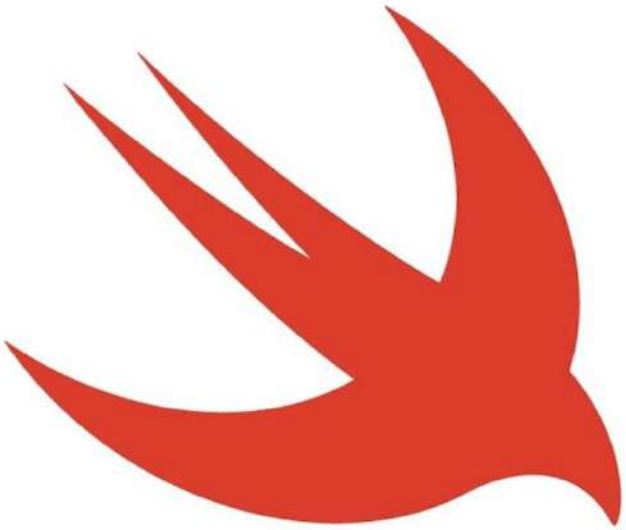
Bêta



Programmer en Swift

Guide pour Swift 2.2

Table des matières

Projet de traduction du guide de programmation Swift en Français

Bienvenue Dans Swift

Découvrir Swift

Swift est un nouveau langage de programmation pour les applications iOS, OS X, watchOS et tvOS qui a été conçu avec le meilleur du C et de l'Objective-C, sans les contraintes de la compatibilité avec le C. Swift propose un modèle de programmation sûr et sécurisé, ajouté à des fonctionnalités modernes ayant pour but de rendre la programmation plus simple, plus flexible et plus fun. Avec l'aide de Cocoa et Cocoa Touch, des frameworks matures et populaires, Swift fait table rase de toutes les anciennes technologies et offre une opportunité pour réinventer le développement logiciel.

La construction de Swift a commencé il y a des années. Chez Apple, nous avons posé ses fondations en avançant nos compilateurs, déboggueurs et infrastructures de frameworks existants. Nous avons simplifié la gestion de la mémoire avec Automatic Reference Counting (ARC). Notre collection de frameworks, construits sur les bases solides que sont Foundation et Cocoa, a été entièrement modernisée et standardisée. L'Objective-C lui-même à évolué pour supporter les blocs, les collections et les modules ; rendant ainsi possible une adoption sans interruption de frameworks contenant les technologies d'un langage moderne. Grâce à ce travail de fond, nous pouvons maintenant introduire un nouveau langage pour le futur du développement de logiciels Apple.

Swift paraît familier aux développeurs Objective-C. Il adopte la facilité de lecture des paramètres et la puissance du modèle de gestion dynamique des objets propres à ce langage. Il fournit un accès continu aux frameworks Cocoa existants et la possibilité d'être mixé avec du code Objective-C (*interpolability*). En plus de cette base commune, Swift introduit de nombreuses nouvelles fonctionnalités et unit les procédures et les portions orientées-objet du langage.

Swift est adapté aux nouveaux programmeurs. C'est le premier langage de programmation pour des systèmes distribués dans le monde entier qui est aussi expressif et plaisant à écrire. Il supporte les Playgrounds, une fonctionnalité innovante qui permet aux programmeurs de faire des expériences avec Swift et d'en voir le résultat immédiatement, sans avoir à construire une app complète

Swift combine ce qu'il y a de mieux dans la philosophie des langages modernes avec la sagesse de la grande culture technologique et de l'ingénierie d'Apple. Le compilateur est optimisé pour la performance, et le langage est optimisé pour le développement, sans que l'un ne soit plus important que l'autre. Swift a été conçu pour supporter toutes sortes de programmes, sur une échelle allant du simple "hello, world" jusqu'à un système d'exploitation tout entier. Tout ceci fait de Swift un investissement intéressant dans l'avenir pour les développeurs et pour Apple.

Swift est un moyen fantastique d'écrire des apps pour iOS, OS X, watchOS, et tvOS, et va continuer d'évoluer avec de nouvelles fonctionnalités et possibilités. Nos objectifs pour Swift sont ambitieux.

Nous avons hâte de découvrir ce que vous aller créer avec lui.

Démarrer avec Swift

La tradition suggère que le premier programme dans un nouveau langage devrait afficher les mots *“Hello, world!”* sur l'écran. En Swift, on peut le faire en une seule ligne :

```
print("Hello, world!")
```

Si vous avez déjà écrit du code en C ou en Objective-C, cette syntaxe vous est familière. En Swift, cette ligne de code est un programme complet. Vous n'avez pas besoin d'importer une bibliothèque séparée pour des fonctionnalités comme la gestion des entrées/sorties ou des chaînes de caractère. Le code écrit dans le champ global est utilisé comme le point d'entrée du le programme, ce qui signifie que vous n'avez pas besoin d'une fonction `main()` . Par ailleurs, les points-virgules à la fin de chaque ligne ne sont pas obligatoires.

Note
Sur un Mac, téléchargez la *Playground* et double-cliquez sur le fichier pour l'ouvrir dans Xcode : https://developer.apple.com/go/?id=swift-tour

Valeurs Simples

Utilisez `let` pour créer une constante et `var` pour créer une variable. La valeur d'une constante n'a pas besoin d'être définie au moment de la compilation, mais sa valeur ne peut être définie qu'une seule et unique fois. Ainsi, vous pouvez utiliser des constantes pour nommer une valeur qui ne sera déclarée qu'une seule fois mais que vous réutiliserez à plusieurs endroits.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

Une constante ou une variable doivent avoir le même type que celui de la valeur que vous voulez lui donner. Toutefois, il n'est pas toujours nécessaire d'écrire le type explicitement. Donner une valeur à une constante ou une variable quand vous la déclarez permet au compilateur de déduire son type. Dans l'exemple ci-dessus, le compilateur à déduit que `myVariable` est de type "Entier" (*Int*) car sa valeur initiale est un entier (42). Si la valeur initiale ne fournit pas assez d'informations (ou si il n'y a pas de valeur initiale), spécifiez le type de la variable/constante en l'écrivant après le symbole `:` .

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

Expérience
Créez une constante de type `Float` avec une valeur `4`.

Les valeurs ne peuvent jamais être converties vers un autre type implicitement. Si vous avez à convertir une valeur vers un autre type, créez explicitement une instance du type désiré.

```
let label = "La largeur est de "
let width = 94
let widthLabel = label + String(width)
```

Expérience
Essayez de retirer la conversion vers le type `String` dans la dernière ligne. Quelle erreur obtenez-vous ?

Il existe une manière encore plus simple d'inclure des valeurs dans du texte : Écrivez la valeur entre parenthèses et ajoutez un *backslash* (`\`) avant les parenthèses. Par exemple :

```
let apples = 3
let oranges = 5
let appleSummary = "J'ai \(apples) pommes."
let fruitSummary = "J'ai \(apples + oranges) morceaux de fruit."
```

Expérience
Utilisez `\()` pour inclure une opération entre deux nombres décimaux dans une chaîne de caractère, puis pour inclure un nom dans un message.

Vous pouvez créer des tableaux (*Array*) et des dictionnaires (*Dictionary*) en utilisant des crochets (`[]`), et accéder à leurs éléments en écrivant leur index ou leur clé dans les crochets. Une virgule peut être placée après le dernier élément.

```
var shoppingList = ["poisson-chat", "eau", "tulipes", "peinture bleue"]
shoppingList[1] = "bouteille d'eau"

var occupations = [
    "Malcolm": "Capitaine",
    "Kaylee": "Mécanicien",
]
occupations["Jayne"] = "Relations Publiques"
```

Pour créer un tableau ou un dictionnaire vide, utilisez la syntaxe d'initialisation.

```
1.  let emptyArray = [String]()
2.  let emptyDictionary = [String: Float]()
```

Si le type peut être suggéré (par exemple lorsque vous changez la valeur d'une variable ou passez un argument à une fonction), un tableau vide s'écrira `[]` et un dictionnaire vide `[:]`.

```
1.  shoppingList = []
2.  occupations = [:]
```

Control Flow

Utilisez `if` et `switch` pour tester des conditions, et utilisez `for`-`in`, `for`, `while`, et `repeat`-`while` pour ajouter des boucles. Il n'est pas obligatoire d'utiliser des parenthèses autour de la variable de la condition ou de la boucle. Des accolades autour du corps de ces dernières sont obligatoires.

```
1.  let individualScores = [75, 43, 103, 87, 12]
2.  var teamScore = 0
3.  for score in individualScores {
4.      if score > 50 {
5.          teamScore += 3
6.      } else {
7.          teamScore += 1
8.      }
9.  }
10.  print(teamScore)
```

Dans une règle `if`, la conditionnelle doit une une expression de type *Boolean* — ce qui signifie que du code comme `if score { ... }` est une erreur, et non une comparaison implicite à zéro.

Vous pouvez combiner `if` et `let` pour traiter des valeurs qui peuvent être manquantes. Ces valeurs manquantes sont nommées *optionnelles*. Une valeur optionnelle contient soit une valeur, soit `nil` dans le cas où la valeur est manquante. Ajoutez un point d'interrogation (`?`) après la déclaration du type d'une variable/constante pour la marquer comme optionnelle.

```
var optionalString: String? = "Bonjour"
print(optionalString == nil)
//Affiche `false` : optionalString contient une valeur
var optionalName: String? = "John Appleseed"
var greeting = "Bonjour!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

Expérience
Assignez `nil` à `optionalName`. Quel message obtenez-vous ? Ajoutez un cas `else` qui affiche in message différent si `optionalName` est égal à `nil`

Si la valeur optionnelle vaut `nil`, alors la condition vérifiée est `false` et le code entre accolades est ignoré. Dans le cas contraire, la valeur optionnelle est *déballée* (on parle d'*optionnal unwrapping*) et assignée à la constante après le mot-clé `let`, ce qui rend la valeur optionnelle disponible au sein du bloc de code.

Il est également possible de gérer les valeurs optionnelles en leur fournissant une valeur par défaut, grâce à l'opérateur `??`. Si la valeur optionnelle est manquante, alors la valeur par défaut sera utilisée à la place.

```
let nickName: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Salut \(nickName ?? fullName)"
```

Les *switchs* supportent tous les types de données et une grande variété d'opérations de comparaisons. Ils ne sont pas limités aux entiers ou aux tests d'égalité.

```
let vegetable = "poivre rouge"
switch vegetable {
case "céleri":
    print("Ajoutez des raisins et ça fera des fourmis sur bûche.")
case "concombre", "cresson":
    print(`"Vivent les sandwiches.")
case let x where x.hasSuffix("poivre"):
    print("Est-ce que ce \(x) est piquant ?")
default:
print("Tout passe dans une soupe.")
}
```

Expérience
Essayez d'enlever le cas `default`. Quelle erreur obtenez-vous ?

Remarque : `let` peut être utilisé pour assigner la valeur qui correspond à un modèle à une constante.

Après avoir exécuté le code contenu dans le cas correspondant, le programme quitte le `switch`. L'exécution ne se propage pas au cas suivant ; vous n'avez donc pas besoin d'ajouter un `break` à la fin du code de chaque cas.

Les boucles `for` - `in` servent à répéter une action pour chaque item dans un Dictionnaire, en fournissant une paire de noms de variables à utiliser pour chaque paire clé-value. Un dictionnaire n'est une collection ordonnée, leurs clés et valeurs sont donc traitées dans un ordre arbitraire.

```
1.  let interestingNumbers = [
2.      "Premiers": [2, 3, 5, 7, 11, 13],
3.      "Fibonacci": [1, 1, 2, 3, 5, 8],
4.      "Carrés Parfaits": [1, 4, 9, 16, 25],
5.  ]
6.  var largest = 0
7.  for (kind, numbers) in interestingNumbers {
8.      for number in numbers {
9.          if number > largest {
10.             largest = number
11.         }
12.     }
13.  }
14.  print(largest)
```

Expérience
Ajoutez une variable qui enregistre quel était le type du plus grand nombre, ainsi que sa valeur.

On utilise `while` pour répéter un bloc de code jusqu'à ce qu'une condition change. On peut écrire la condition en haut, ou en bas, ce qui permet d'être sûr que la boucle est exécutée au moins une fois.

```
1.  var n = 2
2.  while n < 100 {
3.      n = n * 2
4.  }
5.  print(n)

7.  var m = 2
8.  repeat {
9.      m = m * 2
10. } while m < 100
11. print(m)
```

Vous pouvez ajouter un index à une boucle -soit en utilisant `..<` pour créer un intervalle d'index ou en écrivant une suite explicite "initialisation;condition;incrémentation".

Ces deux boucles font la même chose :

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
print(firstForLoop)

var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
print(secondForLoop)
```

Vous pouvez utiliser `..<` pour déclarer un intervalle qui omet la valeur supérieure (intervalle fermé), et utiliser `...` pour un intervalle qui inclut les deux valeurs (intervalle ouvert).

Fonctions et Closures

On utilise le mot-clé `func` pour déclarer une fonction. On lance (ou appelle) une fonction en ajoutant une liste d'arguments entre parenthèses après son nom. On sépare les paramètres du type retourné par la fonction par le symbole `->` .

```
func greet(name: String, day: String) -> String {
    return "Bonjour \(name), aujourd'hui c'est \(day)."
}
greet("Bob", day: "Mardi")
```

Expérience
Retirez le paramètre `day` . Ajoutez un paramètre qui inclut le plat du jour dans un message.

Les tuples sont utiles pour créer une valeur composée - par exemple, pour retourner plusieurs valeurs d'une fonction. Les éléments d'un tuple se référencent soit par nombre ou par nom.

```
1.  func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
2.      var min = scores[0]
3.      var max = scores[0]
4.      var sum = 0

6.      for score in scores {
7.         if score > max {
8.             max = score
9.         } else if score < min {
10.            min = score
11.        }
12.     sum += score
13. }

15.  return (min, max, sum)
16.  }
17.  let statistics = calculateStatistics([5, 3, 100, 3, 9])
18.  print(statistics.sum)
19.  print(statistics.2)
```

Les fonctions peuvent également accepter un nombre variable d'arguments, qui seront collectés dans un tableau.

```
1.  func sumOf(numbers: Int...) -> Int {
2.      var sum = 0
3.      for number in numbers {
4.          sum += number
5.      }
6.      return sum
7.  }
8.  sumOf()
9.  sumOf(42, 597, 12)
```

Expérience
Écrivez une fonction qui calcule la moyenne de ses arguments.

Les fonctions peuvent être combinées. Les fonctions combinées ont accès aux variables déclarées dans les fonctions extérieures. Vous pouvez combiner des fonctions pour organiser le code dans une fonction longue et complexe.

```
1.  func returnFifteen() -> Int {
2.      var y = 10
3.      func add() {
4.          y += 5
5.      }
6.      add()
7.      return y
8.  }
9.  returnFifteen()
```

Les fonctions sont de type *primaire*. Cela signifie qu'une fonction peut retourner une autre fonction.

```
1.  func makeIncrementer() -> ((Int) -> Int) {
2.      func addOne(number: Int) -> Int {
3.          return 1 + number
4.      }
5.      return addOne
6.  }
7.  var increment = makeIncrementer()
8.  increment(7)
```

Une fonction peut avoir une autre fonction comme argument.

```
1.  func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
2.      for item in list {
3.          if condition(item) {
4.              return true
5.          }
6.      }
7.      return false
8.  }
9.  func lessThanTen(number: Int) -> Bool {
10.     return number < 10
11. }
12. var numbers = [20, 19, 7, 12]
13. hasAnyMatches(numbers, condition: lessThanTen)
```

Les fonctions sont en fait une forme spéciale de *closure* : des blocs de code qui peuvent être exécutés plus tard. Le code dans la *closure* ont accès à des éléments comme des variables ou des fonctions disponibles dans le contexte où elle a été créée, même si ce contexte a changé lorsque cette dernière est exécutée. L'exemple des fonctions combinées illustre d'ailleurs ce principe.

Vous pouvez écrire une *closure* sans lui donner de nom, en entourant son code avec des accolades (`{}`). Utilisez `in` pour séparer les arguments et le type de l'élément retourné de son corps.

```
numbers.map({
(number: Int) -> Int in
let result = 3 * number
return result
})
```

Expérience
Ré-écrivez la *closure* de manière à ce qu'elle retourne zéro si *number* est impair.

Vous pouvez raccourcir les *closures* de plusieurs manières. Quand le type d'une *closure* est déjà connu, comme dans le cas d'un *callback* d'un *delegate*, vous pouvez retirer le type de ses paramètres, du type de la valeur qu'elle retourne, où les deux. Les *closures* qui ne déclarent qu'un seul élément retournent cet élément implicitement.

```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
```

Vous pouvez utiliser les paramètres avec des nombres plutôt qu'avec leur nom. Cette approches est particulièrement utile dans les *closures* très courtes. Quand une *closure* est le dernier argument d'une fonction, on peut l'écrire directement après les parenthèses. Quand une *closure* est le seul argument d'une fonction, les parenthèses ne sont plus nécessaires.

```
let sortedNumbers = numbers.sort { $0 > $1 }
print(sortedNumbers)
```

Objets et Classes

On crée une classe en utilisant `class` suivi du nom de la classe. On déclare une propriété au sein d'une classe de la même manière qu'on déclare des variables et des constantes, seul le contexte change. Les méthodes (fonctions) sont également déclarées de la même manière.

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "Une forme avec \(numberOfSides) côtés."
    }
}
```

Expérience
Ajoutez à `Shape` une propriété constante avec `let` et une méthode avec un argument.

On créé une instance de la classe en ajoutant des parenthèses après le nom de la classe. Utilisez un point pour accéder à ses méthodes et propriétés.

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

Il manque quelque chose d'important à cette version de `Shape` : un initiateur qui permet de paramétrer la classe quand l'instance est créée. Pour en créer un, on ajoute la méthode `init` .

```
class NamedShape {
var numberOfSides: Int = 0
var name: String

    init(name: String) {
       self.name = name
    }

    func simpleDescription() -> String {
        return "Une forme avec \(numberOfSides) côtés."
    }
}
```

On peut remarquer que `self` permet de distinguer la propriété `name` de l'argument `name` de l'initiateur. Les arguments de l'initiateur sont passés de la même manière que lors de l'appel d'une fonction quand vous initialisez une classe. Toutes les propriétés doivent avoir été assignées à une valeur, soit au moment de leur déclaration (comme c'est le cas pour `numberOfSizes`), ou dans l'initiateur (comme pour `name`).

Vous pouvez ajouter la fonction `deinit` pour créer un dé-initiateur si vous avez besoin de faire des changements (comme du nettoyage) avant que l'objet soit dé-alloué de la mémoire.

Les sous-classes se créent en ajoutant le nom de la super-classe après deux points (`:`). Les classes ne sont pas obligées d'être des sous-classes de classes standard. Vous pouvez donc inclure ou non une super-classe en fonction de vos besoins.

Les méthodes d'une sous-classe qui ignorent l'implémentation de la super-classe sont marquées avec le mot-clé `override` . Ignorer une méthode par accident, sans `override` est considéré comme une erreur par le compilateur. Le compilateur détecte également les

méthodes marquées `override` qui n'ignorent en rien les méthodes de la super-classe.

```
1.  class Square: NamedShape {
2.      var sideLength: Double

4.      init(sideLength: Double, name: String) {
5.          self.sideLength = sideLength
6.          super.init(name: name)
7.          numberOfSides = 4
8.      }

10.      func area() -> Double {
11.          return sideLength * sideLength
12.      }

14.      override func simpleDescription() -> String {
15.          return "Un carré de \(sideLength) de côté."
16.      }
17.  }
18.  let test = Square(sideLength: 5.2, name: "mon exemple de carré")
19.  test.area()
20.  test.simpleDescription()
```

Expérience
Créez une autre sous-classe de `NamedShape` appelée `Circle` qui est initialisée avec un rayon et un nom. Implémentez-y les méthode `area` et `simpleDescription()`

En plus du simple stockage de valeurs, les propriétés permettent aussi d'ajouter un *getter* et un *setter*.

```
1.  class EquilateralTriangle: NamedShape {
2.      var sideLength: Double = 0.0

4.      init(sideLength: Double, name: String) {
5.          self.sideLength = sideLength
6.          super.init(name: name)
7.          numberOfSides = 3
8.      }

10.     var perimeter: Double {
11.         get {
12.             return 3.0 * sideLength
13.         }
14.         set {
15.             sideLength = newValue / 3.0
16.         }
17.     }

19.     override func simpleDescription() -> String {
20.         return "An equilateral triangle with sides of length \(sideLength)."
21.     }
22. }
23. var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
24. print(triangle.perimeter)
25. triangle.perimeter = 9.9
26. print(triangle.sideLength)
```

Dans le setter de `perimeter`, la nouvelle valeur est implicitement nommée `newValue`. Vous pouvez définir une valeur explicite entre parenthèses après `set`.

On remarque que l'initialisation de `EquilateralTriangle` se fait en trois étapes :

1. Définition de la valeur des propriétés de la sous-classe.
2. Appel de l'initiateur de la super-classe.
3. Changement de la valeur des propriétés de la super-classe. Les tâches qui utilisent des méthodes, des *getters* ou des *setters* peuvent également être effectuées à ce moment.

Si vous avez besoin de calculer une propriété mais avez besoin de fournir du code qui sera exécuté avant et après lui avoir attribué une nouvelle valeur, utilisez `willSet` et `didSet`. Ce code est exécuté à chaque fois que la valeur change en dehors de contexte d'initialisation. Par exemple, la classe ci-dessous s'assure que la taille du côté de son triangle est toujours égale à celle de son carré.

```
1. class TriangleAndSquare {
2.     var triangle: EquilateralTriangle {
3.         willSet {
4.             square.sideLength = newValue.sideLength
5.         }
6.     }
7.     var square: Square {
8.         willSet {
9.             triangle.sideLength = newValue.sideLength
10.        }
11. }
12. init(size: Double, name: String) {
13.      square = Square(sideLength: size, name: name)
14.      triangle = EquilateralTriangle(sideLength: size, name: name)
15.    }
16. }
17. var triangleAndSquare = TriangleAndSquare(size: 10, name: "une autre forme de test")
18. print(triangleAndSquare.square.sideLength)
19. print(triangleAndSquare.triangle.sideLength)
20. triangleAndSquare.square = Square(sideLength: 50, name: "un carré plus large ")
21. print(triangleAndSquare.triangle.sideLength)
```

Lorsque vous utilisez des valeurs optionnelles, vous pouvez écrire un `?` avant des opérations comme des méthodes, des propriétés, ou d'indexation. Si la valeur avant `?` est égale à `nil`, tout ce qui est écrit après le `?` sera ignoré et la valeur de l'expression entière sera égale à `nil`. Dans le cas contraire, la valeur optionnelle et tout ce qui se trouve après `?` se comportera comme une valeur optionnelle. Dans les deux cas, la valeur de l'expression est optionnelle.

```
1.  let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")
2.  let sideLength = optionalSquare?.sideLength
```

Enumérations and Structures

Utilisez le mot-clé `enum` pour créer une énumération. Comme les classes et d'autres types, les énumérations peuvent être associées avec des méthodes.

```
1.  enum Rank: Int {
2.      case Ace = 1
3.      case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
4.      case Jack, Queen, King
5.      func simpleDescription() -> String {
6.          switch self {
7.          case .Ace:
8.          return "ace"
9.          case .Jack:
10.          return "jack"
11.          case .Queen:
12.          return "queen"
13.          case .King:
14.          return "king"
15.          default:
16.          return String(self.rawValue)
17.          }
18.      }
19.  }
20.  let ace = Rank.Ace
21.  let aceRawValue = ace.rawValue
```

Expérience
Écrivez une fonction qui compare deux valeurs `Rank` grâce à leurs valeurs *brutes*

Dans l'exemple ci-dessus, le type de la valeur brute est `Int`. Cela vous permet donc de ne spécifier la valeur brute que pour le premier cas. Les autres sont assignées dans l'ordre. Vous pouvez également utiliser des chaînes de caractères ou des nombres décimaux comme type de valeurs brutes pour une énumération. Utilisez la propriété `rawValue` pour accéder la valeur brute d'un cas d'une énumération.

Utilisez l'initiateur `init?(rawValue:)` pour créer l'instance d'une énumération à partir d'une valeur brute.

```
1.  if let convertedRank = Rank(rawValue: 3) {
2.      let threeDescription = convertedRank.simpleDescription()
3.  }
```

Les cas d'une énumération sont des valeurs à part entière, et non une autre manière d'écrire leurs valeurs brutes. En fait, dans le cas où fournir une valeur brute à une énumération n'a pas de sens particulier, il n'est pas nécessaire de la faire.

```
1.  enum Suit {
2.      case Spades, Hearts, Diamonds, Clubs
3.      func simpleDescription() -> String {
4.          switch self {
5.          case .Spades:
6.          return "spades"
7.          case .Hearts:
8.          return "hearts"
9.          case .Diamonds:
10.          return "diamonds"
11.          case .Clubs:
12.          return "clubs"
13.          }
14.      }
15.  }
16.  let hearts = Suit.Hearts
17.  let heartsDescription = hearts.simpleDescription()
```

Expérience

Ajoutez la méthode `color()` à `Suit` qui retourne *black* pour les cas `Spades` et `Clubs`, et retourne *red* pour les cas `Hearts` et `Diamonds`

On peut remarquer les deux manières de référencer le cas `Hearts` de l'énumération. Quand on assigne sa valeur à la constante `hearts`, le cas `Suit.Hearts` est référencé dans sa forme *complète*, car la constante n'a pas un type explicite. Vous pouvez utiliser la forme abrégée à partir du moment où le type de la propriété est déjà connue.

Utilisez `struct` pour créer une structure. Les structures sont en plusieurs points similaires aux classes, comme par exemple au niveau des méthodes et des initiateurs. Une des différences les plus importantes entre les structures et les classes est que les structures sont toujours copiées quand elles sont passées comme argument/valeur dans votre code, alors que les classes le sont en tant que référence.

```
1.  struct Card {
2.      var rank: Rank
3.      var suit: Suit
4.      func simpleDescription() -> String {
5.          return "Le \(rank.simpleDescription()) de \(suit.simpleDescription())"
6.      }
7.  }
8.  let threeOfSpades = Card(rank: .Three, suit: .Spades)
9.  let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

Expérience
Ajoutez une méthode à `Card` qui créée un jeu de cartes complet, avec un carte de chaque combinaison de couleur et de type.

L'instance d'un cas d'une énumération peut être associé à des valeurs. Les instances d'un même cas peuvent avoir des valeurs associées différentes. Vous fournissez les valeurs associées quand vous créez une instance. Les valeurs associées et les valeurs brutes sont des notions différentes : la valeur brute d'un cas d'une énumération est la même pour toutes les instances, et est fournie lorsque vous définissez/écrivez l'énumération.

Par exemple, considérez le cas où vous récupérez l'heure de lever et de coucher du soleil depuis un serveur. Le serveur renvoie soit une information, soit une erreur.

```
1.  enum ServerResponse {
2.      case Result(String, String)
3.      case Error(String)
4.  }

6.  let success = ServerResponse.Result("6:00 am", "8:09 pm")
7.  let failure = ServerResponse.Error("Out of cheese.")

9.  switch success {
10.      case let .Result(sunrise, sunset):
11.      print("Le soleil se lèvera à \(sunrise) et se couchera à \(sunset).")
12.      case let .Error(error):
13.      print("Échec... \(error)")
14.  }
```

Expérience
Ajoutez un troisième cas à `ServerResponse` et au *switch*.

Retenez comment l'heure de lever et de coucher du soleil sont extraites de la valeur `ServerResponse` et utilisées comme des cas dans le *switch*.

Protocoles and Extensions

On déclare un protocole avec le mot-clé `protocol`.

```
1.  protocol ExampleProtocol {
2.  var simpleDescription: String { get }
3.  mutating func adjust()
4.  }
```

Les classes, les énumérations et structures peuvent suivre des protocoles.

```
class SimpleClass :  ExampleProtocol  {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
 let aDescription = a.simpleDescription

 struct SimpleStructure :  ExampleProtocol {
     var simpleDescription: String = "A simple structure"
     mutating func adjust() {
         simpleDescription += " (adjusted)"
     }
 }
 var b = SimpleStructure()
 b.adjust()
 let bDescription = b.simpleDescription
```

Expérience
Écrivez une énumération qui est conforme à ce protocole.

Remarquez l'utilisation du mot-clé `mutating` dans la déclaration de `SimpleStructure` pour marquer une méthode qui modifie la structure. La déclaration de `SimpleClass` n'a pas besoin de voir ses méthodes marquées comme `mutating`, car les méthodes d'une classe peuvent toujours la modifier.

L'utilisation d' `extension` permet d'ajouter des fonctionnalités à un type existant, telles que de nouvelles méthodes et des propriétés calculées. Vous pouvez utiliser une extension pour permettre à un type déclaré ailleurs d'être conforme à un protocole ; de même pour les types importés depuis une bibliothèque ou un *framework*.

```
extension Int :  ExampleProtocol {
    var simpleDescription: String {
        return "Le nombre \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
print(7.simpleDescription)
```

Expérience
Écrivez une extension pour le type `Double` qui ajoute une propriété `absoluteValue`.

Vous pouvez utiliser un protocole comme tout autre type. Par exemple, vous pouvez créer une collection d'objets qui ont des types différents mais qui sont tous conformes au même protocole. Quand vous travaillez avec des valeurs dont le type est un protocole, il est impossible de déclarer des méthodes en dehors du protocole.

```
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// print(protocolValue.anotherProperty) // Enlevez le commentaire pour voir l'erreur
```

Même si la variable `protocolValue` est de type `SimpleClass` au moment de l'exécution, le compilateur le traite comme une variable de type `ExampleProtocol`. Cela signifie que vous ne pouvez pas accidentellement accéder aux méthodes et aux propriétés que la classe implémente en plus du cadre de la conformité au protocole.

Génériques

Écrivez un nom entre les symboles `<>` pour créer un type ou une fonction générique.

```
func repeatItem<Item>(item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
repeatItem("knock", numberOfTimes:4)
```

Vous pouvez créer des génériques à partir de fonctions et méthodes, mais aussi à partir de classes, d'énumérations et de structures.

```
// Recréez le type optionnel de la bibliothèque standard Swift
enum OptionalValue<Wrapped> {
    case None
    case Some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger` = .Some(100)
```

Utilisez `where` après le nom du type pour spécifier une liste de pré-requis ; par exemple pour que le type doivent impérativement respecter un protocole, que deux types soient identiques, ou qu'une classe ait une super-classe particulière.

```
1.  func anyCommonElements <T: SequenceType, U: SequenceType where T.Generator.Element: E
2.  for lhsItem in lhs {
3.      for rhsItem in rhs {
4.          if lhsItem == rhsItem {
5.              return true
6.          }
7.      }
8.  }
9.  return false
10. }
11. anyCommonElements([1, 2, 3], [3])
```

Expérience
Modifiez la fonction `anyCommonElements(_:_:)` pour qu'elle retourne un tableau des éléments que les deux séquences ont en commun.

Écrire `<T: Equatable>` revient à écrire `<T where T: Equatable>`.

Language Guide