Concept fondamentaux Swift

fonctions, gestion des erreurs, et programmation orientée objet

Antoine Moevus

Table des matières

Références

1. Swift.org. (2023). Le langage de programmation Swift: Fonctions. Consulté le 22 mai 2023, de https://docs.swift.org/swift-book/documentation/the-swift-programming-language/functions
2. Swift.org. (2023). Le langage de programmation Swift: Énumérations. Consulté le 22 mai 2023, de https://docs.swift.org/swift-book/documentation/the-swift-programming-language/enumerations
3. Swift.org. (2023). Le langage de programmation Swift: Classes et structures. Consulté le 22 mai 2023, de https://docs.swift.org/swift-book/documentation/the-swift-programming-language/classesandstructures

Fonctions

En une phrase: les fonctions sont des morceaux de code autonomes qui effectuent une tâche spécifique. Vous donnez à une fonction un nom qui identifie ce qu'elle fait, et ce nom est utilisé pour "appeler" la fonction et lui faire exécuter sa tâche lorsque cela est nécessaire. En programmation, il est important d'utiliser des fonctions pour organiser et réutiliser du code.

La syntaxe unifiée des fonctions de Swift est suffisamment flexible pour modélisr la plupart des cas, depuis une fonction simple de style C sans noms de paramètres jusqu'à une méthode complexe de style Objective-C avec des noms et des étiquettes d'argument pour chaque paramètre. Les paramètres peuvent fournir des valeurs par défaut pour simplifier les appels de fonction et peuvent être passés en tant que paramètres d'entrée-sortie, ce qui modifie une variable passée une fois que la fonction a terminé son exécution.

Chaque fonction en Swift a un type, composé des types de paramètres de la fonction et du type de retour. Vous pouvez utiliser ce type comme n'importe quel autre type en Swift, ce qui facilite le passage de fonctions en tant que paramètres à d'autres fonctions et le retour de fonctions à partir de fonctions. Les fonctions peuvent également être écrites à l'intérieur d'autres fonctions pour encapsuler des fonctionnalités utiles dans une portée de fonction imbriquée.

Création et utilisation de fonctions

Pour créer une fonction en Swift, vous devez spécifier son nom, les paramètres qu'elle accepte et le type de valeur qu'elle renvoie. Voici un exemple de syntaxe de déclaration de fonction :

```
func nomDeLaFonction(parametre1: Type, parametre2: Type) -> TypeDeRetour {
    // Code à exécuter
    return valeurDeRetour
}
```

Dans cet exemple, `nomDeLaFonction` est le nom de la fonction, `parametre1` et `parametre2` sont les noms des paramètres que la fonction accepte avec leurs types respectifs, `TypeDeRetour` est le type de valeur que la fonction renvoie, et `valeurDeRetour` est la valeur renvoyée par la fonction.

Voici un exemple concret :

```
func direBonjour() {
    print("Bonjour !")
}

direBonjour() // Appel de la fonction pour afficher "Bonjour !"
```

Dans cet exemple, nous avons défini une fonction `direBonjour` qui ne prend aucun paramètre et ne renvoie aucune valeur. Elle imprime simplement "Bonjour !" à la console. Nous appelons ensuite cette fonction en utilisant son nom suivi de parenthèses vides pour exécuter le code à l'intérieur de la fonction.

À proprement parler, cette version de la fonction `direBonjour()` retourne quand même une valeur, même si aucun type de retour n'est défini. Les fonctions sans type de retour défini retournent une valeur spéciale de type `Void`. Il s'agit simplement d'un tuple vide, qui est écrit comme ().

```
// forme équivalente
func direBonjour() -> () {
    print("Bonjour !")
}
```

Les fonctions peuvent également prendre des paramètres pour effectuer des opérations spécifiques. Voici un exemple :

```
func direBonjour(prenom: String) {
    print("Bonjour, \(prenom) !")
}

direBonjour(prenom: "Alice") // Appel de la fonction pour afficher "Bonjour, Alice !"
direBonjour(prenom: "Bob") // Appel de la fonction pour afficher "Bonjour, Bob !"
```

Dans cet exemple, la fonction `direBonjour` prend un paramètre de type `String` appelé `prenom`. Elle imprime ensuite un message de salutation en utilisant la valeur du paramètre. Lorsque nous appelons la fonction, nous passons une valeur spécifique pour le paramètre `prenom`.

Les fonctions peuvent également renvoyer une valeur en utilisant le mot-clé `return`. Voici un exemple :

```
func additionner(nombre1: Int, nombre2: Int) -> Int {
    let somme = nombre1 + nombre2
    return somme
}

let resultat = additionner(nombre1: 5, nombre2: 3) // Appel de la fonction et
assignation du résultat à une variable
print("Le résultat de l'addition est \(resultat)") // Affiche "Le résultat de
l'addition est 8"
```

Dans cet exemple, la fonction `additionner` prend deux paramètres de type `Int`, effectue une opération d'addition et renvoie la somme. Lorsque nous appelons la fonction et assignons le résultat à une variable, nous pouvons utiliser cette variable pour afficher le résultat.

Fonctions avec plusieurs valeurs de retour

Vous pouvez utiliser un type de `tuple` comme type de retour d'une fonction pour renvoyer plusieurs valeurs en tant que partie d'une valeur de retour composée.

L'exemple ci-dessous définit une fonction appelée `minMax(array:)`, qui trouve les nombres les plus petits et les plus grands dans un tableau de valeurs `Int` :

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

La fonction `minMax(array:)` renvoie un tuple contenant deux valeurs `Int`. Ces valeurs sont étiquetées `min` et `max` afin qu'elles puissent être accessibles par leur nom lors de la consultation de la valeur de retour de la fonction.

Le corps de la fonction `minMax(array:)` commence par définir deux variables de travail appelées `currentMin` et `currentMax`, qui prennent la valeur du premier entier du tableau. Ensuite, la fonction itère sur les valeurs restantes du tableau et vérifie si chaque valeur est plus petite ou plus grande que les valeurs de `currentMin` et `currentMax` respectivement. Enfin, les valeurs minimale et maximale globales sont renvoyées sous la forme d'un tuple de deux valeurs `Int`.

Étant donné que les membres du tuple sont nommés en tant que partie du type de retour de la fonction, ils peuvent être accessibles à l'aide de la syntaxe du point pour récupérer les valeurs minimale et maximale trouvées :

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("Le minimum est \(bounds.min) et le maximum est \(bounds.max)")
// Affiche "Le minimum est -6 et le maximum est 109"
```

Notez que les membres du tuple n'ont pas besoin d'être nommés au moment où le tuple est renvoyé par la fonction, car leurs noms sont déjà spécifiés en tant que partie du type de retour de la fonction.

Types de retour de tuple optionnels

Si le type de tuple à renvoyer par une fonction a la possibilité de ne pas avoir de "valeur" pour l'ensemble du tuple, vous pouvez utiliser un type de retour de tuple optionnel pour refléter le fait que l'ensemble du tuple peut être `nil`. Un type de retour de tuple optionnel s'écrit en plaçant un point d'interrogation après la parenthèse de fermeture du type de tuple, par exemple `(Int, Int)?` ou `(String, Int, Bool)?`.

NOTE

Tuple d'optionnels ou tuple optionnel?

L'option avec des tuples optionnels est différente d'un tuple qui contient des types optionnels tels que (Int?, Int?). Avec un type de tuple optionnel, l'ensemble du tuple est optionnel, et non chaque valeur individuelle du tuple.

La fonction `minMax(array:)` ci-dessus renvoie un tuple contenant deux valeurs `Int`. Cependant, la fonction ne vérifie pas la sécurité du tableau qui lui est passé. Si l'argument du tableau contient un tableau vide, la fonction `minMax(array:)`, telle que définie ci-dessus, déclenchera une erreur d'exécution lors de la tentative d'accès à `array[0]`.

Pour gérer un tableau vide de manière sécurisée, écrivez la fonction `minMax(array:)` avec un type de retour de tuple optionnel et renvoyez une valeur de `nil` lorsque le tableau est vide :

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Le code suivant permet de vérifier si cette version de la fonction `minMax(array:)` renvoie une valeur de tuple réelle ou `nil` :

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("Le minimum est \(bounds.min) et le maximum est \(bounds.max)")
}
// Affiche "Le minimum est -6 et le maximum est 109"
```

Étiquettes d'argument et paramètres

Les fonctions en Swift peuvent avoir à la fois des étiquettes d'argument et des noms de paramètres. L'étiquette d'argument est utilisée lors de l'appel de la fonction pour spécifier l'argument, tandis que le nom de paramètre est utilisé à l'intérieur de la fonction pour référencer la valeur de l'argument.

Par défaut,les fonctions en Swift utilisent les noms de paramètres comme étiquettes d'argument.

Par exemple, dans la fonction `direBonjour(personne: String)`, le nom de paramètre `personne` est également utilisé comme étiquette d'argument lors de l'appel de la fonction. Cela permet une lecture plus fluide du code.

```
func direBonjour(personne: String) {
    print("Bonjour, \(personne)!")
}
```

```
direBonjour(person: "Alice") // Utilisation de l'étiquette d'argument
```

Cependant, il est possible de spécifier une étiquette d'argument différente pour améliorer la clarté lors de l'appel de la fonction.

Absence d'étiquettes d'arguments

Si vous ne souhaitez pas utiliser d'étiquette d'argument pour un paramètre, vous pouvez utiliser un trait de soulignement _ à la place de l'étiquette d'argument explicite pour ce paramètre.

Dans l'exemple suivant, la fonction `someFunction(::)` prend deux paramètres, `firstParameterName` et `secondParameterName`, sans étiquettes d'arguments :

```
func someFunction(_ firstParameterName: Int, _ secondParameterName: Int) {
    // Dans le corps de la fonction, firstParameterName et secondParameterName
    // font référence aux valeurs des arguments pour les premiers et seconds
paramètres.
}
```

Lors de l'appel de cette fonction, les étiquettes d'arguments sont omises :

```
someFunction(1, 2)
```

Si un paramètre possède une étiquette d'argument, l'argument doit être étiqueté lors de l'appel de la fonction.

Valeurs par défaut des paramètres

Vous pouvez définir une valeur par défaut pour n'importe quel paramètre d'une fonction en lui attribuant une valeur après le type de ce paramètre. Si une valeur par défaut est définie, vous pouvez omettre ce paramètre lors de l'appel de la fonction.

Dans l'exemple suivant, la fonction `someFunction(parametreSansDefaut:parametreAvecDefault:)` possède un paramètre `parametreAvecDefault` avec une valeur par défaut de 12 :

```
func someFunction(parametreSansDefaut: Int, parametreAvecDefaut: Int = 12) {
    // Si vous omettez le deuxième argument lors de l'appel de cette fonction, alors
    // la valeur de parameterWithDefault est de 12 à l'intérieur du corps de la
fonction.
}
```

Lors de l'appel de cette fonction, vous pouvez omettre le paramètre avec valeur par défaut :

```
someFunction(parametreSansDefaut: 3, parametreAvecDefaut: 6) // parametreAvecDefaut
est 6
```

```
someFunction(parametreSansDefaut: 4) // parametreAvecDefaut est 12
```

Placez les paramètres qui n'ont pas de valeurs par défaut au début de la liste des paramètres de la fonction, avant les paramètres qui ont des valeurs par défaut. Les paramètres qui n'ont pas de valeurs par défaut sont généralement plus importants pour la signification de la fonction - les écrire en premier facilite la reconnaissance de l'appel de la même fonction, quels que soient les paramètres par défaut omis ou non.

Paramètres variadiques

Un paramètre variadique accepte zéro ou plusieurs valeurs d'un type spécifié. Vous utilisez un paramètre variadique pour spécifier que le paramètre peut recevoir un nombre variable de valeurs en entrée lors de l'appel de la fonction. Écrivez des paramètres variadiques en insérant trois points de suspension (`...`) après le nom du type du paramètre.

Les valeurs passées à un paramètre variadique sont disponibles à l'intérieur du corps de la fonction sous la forme d'un tableau de type approprié. Par exemple, un paramètre variadique avec un nom `numbers` et un type `Double...` est disponible à l'intérieur du corps de la fonction en tant que tableau constant appelé `numbers` de type `[Double]`.

L'exemple ci-dessous calcule la moyenne arithmétique (également appelée moyenne) pour une liste de nombres de longueur variable :

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// retourne 3.0, qui est la moyenne arithmétique de ces cinq nombres
arithmeticMean(3, 8.25, 18.75)
// retourne 10.0, qui est la moyenne arithmétique de ces trois nombres
```

Une fonction peut avoir plusieurs paramètres variadiques. Le premier paramètre qui vient après un paramètre variadique doit avoir une étiquette d'argument. L'étiquette d'argument permet de distinguer de manière non ambiguë les arguments qui sont passés au paramètre variadique et les arguments qui sont passés aux paramètres qui viennent après le paramètre variadique. Voici un exemple en Swift qui illustre l'utilisation de plusieurs paramètres variadiques, ainsi que l'exigence d'étiquettes d'argument pour les paramètres suivant un paramètre variadique :

```
func imprimerNombres(_ prefixe: String, nombres: Int..., suffixe: String) {
    print(prefixe)
    for nombre in nombres {
        print(nombre)
    }
    print(suffixe)
```

```
}

imprimerNombres("Nombres :", nombres: 1, 2, 3, suffixe: "Fin")
```

Récapitulatif

L'exemple ci-dessous illustre une fonction `multiplier(::)` qui multiplie deux nombres et utilise des étiquettes d'argument personnalisées :

```
func multiplier(_ a: Int, par b: Int) -> Int {
    return a * b
}

let result = multiplier(5, par: 3) // Utilisation des étiquettes d'argument
personnalisées
print(result) // Affiche "15"
```

Dans cet exemple, la fonction `multiplier(::)` a une étiquette d'argument externe `_` pour le premier paramètre `a` et une étiquette d'argument interne `by` pour le deuxième paramètre `b`. Cela permet d'appeler la fonction sans spécifier explicitement l'étiquette d'argument pour le premier paramètre, ce qui améliore la lisibilité du code.

Les étiquettes d'argument et les noms de paramètres en Swift offrent une grande flexibilité pour rendre le code plus expressif et compréhensible.

Types de fonctions

Chaque fonction a un type de fonction spécifique, composé des types de paramètres et du type de retour de la fonction.

Par exemple :

```
func additionnerDeuxEntiers(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func multiplierDeuxEntiers(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

Cet exemple définit deux fonctions mathématiques simples appelées `additionnerDeuxEntiers` et `multiplierDeuxEntiers`. Ces fonctions prennent toutes deux deux valeurs de type `Int` et renvoient une valeur de type `Int`, qui est le résultat d'une opération mathématique appropriée.

Le type de ces deux fonctions est `(Int, Int) → Int`. Cela peut être lu comme :

"Une fonction qui prend deux valeurs de type `Int` et renvoie une valeur de type `Int`."

Voici un autre exemple pour une fonction sans paramètre ni valeur de retour :

```
func imprimerBonjourMonde() {
    print("Bonjour, monde")
}
```

Le type de cette fonction est `() → Void`, ou "une fonction qui n'a pas de paramètres et renvoie `Void`".

Utilisation des types de fonctions

Vous utilisez les types de fonctions de la même manière que tout autre type en Swift. Par exemple, vous pouvez définir une constante ou une variable avec un type de fonction approprié et lui attribuer une fonction appropriée :

```
var fonctionMathematique: (Int, Int) -> Int = additionnerDeuxEntiers
```

Cela peut être lu comme :

> Définir une variable appelée `fonctionMathematique`, qui a un type de 'une fonction qui prend deux valeurs de type `Int` et renvoie une valeur de type `Int`. Attribuer à cette nouvelle variable la référence à la fonction appelée `additionnerDeuxEntiers`.

La fonction `additionnerDeuxEntiers(::)` a le même type que la variable `fonctionMathematique`, et donc cette assignation est autorisée par le vérificateur de types de Swift.

Vous pouvez maintenant appeler la fonction assignée avec le nom `fonctionMathematique` :

```
print("Résultat : \(fonctionMathematique(2, 3))")
// Affiche "Résultat : 5"
```

Une autre fonction avec le même type correspondant peut être assignée à la même variable, de la même manière que pour les types non fonctionnels :

```
fonctionMathematique = multiplierDeuxEntiers
print("Résultat : \(fonctionMathematique(2, 3))")
// Affiche "Résultat : 6"
```

Comme pour n'importe quel autre type, vous pouvez laisser Swift déduire le type de fonction lors de l'assignation d'une fonction à une constante ou une variable :

```
let autreFonctionMathematique = additionnerDeuxEntiers
// autreFonctionMathematique est déduit comme ayant le type `(Int, Int) -> Int`
```

Types de fonctions en tant que types de paramètres

Vous pouvez utiliser un type de fonction tel que `(Int, Int) → Int` en tant que type de paramètre pour une autre fonction. Cela vous permet de passer une fonction en argument à une autre fonction.

Voici un exemple :

```
// Ceci est une fonction qui prend deux Int et retourne un Int
func multiply(_ a: Int, _ b: Int) -> Int {
    return a * b
}

// Ceci est une autre fonction qui accepte n'importe quelle fonction
// qui prend deux Int comme arguments et retourne un Int
func performOperation(_ operation: (Int, Int) -> Int, _ a: Int, _ b: Int) -> Int {
    return operation(a, b)
}

let result = performOperation(multiply, 3, 4)
print(result)  // Affiche "12"
```

Dans cet exemple, `multiply` est une fonction qui est passée en tant qu'argument à la fonction `performOperation`.

De même, vous pouvez passer des closures (qui sont des fonctions sans nom) à une fonction de la même manière. Voici un exemple :

```
let result = performOperation({ (a: Int, b: Int) -> Int in return a * b }, 3, 4)
print(result)  // Affiche "12"
```

Dans cet exemple, le premier argument passé à `performOperation` est une closure qui multiplie ses deux arguments.

Surcharge de fonctions

La surcharge de fonctions permet de définir plusieurs fonctions avec le même nom mais des listes de paramètres différentes. Cela permet d'appeler la fonction appropriée en fonction des types d'arguments fournis.

Par exemple, considérons les deux fonctions suivantes :

```
func afficherMessage(_ message: String) {
    print("Message : \(message)")
}

func afficherMessage(_ message: String, majuscules: Bool) {
```

```
    if majuscules {
        print("Message : \(message.uppercased())")
    } else {
        print("Message : \(message)")
    }
}
```

Ces deux fonctions s'appellent toutes deux `afficherMessage`, mais elles ont des listes de paramètres différentes. La première fonction prend simplement une chaîne de caractères `message` et l'affiche. La deuxième fonction prend une chaîne de caractères `message` et un booléen `majuscules`. Elle affiche la chaîne de caractères en majuscules si `majuscules` est `true`, sinon elle l'affiche telle quelle.

Lorsque vous appelez ces fonctions, Swift détermine la fonction appropriée à appeler en fonction des types d'arguments fournis :

```
afficherMessage("Bonjour")
// Affiche "Message : Bonjour"

afficherMessage("Bonjour", majuscules: true)
// Affiche "Message : BONJOUR"
```

Un autre exemple serait de surcharger les fonctions mathématiques pour gérer les différents type de nombre:

```
func afficherNombres(_ prefixe: String, nombres: Int..., suffixe: String) {
    print(prefixe)
    for nombre in nombres {
        print(nombre)
    }
    print(suffixe)
}

func afficherNombres(_ prefixe: String, nombres: Float..., suffixe: String) {
    print(prefixe)
    for nombre in nombres {
        print(nombre)
    }
    print(suffixe)
}

afficherNombres("Nombres :", nombres: 1, 2, 3, suffixe: "Fin")
afficherNombres("Nombres :", nombres: 1.3, 2.5, 3.1, suffixe: "Fin")
```

Ce code définit deux fonctions appelées afficherNombres qui prennent des listes de nombres entiers et de nombres à virgule flottante respectivement. Chaque fonction affiche les nombres avec un préfixe donné et un suffixe donné.

Les deux fonctions sont des surcharges, car elles ont le même nom mais des types de paramètres

différents. La première fonction `afficherNombres` prend une liste de nombres entiers, tandis que la deuxième fonction `afficherNombres` prend une liste de nombres à virgule flottante.

En conclusion, la surcharge de fonctions permet de fournir une interface plus flexible et intuitive, en permettant aux développeurs d'appeler une fonction avec différents ensembles de paramètres en fonction de leurs besoins.

Notez que la surcharge de fonctions est basée sur les types des paramètres, ce qui signifie que les fonctions avec les mêmes types de paramètres mais des types de retour différents ne sont pas considérées comme des surcharges. Les fonctions doivent avoir des listes de paramètres différentes pour être considérées comme des surcharges distinctes.

Gestion des erreurs

Swift offre un modèle de gestion des erreurs qui permet de gérer les erreurs de manière élégante et explicite. Contrairement à d'autres langages de programmation qui utilisent des exceptions pour gérer les erreurs, Swift utilise des valeurs d'énumération pour représenter les erreurs. Cette approche permet d'éviter les erreurs inattendues et les plantages de l'application.

Déclaration d'une fonction pouvant lancer des erreurs

Pour indiquer qu'une fonction peut lancer des erreurs, vous devez utiliser le mot-clé `throws` dans la déclaration de la fonction. Par exemple :

```
func lancerUneErreur() throws {
    // Code pouvant lancer une erreur
}
```

Dans cet exemple, la fonction `lancerUneErreur` est déclarée avec le mot-clé `throws`, ce qui signifie qu'elle peut lancer des erreurs. Le code à l'intérieur de la fonction peut contenir des instructions pouvant générer des erreurs. Lors de l'appel de cette fonction, vous devez utiliser un bloc `do-catch` pour capturer et gérer les erreurs éventuelles.

Capture et gestion des erreurs

Pour capturer et gérer les erreurs lancées par une fonction, vous utilisez un bloc `do-catch`. Le bloc `do` contient le code qui peut potentiellement lancer une erreur, tandis que le bloc `catch` capture et traite les erreurs éventuelles.

Voici un exemple d'utilisation d'un bloc `do-catch` :

```
do {
    try lancerUneErreur()
    // Code à exécuter si aucune erreur n'est lancée
} catch {
    // Code à exécuter si une erreur est capturée
}
```

Dans cet exemple, la fonction `lancerUneErreur` est appelée à l'intérieur du bloc `do`. Si aucune erreur n'est lancée, le code à l'intérieur du bloc `do` est exécuté normalement. Si une erreur est lancée, elle est capturée par le bloc `catch` et le code à l'intérieur du bloc `catch` est exécuté pour traiter l'erreur.

Représentation et lancement d'erreurs

En Swift, les erreurs sont représentées par des valeurs de types qui se conforment au protocole `Error`. Ce protocole vide indique qu'un type peut être utilisé pour la gestion des erreurs.

Les énumérations en Swift sont particulièrement adaptées pour modéliser un groupe de conditions d'erreur liées, avec des valeurs associées permettant de communiquer des informations supplémentaires sur la nature d'une erreur. Par exemple, voici comment vous pourriez représenter les conditions d'erreur lors de l'utilisation d'un distributeur automatique :

```
enum ErreurDistributeurAutomatique: Error {
    case selectionInvalide
    case fondsInsuffisants(piecesNecessaires: Int)
    case enRuptureDeStock
}
```

Lancer une erreur vous permet d'indiquer qu'un événement inattendu s'est produit et que le déroulement normal de l'exécution ne peut pas se poursuivre. Vous utilisez une instruction throw pour lancer une erreur. Par exemple, le code suivant lance une erreur pour indiquer que cinq pièces supplémentaires sont nécessaires par le distributeur automatique :

```
throw ErreurDistributeurAutomatique.fondsInsuffisants(piecesNecessaires: 5)
```

Lorsqu'une erreur est lancée, une partie du code environnante doit être responsable de la gestion de l'erreur, en corrigeant le problème, en essayant une approche alternative ou en informant l'utilisateur de l'échec.

Il existe quatre façons de gérer les erreurs en Swift. Vous pouvez:

- Propager l'erreur depuis une fonction vers le code qui appelle cette fonction,
- Gérer l'erreur à l'aide d'une instruction do-catch,
- Gérer l'erreur en tant que valeur optionnelle, ou
- Affirmer que l'erreur ne se produira pas.

Chaque approche est décrite dans une section ci-dessous.

Lorsqu'une fonction lance une erreur, cela modifie le flux de votre programme, il est donc important que vous puissiez rapidement identifier les endroits dans votre code où des erreurs peuvent être lancées. Pour identifier ces endroits dans votre code, écrivez le mot-clé `try` — ou la variation `try?` ou `try!` — avant un morceau de code qui appelle une fonction, une méthode ou un initialiseur pouvant lancer une erreur. Ces mots-clés sont décrits dans les sections ci-dessous.

Propagation des erreurs à l'aide de fonctions lançant des erreurs

Pour indiquer qu'une fonction, une méthode ou un initialiseur peut lancer une erreur, vous écrivez le mot-clé `throws` dans la déclaration de la fonction après ses paramètres. Une fonction marquée avec `throws` est appelée une fonction lançant une erreur. Si la fonction spécifie un type de retour, vous écrivez le mot-clé `throws` avant la flèche de retour (`->`).

```
func peutLancerDesErreurs() throws -> String
```

```
func nePeutPasLancerDErreurs() -> String
```

Une fonction lançant une erreur propage les erreurs qui sont lancées à l'intérieur d'elle vers la portée depuis laquelle elle est appelée.

Note: Seules les fonctions lançant des erreurs peuvent propager des erreurs. Toute erreur lancée à l'intérieur d'une fonction non lançante doit être gérée à l'intérieur de la fonction.

Dans l'exemple ci-dessous, la classe `DistributeurAutomatique` a une méthode `vendre(articleNom:)` qui lance une `ErreurDistributeurAutomatique` appropriée si l'article demandé n'est pas disponible, est en rupture de stock ou a un coût supérieur au montant actuellement déposé :

```
struct Article {
    var prix: Int
    var quantite: Int
}

class DistributeurAutomatique {
    var inventaire = [
        "Barre de chocolat": Article(prix: 12, quantite: 7),
        "Chips": Article(prix: 10, quantite: 4),
        "Bretzels": Article(prix: 7, quantite: 11)
    ]
    var montantDepose = 0

    func vendre(articleNom nom: String) throws {
        guard let article = inventaire[nom] else {
            throw ErreurDistributeurAutomatique.selectionInvalide
        }

        guard article.quantite > 0 else {
            throw ErreurDistributeurAutomatique.enRuptureDeStock
        }

        guard article.prix <= montantDepose else {
            throw ErreurDistributeurAutomatique.fondsInsuffisants(piecesNecessaires:
article.prix - montantDepose)
        }

        montantDepose -= article.prix

        var nouvelArticle = article
        nouvelArticle.quantite -= 1
        inventaire[nom] = nouvelArticle

        print("Distribution de \(nom)")
    }
}
```

La mise en œuvre de la méthode `vendre(articleNom:)` utilise des déclarations `guard` pour sortir de la méthode prématurément et lancer des erreurs appropriées si l'une des conditions d'achat d'une collation n'est pas remplie. Étant donné qu'une instruction `throw` transfère immédiatement le contrôle du programme, un article ne sera vendu que si toutes ces conditions sont remplies.

Comme la méthode `vendre(articleNom:)` propage les erreurs qu'elle lance, tout code appelant cette méthode doit soit gérer les erreurs — en utilisant une instruction `do-catch`, `try?`, ou `try!` — soit les propager. Par exemple, la fonction `acheterCollationPreferee(personne:distributeurAutomatique:)` dans l'exemple ci-dessous est également une fonction lançant une erreur, et toutes les erreurs que la méthode `vendre(articleNom:)` lance seront propagées jusqu'au point où la fonction `acheterCollationPreferee(personne:distributeurAutomatique:) `est appelée.

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Réglisse",
    "Eve": "Bretzels",
]

func acheterCollationPreferee(personne: String, distributeurAutomatique:
DistributeurAutomatique) throws {
    let nomCollation = favoriteSnacks[personne] ?? "Barre de bonbon"
    try distributeurAutomatique.vendre(articleNom: nomCollation)
}
```

Dans cet exemple, la fonction `acheterCollationPreferee(personne:distributeurAutomatique:)` recherche la collation préférée d'une personne donnée et essaie de l'acheter pour elle en appelant la méthode `vendre(articleNom:)` du distributeur automatique. Comme la méthode `vendre(articleNom:)` peut lancer une erreur, elle est appelée avec le mot-clé `try` devant.

Les initialiseurs lançant des erreurs peuvent propager les erreurs de la même manière que les fonctions lançant des erreurs. Par exemple, l'initialiseur de la structure `CollationAchetée` dans l'exemple ci-dessous appelle une fonction lançant des erreurs dans le processus d'initialisation et gère les erreurs qu'il rencontre en les propageant à son appelant.

```
struct CollationAchetée {
    let nom: String

    init(nom: String, distributeurAutomatique: DistributeurAutomatique) throws {
        try distributeurAutomatique.vendre(articleNom: nom)
        self.nom = nom
    }
}
```

Gestion des erreurs avec do-catch

Vous utilisez une instruction `do-catch` pour gérer les erreurs en exécutant un bloc de code. Si une erreur est lancée par le code dans la clause `do`, elle est comparée avec les clauses catch pour

déterminer celle qui peut gérer l'erreur.

Voici la forme générale d'une instruction `do-catch` :

```
do {
    try <#expression#>
    <#statements#>
} catch <#pattern 1#> {
    <#statements#>
} catch <#pattern 2#> where <#condition#> {
    <#statements#>
} catch <#pattern 3#>, <#pattern 4#> where <#condition#> {
    <#statements#>
} catch {
    <#statements#>
}
```

Vous écrivez un motif après `catch` pour indiquer quelles erreurs cette clause peut gérer. Si une clause `catch` n'a pas de motif, elle correspond à n'importe quelle erreur et lie l'erreur à une constante locale nommée error. Pour plus d'informations sur la correspondance de motifs, consultez les sections correspondantes dans la documentation.

Par exemple, le code suivant gère les trois cas de l'énumération `ErreurDistributeurAutomatique`:

```
var distributeurAutomatique = DistributeurAutomatique()
distributeurAutomatique.piecesDeposees = 8

do {
    try acheterCollationPreferee(personne: "Alice", distributeurAutomatique:
distributeurAutomatique)
    print("Succès ! Miam.")
} catch ErreurDistributeurAutomatique.selectionInvalide {
    print("Sélection invalide.")
} catch ErreurDistributeurAutomatique.enRuptureDeStock {
    print("Rupture de stock.")
} catch ErreurDistributeurAutomatique.fondsInsuffisants(let piecesNecessaires) {
    print("Fonds insuffisants. Veuillez insérer \(piecesNecessaires) pièces
supplémentaires.")
} catch {
    print("Erreur inattendue : \(error).")
}
// Affiche "Fonds insuffisants. Veuillez insérer 2 pièces supplémentaires."
```

Dans l'exemple ci-dessus, la fonction `acheterCollationPreferee(personne:distributeurAutomatique:)` est appelée dans une expression `try`, car elle peut lancer une erreur. Si une erreur est lancée, l'exécution est immédiatement transférée aux clauses `catch`, qui décident si la propagation doit se poursuivre. Si aucun motif n'est correspondant, l'erreur est attrapée par la clause `catch` finale et liée à une constante locale `error`. Si aucune erreur n'est lancée, les autres instructions dans la

clause `do` sont exécutées.

Les clauses `catch` ne doivent pas nécessairement gérer toutes les erreurs possibles que le code dans la clause `do` peut lancer. Si aucune des clauses `catch` ne gère l'erreur, l'erreur se propage à la portée environnante. Cependant, l'erreur propagée doit être gérée par une portée environnante. Dans une fonction non lançante, une instruction `do-catch` englobante doit gérer l'erreur. Dans une fonction lançante: soit une instruction `do-catch` englobante, soit l'appelant doit gérer l'erreur. Si l'erreur se propage au niveau de portée le plus élevé sans être gérée, une erreur d'exécution se produira.

Pour exemple, l'exemple ci-dessus peut être écrit de manière à ce que toute erreur qui n'est pas une `VendingMachineError` soit capturée par la fonction appelante :

```
func nourrir(avec item: String) throws {
    do {
        try vendingMachine.vendre(articleNom: item)
    } catch is VendingMachineError {
        print("Impossible d'acheter cela depuis le distributeur automatique.")
    }
}

do {
    try nourrir(avec: "Chips à la betterave")
} catch {
    print("Erreur inattendue non liée au distributeur automatique : \(error)")
}
// Affiche "Impossible d'acheter cela depuis le distributeur automatique."
```

Dans la fonction `nourrir(avec:)`, si `vendre(articleNom:)` lance une erreur qui correspond à l'une des valeurs de l'énumération `VendingMachineError`, `nourrir(avec:)` gère l'erreur en affichant un message. Sinon, `nourrir(avec:)` propage l'erreur à son emplacement d'appel. L'erreur est ensuite capturée par la clause `catch` générale.

Une autre façon de capturer plusieurs erreurs liées consiste à les lister après le mot-clé `catch`, séparées par des virgules. Par exemple :

```
func manger(article: String) throws {
    do {
        try vendingMachine.vendre(articleNom: article)
    } catch VendingMachineError.invalidSelection, VendingMachineError
.insufficientFunds, VendingMachineError.outOfStock {
        print("Sélection invalide, rupture de stock ou fonds insuffisants.")
    }
}
```

La fonction `manger(article:)` liste les erreurs du distributeur automatique à capturer, et son texte d'erreur correspond aux éléments de cette liste. Si l'une des trois erreurs listées est lancée, cette clause `catch` les gère en affichant un message. Toutes les autres erreurs sont propagées au bloc entourant, y compris les erreurs du distributeur automatique qui pourraient être ajoutées

ultérieurement.

Conversion des erreurs en valeurs optionnelles

On utilise `try?` pour gérer une erreur en la convertissant en une valeur optionnelle. Si une erreur est lancée lors de l'évaluation de l'expression `try?`, la valeur de l'expression est `nil`. Par exemple, dans le code suivant, `x` et `y` ont la même valeur et le même comportement :

```
func uneFonctionLancantUneErreur() throws -> Int {
    // ...
}

let x = try? uneFonctionLancantUneErreur()

let y: Int?
do {
    y = try uneFonctionLancantUneErreur()
} catch {
    y = nil
}
```

Si `uneFonctionLancantUneErreur()` lance une erreur, la valeur de `x` et `y` est `nil`. Sinon, la valeur de `x` et `y` est la valeur que la fonction a retournée. Notez que `x` et `y` sont optionnels du type retourné par `uneFonctionLancantUneErreur()`. Ici, la fonction retourne un entier, donc `x` et `y` sont des entiers optionnels.

L'utilisation de `try?` vous permet d'écrire un code de gestion des erreurs concis lorsque vous souhaitez traiter toutes les erreurs de la même manière. Par exemple, le code suivant utilise plusieurs approches pour récupérer des données, ou renvoie `nil` si toutes les approches échouent.

```
func récupérerDonnées() -> Data? {
    if let données = try? récupérerDonnéesDepuisDisque() { return données }
    if let données = try? récupérerDonnéesDepuisServeur() { return données }
    return nil
}
```

Désactivation de la propagation des erreurs

Parfois, vous savez qu'une fonction ou une méthode lancée ne lancera en réalité aucune erreur à l'exécution. Dans ces cas-là, vous pouvez écrire `try!` avant l'expression pour désactiver la propagation des erreurs et envelopper l'appel dans une assertion à l'exécution selon laquelle aucune erreur ne sera lancée. Si une erreur est effectivement lancée, vous obtiendrez une erreur à l'exécution.

Par exemple, le code suivant utilise une fonction `loadImage(atPath:)`, qui charge la ressource image à partir d'un chemin donné ou lance une erreur si l'image ne peut pas être chargée. Dans ce cas, étant donné que l'image est fournie avec l'application, aucune erreur ne sera lancée à l'exécution, il

est donc approprié de désactiver la propagation des erreurs.

```
let photo = try! loadImage(atPath: "./Resources/John_Appleseed.jpg")
```

Programmation orientée objet

Introduction à la POO

La Programmation Orientée Objet (POO) est un paradigme de programmation qui traite des concepts de "objets" et de leurs interactions. Il est utilisé pour structurer un logiciel autour de données, ou objets, et de données associées à ces objets.

Définition et principes fondamentaux

La POO est basée sur plusieurs principes fondamentaux, dont l'encapsulation, l'héritage, le polymorphisme et l'abstraction.

```
class Animal {
    var nom: String
    init(nom: String) {
        self.nom = nom
    }
    func description() -> String {
        return "Un animal nommé \(nom)"
    }
}
```

Dans cet exemple, `Animal` est une classe qui représente un concept général d'animal. L'objet `Animal` a une propriété `nom` et une méthode `description`.

Comparaison avec la programmation procédurale

La programmation procédurale est un autre paradigme de programmation où le logiciel est organisé autour de procédures ou fonctions. À la différence de la POO, les données et les fonctions sont séparées et non regroupées en objets.

Avantages et utilisation de la POO

La POO offre plusieurs avantages, dont la modularité, la réutilisabilité et une structure facile à comprendre et à maintenir. Elle est largement utilisée dans le développement de logiciels et est soutenue par de nombreux langages de programmation modernes, dont Swift.

Classes et Objets en Swift

Swift supporte la Programmation Orientée Objet (POO), qui est un paradigme de programmation qui permet de structurer le logiciel autour d'objets.

Définition d'une classe et création d'objets

```
class Animal {
```

```
    var nom: String
    init(nom: String) {
        self.nom = nom
    }
}

let monAnimal = Animal(nom: "Bella")
```

Dans cet exemple, `Animal` est une classe et `monAnimal` est un objet (ou une instance) de la classe `Animal`.

Propriétés et méthodes

```
class Animal {
    var nom: String // Propriété

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

    func saluer() { // Méthode
        print("Bonjour, je suis \(nom)!")
    }
}

let monAnimal = Animal(nom: "Bella")
monAnimal.saluer()
```

Ici, `nom` est une propriété de la classe `Animal` et `saluer` est une méthode.

Initialisation et déinitialisation

```
class Animal {
    var nom: String

    init(nom: String) { // Initialisateur
        self.nom = nom
    }

    deinit { // Désinitialisateur
        print("\(nom) a été détruit.")
    }
}

do {
    let monAnimal = Animal(nom: "Bella")
} // En sortant de bloc, on désalloue l'instance
```

Dans cet exemple, l'initialiseur est utilisé pour créer une nouvelle instance de `Animal` et le désinitialiseur est appelé lorsque l'instance est désallouée.

Passage par référence

```
let animal1 = Animal(nom: "Bella")
let animal2 = animal1
animal2.nom = "Lucy"

print(animal1.nom) // Imprime "Lucy"
```

Swift traite les objets comme des références. Dans cet exemple, lorsque nous modifions le `nom` d'`animal2`, cela modifie également le `nom` d'`animal1` car ils référencent le même objet.

Structures en Swift

En plus des classes, Swift propose également des structures, qui sont similaires aux classes mais ont quelques différences clés.

Définition d'une strucutre

```
struct Animal {
    var nom: String
}
let monAnimal = Animal(nom: "Bella")
```

Dans cet exemple, `Animal` est une structure et `monAnimal` est une instance de la structure `Animal`.

Propriétés et méthodes

```
struct Animal {
    var nom: String // Propriété

    func saluer() { // Méthode
        print("Bonjour, je suis \(nom)!")
    }
}
let monAnimal = Animal(nom: "Bella")
monAnimal.saluer()
```

Ici, `nom` est une propriété de la structure `Animal` et `saluer` est une méthode.

Initialisation et déinitialisation

```
struct Animal {
```

```
    var nom: String

    init(nom: String) { // Initialisateur
        self.nom = nom
    }
}
let monAnimal = Animal(nom: "Bella")
```

Dans cet exemple, l'initialiseur est utilisé pour créer une nouvelle instance de `Animal`. Contrairement aux classes, les structures n'ont pas de désinitialiseur.

Passage par référence

```
var animal1 = Animal(nom: "Bella")
var animal2 = animal1
animal2.nom = "Lucy"

print(animal1.nom) // Imprime "Bella"
```

Contrairement aux classes, les structures sont passées par valeur, pas par référence. Donc, dans cet exemple, lorsque nous modifions le `nom` d'`animal2`, cela n'affecte pas le `nom` d'`animal1`.

Énumérations en Swift

Les énumérations, ou "enums", sont un moyen de grouper un ensemble de valeurs connexes. Les énumérations en Swift sont extrêmement flexibles et ne se limitent pas à une liste de valeurs de type Integer.

Définition et utilisation d'une énumération

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

let maDirection = Direction.nord
```

Dans cet exemple, `Direction` est une énumération qui a quatre cas possibles. Nous pouvons assigner l'un de ces cas à une variable.

Types bruts

Il est possible de donner une valeur aux cas de l'énumération.

```
enum Direction: String {
    case nord = "Nord"
    case sud = "Sud"
    case est = "Est"
    case ouest = "Ouest"
}

let maDirection = Direction.nord
print(maDirection.rawValue)  // Affiche "Nord"
```

Dans cet exemple, nous définissons une énumération `Direction` qui représente les quatre points cardinaux : nord, sud, est et ouest. Cette énumération est déclarée avec un type brut `String`, ce qui signifie que chaque cas de l'énumération a une valeur associée de type `String`.

Chaque cas de l'énumération est explicitement associé à une valeur brute. Par exemple, le cas `nord` est associé à la chaîne de caractères "Nord". De même, les autres cas sont associés à leurs respectives chaînes de caractères.

Une fois que vous avez déclaré une variable (dans cet exemple, `maDirection`) pour représenter un cas particulier de l'énumération (ici, `Direction.nord`), vous pouvez utiliser la propriété `.rawValue` pour accéder à la valeur brute associée à ce cas. Dans cet exemple, `maDirection.rawValue` renvoie la chaîne "Nord".

Cela permet une plus grande flexibilité dans l'utilisation des énumérations, car vous pouvez associer des informations supplémentaires à chaque cas. Les valeurs brutes doivent être uniques à chaque cas et peuvent être de n'importe quel type qui est conforme au protocole `RawRepresentable`, bien que les types les plus couramment utilisés soient `Int` et `String`.

Swift est aussi capable de déduire les valeurs des types bruts:

```
enum JourDeLaSemaine: Int {
    case lundi = 2
    case mardi, mercredi, jeudi, vendredi, samedi, dimanche
}

let premierJour = JourDeLaSemaine.lundi
print(premierJour.rawValue)  // Affiche "2"

let deuxiemeJour = JourDeLaSemaine.mardi
print(deuxiemeJour.rawValue)  // Affiche "3"
```

Dans cet exemple, `JourDeLaSemaine` est une énumération qui représente les jours de la semaine. Chaque cas de l'énumération a une valeur associée de type `Int`. Le premier jour `lundi` a une valeur brute de `2`, puis chaque jour suivant a une valeur brute qui est incrémentée de un par rapport au jour précédent. Par exemple, `mardi` a une valeur brute de `3`, `mercredi` a une valeur brute de `4`, et ainsi de suite.

Quand vous déclarez une variable (dans cet exemple, `premierJour` et `deuxiemeJour`) pour représenter

un cas particulier de l'énumération (ici, `JourDeLaSemaine.lundi` et `JourDeLaSemaine.mardi`), vous pouvez utiliser la propriété `.rawValue` pour accéder à la valeur brute associée à ce cas. Par exemple, `premierJour.rawValue` renvoie l'entier `2` et `deuxiemeJour.rawValue` renvoie l'entier `3`.

Utilisation d'un switch avec une énumération

```
switch maDirection {
case .nord:
    print("Nous allons vers le nord")
case .sud:
    print("Nous allons vers le sud")
case .est:
    print("Nous allons vers l'est")
case .ouest:
    print("Nous allons vers l'ouest")
}
```

Avec une énumération, nous pouvons utiliser une instruction switch pour effectuer différentes actions en fonction du cas de l'énumération.

Utilisation d'une énumération dans une classe

```
class Voiture {
    var direction: Direction

    init(direction: Direction) {
        self.direction = direction
    }

    func changerDirection(nouvelleDirection: Direction) {
        direction = nouvelleDirection
    }
}

let maVoiture = Voiture(direction: .nord)
maVoiture.changerDirection(nouvelleDirection: .sud)
```

Dans cet exemple, la classe `Voiture` utilise l'énumération `Direction` pour représenter sa direction actuelle. Nous pouvons également changer la direction en utilisant la méthode `changerDirection`.

Encapsulation

L'encapsulation est un principe fondamental de la programmation orientée objet. Il permet de cacher les détails d'implémentation d'un objet et d'exposer uniquement ce qui est nécessaire. Cela protège les données et rend le code plus facile à utiliser et à gérer.

Getters et setters

En Swift, vous pouvez définir des getters et des setters pour les propriétés d'une classe ou d'une structure. Voici un exemple :

```
class Cercle {
    private var _rayon: Double

    var rayon: Double {
        get {
            return _rayon
        }
        set(nouveauRayon) {
            _rayon = nouveauRayon < 0 ? 0 : nouveauRayon
        }
    }

    init(rayon: Double) {
        self._rayon = rayon
    }
}
```

Dans cet exemple, `_rayon` est une propriété privée. `rayon` est une propriété publique avec un getter et un setter. Le setter s'assure que le `rayon` ne peut jamais être négatif.

Propriétés calculées

Une propriété calculée est une propriété dont la valeur est dérivée d'autres propriétés. Swift permet de créer des propriétés calculées avec des getters et des setters.

```
class Cercle {
    // ...

    var aire: Double {
        return 3.14 * rayon * rayon
    }
}
```

Dans cet exemple, `aire` est une propriété calculée qui dépend du `rayon` du cercle.

Modificateurs d'accès : public, internal, fileprivate et private

Les modificateurs d'accès en Swift contrôlent la portée d'une entité (c'est-à-dire une classe, une structure, une énumération, ou une propriété). Ils définissent le contexte dans lequel une entité peut être accédée.

```
public class ClassePublique {
```

```
    public var proprietePublique = "Je suis publique"
    private var proprietePrivee = "Je suis privée"
    fileprivate var proprietePriveeFichier = "Je suis privée au fichier"
    internal var proprieteInterne = "Je suis interne"
}
```

Dans cet exemple, nous avons quatre modificateurs d'accès.

- `public` : Les entités marquées comme publiques peuvent être accédées de n'importe où, y compris en dehors du module (c'est-à-dire du paquet ou de la bibliothèque) dans lequel elles sont définies. Ceci est utile pour les API qui doivent être accessibles à d'autres modules.
- `internal` : **C'est le modificateur d'accès par défaut.** Les entités marquées comme internes peuvent être accédées n'importe où dans le même module, mais pas en dehors. C'est généralement utilisé pour les détails d'implémentation qui doivent être cachés de l'extérieur du module.
- `fileprivate` : Les entités marquées comme `fileprivate` peuvent seulement être accédées à l'intérieur du fichier source dans lequel elles sont définies. C'est utile pour cacher les détails d'implémentation qui sont partagés entre plusieurs entités dans le même fichier.
- `private` : Les entités marquées comme privées peuvent seulement être accédées dans leur propre corps et dans les extensions de leur définition dans le même fichier source. C'est le niveau d'accès le plus restrictif et c'est utile pour cacher les détails d'implémentation à toutes les autres entités.

Notez que Swift n'a pas de modificateur `protected` comme d'autres langages de programmation orientée objet tels que Java. En Swift, une entité ne peut pas être accessible uniquement par une sous-classe.

Héritage

L'héritage est un principe fondamental de la programmation orientée objet. C'est un mécanisme par lequel une nouvelle classe peut être dérivée d'une classe existante.

Définition et utilité de l'héritage

L'héritage permet à une classe de gagner (ou d'"hériter") les propriétés et méthodes d'une autre classe. Cela permet de créer des hiérarchies de classes où les sous-classes partagent un ensemble commun de fonctionnalités avec leur classe parente, tout en étant capable de définir leur propre comportement et leurs propres propriétés uniques.

La classe qui est héritée est appelée la "classe parente" ou "superclasse", et la classe qui hérite est appelée la "sous-classe".

```
class Vehicule {   // classe parente ou superclasse
    var vitesseMax = 100
}

class Voiture: Vehicule {  // sous-classe
```

```
    var nombreDePortes = 4
}
```

Syntaxe d'héritage en Swift

En Swift, l'héritage est défini en utilisant le caractère deux points `:`. La nouvelle classe est définie normalement, mais suivie de `:`, puis du nom de la classe parente.

```
class Vehicule {
    var vitesseMax = 100
}

class Voiture: Vehicule {
    var nombreDePortes = 4
}
```

Surcharge de méthodes

La surcharge de méthode est un concept qui permet à une sous-classe de fournir une implémentation spécifique d'une méthode qui est déjà fournie par sa classe parente.

```
class Vehicule {
    var vitesseMax = 100

    func afficherVitesse() {
        print("La vitesse maximale est \(vitesseMax) km/h.")
    }
}

class Voiture: Vehicule {
    var nombreDePortes = 4

    override func afficherVitesse() {
        print("La vitesse maximale de cette voiture est \(vitesseMax) km/h.")
    }
}
```

Dans cet exemple, la classe `Voiture` surcharge la méthode `afficherVitesse()` de la classe `Vehicule` pour fournir un message plus spécifique.

Héritage de classes, de propriétés et de méthodes

En Swift, lorsqu'une classe est dérivée d'une superclasse, elle hérite de toutes les propriétés et méthodes de la superclasse. Elle peut alors ajouter ses propres propriétés et méthodes ou surcharger les méthodes existantes de la superclasse pour changer leur comportement.

Notez que Swift n'autorise pas l'héritage multiple (une classe ne peut pas hériter de plus d'une

superclasse). Cependant, Swift offre une autre fonctionnalité appelée protocoles pour fournir une partie de la fonctionnalité de l'héritage multiple.

Polymorphisme

Le polymorphisme est un autre concept clé de la programmation orientée objet. Il désigne la capacité d'une entité de prendre plusieurs formes. En Swift, le polymorphisme se manifeste lorsque une classe hérite d'une autre.

Polymorphisme statique et dynamique

Le polymorphisme statique, aussi appelé polymorphisme en temps de compilation, se produit lorsque la méthode à exécuter est déterminée à la compilation. En Swift, cela se produit généralement par le biais de la surcharge de méthodes.

Le polymorphisme dynamique, aussi appelé polymorphisme en temps d'exécution, se produit lorsque la méthode à exécuter est déterminée en temps d'exécution. En Swift, cela se produit généralement par le biais de la substitution de méthodes.

Surcharge et substitution

La surcharge de méthodes est une forme de polymorphisme statique qui permet à plusieurs méthodes de partager le même nom, mais de différer par le nombre ou le type de leurs paramètres. En Swift, il est possible de surcharger des méthodes ainsi que des initialisateurs de classe.

Voici un exemple de surcharge de méthodes en Swift :

```
class Calculatrice {
    func additionner(_ a: Int, _ b: Int) -> Int {
        return a + b
    }

    func additionner(_ a: Double, _ b: Double) -> Double {
        return a + b
    }
}
```

Dans cet exemple, la classe `Calculatrice` possède deux méthodes `additionner`. La première méthode prend deux paramètres de type `Int`, tandis que la deuxième méthode prend deux paramètres de type `Double`. Cette surcharge permet d'appeler la méthode `additionner` avec des paramètres de types différents.

La substitution de méthodes, en revanche, est une forme de polymorphisme dynamique qui permet à une sous-classe de fournir une implémentation spécifique d'une méthode qui est déjà fournie par sa superclasse. La méthode dans la sous-classe qui surcharge la méthode de la superclasse doit être précédée du mot-clé `override`.

Reprenons notre exemple précédent avec les classes `Vehicule` et `Voiture` :

```
class Vehicule {
    func afficherVitesse() {
        print("La vitesse du véhicule est inconnue.")
    }
}

class Voiture: Vehicule {
    var vitesseMax = 100

    override func afficherVitesse() {
        print("La vitesse maximale de cette voiture est \(vitesseMax) km/h.")
    }
}
```

Ici, la classe `Voiture` est une sous-classe de `Vehicule` et surcharge la méthode `afficherVitesse()`. Dans la classe `Voiture`, cette méthode utilise la propriété `vitesseMax` pour afficher une information plus précise sur la vitesse du véhicule.

Utilisation de `super`

Le mot-clé `super` est utilisé pour appeler une méthode de la superclasse. Cela est particulièrement utile lorsque vous voulez augmenter la fonctionnalité d'une méthode héritée plutôt que de la remplacer complètement.

```
class Vehicule {
    var vitesseMax = 100

    func afficherVitesse() {
        print("La vitesse maximale est \(vitesseMax) km/h.")
    }
}

class Voiture: Vehicule {
    var nombreDePortes = 4

    override func afficherVitesse() {
        super.afficherVitesse()  // Appelle la méthode de la superclasse
        print("Et cette voiture a \(nombreDePortes) portes.")
    }
}
```

Dans cet exemple, la méthode `afficherVitesse()` de la classe `Voiture` appelle d'abord la méthode `afficherVitesse()` de sa superclasse, puis ajoute une information supplémentaire.

Abstraction

L'abstraction est un concept clé de la programmation orientée objet. C'est le processus de cacher les

détails d'implémentation et de montrer seulement la fonctionnalité à l'utilisateur. En d'autres termes, l'abstraction permet de se concentrer sur "quoi" fait l'objet plutôt que "comment" il le fait.

Définition et utilité de l'abstraction

L'abstraction permet de réduire la complexité en cachant les détails inutiles, ce qui permet de concevoir des systèmes plus simples et plus faciles à comprendre. Par exemple, lorsque vous utilisez un téléphone, vous n'avez pas besoin de comprendre comment les pixels s'affichent, ou comme le microphone convertit le son en données digitales, pour pouvoir l'utiliser.

Protocoles (similaire à des interfaces dans d'autres langages)

En Swift, on utilise des protocoles pour définir une interface commune pour des classes, structures ou énumérations. Un protocole peut définir des méthodes et des propriétés qui doivent être implémentées par toute entité qui adopte ce protocole.

```
protocol Volant {
    var aDesAiles: Bool { get set }
    func voler()
}
```

Dans cet exemple, toute entité qui adopte le protocole `Volant` doit avoir une propriété `aDesAiles` et une méthode `voler()`.

Implémentation d'interfaces (protocoles)

Une classe, une structure ou une énumération peut adopter un ou plusieurs protocoles et doit fournir une implémentation pour toutes les méthodes et propriétés définies dans ces protocoles.

```
class Oiseau: Volant {
    var aDesAiles: Bool = true

    func voler() {
        print("L'oiseau vole dans le ciel.")
    }
}
```

Dans cet exemple, la classe `Oiseau` adopte le protocole `Volant` et fournit une implémentation pour la propriété `aDesAiles` et la méthode `voler()`.

Génériques en Swift

Les génériques sont un moyen de créer du code flexible et réutilisable qui peut fonctionner avec n'importe quel type. Ils vous permettent de créer des fonctions, des méthodes, des classes, des structures et des énumérations qui peuvent travailler avec n'importe quel type, tout en maintenant la sécurité du type.

Introduction aux types génériques

Les types génériques sont une façon d'écrire du code flexible et réutilisable qui peut interagir avec différents types de manière sécurisée. Ils sont souvent utilisés dans la création de structures de données.

Par exemple, Swift a un Array générique qui peut être utilisé pour créer un tableau de n'importe quel type :

```
var tableauDeStrings = Array<String>()
tableauDeStrings.append("Bonjour")
```

Création de fonctions, méthodes et classes génériques

Vous pouvez également créer vos propres fonctions génériques, méthodes et classes. Pour créer une fonction générique, vous utilisez une "balise de type générique" (une balise entre chevrons) après le nom de la fonction.

Voici un exemple de fonction générique qui peut échanger les valeurs de deux variables de n'importe quel type :

```
func echanger<T>(a: inout T, b: inout T) {
    let temporaire = a
    a = b
    b = temporaire
}

var un = 1
var deux = 2
echanger(a: &un, b: &deux)
```

Dans cet exemple, `<T>` est la balise de type générique. Cette fonction utilise `T` comme type de ses deux paramètres et peut donc être utilisée avec n'importe quel type.

De la même manière, vous pouvez créer des classes, structures et énumérations génériques en utilisant une balise de type générique après le nom de la classe, structure ou énumération.

```
class Pile<T> {
    var elements = [T]()

    func ajouter(_ element: T) {
        elements.append(element)
    }

    func retirer() -> T? {
        return elements.popLast()
    }
}
```

```
var pileDeNombres = Pile<Int>()
pileDeNombres.ajouter(5)
pileDeNombres.ajouter(6)
```

Dans cet exemple, `Pile<T>` est une classe générique qui peut être utilisée pour créer une pile de n'importe quel type. `T` est utilisé comme type des éléments dans la pile.

Exercices

Fonctions

Exercice 1 : Calcul de la somme

Écrivez une fonction nommée `calculerSomme` qui prend deux paramètres entiers `a` et `b`, et retourne la somme de ces deux nombres.

Exercice 2 : Vérification de la parité

Écrivez une fonction nommée `estPair` qui prend un paramètre entier `nombre` et retourne `true` si le nombre est pair, et `false` sinon.

Exercice 3 : Conversion de Celsius en Fahrenheit

Écrivez une fonction nommée `convertirEnFahrenheit` qui prend un paramètre de type `Double` représentant une température en degrés Celsius, et retourne la valeur équivalente en degrés Fahrenheit.

Exercice 4 : Calcul de la factorielle

Écrivez une fonction nommée `calculerFactorielle` qui prend un paramètre entier `n` et retourne la factorielle de ce nombre.

Exercice 5 : Vérification de la présence d'un élément dans un tableau

Écrivez une fonction nommée `verifierPresence` qui prend un paramètre de type `Int` nommé `element` et un tableau d'entiers nommé `tableau`, et retourne `true` si l'élément est présent dans le tableau, et `false` sinon.

Gestion des erreurs

Exercice 1 : Validation d'un code PIN

Écrivez une fonction nommée `validerCodePIN` qui prend un paramètre de type `String` représentant un code PIN à quatre chiffres. La fonction doit vérifier si le code PIN est valide en appliquant les règles suivantes : - Le code PIN doit avoir exactement quatre caractères. - Tous les caractères du code PIN doivent être des chiffres de 0 à 9. Si le code PIN est valide, la fonction doit renvoyer `true`, sinon elle doit renvoyer `false`.

Exercice 2 : Conversion de chaîne en entier

Écrivez une fonction nommée `convertirEnEntier` qui prend un paramètre de type `String` représentant un nombre entier. La fonction doit convertir la chaîne en un entier et le renvoyer. Si la conversion échoue, la fonction doit renvoyer `nil`.

Exercice 3 : Division sécurisée

Écrivez une fonction nommée `diviserSécurisée` qui prend deux paramètres de type `Double`, `numérateur` et `dénominateur`, et effectue la division entre eux. La fonction doit gérer l'erreur lorsque le `dénominateur` est égal à zéro en lançant une erreur de type `ErreurDivisionParZero`. Cette erreur devrait être définie comme une énumération avec un cas unique.

Exercice 4 : Recherche d'un élément dans un tableau

Écrivez une fonction nommée `rechercherElement` qui prend un paramètre de type `Int` nommé `élément` et un tableau d'entiers nommé `tableau`. La fonction doit rechercher la première occurrence de l'élément dans le tableau et renvoyer son index. Si l'élément n'est pas trouvé, la fonction doit lancer une erreur de type `ElementNonTrouvé`.

Exercice 5 : Traitement d'un fichier CSV

Imaginez que vous devez écrire une fonction qui lit un fichier CSV et effectue un traitement sur les données. Écrivez le prototype de cette fonction en précisant les paramètres nécessaires et le type de valeur renvoyée. Indiquez également les erreurs potentielles qui peuvent se produire lors de la lecture du fichier.

POO

Classes

Exercice 1

Créez une classe `Personne` avec des propriétés `nom`, `prenom` et `age`. Ajoutez une méthode `sePresenter()` qui affiche une courte description de la personne.

Exercice 2

Créez une classe `Voiture` avec des propriétés `marque`, `modele`, `odometre`, et `annee`. Ajoutez une méthode `demarrer()` qui affiche un message indiquant que la voiture démarre. Ajoutez une fonction `avancer(de kilometres: Int)` qui incremente la valeur d'odomètre *si* la voiture est démarrée.

Exercice 3

Définissez une classe `Cercle` avec une propriété `rayon`. Ajoutez deux méthodes pour calculer et retourner le périmètre et l'aire du cercle.

Structures

Exercice 4

Définissez une structure `Point` qui représente un point dans un système de coordonnées 3D. Ajoutez une méthode `distanceJusqua(autrePoint: Point)` pour calculer la distance entre deux points.

Exercice 5

Créez une structure `Rectangle` qui a deux propriétés `longueur` et `largeur`. Ajoutez des méthodes pour calculer l'aire et le périmètre du rectangle. Ajoutez une méthode pour comparer l'instance à une autre instance et retourner si son aire est plus petite ou non.

Gérer les erreurs si `longueur` ou `largeur` est négatif.

Intancier 5 rectangles et faire 5 comparaisons (de votre choix).

Enums

Exercice 6

Définissez une énumération `Jour` qui représente les jours de la semaine.

Exercice 7

Définissez une énumération `Note` pour représenter les notes d'un étudiant. Chaque cas doit avoir une valeur brute de type `Int`.

Intégration

Exercice 8

Créez une classe `Eleve` qui hérite de la classe `Personne` et ajoute une propriété `note`. Ajoutez une méthode `passerExamen()` qui détermine si l'élève a réussi l'examen en fonction de sa note.

Exercice 9

Créez une structure `Vecteur` qui représente un vecteur dans un système de coordonnées 3D. Ajoutez des méthodes pour calculer la longueur du vecteur et une autre pour additionner deux vecteurs.

Exercice 10

Créez une classe `Livre` qui a des propriétés pour le `titre`, l'`auteur` et le `nombreDePages`. Créez une classe `Bibliothèque` qui a une propriété pour une `collectionDeLivres`, qui est une liste de `Livre`s. Ajoutez une méthode à `Bibliothèque` pour `ajouterLivre(livre: Livre)`, qui ajoute un livre à la collection.

Exercice 11

Définissez une structure `Point` qui représente un point dans un système de coordonnées 2D avec des propriétés `x` et `y`. Créez une structure `Ligne` qui a deux propriétés de type `Point`, `start` et `end`. Ajoutez une méthode à `Ligne` pour calculer la longueur de la ligne.

Exercice 12

Créez une classe `CompteBancaire` avec des propriétés pour le `solde` et le `titulaire`. Ajoutez des méthodes pour `déposer(montant: Double)` et `retirer(montant: Double)`. Assurez-vous que le solde ne peut pas descendre en dessous de 0.

Exercice 13

Définissez une énumération `TypeDeVoiture` avec des cas pour différents types de voitures (par exemple, `berline`, `SUV`, `cabriolet`). Chaque cas doit avoir une valeur associée qui est une instance de la classe `Voiture` correspondante.