TD n°1 - Développement natif sous iOS

Exercice 1 : Hello, World!

Le but de cet exercice est de commencer à prendre en main Swift simplement. Selon votre préférence, créez un nouveau fichier Swift sur votre machine que vous compilerez avec la commande `swiftc` ou utilisez Xcode sous macOS et créez un nouveau « Playground ».

1. Faîtes appel à la fonction `print` pour afficher le message « Hello, World! »

2. Créez une fonction `hello` prenant en paramètre un nom et affichant « Hello » suivi du nom passé en paramètre.

   Rappel : Pour injecter (i.e. interpoler) une variable dans une chaîne, on peut faire comme suit :

   ```
   "Il vous manque \(manquant) euros"
   ```

3. Faîtes en sorte que la fonction créée dans la question précédente puisse être appelée sans spécifier le label.

   Rappel : Pour permettre de spécifier un paramètre sans spécifier son label, il suffit de placer un *underscore* (_) devant le label :

   ```
   func fonction(_ label : String)
   ```

Exercice 2 : Liste de tâches à faire

Le but de cet exercice est d'afficher une liste de tâches à faire, représentées sous forme d'instances d'une classe que vous créerez.

1. Créez une classe `Tache` représentant une tâche.

   Une tâche comporte un titre (`String`) et un champ permettant de savoir si la tâche a été réalisée ou non (`Bool`). Laissez la possibilité de définir ses valeurs via le constructeur de la classe.

2. Créez un tableau contenant quelques tâches en créant des instances de la classe précédemment définie.

3. Utilisez une boucle afin d'afficher la liste précédemment définie. Utilisez une condition dans le corps de la boucle pour préciser si une tâche est réalisée ou non.

   Vous pouvez par exemple afficher `[X]` devant le titre d'une tâche réalisée et `[ ]` devant une tâche qui ne l'est pas.

   Utilisez l'interpolation pour placer le titre dans la chaîne affichée.

Exercice 3 : Traitements sur les tableaux avec les closures

La bibliothèque standard Swift propose tout un tas de méthodes permettant de réaliser des traitements sur les tableaux.

Une des méthodes les plus utiles est `map`. Celle-ci permet de générer un nouveau tableau auquel on aura appliqué un traitement à chaque élément. Par exemple, dans le cas de l'exercice précédent, si on voulait générer un tableau ne contenant que les titres à partir du tableau contenant les tâches :

```
let taches = [
     Tache(titre: "Faire les courses", realisee: false),
     Tache(titre: "Dormir", realisee: true)
]

let titres = taches.map { $0.titre }
```

Le tableau `titres` contiendrait alors les valeurs « Faire les courses » et « Dormir ».

Une autre méthode utile est la méthode `reduce` même si elle peut parfois être mal utilisée et produire un code difficilement lisible. Par exemple, si on avait un tableau de notes :

```
let notes = [12, 15, 18]
```

Si on voulait réaliser la somme de ce tableau, on pourrait utiliser la méthode `reduce` comme suit :

```
let somme = notes.reduce(0, {acc, note in
     acc + note
})
```

Le premier paramètre passé est la valeur initiale, ici 0. Ensuite, une *closure* est passée ; cette dernière prend deux paramètres en entrée : la valeur calculée aux précédentes itérations (appelée l'accumulateur) et la valeur du tableau à l'itération actuelle.

Ici, on aurait donc, à la première itération `acc` qui vaut 0 et `note` qui vaut 12. Au tour suivant, `acc` vaudrait 12 (0 + 12) et `note` 15. A la dernière itération, `acc` vaudrait 27 (12 + 15) et `note` vaudrait 18.

1. Définissez une classe `Note` ayant un attribut `valeur` et un autre `coefficient`.

   Ces deux attributs seront de type `Int`.

2. Définissez un tableau contenant plusieurs notes, étant des instances de la classe précédemment définie.

3. Utilisez la méthode `map` sur le tableau précédemment défini pour créer un tableau contenant les notes coefficientées (c'est à dire en multipliant leur coefficient et leur valeur).

4. Utilisez la méthode `reduce` pour calculer le total des notes coefficientées.

5. Divisez le résultat précédemment obtenu par le nombre de notes et utilisez la méthode `print` pour afficher la moyenne calculée.

   Note : En Swift, l'attribut `count` permet d'obtenir la taille d'un tableau.

Exercice 4 : Quelques fonctions simples

Pour chaque fonction de cet exercice, réalisez au moins un appel à celle-ci dans votre programme.

1. Définissez une fonction `estPair` renvoyant vrai si un nombre est pair et faux s'il ne l'est pas.

2. Définissez une fonction permettant de calculer la suite de Fibonacci. Pour rappel, cette suit est définie comme suit :

   1. Pour 0, la valeur est 0. Pour 1, la valeur est 1
   2. Pour toute autre valeur, la suite est calculée par `fib(n -1) + fib(n -2)` en considérant que `fib` est le nom de la fonction calculant cette suite

3. Définissez une fonction prenant en paramètre un tableau d'entiers et permettant d'en calculer la moyenne.

4. Définissez une fonction prenant en paramètre un tableau d'entiers et permettant de connaître la valeur maximale (sans utiliser la méthode `max` fournie par Swift).

Exercice 5 : Enumérations

Les énumérations permettent de définir un groupe de valeurs étant en relation les unes avec leurs autres. Par exemple, on peut imaginer une énumération contenant des unités de mesure de poids, une contenant des matières de cours, une contenant des rangs d'utilisateurs (administrateur, modérateur, simple utilisateur), etc.

Par exemple, une énumération représentant les points cardinaux serait définie comme suit :

```
enum PointCardinal {
    case nord
    case sud
    case est
    case ouest
}
```

On peut ensuite utiliser une valeur de l'énumération comme valeur d'une variable :

```
var orientation = PointCardinal.nord
```

Swift permet un raccourci une fois que le type de la variable est connu pour ne pas avoir à retaper le nom de l'énumération :

```
orientation = .sud
```

1. Définissez une énumération `Etat` ayant 3 valeurs possibles : `nouvelle`, `encours` et `finie`.

2. Reprenez la classe Tache de l'exercice n°2 et changez l'attribut booléen pour que l'énumération précédemment définie soit utilisée.

3. Comme expliqué dans le cours, les énumérations permettent également de définir des erreurs lancées par nos programmes.

   Définissez une énumération `FibonacciErreur` avec un cas `mauvaisInterval` prenant un paramètre `valeur` de type `Int`.

   Reprenez la fonction définissant la suite de Fibonacci de l'exercice précédent et adaptez là pour que l'erreur que vous venez de définir soit lancée si jamais la valeur passée en paramètre est négative.

   Réalisez ensuite un appel à cette fonction en passant une valeur négative le tout, englobé dans un bloque traitant cette erreur. Le message affiché devra comporter la valeur passée en paramètre.