Semaine 02 - Les bases du langage Swift

Syntaxe, Variables, Contrôles, Collections

Antoine Moevus

Table des matières

Introduction

Swift est un langage de programmation créé par Apple en 2014 pour faciliter et moderniser le développement d'applications iOS et macOS. Avant l'introduction de Swift, le langage de programmation Objective-C était largement utilisé pour créer des applications pour les plateformes Apple. Swift a été conçu pour être plus rapide, plus sûr, et plus facile à lire et à écrire que son prédécesseur. Il combine la simplicité de Python, la performance du C, et la modularité des programmes orientés objet modernes, tels que PH, Scala, ou Rust. Swift permet aux développeurs de créer des applications performantes et élégantes avec un effort moindre.

Son prédécesseur, l'Objective-C, né dans les années 1980, est basé sur le langage de programmation C et ajoute des fonctionnalités de programmation orientée objet inspirées par le langage Smalltalk. Bien qu'Objective-C ait été largement adopté et utilisé pendant de nombreuses années, il présentait certaines limitations et complexités dans sa syntaxe, ce qui rendait le développement d'applications plus difficile et plus long.

Avec Swift, Apple a cherché à créer un langage plus moderne, élégant et intuitif pour résoudre les problèmes rencontrés avec Objective-C. Swift a rapidement gagné en popularité et a été adopté par de nombreux développeurs d'applications iOS et macOS. Selon le classement des langages de programmation TIOBE, Swift se situe actuellement parmi les 20 langages les plus populaires[1].

Grâce à la facilité d'utilisation et aux performances améliorées de Swift, Apple a pu maintenir et étendre sa part de marché dans le domaine des systèmes d'exploitation mobiles et de bureau. Swift est aujourd'hui le langage de choix pour le développement d'applications sur les plateformes Apple et est également utilisé dans d'autres domaines, tels que le développement de serveurs ou la programmation de systèmes embarqués.

Dans cette leçon, nous allons explorer les bases du langage de programmation Swift, utilisé principalement pour le développement d'applications iOS et macOS. Nous aborderons la syntaxe de base, les variables, les constantes, les types de données, les instructions conditionnelles, les boucles et les collections. Ce cours vous fournira les connaissances nécessaires pour commencer à créer des applications avec Swift.

[1] TIOBE Software BV. (s. d.). Indice TIOBE pour mai 2023. TIOBE - The Software Quality Company. Consulté le 2023-05-10, à partir de https://www.tiobe.com/tiobe-index/

Syntaxe de base de Swift

Voici quelques éléments clés de la syntaxe Swift à connaître pour bien débuter.

Commentaires

Les commentaires sont utilisés pour documenter le code et expliquer son fonctionnement. En Swift, les commentaires peuvent être créés en utilisant les caractères `//` pour un commentaire sur une seule ligne, et `/* */` pour un commentaire multi-lignes.

```
// Ceci est un commentaire sur une seule ligne

/*
Ceci est un commentaire
sur plusieurs lignes
*/
```

Point-virgule

En Swift, il n'est généralement pas nécessaire d'utiliser un point-virgule (`;`) pour séparer les instructions, contrairement à d'autres langages tels que C et Objective-C. Cependant, si vous souhaitez écrire plusieurs instructions sur une seule ligne, vous devrez utiliser un point-virgule pour les séparer.

```
// le ; n'est pas obligatoire
let a = 0
let b = 1
let c = 2; // mais vous pouvez le mettre
// En fait si vous voulez faire plusieurs
// instructions sur une ligne, il faut le mettre
let x = 10; let y = 20
```

Noms de variables et constantes

Les noms de variables et de constantes en Swift doivent commencer par une lettre ou un caractère de soulignement `_` et peuvent inclure des lettres, des chiffres et des caractères de soulignement. Les noms de variables et de constantes sont sensibles à la casse.

```
let maConstante = 42
var maVariable = "Bonjour"
```

Déclaration de fonctions

Les fonctions sont déclarées en utilisant le mot-clé `func`, suivi du nom de la fonction, d'une paire de parenthèses contenant les paramètres (le cas échéant) et d'une flèche (`->`) indiquant le type de retour (si la fonction retourne une valeur).

```
func maFonction(parametre1: Int, parametre2: String) -> String {
    // Code de la fonction
}
```

Imprimer des messages

La fonction `print()` est utilisée pour afficher des messages dans la console. Vous pouvez imprimer des chaînes de caractères, des variables ou des constantes.

```
let message = "Bonjour le monde"
print(message) // Affiche "Bonjour le monde"
```

Chaînes de caractères multilingues

Swift utilise le type `String` pour représenter les chaînes de caractères, qui supportent nativement les caractères Unicode. Cela permet de travailler facilement avec des chaînes de caractères multilingues.

```
let salutation = "□□□□□□□□" // "Bonjour" en japonais
print(salutation) // Affiche "□□□□□□□□"
```

Interpolation de chaînes

L'interpolation de chaînes est une méthode pour insérer des variables ou des expressions directement dans une chaîne de caractères, en utilisant une syntaxe simple. Pour interpoler une valeur dans une chaîne, utilisez la syntaxe `\()`.

```
let age = 30
let messageAnniversaire = "Joyeux \(age)ème anniversaire!"
print(messageAnniversaire) // Affiche "Joyeux 30ème anniversaire!"
```

Chaînes multilignes

Swift permet de créer des chaînes de caractères multilignes en utilisant des triples guillemets (`"""`). Vous pouvez ainsi définir une chaîne de caractères sur plusieurs lignes sans avoir à utiliser de séquences d'échappement pour les sauts de ligne.

```
let poeme = """
    Un petit poème
    Écrit sur plusieurs lignes
    En toute simplicité
    """
print(poeme)
// Affiche :
// Un petit poème
// Écrit sur plusieurs lignes
// En toute simplicité
```

NOTE La première et la dernière ligne de votre multichaine doivent être uniquement le triple guillements. Le contenu commence à la ligne suivant le premier triple guillemets et fini avant le dernier triple guillemets.

Variables, constantes et types de données

Dans Swift, les données sont stockées sous forme de variables et de constantes. Une variable est une information qui peut changer, tandis qu'une constante est une information qui reste inchangée une fois définie.

Variables

Une variable est définie en utilisant le mot-clé `var`. Voici comment on peut définir une variable en Swift :

```
var salutation = "Bonjour"
```

Constantes

Une constante est définie en utilisant le mot-clé `let`. Une fois définie, la valeur d'une constante ne peut pas être modifiée.

```
let pi = 3.14159
```

L'absence de valeur : `nil`

En Swift, `nil` n'est pas la même chose que zéro, ni la même chose qu'une chaîne de caractères vide. Au lieu de cela, `nil` signifie "absence de valeur valide". Par exemple, si vous déclarez une variable de type optionnel mais que vous ne lui donnez pas de valeur, sa valeur par défaut sera `nil`.

```
var optionalString: String?  // optionalString est de type optionnel String
print(optionalString)  // imprime "nil", car optionalString n'a pas encore de valeur
```

Les variables non optionnelles doivent toujours avoir une valeur. Par exemple le code ci-dessous va produire une erreur de compilation :

```
var hey: String  // hey est de type String et n'est pas initialisée
print(hey)  // Erreur de compilation : Variable 'hey' used before being initialized
```

NOTE	Il est important de noter que vous ne pouvez assigner `nil` qu'aux variables optionnelles. Si vous essayez d'assigner `nil` à une variable non optionnelle, vous obtiendrez une erreur de compilation. C'est une façon pour Swift de vous assurer que vous traitez toujours correctement les cas où une valeur peut être absente.

Typage fort et compilation

Le système de typage en Swift est statique et fortement typé. Cela signifie que le type de chaque variable et expression est connu au moment de la compilation, et qu'il ne peut pas être changé par la suite. C'est une caractéristique clé qui permet à Swift d'être à la fois puissant et sûr.

Une variable en Swift doit toujours avoir un type explicite ou implicite. Vous pouvez spécifier le type de manière explicite lors de la déclaration de la variable, comme ceci :

```
var name: String = "John Doe"
```

Ici, `name` est explicitement déclaré comme une `String`. Cependant, Swift dispose également d'une fonctionnalité appelée inférence de type, qui lui permet de déterminer automatiquement le type d'une variable en fonction de sa valeur initiale. Ainsi, vous pourriez également écrire :

```
var name = "John Doe"  // name est automatiquement de type String
```

Dans ce cas, Swift détermine que `name` doit être une `String` parce que sa valeur initiale est une `String`.

Le typage statique a plusieurs avantages. Il permet au compilateur de détecter et de signaler les erreurs de typage au moment de la compilation, plutôt que de les laisser se produire pendant l'exécution. Cela peut aider à prévenir de nombreux bugs courants. De plus, il permet au compilateur d'optimiser le code plus efficacement, car il connaît le type exact de chaque variable à l'avance.

En revanche, le typage statique est moins flexible que le typage dynamique, car une variable ne peut pas changer de type après sa déclaration. Cela signifie que vous devez faire attention à déclarer vos variables avec le bon type, et que vous devrez parfois utiliser des conversions de type explicites si vous devez changer le type d'une variable, comme par exemples avec les collections.

Typage fort avec les collections

Lorsqu'il s'agit de collections, comme les tableaux (`Array`), le typage fort en Swift nécessite un effort supplémentaire pour garantir la cohérence des types. En Swift, un tableau est toujours d'un seul type. Par exemple, si vous déclarez un tableau d'entiers (`Int`), vous ne pouvez y ajouter que des entiers. Si vous essayez d'y ajouter une chaîne de caractères (`String`), par exemple, le compilateur vous arrêtera avec une erreur.

```
var numbers = [1, 2, 3]  // numbers est de type [Int]
numbers.append("four")  // erreur : 'String' n'est pas compatible avec 'Int'
```

De plus la gestion des valeurs inexistante est problématique. Il existe des situations où une variable peut ne pas avoir de valeur. Par exemple, si vous essayez de récupérer un élément à un index qui n'existe pas dans le tableau. Pour gérer ces cas, Swift utilise le concept des "optionnels". Un "optionnel" en Swift signifie que la variable peut contenir une valeur d'un certain type, ou qu'elle

peut ne contenir aucune valeur (elle serait alors `nil`). Les optionnels sont déclarés en ajoutant un point d'interrogation (`?`) après le type de la variable.

```
let names = ["John", "Jane", "Joe"]
let index = names.firstIndex(of: "Mary")  // index est de type Int?
```

Dans l'exemple ci-dessus, `firstIndex(of:)` renvoie un `Int?` (un entier optionnel). Si "Mary" se trouve dans le tableau, `index` contiendra son indice. Sinon, `index` sera `nil`, indiquant qu'il n'y a pas de valeur. Le typage fort et les optionnels en Swift garantissent ainsi la sécurité et la prévisibilité de votre code.

Types de données

Swift est un langage à typage statique, ce qui signifie que le type de chaque variable ou constante doit être connu au moment de la compilation. Swift fournit plusieurs types de données intégrés, notamment :

- `Int` pour les nombres entiers. Ils peuvent être signés (positifs, zéros ou négatifs) ou non signés (toujours positifs). Swift fournit également des entiers de différentes tailles : `Int8`, `Int16`, `Int32`, `Int64` pour les entiers signés et `UInt8`, `UInt16`, `UInt32`, `UInt64` pour les entiers non signés.

```
var petitNombre: Int8 = 127
var grandNombre: UInt64 = 18446744073709551615
```

- `Double` et `Float` pour les nombres à virgule flottante. `Double` a une précision d'au moins 15 chiffres décimaux, tandis que `Float` peut être aussi précis que 6 chiffres décimaux.

```
var pi: Double = 3.141592653589793
var approximationPi: Float = 3.14159
```

- `Bool` pour les valeurs booléennes. Un `Bool` peut être soit `true`, soit `false`.

```
var lumiereAllumee: Bool = true
```

- `String` pour les chaînes de caractères. Les `String` en Swift sont Unicode, ce qui signifie qu'elles peuvent stocker n'importe quel caractère de n'importe quelle langue.

```
var salutation: String = "Bonjour, monde!"
```

En plus de ces types, Swift offre également des types plus complexes tels que les collections (`Array`, `Set`, `Dictionary`), les optionnels, les tuples, et des types personnalisés que vous pouvez définir à l'aide de `class`, `struct`, et `enum`.

Instructions conditionnelles et boucles

Dans cette section, nous allons explorer les structures de contrôle de Swift, qui sont essentielles pour effectuer différentes opérations de manière automatisée en fonction des conditions dans lesquelles le programme s'exécute.

Structures de décisions

Condition `if`, `else` et `else if`

La structure de contrôle `if` permet d'exécuter un bloc de code si une certaine condition est remplie. Le bloc `else` est exécuté si la condition `if` n'est pas remplie. Vous pouvez également utiliser `else if` pour vérifier plusieurs conditions.

```
var temperature = 30

if temperature > 28 {
    print("Il fait chaud !")
} else if temperature < 10 {
    print("Il fait froid.")
} else {
    print("Il fait bon.")
}
```

Condition `switch`

Swift offre également une structure de contrôle `switch` pour comparer une valeur à plusieurs cas possibles. `switch` est particulièrement puissant en Swift et peut être utilisé avec différents types de données, pas seulement des nombres, et permet des comparaisons plus complexes.

```
var temperature = 30

switch temperature {
case 0..<10:
    print("Il fait froid.")
case 10..<28:
    print("Il fait bon.")
default:
    print("Il fait chaud !")
}
```

Dans cet exemple, `switch` compare la valeur de `temperature` à plusieurs plages de valeurs (appelées "intervalles" en Swift). Si `temperature` se trouve dans l'un de ces intervalles, le bloc de code correspondant est exécuté. Si aucune correspondance n'est trouvée, le bloc `default` est exécuté.

Les structures de répétitions en Swift

Boucle `for-in` et intervalles

La boucle `for-in` est très flexible et peut être utilisée avec divers types de séquences, y compris les intervalles. Un intervalle est une séquence de valeurs entre une valeur de début et une valeur de fin. En Swift, vous pouvez créer des intervalles à l'aide de l'opérateur `...` (intervalles fermés) ou `..<` (intervalles semi-ouverts).

```
// Intervalles fermés incluent la valeur de fin
for i in 1...5 {
    print(i)  // Affiche les nombres de 1 à 5
}

// Intervalles semi-ouverts excluent la valeur de fin
for i in 1..<5 {
    print(i)  // Affiche les nombres de 1 à 4
}
```

Vous pouvez également utiliser la boucle `for-in` pour itérer sur des éléments dans un tableau :

```
let noms = ["Anna", "Alex", "Brian", "Jack"]
for nom in noms {
    print("Hello, \(nom)!")
}
```

Dans cet exemple, la boucle `for-in` parcourt chaque élément du tableau `noms` et affiche un message de bienvenue pour chaque nom.

Il est également possible d'utiliser la boucle `for-in` pour itérer sur un dictionnaire, qui est une collection non ordonnée de paires clé-valeur. Lors de l'itération sur un dictionnaire, chaque élément est une paire de tuple `(clé, valeur)`.

```
let nombreDePattes = ["araignée": 8, "fourmi": 6, "chat": 4]
for (nomAnimal, comptePattes) in nombreDePattes {
    print("Les \(nomAnimal)s ont \(comptePattes / 2) pairs de pattes")
}
```

Dans cet exemple, la boucle `for-in` parcourt chaque paire `(clé, valeur)` du dictionnaire `nombreDePattes` et affiche le nombre de pairs de pattes pour chaque animal.

Boucle `while`

La boucle `while` exécute un bloc de code tant qu'une condition est vraie. Voici un exemple :

```
var nombre = 1
while nombre <= 5 {
    print(nombre)
    nombre += 1
}
```

Dans cet exemple, la boucle `while` continue de s'exécuter tant que la variable `nombre` est inférieure ou égale à 5, affichant ainsi les nombres de 1 à 5.

Boucle `repeat-while`

La boucle `repeat-while`, aussi connue sous le nom de boucle `do-while` dans d'autres langages, est similaire à une boucle `while`, à la différence qu'elle vérifie la condition après avoir exécuté le bloc de code. Cela garantit que le bloc de code est exécuté au moins une fois. Voici un exemple :

```
var nombre = 1
repeat {
    print(nombre)
    nombre += 1
} while nombre <= 5
```

Dans cet exemple, la boucle `repeat-while` affiche les nombres de 1 à 5, tout comme dans l'exemple précédent. Cependant, si la condition était fausse dès le début (par exemple, si `nombre` était initialement 6), le bloc de code serait quand même exécuté une fois avec une boucle `repeat-while`, mais pas avec une boucle `while`.

Forçage de type pour les booléens

En Swift, une valeur booléenne est soit vraie (`true`), soit fausse (`false`). Cependant, contrairement à certains autres langages de programmation, Swift est strict sur le type de valeurs qu'il accepte dans les expressions conditionnelles. Seules les valeurs booléennes `true` et `false` peuvent être utilisées. Par conséquent, Swift n'a pas de concept de "Falsy" ou "Truthy" pour des valeurs non booléennes comme en Javascript.

Par exemple, le code suivant générera une erreur en Swift :

```
let i = 1
if i {
    // Ceci est une erreur
}
```

Dans ce cas, Swift générera une erreur car `i` n'est pas une valeur booléenne.

Cela peut sembler restrictif, mais cela aide à prévenir les erreurs. En rendant obligatoire l'utilisation de valeurs booléennes dans les conditions, Swift nous oblige à être explicites sur nos intentions, ce qui peut aider à rendre notre code plus clair et moins sujet à des bugs.

Ainsi, si nous voulons vérifier si `i` est différent de zéro, nous devons le faire explicitement :

```
let i = 1
if i != 0 {
    // Ceci est correct
}
```

De cette manière, Swift garantit que les expressions conditionnelles sont toujours claires et sans ambiguïté.

Cependant, dans le cas des types optionnels, Swift introduit le concept de `nil`. Une variable optionnelle est considérée "falsy" lorsqu'elle est `nil`, et "truthy" lorsqu'elle a une valeur.

Par exemple, considérez le code suivant :

```
var maVariableOptionnelle: String?
if maVariableOptionnelle {
    print("maVariableOptionnelle a une valeur")
} else {
    print("maVariableOptionnelle est nil")
}
```

Dans ce cas, Swift affichera "maVariableOptionnelle est nil", car `maVariableOptionnelle` n'a pas de valeur (elle est `nil`) et est donc considérée comme "falsy".

Si nous attribuons une valeur à `maVariableOptionnelle` et exécutons à nouveau le code :

```
var maVariableOptionnelle: String? = "Bonjour"
if maVariableOptionnelle {
    print("maVariableOptionnelle a une valeur")
} else {
    print("maVariableOptionnelle est nil")
}
```

Swift affiche maintenant "maVariableOptionnelle a une valeur", car `maVariableOptionnelle` a une valeur et est donc considérée comme "truthy".

Collections : Array, Set, Dictionary

Array

Un `Array` en Swift est une collection ordonnée d'éléments du même type. Voici comment déclarer, initialiser et manipuler un tableau en Swift :

```swift
// Déclaration et initialisation d'un tableau
var nombres: [Int] = [1, 2, 3, 4, 5]

// Ajout d'un élément à la fin du tableau
nombres.append(6)

// Accès à un élément du tableau
let premierNombre = nombres[0]

// Modification d'un élément du tableau
nombres[0] = 7

// Suppression d'un élément du tableau
nombres.remove(at: 0)
```

Les différentes manière d'initialiser un Array

```swift
// Initialisation vide
var arrayVide1: [Int] = []
var arrayVide2 = [Int]()

// Initialisation avec des valeurs par défaut
var arrayDefaut = [Int](repeating: 0, count: 5)

// Initialisation avec une liste de valeurs
var arrayValeurs: [Int] = [1, 2, 3, 4, 5]
var arrayValeurs2 = [1, 2, 3, 4, 5]

// Initialisation avec une plage de valeurs
var arrayPlage = Array(1...5)
```

Set

Un `Set` en Swift est une collection non ordonnée d'éléments uniques. Voici comment déclarer, initialiser et manipuler un ensemble en Swift :

```swift
// Déclaration et initialisation d'un ensemble
var lettres: Set<Character> = ["a", "b", "c"]

// Ajout d'un élément à l'ensemble
lettres.insert("d")

// Vérification de la présence d'un élément dans l'ensemble
let contientD = lettres.contains("d")

// Suppression d'un élément de l'ensemble
lettres.remove("d")
```

Dictionary

Un `Dictionary` en Swift est une collection non ordonnée d'associations clé-valeur. Voici comment déclarer, initialiser et manipuler un dictionnaire en Swift :

```swift
// Déclaration et initialisation d'un dictionnaire
var ages: [String: Int] = ["Alice": 30, "Bob": 25]

// Accès à la valeur associée à une clé
let ageAlice = ages["Alice"]

// Modification de la valeur associée à une clé
ages["Alice"] = 31

// Suppression d'une association clé-valeur
ages.removeValue(forKey: "Alice")
```

Exemple récapitulatif

Voici un exemple de code Swift qui utilise les notions vues dans ce cours.

Cet exemple est une simple simulation du jeu "Serpents et Échelles" [2]:

Les règles du jeu sont les suivantes :

- Le plateau de jeu compte 25 cases et le but est d'atteindre ou de dépasser la case 25.
- La case de départ du joueur est la "case zéro", qui se trouve juste en dehors du coin inférieur gauche du plateau.
- A chaque tour, vous lancez un dé à six faces et vous avancez du nombre de cases indiqué par le dé.
- Si votre tour se termine au bas d'une échelle, vous montez à la case correspondante au haut de l'échelle.
- Si votre tour se termine à la tête d'un serpent, vous descendez à la case correspondante à la queue du serpent.
- Le dernier lancer de dé doit arriver à la case 25 exactement. Si le lancer dépasse il est annulé et vous devez réessayer au prochain tour.

Initialisation du plateau de jeu

Commençons par initialiser notre plateau de jeu. Pour l'instant, toutes les cases sont neutres.

```
let caseFinale = 25
var plateau = [Int](repeating: 0, count: caseFinale + 1)
```

Le plateau de jeu est représenté par un tableau de valeurs entière. Sa taille est basée sur une constante appelée `caseFinale`, qui est utilisée pour initialiser le tableau et également pour vérifier une condition de victoire plus tard dans l'exemple. Comme les joueurs commencent hors du plateau, sur la "case zéro", le plateau est initialisé avec 26 valeurs Int, et non 25.

Initialisation des variables de jeu

Ensuite, nous initialisons les variables pour la case courante et le jet de dé. Nous commençons à la case 0 et le jet de dé est également défini à 0.

```
var caseCourante = 0
var jetDe = 0
```

Boucle de jeu

Maintenant, nous créons la boucle principale de notre jeu. Pour l'instant, le joueur avance

simplement du nombre de points qu'il obtient au dé.

```
while caseCourante < caseFinale {
    // lance le dé
    jetDe = 1 //FIXIT: Rendre le lançage de dé aléatoire
    // avance du montant du dé
    caseCourante += jetDe
}
```

Ajout des serpents et des échelles

Maintenant, nous ajoutons les serpents et les échelles à notre plateau. Les cases avec la base d'une échelle ont un nombre positif pour vous faire monter sur le plateau, tandis que les cases avec la tête d'un serpent ont un nombre négatif pour vous faire descendre sur le plateau.

```
plateau[03] = +08; plateau[06] = +11; plateau[09] = +09; plateau[10] = +02
plateau[14] = -10; plateau[19] = -11; plateau[22] = -02; plateau[24] = -08
```

Gestion des serpents et des échelles

Ensuite, nous modifions notre boucle de jeu pour prendre en compte les serpents et les échelles. Si le joueur atterrit sur une case avec un serpent ou une échelle, il monte ou descend en conséquence.

```
while caseCourante < caseFinale {
    // lance le dé
    jetDe = 1 //FIXIT: Rendre le lançage de dé aléatoire
    // avance du montant du dé
    caseCourante += jetDe
    // si on est toujours sur le plateau, monte ou descend pour un serpent ou une
échelle
    caseCourante += plateau[caseCourante]
}
```

Gestion de la dernière case

Il se peut que le jet de dé fasse dépasser le joueur de la dernière case. Dans ce cas, nous ne voulons pas que le joueur avance. Pour gérer cela, nous vérifions si `caseCourante` dépasse `caseFinale` après le déplacement par le montant de `jetDe`. Si c'est le cas, nous annulons le déplacement (en pratique, la case courante n'est pas incrémentée).

```
while caseCourante < caseFinale {
    // lance le dé
    jetDe = 1 //FIXIT: Rendre le lançage de dé aléatoire
    // avance du montant du dé
    if caseCourante + jetDe <= caseFinale {
        caseCourante += jetDe
        caseCourante += plateau[caseCourante]
    }
}
```

Jet de dé aléatoire

Maintenant, nous allons rendre le lancer de dé aléatoire. Swift fournit la fonction `Int.random(in:)` pour générer un nombre entier aléatoire dans un certain intervalle. Nous allons l'utiliser pour simuler le lancer d'un dé à six faces.

```
while caseCourante < caseFinale {
    // lance le dé
    jetDe = Int.random(in: 1...6)
    // avance du montant du dé
    if caseCourante + jetDe <= caseFinale {
        caseCourante += jetDe
        caseCourante += plateau[caseCourante]
    }
}
```

Fin de la partie

Enfin, une fois que la boucle de jeu est terminée, nous imprimons un message pour indiquer que la partie est terminée.

```
print("Fin de la partie !")
```

Code Final

```
let caseFinale = 25
var plateau = [Int](repeating: 0, count: caseFinale + 1)

plateau[03] = +08; plateau[06] = +11; plateau[09] = +09; plateau[10] = +02
plateau[14] = -10; plateau[19] = -11; plateau[22] = -02; plateau[24] = -08

var caseCourante = 0
var jetDe = 0
while caseCourante < caseFinale {
    // lance le dé
    jetDe = Int.random(in: 1...6)
    if caseCourante + jetDe <= caseFinale {
        // avance du montant du dé
        caseCourante += jetDe
        // si on est toujours sur le plateau, monte ou descend pour un serpent ou une
échelle
        caseCourante += plateau[caseCourante]
    }
}
print("Fin de la partie !")
```

[2] https://fr.wikipedia.org/wiki/Serpents_et_%C3%A9chelles

Conclusion

Il est crucial de comprendre que l'apprentissage d'un nouveau langage de programmation, comme Swift, est une exploration. Chaque concept que vous avez appris ici, chaque ligne de code que vous avez écrite, chaque erreur que vous avez corrigée, c'est une étape sur le chemin de la maîtrise. Mais ce chemin ne s'arrête jamais vraiment. Comme certains disent, "L'apprentissage constant est la seule chose qui ne vieillit jamais". L'informatique évolue constamment, et il est crucial d'être adaptable et de continuer à apprendre.

Cela dit, ne vous précipitez pas pour apprendre tous les concepts en une seule fois. Prenez le temps de bien comprendre chaque concept, de faire des exercices, de lire et de relire si nécessaire. Warren Buffet a dit : "Ne pas investir dans ce que l'on ne comprend pas", et cela s'applique aussi à l'apprentissage. Vous devez investir votre temps et vos efforts pour comprendre ces concepts, car ce sont eux qui formeront la base de votre carrière en développement Swift.

Références

1. Swift.org. (n.d.). Control Flow — The Swift Programming Language (Swift 5.5). Récupéré de https://docs.swift.org/swift-book/documentation/the-swift-programming-language/controlflow/
2. Swift.org. (n.d.). Collection Types — The Swift Programming Language (Swift 5.5). Récupéré de https://docs.swift.org/swift-book/documentation/the-swift-programming-language/collectiontypes/

Exercices

Partie 1: Structures de contrôles

1. **Exercice 1 : Condition `if-else`** Écrivez un programme qui détermine si un nombre est pair ou impair.
2. **Exercice 2 : Boucle `for-in`** Écrivez un programme qui imprime les 10 premiers nombres de la suite de Fibonacci.
3. **Exercice 3 : Boucle `while`** Écrivez un programme qui trouve le plus petit nombre entier `n` tel que `2^n` dépasse un million.
4. **Exercice 4 : Boucle `repeat-while`** Écrivez un programme qui demande à l'utilisateur de saisir un nombre jusqu'à ce qu'il saisisse un nombre dans la plage de 1 à 10.
5. **Exercice 5 : Conditions `if-else` `if-else`** Écrivez un programme qui simule un système de notation. Si le score est supérieur à 90, imprimez "A". Si le score est supérieur à 80, imprimez "B", etc. Traitez le cas où le score est inférieur à 50 comme "F".
6. **Exercice 6 : Condition `switch`** Écrivez un programme qui traduit les jours de la semaine de l'anglais vers le français en utilisant `switch`.

Partie 2 : Les collections

1. Créez un tableau d'entiers appelé "nombres" avec cinq éléments. Ajoutez ensuite un sixième élément à ce tableau.
2. Créez un `Set` de chaînes de caractères appelé "fruits" avec trois éléments. Vérifiez ensuite si le set contient l'élément "pomme". .Créez un dictionnaire qui contient le nom des jours de la semaine en français et leur équivalent en anglais. Par exemple, "lundi" serait la clé et "Monday" serait la valeur. Essayez ensuite de récupérer la traduction en anglais de "mercredi".
3. Soit un tableau d'entiers appelé "notes" avec les éléments `[12, 15, 13, 14, 16, 15, 14]`. Calculez la note moyenne.
4. Créez un tableau d'entiers appelé "nombresDupliqués" qui contient des doublons. Utilisez un `Set` pour éliminer les doublons, et reconvertissez en tableau le résultat sans doublons.

Partie 3

Exercice 1 : Structures de contrôle

Écrivez un programme qui utilise une boucle `for-in` pour imprimer les 10 premiers nombres de la séquence de Fibonacci.

Exercice 2 : Génération de couleurs hexadécimales

Écrivez un programme qui génère un code couleur hexadécimal aléatoire. Un code couleur hexadécimal est une chaîne de 6 caractères où chaque caractère peut être l'un des suivants : 0, 1, 2,

3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Chaque code couleur hexadécimal commence par un dièse (#).

Exercice 3 : Structures de contrôle et collections

Utilisez une boucle `for-in` pour trouver le nombre le plus grand dans un tableau d'entiers.

Exercice 4 : Fonctions

Créez une fonction qui prend un nombre entier et renvoie un tableau de tous les diviseurs de ce nombre.

Exercice 5 : Structures de contrôle - Instruction switch

Écrivez un programme qui utilise une instruction switch pour imprimer le nom d'une note musicale (do, ré, mi, fa, sol, la, si) en fonction de son numéro (1 pour do, 2 pour ré, etc.). Le programme doit gérer les cas où le numéro n'est pas compris entre 1 et 7.

Exercice 6 : Types personnalisés et collections

Créez un type personnalisé `Étudiant` avec des propriétés pour le nom, l'âge et la note moyenne. Ensuite, créez un tableau d'objets `Étudiant`. Utilisez une boucle `for-in` pour calculer la note moyenne de tous les étudiants.

Partie 4

1. En quoi l'utilisation de `let` pour déclarer une constante peut-elle améliorer la lisibilité et la sécurité du code dans Swift? Quels sont les cas où il serait préférable d'utiliser `var` pour déclarer une variable ?
2. Comment l'utilisation des optionnels en Swift aide-t-elle à gérer les situations où une valeur peut être présente ou absente ? Quels problèmes peut-elle résoudre par rapport à d'autres langages de programmation qui n'ont pas ce concept ?
3. Quel est l'impact de l'utilisation de différentes structures de contrôle, telles que `for`, `while`, `repeat-while`, `if`, `else if` et `switch` dans la gestion du flux de contrôle dans un programme Swift ? Pouvez-vous donner un exemple concret d'une situation où l'utilisation d'une certaine structure de contrôle serait plus appropriée que les autres ?