Cours n°5 — Gestion des données locales

Partie 1 - Côté Swift

Property observers

Les property observers servent à déclencher des actions avant ou après que la valeur d'une propriété soit modifiée. Deux *observers* sont disponibles : `willSet`, qui est exécuté avant qu'une propriété ne soit modifiée et `didSet` qui est exécuté après qu'une propriété soit modifiée.

On retrouve la même convention de nommage que pour les méthodes qui interviennent dans le cycle de vie des vues ou contrôleurs (`viewDidLoad` par exemple).

La diapositive 4 montre un exemple utilisant une classe permettant de représenter un compte en banque. L'*observer* `willSet` va mettre en pause toutes les transactions sur ce compte ; `didSet` va reprendre toutes les transactions qui étaient en pause et définir la valeur du champ booléen `aDecouvert` si le solde est négatif.

La particularité de `didSet` est que Swift rend accessible, via la variable `oldValue`, la précédente valeur du champ comme on peut le voir sur l'exemple de la diapositive 5.

Propriétés évaluées et *property wrappers*

Les propriétés évaluées sont des propriétés dont la valeur est déterminée dynamiquement ; c'est un peu comme définir une méthode de getter en Java.

Pour définir de telles propriétés, on utilise des *property wrappers*. Deux sont disponibles : `get` et `set`. L'exemple de la diapositive 7 montre une implémentation possible d'une classe représentant un compte empêchant le client d'être à découvert.

Lazy properties

Les *lazy properties* sont des propriétés qui sont évaluées paresseusement, c'est à dire seulement à partir du moment où on y accède. Normalement, le compilateur Swift requiert que la valeur de toutes les propriétés d'une instance soient définies au moment de la création de cette instance.

Ces propriétés permettent d'éviter de réaliser des calculs si cela ne s'avère pas nécessaire à l'exécution du programme et de plus, la valeur de ces propriétés peut se baser sur la valeur d'autres propriétés au moment où elles sont calculées.

La diapositive 9 montre un exemple avec les deux façons permettant de définir des propriétés évaluées paresseusement. On peut soit directement spécifier une valeur (`connection1`) ou spécifier une *closure* qu'on appelle directement si le calcul de la valeur doit se faire sur plusieurs lignes (`connection2`).

Les structures

Les structures sont un concept similaire aux classes ayant la différence notable d'être utilisée par valeur plutôt que par référence. D'autres différences comme l'absence d'héritage les rendent différentes des classes.

Pour rappel, lorsqu'un objet est utilisé par valeur, le fait de l'assigner à une variable ou de le passer en paramètre d'une fonction va créer une copie en mémoire de cet objet. Lorsqu'un objet est utilisé par référence, c'est le même espace mémoire qui sera partagé entre les différentes variables ou appel de fonctions. (c.f. diapositives 11 et 12)

L'avantage notable des structures est que le compilateur Swift fournit un constructeur par défaut prenant en paramètre tous les attributs d'une structure contrairement à une classe. (c.f. diapositive 13)

La classe `Optional`

Pour pallier le fait qu'une valeur peut être absente, Swift propose la classe `Optional` qui peut être vue comme une boîte. On connaît le contenu de la boîte (le type de la variable dans notre cas) mais la boîte peut être remplie ou vide. En Swift, l'absence de valeur est représentée avec la constante `nil`.

La diapositive 15 montre les syntaxes possibles pour définir une variable en tant qu'instance de la classe `Optional`.

Pour contourner le fait que l'on utilise de telles valeurs dans nos programmes et que des erreurs peuvent être levées si une valeur est absente, plusieurs opérateurs sont mis à notre disposition comme l'opérateur `??`, le chaînage optionnel ou le chaînage sans condition. (c.f. diapositive 16 à 19)

Il est également possible de fournir des constructeurs qui créent un `Optional` de la classe en question et qui renvoient `nil` dans les cas voulus. La diapositive 20 reprend un exemple du compte ne pouvant pas avoir de découvert. La création d'une instance renvoie `nil` si le solde passé au constructeur est négatif.

Le typage

Comme dans de nombreux langages, l'héritage et le polymorphisme font que le type de certains objets peut parfois s'avérer trop ou pas assez spécifique pour nos besoins. En prenant un exemple d'une classe `Document` et d'une classe `Livre` héritant de la première (diapositive 21), les diapositives 22 et 23 exposent les différentes opérations de conversion de type qu'il est possible d'effectuer en Swift.

Il est également possible de déterminer le type d'un objet grâce à l'opérateur `is`. Ceci peut s'avérer très pratique avec des collections hétérogènes comme le tableau de documents parcouru à la diapositive 24. Il faudra cependant veiller à vérifier les types les plus spécifiques (les types enfants) en premier.

Partie 2 - Côté iOS

Affichage des données

Table View Controller

Une des classes de prédilection des applications iOS est la classe `UITableViewController` permettant d'afficher simplement, sous forme de tableau une liste d'éléments. L'avantage est que des modèles pré-existants sont déjà proposés dans Xcode mais il est également possible de définir les éléments de nos cellules de manière personnalisée.

Pour créer un tel contrôleur, il suffit de glisser-déposer sur le storyboard un « Table View Controller » depuis la bibliothèque d'éléments de Xcode (icône + en haut à droite lorsque le storyboard est ouvert). La personnalisation des éléments et l'association avec le code sont exposés entre les diapositives 27 et 31.

Une fois tout placé côté storyboard, il suffit d'implémenter les méthodes présentées en diapositives 34 et 35 pour alimenter la « Table View » en données et ainsi réaliser l'affichage.

Collection View Controller

À l'instar des « Table View Controllers », les « Collection View Controllers », représentés avec la classe `UICollectionViewController`, permettent d'afficher simplement une grille d'éléments. Contrairement à la classe vue précédemment, il n'y a pas de modèle pré-défini, c'est au développeur de fournir le patron de chaque cellule et les espacements entre elles via le storyboard comme présenté dans les diapositives 39 et 40.

Comme pour le précédent contrôleur, pour récupérer la cellule côté code, il faut lui donner un identifiant. (c.f. diapositive 41) Même chose en ce qui concerne l'alimentation en données du contrôleur. (c.f. diapositives 42 à 46)

La seule différence ici est qu'il faut attacher au modèle de la cellule une classe de notre projet permettant de la représenter dans notre code puisque l'on souhaite récupérer ses éléments qui ont été placés dans le storyboard (c.f. diapositives 42 et 43).

Gestion des données en local

Préservation des états de l'application

Lorsque l'utilisateur change d'application ou verrouille son téléphone, l'application passe en mode pause. Par défaut, tous les états de l'application (niveau de scroll d'une vue, contenu des champs textes, etc.) ne sont pas conservés lorsque l'application passe dans un tel mode.

Il est cependant possible de faire en sorte que ce soit le cas en implémentant deux méthodes au niveau du `AppDelegate` de notre application. (c.f. diapositive 50)

La méthode ayant en paramètre `shouldSaveApplicationState` détermine s'il faut conserver ou non les états de l'application au moment du passage en mode pause. Celle possédant le paramètre `shouldRestoreApplicationState` détermine s'il faut les restaurer ou non.

On peut imaginer un système de validité par date, une connexion à un service distant pour vérifier si l'utilisateur ne s'est pas connecté à un autre appareil, etc pour déterminer s'il faut les restaurer ou non.

Préférences simples

Le SDK iOS fournit une classe très facile à utiliser pour stocker des valeurs comme des nombres, des chaînes de caractères, etc.

La classe `UserDefaults` permet le stockage et le chargement de ces valeurs. La méthode `set` permet de définir une valeur en lui donnant un nom (une clé) comme un volume d'écoute par exemple. La méthode associée au type de la donnée (`float` par exemple) permet de récupérer cette valeur une fois stockée. (c.f. diapositive 53)

Stockage des fichiers

Chaque application iOS possède une *sandbox*, représentant l'espace lui étant dédié sur le système de fichiers de l'appareil. La structure des fichiers est présentée diapositive 54. Les applications ne peuvent se partager des fichiers entre leurs sandboxes.

Il peut être intéressant de stocker des données localement sur l'application sous forme de fichiers. De nombreuses facilités pour stocker les données existent, dans principalement deux formats : les Property Lists et le JSON.

Un fichier Property List (plist) est un fichier XML suivant une structure particulière. Le nom du tag décrit le type de données que l'on utilise et on retrouve en général à la racine du fichier un tag `<array>` représentant le tableau de toutes les valeurs stockées. Au sein des dictionnaires, on utilise le tag `<key>` suivit d'un tag ayant pour le nom le type de la valeur pour représenter l'alternance entre clés et valeurs.

La particularité du format plist est qu'il est prépondérant dans le monde Apple mais pas ailleurs. Pour faciliter l'inter-opérabilité avec Android par exemple, il peut être plus judicieux d'utiliser le format JSON.

Il est également possible d'enregistrer les fichiers au format binaire sur le disque, ce qui permet d'enregistrer des structures plus complexes que les types primaires que supporte les fichiers au format texte. Il faut que ces structures implémentent le protocole `Codable`.

Bases SQLite 3

Dans le domaine de l'embarqué, le système de base de données SQLite 3 est relativement populaire car léger et ne nécessite pas de serveur ; la base de données est stockée dans un simple fichier, ce fichier étant créé dynamiquement ou bien créé par vos soins et ajouté avec les autres fichiers de l'application. Il est possible de l'intégrer dans une application iOS, il suffit pour cela de requérir que la bibliothèque soit importée dans le binaire de l'application. (c.f. démarche à suivre diapositives 60 et 61)

Pour accéder à la base SQLite, il faudra ensuite faire appel à une bibliothèque tierce. Le SDK d'Apple ne fournit aucune facilité pour accéder à une base SQLite. Plusieurs options sont listées en diapositive 62 en sachant que fmdb est l'option la plus populaire (pour des raisons d'ancienneté).

Core Data

Finalement, au delà des options précédemment présentées, la solution la plus complète et la plus avancée pour stocker des données est Core Data, un ORM fourni par Apple extrêmement puissant, pouvant être comparé à quelque chose tel que JPA en Java.

La marche à suivre pour utiliser Core Data dans le cas d'une nouvelle application ou d'une application existante est expliquée dans les diapositives 64 à 68.

Un des avantages de Core Data est qu'il est complètement intégré à Xcode. Il est possible de définir et connecter les entités entre elles directement depuis l'IDE. (c.f. diapositives 69 à 72)

Dans le cadre d'une application utilisant Core Data, deux éléments sont mis à disposition dans le `AppDelegate` : le persistent container et la méthode `saveContext`.

Le persistent container est l'objet central nous permettant d'interagir avec les fonctionnalités que propose Core Data.

La méthode `saveContext` sert tout simplement à enregistrer toutes les modifications qui ont été faites depuis son dernier appel.

Finalement, pour récupérer les données enregistrées, on fait appel à des éléments qui portent également le nom de contrôleur mais que l'on appellera « Fetched Results Controller ». C'est via ces contrôleurs qu'il est possible de construire une requête permettant de filtrer, trier ou limiter les données que l'on récupère.

De manière générale, on stocke le contrôleur dans une propriété évaluée paresseusement pour ne pas exécuter la requête tant que ce n'est pas nécessaire. (c.f. diapositive 75)

La construction du contrôleur va ensuite se faire avec une requête qui va contenir les éléments permettant de filtrer, limiter ou trier les éléments. (c.f. diapositive 76 et 77) Il sera ensuite possible de parcourir les éléments récupérés via l'attribut `fetchedObjects` du contrôleur. (c.f. diapositive 78)