Dev mobile swift (cours 1)

Sommaire :

[Dev mobile swift (cours 1) 1](#_Toc133944497)

[Sommaire : 1](#_Toc133944498)

[Introduction 3](#_Toc133944499)

[M : pour Modèle 3](#_Toc133944500)

[V : pour Vue 3](#_Toc133944501)

[C : pour Contrôleur 3](#_Toc133944502)

[Créer un projet avec Xcode 4](#_Toc133944503)

[Fichier ViewControlller.swift 5](#_Toc133944504)

[Le fichier AppDelegate.swift 5](#_Toc133944505)

[L’application : 5](#_Toc133944506)

[Création d’une interface (Vue) 6](#_Toc133944507)

[Interaction entre la vue et le contrôleur 6](#_Toc133944508)

[Comment réaliser une prise ? 6](#_Toc133944509)

[Comment réaliser une action 7](#_Toc133944510)

[Ecriture du contrôleur 7](#_Toc133944511)

[Pour le CC : 7](#_Toc133944512)

[Création d’une table View pour afficher des données (cours 2) 8](#_Toc133944513)

[Le *table view* 8](#_Toc133944514)

[Configuration de l’objet UITableViewCell 9](#_Toc133944515)

[Écriture de UITableViewController 10](#_Toc133944516)

[Suite de l’écriture du contrôleur TCVMessage 10](#_Toc133944517)

[Dans le cas du style Subtile : 11](#_Toc133944518)

[Chapitre 3 La combinaison de Vues (cours 3) 11](#_Toc133944519)

[Les combinaisons de vues la Tab Bar Controller 12](#_Toc133944520)

[Les différents contrôleurs (ViewController) 12](#_Toc133944521)

[Les relations entre les vues 13](#_Toc133944522)

[Chapitre 4 La persistance des données (cours 4) 14](#_Toc133944523)

[Le format Json 14](#_Toc133944524)

[Le Bundler Container 15](#_Toc133944525)

[Le Data Container 16](#_Toc133944526)

[Accéder aux ressources dans le Data Container 16](#_Toc133944527)

[Accès a un document 17](#_Toc133944528)

[La lecture et l’écriture de données 17](#_Toc133944529)

[La modification du modèle pour JSON 18](#_Toc133944530)

[Lecture dans un fichier JSON (vue en L1) 18](#_Toc133944531)

[Écriture dans un fichier JSON 18](#_Toc133944532)

[Chapitre 5 Déballage des variables 19](#_Toc133944533)

[Les types facultatifs 19](#_Toc133944534)

[Exemple : 19](#_Toc133944535)

[Déballage par la force (nom non officiel) 20](#_Toc133944536)

[Déballage en douceur (nom non officiel) 20](#_Toc133944537)

[Raccourcis : 20](#_Toc133944538)

[Pour utiliser les variables raccourcies dans un if (non optionnel) de façon non locale (utilisable en dehors du if) 21](#_Toc133944539)

[Les tests Unitaires 21](#_Toc133944540)

[Dans le TDD (Test-Driven Development) 21](#_Toc133944541)

[L’écriture des test unitaires 21](#_Toc133944542)

[Le travail de développement 22](#_Toc133944543)

[Le cycle red/green/blue 22](#_Toc133944544)

[La structure des 3A 22](#_Toc133944545)

[Le fremework XCTest 23](#_Toc133944546)

[La création dun cas de tests 23](#_Toc133944547)

[Les assertions 24](#_Toc133944548)

[Type de fichiers a créer : 24](#_Toc133944549)

[Résumé : 24](#_Toc133944550)

[End Project (td5): 25](#_Toc133944551)

[Astuces très utiles (pas dans le cours de la prof) 25](#_Toc133944552)

[Pour faire une DocString 25](#_Toc133944553)

Introduction

L’architecture MVC appartient à la famille des Patrons de conception

Ils décrivent des solutions standards face a des problèmes de conception qui se sont déjà produits par le passé

Patrons de conception = capitalisation d’expérience

Exemple :

En php on avait vu fetch() comme patron de conception

L’architecture MVC sépare la logique du code de l’application en 3 parties :

M : pour Modèle

Objectif : encapsule la logique métier et extrait les données brutes provenant de la BD, de fichiers textes.

V : pour Vue

Objectif : afficher et récupérer les données

C’est la couche ou l’on trouve les composants ergonomiques

C : pour Contrôleur

Objectif : gérer toute la logique applicative. Il pilote la vue, en réalité aux actions de l’utilisateur, demande des données au modèle. C’est le chef d’orchestre de l’application.

Model view controller

User 🡪(actions) views 🡪 control 🡪 data

🡨(data) 🡪

On utilise l’utilitaire (IDE)

Application Xcode

Créer un projet avec Xcode

Lancer l’application Xcode puis choisir l’option « create new pXcode project »

Dans le choix du template 🡪 IOS 🡪 (volet droit APP) puis cliquer sur next.

Sélectionner l’option (du nouveau projet) interface : storyboard

Language : Swift

Décocher le reste en dessous

Ensuite ont choisi le répertoire du projet

Les fichiers :

**AppDelegate.swift** : fichier source qui mémorise les méthodes appelées à différents moments du cycle de vie des applications. Il gère le cycle de vie des applications.

**ViewController.swift** : fichier source permettant de contrôler la vue par défaut de l’application. A ce stade du projet, ce fichier est le contrôleur de l’application.

**Main.storyboard** : le fichier d’interface pouvant contenir plusieurs écrans liés entre eux. Il décrit les différentes vues de l’application.

Assets.Xassets : répertoire destiné à stocker des éléments graphiques tels que des images, icones etc…

LancerScree.storyboard : autre fichier d’interface qui permet de dessiner l’écran a afficher, au démarrage de l’application

Info.plist : fichier CML contenant les finformations …

Fichier ViewControlller.swift

* Code du fichier ViewControlleur.swift

On créé une classe en swift avec un héritage **UIViewController**

Avec override func viewDidLoad(){

Super.viewDidLoad()

}

Chaque vue est associé son controlleur

Rappel : override = redéfinir

Le fichier AppDelegate.swift

Cette classe gère le cycle de vie de l’application. Elle utilise le protocole UIApplicationDelegate lui permettre si besoin d’implémenter les méthodes quand :

L’application :

* + vient de s’ouvrir
  + Est passé en arrière-plan
  + Sur le point de se fermer
  + N’est plus active

**C’est dans ce fichier que l’on va définir les variables globales**

(Visible dans tous les autres contrôleurs)

* La classe AppDelegate permet d’avoir accès à des variables globales (accessibles partout dans le code) grâce à la classe UIApplication.
* UIApplication est la classe qui représente notre application et elle utilise le patron de conception singleton. Il est donc toujours possible d’en récupérer l’instance grâce à la propriété **UIApplication.shared**
* Pour accéder a notre AppDelegate dans le code de l’application, il faudra écrire : **UIApplication.shared.delegate**

Création d’une interface (Vue)

* L’interface Builder est un éditeur permettant de créé et modifier des vues en WYSIWYG (**w**hat **y**ou **s**ee **i**s **w**hat **y**ou **g**et)
* Il est accessible grâce a la sélection du fichier Main.storyboard
* A partir de la bibliothèque d’objets fournie par le framework, il est possible pas un cliquer/glicer d’ajouter des éléments graphiques sur la vue.

Il faudra à la fois modifier l’interface que l’on nous donne pour changer les attributs des boutons (dans l’inspecteur d’attributs) et il y a une partie a coder.

Interaction entre la vue et le contrôleur

* Les interactions entre la vue et le contrôleur se font dans le framework grâce à la notion de prises et d’actions :
  + Les prises également appelées **Outlets** sont des liens permettant de référencer un élément graphique de la vue
  + Mes actions permettent de lancer des méthodes du contrôleur à la suite d’action de l’utilisateur.

Comment réaliser une prise ?

1. Afficher le code du contrôleur grâce à l’éditeur d’assistant

Le menu est une icône ressemblant à aligner à gauche de Word (pour le texte)

1. Création d’une prise :
   1. Par un **ctrl clic** (ou à partir du menu contextuel), déplacer la souris vers le code du contrôleur. Un trait bleu symbolise la création de la prise. (ou avec le raccourci clique droit + glicer)
   2. Une annotation **@IBOutlet** avec le nom de la prise est alors automatiquement créée

Rappel : nom\_var : typed\_donné

Est une variable pointant sur un type de type : typed\_donné

Pour vérifier si on a aucune erreur : on fait cliquer droit sur la partie édition d’interface.

On est obligé de supprimer une prise et la recréer pour pouvoir avoir un nom différent on ne peut pas juste modifier le code !!!

Comment réaliser une action

* La création d’une action se déroule de la même manière

1. Afficher le code du contrôleur grâce a l’éditeur d’assistant :
2. Création d’une action
   1. Par ctrl + clic (ou a partir du menu contextuel), déplacer la souris

Vers le code du contrôleur. Un trait bleu…

Une annotation @quelquechose

Pour voir toutes les prises : aller voir l’inspecteur de connexion (l’icône est une sphère à côté de l’équerre en haut à droite de la fenêtre)

Ecriture du contrôleur

Le point d’entrée de l’application est spécifié dans MainStory par une flèche.

C’est le contrôleur associé à cette vue qui sera exécuté au lancement de l’application et plus précisément sa méthode viewDidLoad()

Pour le CC :

* Sur papier (50% note finale)
* + Un projet à réaliser (50%)

Création d’une table View pour afficher des données (cours 2)

(Cours 2 de dev mobile)

Le *table view*

* **C’est un composent qui permet d’afficher une liste d’élément, organisées en section sans limite de taille.**
* Une infinité de cellules ne sera pas créée, uniquement celles qui seront vues à l’écran, les autres seront réutilisées. Quand on fait défiler l’écran la première cellule disparait et va être.

Dans la table view, Il y a :

* Une section
  + Des cellules
  + Des lignes
  + Etc…
* Une autre section
  + Des cellules
  + Des lignes
  + Etc..
* La table view ne gère toute la présentation de leurs données **pas les données elles-mêmes.**

- En fait la vue délègue toutes les prises de décision au contrôleur d’où le nom

de mécanisme de délégation.

- le contrôleur devient ainsi « délégué » celui à qui on délègue la responsabilité. Quand la vue aura besoin de quelque chose cela sera c

Au contrôleur de répondre à son besoin

* La classe UITableView qui se situent dans la couche Vue qui va afficher les données
* Et c’est la classe UITableViewController qui pilotera la vue et qui configurera la liste complète.
* Le contrôleur issu de la classe UITableViewController, utilisera deux protocoles pour fonctionner :
  + UITableViewDataSource pour fournir les données à la table view
  + UITableViewController
* Un protocole en swift se rapproche de la notion d’interface dans les autres langages de programmation.
* Une interface peut être désignée comme un contrat : toute classe qui souhaite utiliser un protocole s’engage à implémenter les méthodes dont les signatures ont étés définies dans le protocole.

Exemple :

UITableViewDataSource fournit à la table View impose au contrôleur de coder 2 méthodes :

* L’une qui retourne le nombre de sections :

Override func numberOfSection(etc…){

Return 1 // pour 1 section

}

* L’autre qui retourne le nombre de lignes :

Override func tabView(etc…){

Return 0 // pour le nombre de lignes

}

* Une autre classe est généralement nécessaire pour afficher une liste d’éléments :
  + La classe UITableViewCell qui va permettre de configurer un élément de la table UITableView

C’est la seule fois ou le UITableViewController le fait automatiquement

Configuration de l’objet UITableViewCell

* **La solution la plus simple :**

Sélectionner la cellule :

Choisir dans l’inspection d’attributs :

subtile (automatiquement le titre et le sous-titre)

Il faut absolument donner un identifiant a la cellule.

* **Solution moins simple : *(CUSTOM)***

On choisit custom ou on ne verra pas les emplacements des titres et des sous titres

* Il faut ensuite déposer les labels
* Et déposer les prises

Écriture de UITableViewController

3 méthodes au minimum sont à coder pour

afficher les données dans la **table view**

🡪 Au minimum :

**override** **func** numberOfSections(in tableView: UITableView) -> Int {

// il peut y avoir plusieurs sections (doit être du type Int comme écrit !)

**Return** nombreDeSectionAEcrireIci

}

**override** **func** tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

**return** nombreDeLignesParSectionsAEcrireIci

}

**override** **func** tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

**let** cell = tableView.dequeueReusableCell(withIdentifier: "affichageDesElevesParClasses", for: indexPath)

**let** sections = UneListeDElements // pour chaque section(s)

**let** lignes = UneListeDElements // pour chaque lignes

// Configure the cell...

// chaque element doit être une chaine a chaque lignes

cell.textLabel?.text = sections[indexPath.section]

cell.detailTextLabel?.text = lignes[indexPath.row].get()

// on renvoie la cellule modifié

**return** cell

}

Attention avec l’identifier de UITableViewController

L’identifier présent dans la cellule « cell » doit avoir le même identifiant que celui présent dans la cellule d’exemple (que l’on peut voir sur la vue « Controller ») il faut aller dans la vue sur la cellule d’exemple et ensuite dans les paramètres (droite) c’est identifier (avec un placeholder **reuseIdentifier**)

indexPath est une structure : contenant nb de lignes et nb de colonnes pour identifier les coordonnées d’une cellule la table view

Suite de l’écriture du contrôleur TCVMessage

* Implémentation de la méthode qui permet de donner corps à une cellule pour un index donné

Override func tabView(){

…

Var lemessage : Message

Let unecellule : …tableView.dequereusableCell(etc..) as ! MessageCell

}

Dans le cas du style Subtile :

cell.testLabel ?.text

Cell.detailTextLabel?.text

* C’est la méthode “dequeuReusableCell” qui instancie une cellule pour un index donné. Ne pas oublier le cast
* Pour des soucis de performance, la méthode

**POO dev mobile :**

Title

Machin et section (c’est le header de la page)

Faire un TableViewControler customisé :

* + on créé un projet
  + on supprime ViewControler
  + On créer un nouveau fichier avec l’onglet fichier en haut a gauche
  + On créer un template Cocoa Touch Class
  + On créer tout ça dans le bon répertoire qui est demandé

Chapitre 3 La combinaison de Vues (cours 3)

Pour l’Accès au server mac-mini voir ordi Windows.

* Si certaines applications ne comportent qu’une seule vue, la taille d’un iPhone impose généralement la navigation d’une vue à l’autre.
* Des liens de parenté seront créés entre les contrôleurs afin de ne pas donner à un seul de trop grandes responsabilités
* Les contrôleurs qui permettent la navigation d’une vue à l’autre sont des ViewController (tab bar ou Controller)
* Une application Iphone peut combiner à la fois des tab Bar controller et des navigation bar Controller
* En général les Tab Bar Controller sont utilisées pour accéder aux différentes fonctionnalités de l’application grâce à sa barre d’onglet
* Les Navigation Bar Controller permettent de naviguer entre les vues en apportant plus de détails entre chaque navigation.

Les combinaisons de vues la Tab Bar Controller

* La Tab Bar Controller présente à l’utilisateur une barre d’onglet visible à chaque instant.
* Pour créer une Tab Bar Controller, il est nécessaire de déposer un objet Tab Bar Controller sur le fichier main.storyboard, ce qui à pour effet de créer automatiquement 3 scènes.
* La scène « à gauche » qui contiendra entièrement l’onglet.

Les différents contrôleurs (ViewController)

Les liens qui existent entre les différents contrôleurs s’appellent les segues (/seugways/)

* C’est un view controller qui a la capacité de contenir et d’organiser d’autres view controller qui seront organisés sous forme d’une pile
* Le premier élément de la pile est l’élément racine, au fur et a mesure de l navigation, les View Controllers sont empilés les uns sur les autres et la navigation a lieu
* Le view controlller qui est visible est celui qui est au sommet de la pile, soit le dernier ajouté dans la pile.
* Tapoter sur le bouton retour effectue un « dépilage » le dernier view controller est ôté de l’animation de retour.
* Les méthodes d’empilage et de dépilage seront appelés par ces deux méthodes.

Les relations entre les vues

* Il existe différents types de segue :
  + Les segues de relation
  + Les segues de navigation
  + Les segues de débobinage (non étudiés pour le moment)
* Dans les Tab Bar Controller et les navigation Controller, le segue de relation permettent de définir un lien parent enfant entre deux Controller.
* La relation se fait grâce a ctrl + clic + glicer d’une vue à l’autre.
* A l’ouverture du menu, choisir l’option Relationship segues – view Controller.
* Un segue de navigation relie deux contrôleurs et permet la navigation entre l’un et l’autre
* Il permet également de transmettre des données via le code
* 2 possibilités pour créer un segue de navigation :
  + Via un composant ergonomique (comme un bouton) associé a une action (tapsurbtncliquer)
  + De contrôleur à contrôleur en nommant de manière précise le nom du contrôleur de navigation

Pour pouvoir transmettre des informations avec une segue il faut la préparer et ensuite la traiter.

1. Pour préparer la segue
   1. On appelle la méthode prépare

Chapitre 4 La persistance des données (cours 4)

MVC = model view controller (rappel)

Nouvelles verrons les structures de données JSON

* Rappel : Dans une architecture MVC, de modèle se charge de générer la persistance des données.
* Le format JSON pour JavaScript Object Notation s’est imposé comme un standard pour communiquer avec un service web.
* Il permet d’échanger des données sous forme d’un fichier texte
* Il est également utilisé pour stocker des données dans la partie Modèle de l’application

Le format Json

* Les objets ou dictionnaires
* Les tableaux
* Les valeurs
* Un objet est une semble de paires clé/valeurs séparées par des virgules entourées par des accolades :

{« clé » : valeur, « clef2 » :valeur}

Les clés sont toujours des chaines de caractères



* Chaque application est rangée dans son propre bac à sable.
* C’est le seul espace ou l’application peut lire et écrire des informations.
* Le concept du **bac à sable** garantit la sécurité de l’utilisateur : une application ne peut pas accéder ou falsifier les données d’une autre application
* Il est composé de 2 parties principales : le Bundle Container et le Data Container.

Le Bundler Container

* Le **Bundler Container** comporte le code compilé et les ressources de l’application comme les images, les fichier Storyboards et d’autres fichiers tels que les fichiers de propriétés.
* Il n’est ouvert qu’en lecture

Le Data Container

* Il comporte plusieurs répertoires accessibles en lecture et écriture
  + Le répertoire **Documents** permet de stocker et de lire des informations crées par l’utilisateur
  + Le répertoire **Library** contient plusieurs répertoires standards comme informations qui persistent d’un lancement sur l’autre et qui ne doivent pas être exposées à l’utilisateur
  + Le répertoire **tmp** destiné à recevoir des données temporaires non persistants d’un lancement a un autre

Schéma :

* Bundler Container
  + Lecture
* Data Container
  + Document
  + Library
  + Tmp

Lecture/Ecriture

Problème : On ne connait pas le chemin d’accès de ces différents répertoires

Accéder aux ressources dans le Data Container

* On utilise la classe FileManager qui permet de localiser, créer et modifier un fichier.
* La classe FileManager instancie un singleton auquel on accède grâce à la propriété default de la classe

let leFileManager = FileManager.default

* Afin de se situer dans les répertoires-clé du Data Container, on utilise la méthode urls(for :,in :) qui renvoie un tableau d’url et qui prend 2 paramètres
  + Directory : qui correspond au répertoire que l’on recherche dans le data container : .documentDirectory, .cachesDirectory, .libraryDirectory, .applicationSupportDirectory
  + domainMask : sur IPhone, une seule valeur existe : .userDomain.Mask
* Par exemple pour obtenir le répertoire des documents de l’utilisateur

Accès a un document

* L’url est l’élément qui permet de cibler un fichier ou un répertoire par son chemin qu’il se trouve dans le Bundler ou le Data Container
* Pour cibler un document en particulier, il faut d’abord reconstruire son url à partir de celle de son répertoire. Pour cela on utilise la méthode appendingPathComponent

// (sigleton)

let leFileManager = FileManager.default

let urls = leFileManager.urls(for: .documentDirectory, in: .userDomainMask)

let urlDocument = urls.first!.appendingPathComponent(“nom\_fichier.json”)

* On peut ensuite vérifier que le fichier existe avec la méthode fileExists(atPath :)

La lecture et l’écriture de données

* Une fois qu’une URL vers un fichier a été obtenue, on utilise l’objet Data pour lire ou écrire des données dans ce fichier
* Pour lire un fichier, un constructeur de Data prend une url en paramètre :

Let mesDonnées : Data = try Data(contentsOF: urlDocument)

* Pour créer ou remplacer un fichier avec le contenu d’un objet Data, on utilise FileManager

leFileManager.createFile(atPath : urlDocument.path, contents: mesDonnées, attributes: nil)

La modification du modèle pour JSON

* Ajouter le protocole Codable à la définition de la classe « Modèle »
* Ajouter deux méthodes statiques pour manipuler le modèle :
  + Une méthode pour y écrire
  + Une méthode pour y lire

Lecture dans un fichier JSON (vue en L1)

* Lecture du fichier JSON

1. Atteindre le fichier via son url : une fois qu’une URL vers un fichier a été obtenue, on utilisera l’objet Data pour lire dans ce fichier
2. Création d’objet **JSONDecoder** qui va nous permettre de « décoder » le fichier JSON et créer une **collection d’objets facilement manipulable**.

Écriture dans un fichier JSON

1. Encoder au format JSON, tous les objets à sauvegarder grâce à un objet de la classe JSONEncoder
2. Récupération du chemin du fichier
3. Tentative d’écriture dans le fichier

* Dans la classe Message

// (donné)

class Message : Codable{

public static func ecritureJSON(tousLEsMEssages: [Message]){

…

}

}

Notes pour le td 5 :

Fichiers –

* Les mots
* Appdelegate
* Mot
* Scenedelegate

Tab bar controller plusieurs éléments pour nav dans les pages (segue)

Liste des mots déjà saisi par ordre alphabétique (dans une segue avec Controller sous forme de liste)

Ordre pour faire le projet :

* Faire la classe mot
* Puis les mots…
* Puis ajouter mot
* Puis tab bar contrôler
* Puis ajouter mot (fonctionnalité) et
* Ensuite tab bar contrôler
* Ensuite la recherche

Chapitre 5 Déballage des variables

Les types facultatifs

* Dans la déclaration des variables, lorsqu’un type est suivi d’un point d’interrogation, cela signifie que la valeur contient une valeur facultative
* La variable peut donc contenir soit une valeur, soit NIL qui correspond à l’absence de valeur

Exemple :

Var laChaineOptionelle : String ?

Var nb : Int?

Nb et laChaineOptionelle sont optionels !!

* Un type optionnel est donc un « paquet cadeau », on ne sait pas à l’avance ce que contient la variable.

Déballage par la force (nom non officiel)

* Une variable de type « chaine facultative » (« string ? ») ne peut pas être utiliséesimplement comme une chaine de caractère classique, il faut avant la déballer
* La façon la plus simple de déballer une variable possédant une type optionnel est d’jouter un poit d’exclamation ( !) après le nom de la variable.

Déballage en douceur (nom non officiel)

* Pcq Swift impose de vérifier le contenu de la variable avant de la déballer. Voici le code à appliquer :

Var lachaineoptionelle : String? = « LesL1 NEC au travail ! »

If lachaineoptionelle = ! nil {

Let lachainedébalée : String = lachaineoptionelle!

Print(« message\(chainedébalée ) »)

}

Raccourcis :

C’est tellement utilisé qu’il existe un raccourci :

If let chainedéballée : String = laChaineOptiuonelle ! {

On fait si on peut la débaler ….

ATTENTION : la variable N’est visible que dans le if

}

Pour utiliser les variables raccourcies dans un if (non optionnel) de façon non locale (utilisable en dehors du if)

**guard** Permet d’utiliser une variable définie dans une condition en dehors de la condition elle-même :

Il faut utiliser « **guard** » de cette façon :

// point d’exclamation dans le if **n’est plus nécessaire**

**guard** **let** chainedébalée = lachaineOptionelle **else** {

**return**

**// le programme s’arrète**

}

Les tests Unitaires

Dans le TDD (Test-Driven Development)

Les tests unitaires jouent plusieurs rôles :

* Ils rythment le travail du développeur
* Ils guident la conception du système
* Ils servent de documentation au code produit

L’écriture des test unitaires

1. Écriture des tests unitaires avant le code permet au développeur de progresser à la façon d’un alpiniste.
   * On grimpe de quelques pas
   * On écrit donc un test
   * On pose un piton pour se prémunir contre la chute
   * ….

* L’écriture de test conduit à décrire de façon formelle ce qui se passe….

Le travail de développement

Le principe du travail de développement est simple, il consiste à suivre le cycle red/green/refactor :

1. On ajoute un test au système
2. On le teste, le test échoue
3. On écrit la fonctionnalité correspondante le plus simple possible qui fasse passer le test
4. On vérifie que le test passe
5. On simplifie avec les techniques de remaniement (refactor)

Le cycle red/green/blue

* Ce cycle porte l’appellation “red/green/blue » car les développeurs qui utilisent des outils de test logiciel, reçoivent en réponse au lancement de leurs tests une réponse claire : soit rouge (le test ne passe pas), soit vert (test passe).
* Les indicateurs
* …

La structure des 3A

La structure d’un simple test : la structure des 3A

1. Acteur : On instancie un ou plusieurs objets, un peu comme on placerait des acteurs sur une scène
   1. LaFactorielleCinq : Factorielle
   2. LaFactorielleCinq = Factorielle(5)
2. Action : On appelle en général une seule méthode d’un de ces objets, ce qui déclenche une action (ou plusieurs) bien précise de leur part.
   1. Var résultat : Int
   2. résultat = LaFactorielleCinq.calculer()
3. On termine par une assertion ou une série d’assertions Une assertion est une condition définie par contrat qui vérifie que compte tenu des conditions initiales, l’action est menée à bien ou non.

Le fremework XCTest

Le Framework XCTest permet de réaliser des tests unitaires en Swift

Il permet de créer des tests unitaires tout en respectant trois critères essentiels :

* Les tests unitaires doivent pouvoir être exécutés rapidement
* Leurs résultats doivent être déterministes sans ambiguïtés

2 classes de Framework XCTest sont importantes pour réaliser des tests unitaires :

La classe XCTestCasse est la classe principale pour définir les cas de test, les méthodes de test et les tests de performances.

La création d’un cas de tests

En général un test est une classe qui par convention porte le nom de la classe à tester suivi du mot « Test ».

Cette nouvelle classe doit hériter de la classe EXTestCase

Rappel un test est une classe qui par convention porte le nom de la classe à tester suivi du mot « Test ».

Final class NomDeLaClasseATester : XCTestCase {

…

}

Cette définition doit obligatoirement être précédée de l’instruction :

**@testable import NomDuProjetXcode**

Cette instruction permet d’accéder …. .

Le nom de chaque méthode de test doit être précédé du mot « test » pour que le test fonctionne.

func testExample() throws {

//…

}

Les assertions

Selon lepetit Robert, une assertion est : Phrase affirmative ou négative, censé dire le vrai

C’est donc une vérité que nous devons de vérifier.

Le Framework SCTest fournit aux développeurs un certain nombre de méthodes permettant de vérifier des assertions.

Quelques méthodes sont résumées dans le tableau suivant :

|  |  |
| --- | --- |
| Méthodes | Rôle |
|  |  |
|  |  |
|  |  |

Type de fichiers a créer :

UI test case (pour tester les interface)

Unit test case (pour tester notre code)

Résumé :

Test unitaires : permet de tester si tout fonctionne de façon automatique sans les faire manuellement pour chaque cas

(Pratique si on modifie le code et vérifier si c’est toujours valide avec les test unitaires que l’on a fait)

End Project (td5):

* Contraintes : Intégrer ce que l’on a vu depuis le début en développement d’applis mobile
* Créer Application de notre choix

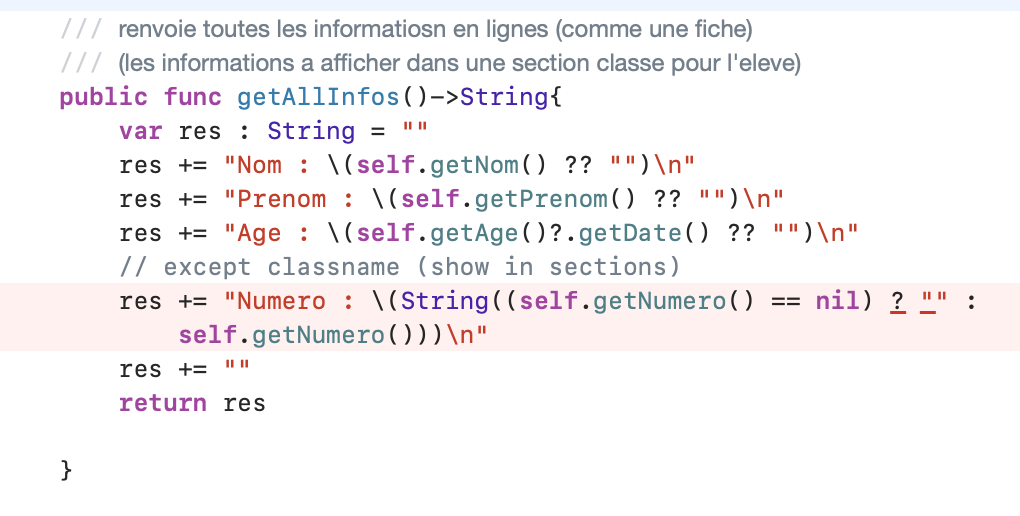
Astuces très utiles (pas dans le cours de la prof)

Pour faire une DocString

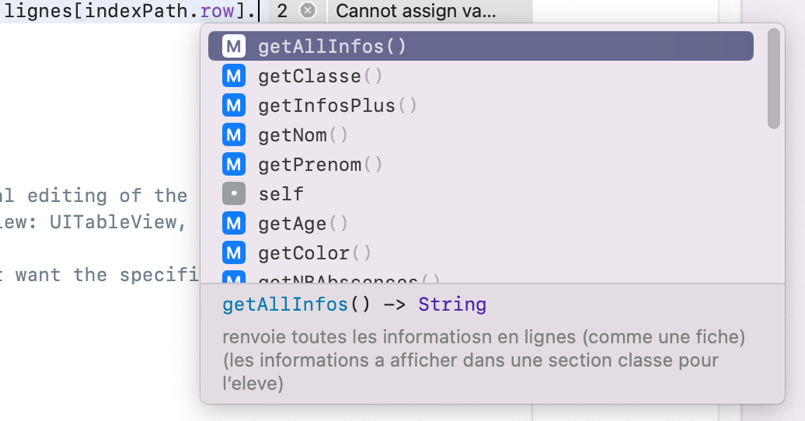
Une **DocString** (en python, Swift etc…) est une chaine placée sous la fonction qui est visible lorsque l’on utilise cette fonction ailleurs et qui permet de donner une description de la fonction

En python elle se fait sous la fonction avec « « « texte » » »

En **Swift** elle se fait avec /// texte sur une ligne avant et **collé** à la fonction

Exemple :

Lors de l’appel de la fonction :



Les remplacement optionnels :

Si une variable déballée est « nil » alors il est possible de mettre du contenu dedans dans ce cas on utilise l’opérateur ?? suivi du contenu à insérer

Exemple :

**var** valeur : String? = **self**.donneUneChaine()

**var** texte : String = valeur ?? ""

si « valeur » existe alors on sera mis dans « texte » sinon « texte » sera vide («») c’est pour cela que texte n’est jamais vide

# end page