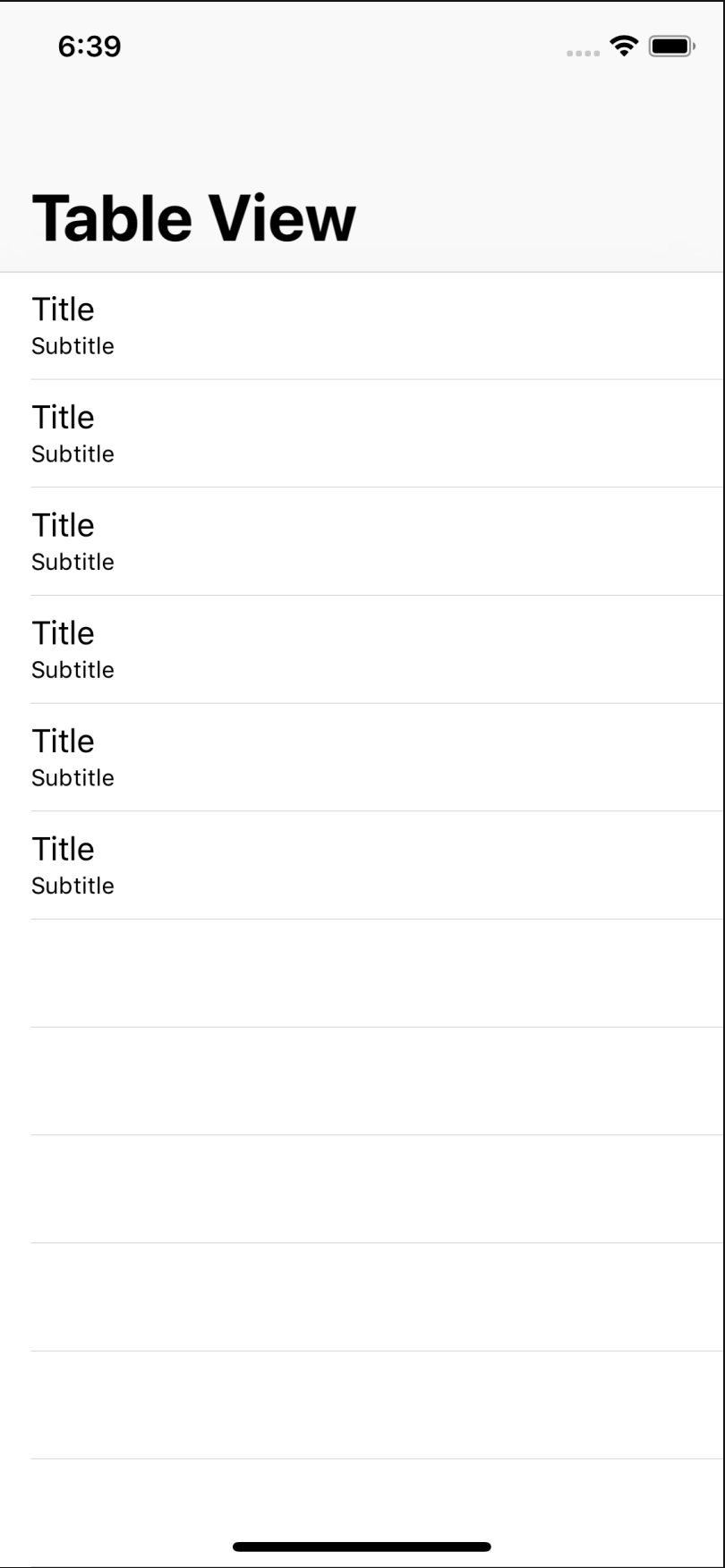
# Sur iOS

- ✦ UINavigationController
  - ✦ Une sous classe par écran ou fonctionnalité

# Objets assistants

- ✦ Déléguer certains fonctionnements à un objet externe
- ✦ Prévoir une modification du comportement sans sous classer
- ✦ Nécessite l'utilisation de protocole

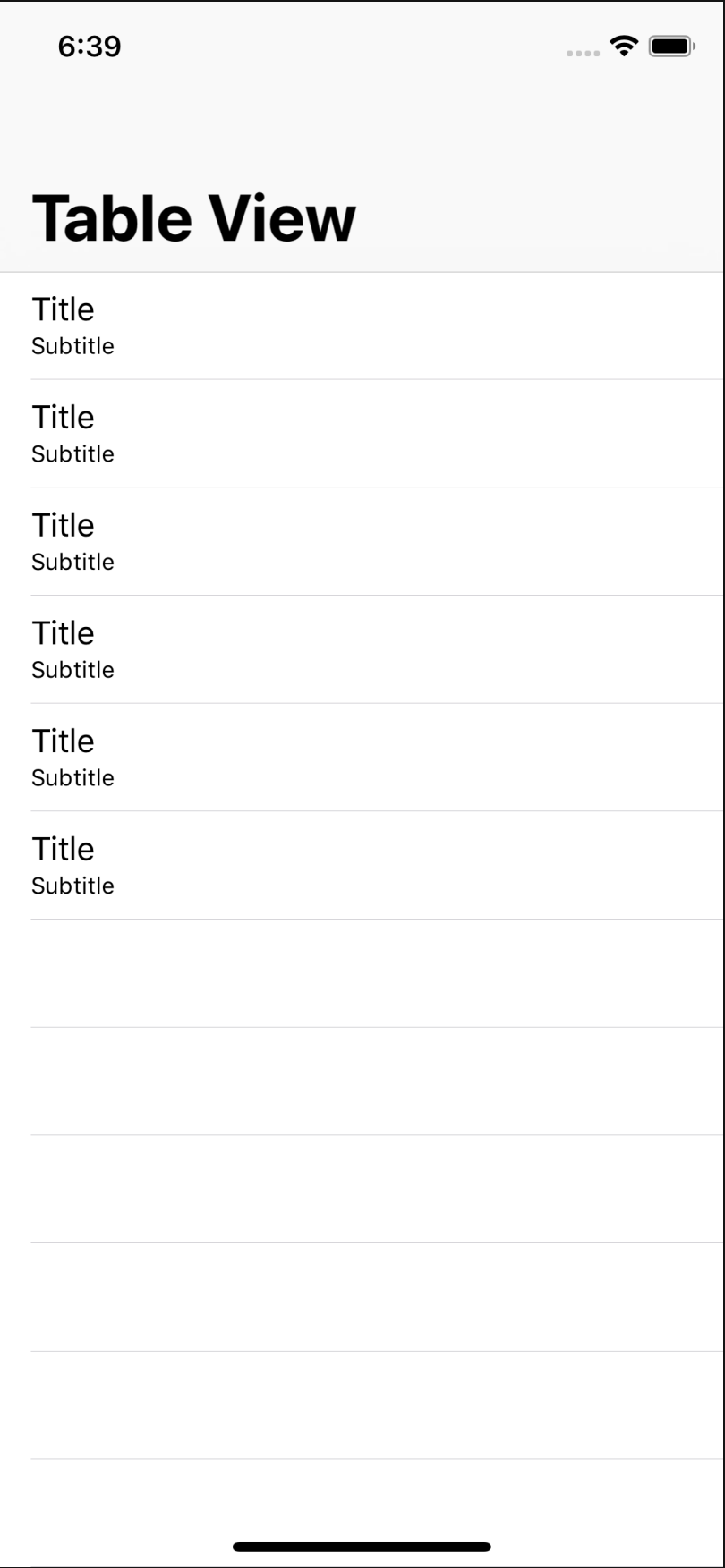
- ✦ 2 types principaux d'objets assistants
  - ✦ DataSource
    - ✦ Sert de source de données à afficher
  - ✦ Delegate
    - ✦ Effectue certaines actions pour le compte d'un autre objet



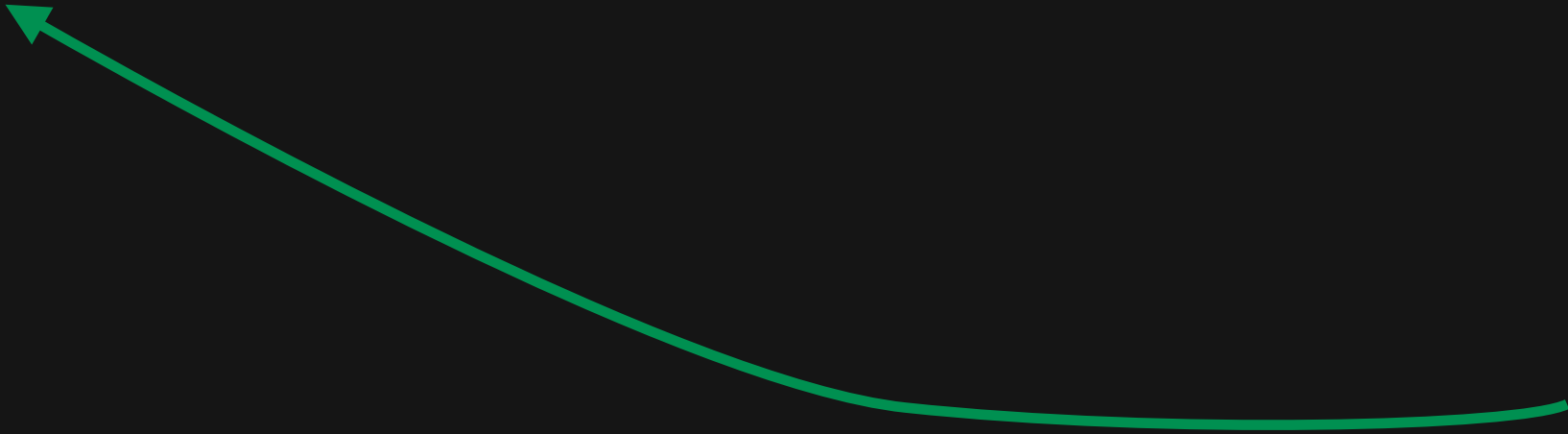
Combien de lignes ?



Exemple avec UITableView et son protocole  
UITableViewDataSource

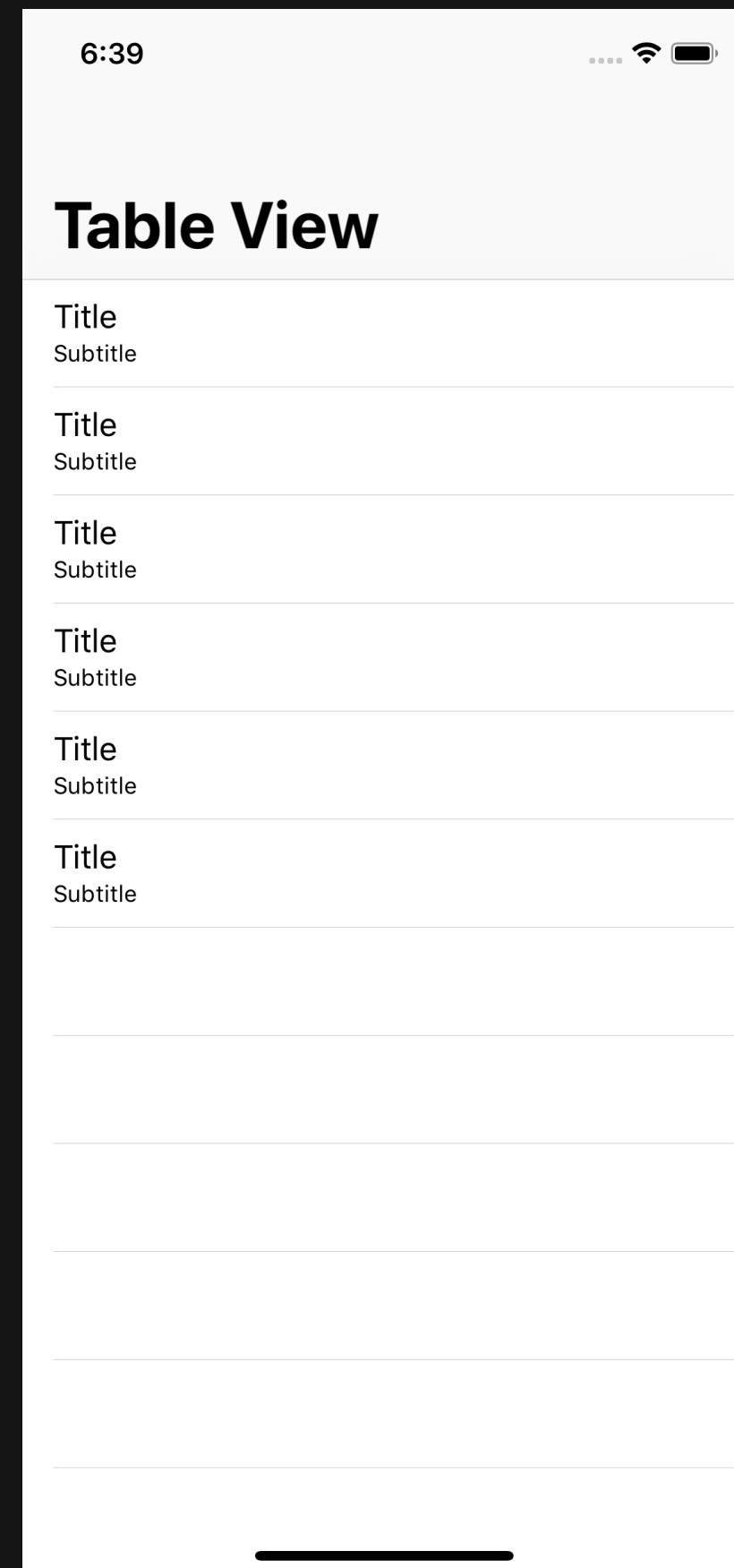


7 !



Exemple avec UITableView et son protocole  
UITableViewDataSource

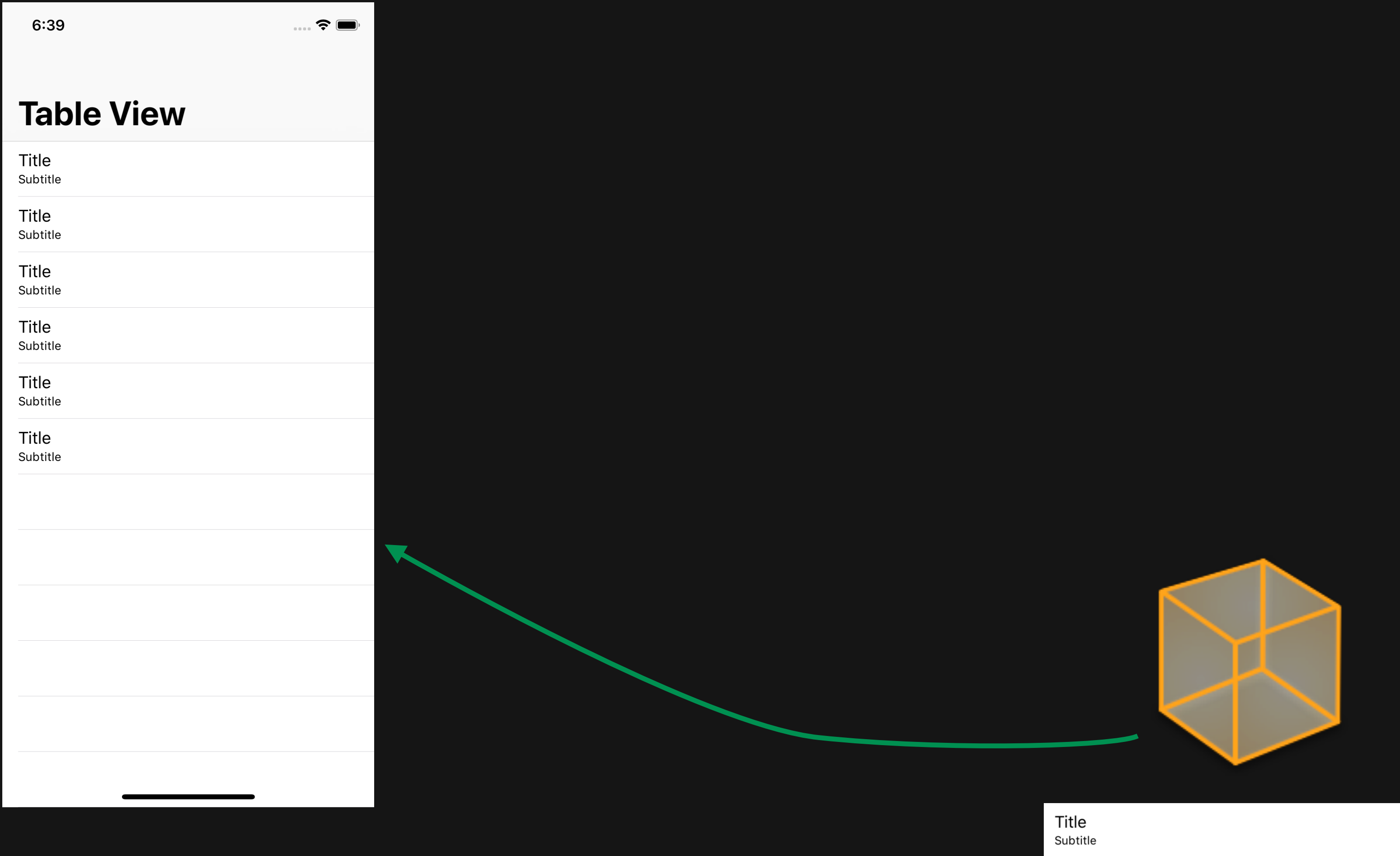




Quel contenu pour  
la ligne X ?



Exemple avec UITableView et son protocole  
UITableViewDataSource



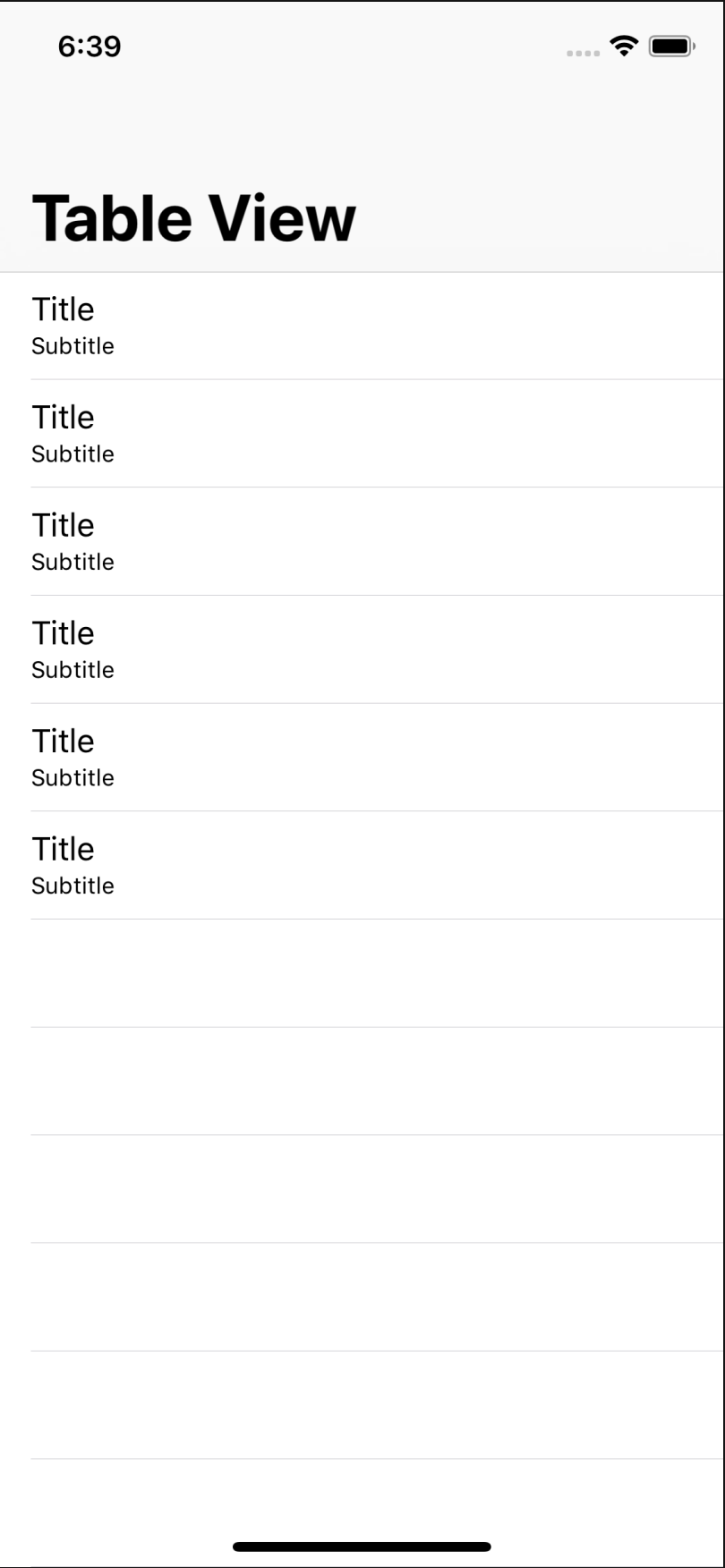
Exemple avec UITableView et son protocole  
UITableViewDataSource

# UITableViewDataSource

- ✦ 2 méthodes obligatoires à implémenter

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int
```

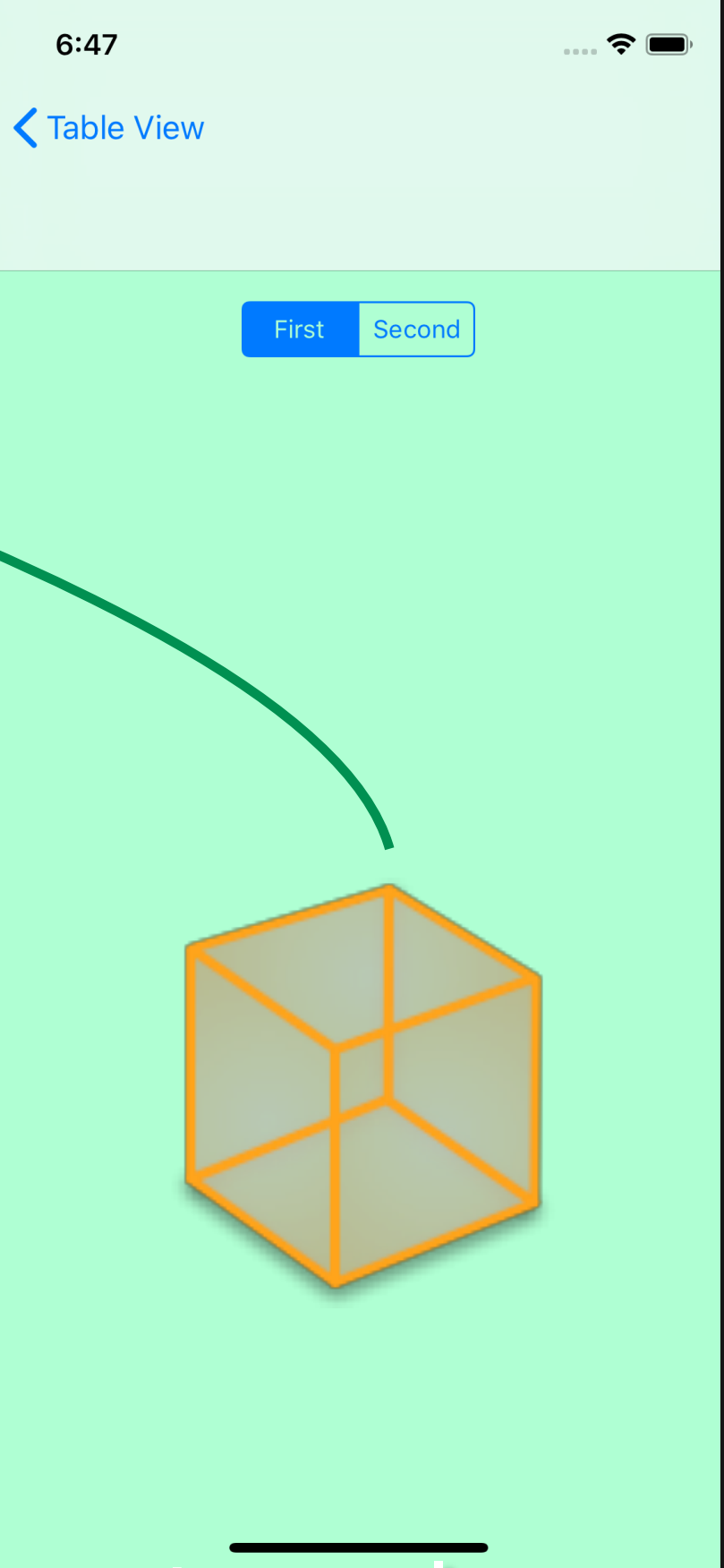
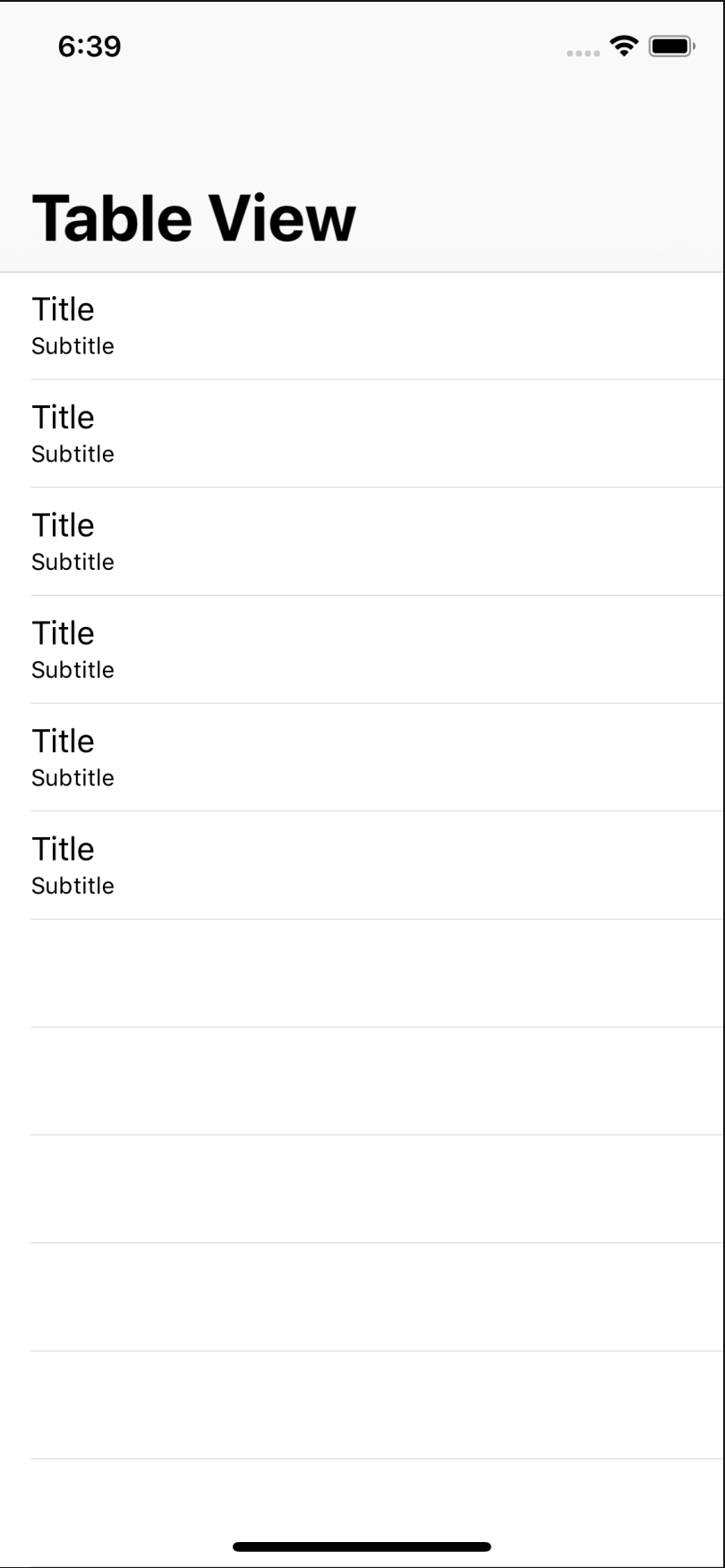
```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
```



On a appuyé sur la  
cellule X, je fais quoi ?



Exemple avec UITableView et son protocole  
UITableViewDelegate



Affiche cette vue.

Exemple avec UITableView et son protocole  
UITableViewDelegate

# Singleton

- ✦ Une instance unique partagée
  - ✦ Peut-être utile dans certains cas
  - ✦ Attention à ne pas en abuser !

## Singleton

```
class MyClass {  
    //:Cette propriété sera un singleton  
    static let instance = MyClass()  
  
    //:Continuer l'implémentation des méthodes et propriétés  
    normalement  
}
```