Table des matières

Introduction

Chapitre 1
Premier programme Kotlin

Installation de Kotlin

Pour pouvoir compiler et exécuter des programmes écrits en Kotlin, vous aurez besoin d'un compilateur. Pour ce faire, récupérez la dernière version du compilateur Kotlin à l'adresse suivante https://github.com/JetBrains/kotlin/releases/latest.

Une fois que vous aurez téléchargé le fichier .zip, extrayez-le dans un emplacement de votre choix. Il vous suffira ensuite d'ajouter l'emplacement du dossier `bin` qu'il contient, à la variable d'environnement `PATH` de votre système. La procédure dépend de votre système d'exploitation.

Windows

- Dans le panneau de configuration, choisissez *Afficher par : Petites icônes* en haut à droite ;
- Cliquez sur *Système* ;
- À gauche, cliquez sur *Paramètres Systèmes Avancés* ;
- Dans la boîte de dialogue, cliquez sur le bouton `Variables d'environnement…` ;
- Dans le cadre du bas, cliquez sur la ligne dont la valeur pour la colonne Variable vaut `Path` ;
- Cliquez sur le bouton `Modifier…` ;
- Ajoutez le chemin vers le dossier `bin` dans une nouvelle ligne ;
- Cliquez sur `OK` afin de fermer chacune des trois boîtes de dialogue.

macOS

- Ouvrez un terminal ;
- Exécutez la commande `sudo nano /etc/paths` ;
- Ajoutez le chemin vers votre dossier `bin` sur une nouvelle ligne ;
- Tapez `Ctrl+O` pour enregistrer, puis `Ctrl+X` pour quitter.

Linux

La procédure dépend de votre distribution. Mais si vous êtes sous Linux, je suppose que vous savez modifier la variable d'environnement `PATH` vous-même.

Hello, World!

Code source

Un programme **Hello, World!** est un programme basique, ce qui se fait de plus élémentaire dans chaque langage de programmation. Il s'agit simplement d'afficher à l'écran la chaine de caractères `Hello, World!`. Son objectif est de fournir une première approche minimaliste du langage étudié.

Ouvrez un éditeur de texte (comme l'excellent Atom par exemple), et écrivez le contenu suivant, que vous sauvegarderez dans un fichier nommé `HelloWorld.kt`.

HelloWorld.kt

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

Du code source à l'exécution

Pour exécuter un programme en Kotlin, nous devons passer par trois étapes.

L'écriture du code source

C'est ce que nous venons de faire. Il s'agit d'écrire un programme dans un langage lisible par un développeur. Ce que vous avez écrit, n'importe quel développeur est capable de le lire et de le modifier assez facilement pour modifier le comportement du programme.

La compilation

Il s'agit d'une étape qui vise à transformer votre code source en un fichier compréhensible par la machine. Pour certains langages, comme le PHP ou le JavaScript, il n'est pas nécessaire de compiler le code source. Le système va être capable de lire et d'interpréter directement votre programme. Toutefois, l'inconvénient de ce système est qu'il présente des lacunes en termes de performances.

En effet, en tant qu'humains, nous sommes capables d'écrire du code directement compréhensible par l'ordinateur. Le désagrément d'écrire un tel programme est qu'il va être très complexe à lire et à modifier par un être humain.

HelloWorld.exe

```
4D 5A 3B 00 01 00 00 00 02 00 00 01 FF FF 02 00
00 10 00 00 00 00 00 00 1C 00 00 00 00 00 00 00
0E 1F B4 09 BA 0E 00 CD 21 B8 00 4C CD 21 48 65
6C 6C 6F 20 57 6F 72 6C 64 21 24
```

Regardez le programme ci-dessus : il y a très peu de chances pour que vous compreniez quoi que ce soit. C'est la retranscription en hexadécimal d'un `Hello, World!` pour les systèmes DOS 32 bits. Vous voyez aisément que si l'on écrit dans la *«langue naturelle»* pour le système, il devient très difficile de comprendre ou de modifier ce message.

Le rôle du compilateur va donc être de générer un fichier plus facilement compréhensible par votre système, à partir d'un ou plusieurs fichiers facilement compréhensibles par un être humain.

Système et environnement d'exécution

Je viens de parler de codes plus facilement compréhensibles par le système. En fait, le système décrit l'environnement dans lequel s'exécute le programme. Il peut s'agir par exemple de votre système d'exploitation, ou bien encore, comme c'est le cas en Java (et donc en Kotlin qui se base sur Java), d'un environnement d'exécution dédié.

Le principal avantage de cet environnement d'exécution dédié est que vous pouvez écrire des programmes et partager les binaires sans vous soucier du système d'exploitation sur lequel il sera exécuté.

Par exemple, un fichier exécutable `.exe` pourra être exécuté sous Windows, mais si vous tentez de l'exécuter sous macOS, cela ne fonctionnera pas.

Cependant, un fichier exécutable Java ou Kotlin, qui se matérialise par un fichier `.class`, pourra être exécuté aussi bien sous Windows, que macOS ou encore Android.

L'exécution

Enfin, vient l'étape de l'exécution du programme. C'est cette étape qui produit le résultat final.

Compilation Kotlin

Sous Kotlin, pour compiler un programme, il suffit d'exécuter la commande suivante dans un terminal :

```
kotlinc HelloWorld.kt
```

Cette commande produira normalement un fichier `HelloWorldKt.class`. Il s'agit d'un fichier exécutable Kotlin.

Exécution Kotlin

Pour exécuter votre programme Kotlin désormais compilé, il suffit d'exécuter la commande suivante :

```
kotlin HelloWorldKt
```

Le résultat produit sera le suivant :

```
Hello, World!
```

Félicitations, vous venez d'écrire le code source, de compiler, et d'exécuter votre première application Kotlin.

Explications du code source

Revenons un peu sur notre code source, et nous allons chercher à savoir ce qu'il fait exactement.

HelloWorld.kt

```kotlin
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

Fonctions, instructions

Le mot-clé `fun` indique à Kotlin que nous allons définir une fonction. En programmation, on désigne par fonction un groupe contenant une ou plusieurs instructions à exécuter dans un ordre défini.

Une instruction étant un appel à une action de la part de l'ordinateur, cette action peut soit être une action exécutée par le système (créer un fichier, afficher du texte à l'écran, etc.), soit un appel à une fonction.

Juste après ce mot-clé `fun`, on trouve le nom donné à la fonction, dans notre cas, `main`. Le fait d'appeler cette fonction `main` permet d'indiquer à Kotlin que c'est cette fonction qui doit être exécutée lorsque l'on appelle le programme `HelloWorldKt`. En effet, nous aurions pu écrire un programme contenant plusieurs fonctions, et si tel avait été le cas, il aurait bien fallu indiquer à Kotlin quelle fonction exécuter en premier.

Entre parenthèses, nous retrouvons les arguments de notre fonction, ici `args: Array<String>`. Les arguments, ce sont les données avec lesquelles notre fonction va interagir. Ici, ces données ne servent pas, donc nous n'allons pas entrer plus en détails sur l'argument de cette fonction `main`, mais cet argument est obligatoire. Si nous avions écrit notre fonction `main` comme ceci : `fun main()`, cela aurait provoqué un bug lors de l'exécution, même si la notation écrite ici correspond à une fonction `main` sans argument, ce qui est tout à fait valide. Il s'agit plus d'un bug dû à la spécificité de la fonction `main`, qui est celle appelée lors de l'exécution d'un programme.

Ensuite, nous retrouvons une accolade ouvrante et une accolade fermante à la fin du fichier. Ces accolades servent à délimiter les instructions d'une fonction. Ainsi, toutes les instructions comprises entre ces accolades sont les instructions de la fonction main.

Println

Ensuite, nous avons un appel à la fonction `println`. Cette fonction sert à afficher un texte (`print`) sur une nouvelle ligne (`ln`). Cette fonction prend en argument la chaine de caractères `"Hello, World!"`. On voit bien ici l'utilité de l'argument d'une fonction. Ici, l'argument représente les données que va afficher la fonction `println`.

Conclusion

Et voilà, nous avons vu comment écrire un code source en Kotlin, comment le compiler et l'exécuter. Nous en avons profité pour voir ce que sont que les fonctions, les instructions et les arguments.

Juste une petite note pour vous rassurer : le début de cet livre contient beaucoup de définitions, qui peuvent vous paraitre abstraites, et que vous pouvez avoir du mal à comprendre. Cela est tout à fait normal, et ne doit pas vous décourager.

Autant il est nécessaire que tous ces termes soient définis lors de leur premier usage, autant il n'est pas nécessaire d'en connaitre la définition par cœur pour appréhender la suite du livre.

Afin de vous rassurer sur ce point, j'ai été contraint d'aller sur Wikipédia pour définir ces notions, afin d'avoir une idée de la définition qu'il donnait de ces mots. Une fois que vous aurez développé quelques programmes, vous verrez rapidement que vous saurez ce qu'est une fonction, sans forcément savoir définir ce que c'est précisément. Vous aurez une idée très claire de ce que c'est et de ce que ce n'est pas.

Exercices

Chaque chapitre se terminera par un ou plusieurs exercices, avec une mention à propos de la difficulté : facile, moyen ou difficile. Il ne s'agit pas vraiment de la complexité du programme à écrire (car cette valeur est subjective et propre à chacun), mais plus d'une échelle permettant de savoir à quoi vous attendre :

- **Facile** : Tout ce dont vous avez besoin pour compléter l'exercice est dans le chapitre. Vous n'avez qu'à appliquer la théorie à la problématique de l'exercice pour comprendre.
- **Moyen** : Vous pouvez compléter cet exercice avec ce qui est dit dans le chapitre, mais vous aurez besoin pour cela d'expérimenter, de tenter des choses, de vous tromper, car tout n'est pas forcément indiqué.
- **Difficile** : Le seul contenu de ce livre ne vous permettra pas de compléter l'exercice. Vous aurez besoin de chercher comment faire sur Internet, ou bien sur d'autres sources.

Bien sûr, j'aurais pu écrire un livre exhaustif dont le contenu puisse directement vous permettre de compléter les exercices. Mais c'est une volonté que de vous mettre face à de telles épreuves. Encore une fois, le but de ce livre est de faire de vous des développeurs débutants. Or, un développeur passe ses journées à essayer à tâtons, à aider ses collègues et à se faire aider par ses collègues, et à chercher sur Internet des solutions aux problèmes qu'il rencontre. Autant vous y habituer tout de suite…

Exercice 1 - Moyen

Reprenez le programme `HelloWorld.kt`, et modifiez-le de sorte que :

- Il contienne deux fonctions, `main` et `displayHelloWorld`.
  - `displayHelloWorld`, sans argument, sera une fonction qui affiche `Hello, World!`,
  - `main` se contentera d'appeler displayHelloWorld.

Exercice 2 - Difficile

Nous l'avons vu, lorsque nous compilons notre programme, cela nous produit un fichier `HelloWorldKt.class` dans le répertoire actuel. Faites en sorte que le fichier `HelloWorldKt.class` soit généré dans un dossier `exercice`, en utilisant uniquement la commande `kotlinc` (donc sans créer le dossier `exercice` autrement qu'en utilisant la commande `kotlinc`).

Chapitre 2
Variables, Types, Constantes et Opérateurs

Définition d'une variable

Une variable, c'est un symbole qui associe un nom à une valeur. Les variables, vous les utilisez déjà en mathématiques. Lorsqu'on vous demande la formule de la circonférence d'un cercle, vous répondez π multiplié par le diamètre. Or, lorsque vous dites cela, vous associez le mot *diamètre* à une valeur.

Types de variable

En informatique, une valeur correspond à un emplacement de mémoire. Cependant, pour optimiser les performances d'une application, il va falloir faire en sorte d'utiliser au mieux l'espace mémoire.

Imaginons que vous souhaitiez faire un quizz ayant plus ou moins pour but de faire deviner un nombre entre 1 et 100. Ce nombre peut très bien être stocké sur un octet (2^8, soit 256 valeurs possibles). Vous allez donc affecter à ce nombre une valeur pouvant être stockée sur un octet, afin d'optimiser la mémoire consommée par votre application.

Si maintenant, vous souhaitez associer à chaque participant d'une convention un numéro, sachant que cette convention rassemble entre 2 500 et 3 000 personnes. Vous vous apercevez très vite qu'il n'est pas possible de stocker le numéro de chaque personne sur un octet. Très vite, vous serez dépassé. Vous aurez besoin de les stocker dans un type pouvant englober plus de données.

Nombres

C'est pourquoi, en Kotlin, nous avons 8 types de variables possibles pour stocker des nombres :

- `Byte` : une valeur allant de -128 à +127, stockée sur un octet,
- `Short` : une valeur allant de -32 768 à 32 767, stockée sur deux octets,
- `Int` : une valeur allant de -231 à 231-1, stockée sur quatre octets,
- `Long` : une valeur allant de -263 à 263-1, stockée sur huit octets,
- `Float` : utilisée pour stocker des valeurs décimales avec une précision relative, stockée sur seize octets,
- `Double` : utilisée pour stocker des valeurs décimales avec une plus grande précision, stockée sur trente-deux octets.

Attention, `Float` et `Double` doivent toujours être considérées comme des valeurs approximatives (`Double` étant plus précis), et jamais comme des valeurs exactes.

Caractères

Certes, nous pouvons stocker des nombres dans une variable, mais également des caractères. Ce qu'on appelle un caractère, c'est grosso modo un signe qui peut être affiché à l'écran, comme par exemple une lettre, une tabulation, un retour à la ligne ou encore un signe de ponctuation.

- `Char` : valeur utilisée pour stocker des caractères, stockée sur deux octets.

Booléen

Enfin, le type booléen ne peut prendre que deux valeurs, soit `true` (vrai) soit `false` (faux).

- `Boolean` : valeur utilisée pour stocker une valeur binaire, stockée sur un emplacement très faible en mémoire (inférieur ou égal à un octet), mais de taille variable.

Objet

Enfin, en plus de ces dix types primitifs, les variables peuvent également stocker des instances d'objets. Mais la notion d'objet est un concept que nous verrons plus loin dans ce livre.

Notation

Pour définir une variable, en Kotlin, on utilise le mot clé `var`, suivi du nom de la variable, de deux points, et de son type, suivi du signe `=` et de sa valeur.

Ainsi, pour définir une variable de type `Double`, qui a pour nom `diameter`, nous l'écrirons ainsi :

```
var diameter: Double = 3.78
```

Définition d'une valeur invariante

Une valeur invariante, c'est une variable qui ne change pas de valeur. Vous initialisez sa valeur, et vous ne pouvez plus la modifier. Si vous la modifiez, le compilateur refusera de compiler votre programme.

En dehors du fait que vous ne pouvez pas modifier sa valeur, tout ce qui a été dit à propos des variables est également valable pour une valeur invariante.

Notation

Pour définir une valeur invariante, en Kotlin, on utilise la même notation que pour une variable, à l'exception du mot `var` qui devient `val`.

```
val pi: Double = 3.14
```

Définition d'une constante

Une constante, c'est un identifiant qui va être remplacé par sa valeur par le préprocesseur du compilateur avant la compilation.

Nous avons vu les différentes étapes pour passer d'un code source au résultat final, et parmi elle, la compilation.

Juste avant la compilation, il y a un préprocesseur qui va passer pour modifier votre code source. Et ce préprocesseur va remplacer tous les appels à une constante par leur valeur.

Cette contrainte implique que vous ne pouvez spécifier qu'une valeur brute à une constante. Vous ne pouvez pas, par exemple, affecter à une constante la valeur de retour d'une fonction (notion que nous verrons dans la suite de ce chapitre), mais simplement un nombre ou une chaine de caractère…

Afin de produire un code plus clair, il est d'usage d'utiliser pour une valeur :

- Une constante,
- S'il n'est pas possible d'utiliser une constante, alors une valeur invariante,
- S'il n'est pas possible d'utiliser une valeur invariante, alors une variable.

Notation

Pour définir une constante en Kotlin, la notation est la même que pour une valeur invariante, sauf que l'on fait précéder le mot clé val par le mot clé const.

```
const val PI: Double = 3.14
```

Définition d'un argument

Un argument, c'est une variable que l'on passe à une fonction pour que cette dernière interagisse avec. Par exemple, si l'on écrit une fonction pour calculer la circonférence d'un cercle, alors nous aurons besoin de connaitre le diamètre du cercle. Ce dernier pourra être passé en argument à la fonction.

Notation

Un argument s'écrit entre deux parenthèses dans la définition d'une fonction. Tout comme une variable, on écrit le nom de l'argument, suivi de deux points, suivi de son type. Pour une fonction calculant la circonférence d'un cercle, cela s'écrira :

```
fun circumference(diameter: Double){

}
```

Nous avons vu ici comment écrire une fonction avec un argument. Mais une fonction peut prendre plusieurs arguments. Dans ce cas, les arguments sont séparés par une virgule. Ainsi, si nous voulons écrire une fonction qui calcule le périmètre d'un rectangle, nous aurons besoin de la longueur et de la largeur de ce dernier. La fonction s'écrira donc ainsi :

```
fun rectanglePerimeter(width: Double, height: Double){

}
```

Valeur de retour d'une fonction

Une fonction peut retourner une valeur. Certes, dans notre premier exemple Hello World, la fonction `main` ne retournait aucune valeur. Mais dans le cas d'une fonction calculant la circonférence d'un cercle, nous pourrions souhaiter que la fonction retourne la valeur de ladite circonférence. Pour indiquer une valeur de retour, après la parenthèse fermante, nous ajoutons deux points, suivi du type de la valeur de retour. Ainsi, pour la fonction calculant la circonférence d'un cercle, cela donne :

```
fun circumference(diameter: Double): Double{

}
```

Une fonction ne peut avoir qu'une seule valeur de retour, et cette dernière ne peut être que d'un type donné.

Maintenant, vous pouvez tout à fait choisir d'affecter à une variable, la valeur de retour d'une fonction. Cela se note ainsi :

```
val diameter : Double = 3.78
val circumference : Double = circumference(diameter)
```

Types implicites et littéraux

Nous avons vu comment définir une variable :

```
val someValue: Int = 4
```

Dans ce cas précis, nous ne sommes pas obligés de déclarer le type `Int`, nous pouvons nous contenter d'écrire :

```
val someValue = 4
```

Cette notation est tout à fait correcte, et est équivalente à la première. Attention toutefois, si le type n'est pas explicitement indiqué, il n'en demeure pas moins que 4 est un nombre entier. Pour vous en convaincre, vous pouvez écrire le programme suivant :

TypeError.kt

```
fun main(args: Array<String>) {
   var someValue = 4
   someValue = 5.56
   println(someValue)
}
```

En essayant de compiler ce programme, vous obtiendrez l'erreur suivante :

```
TypeError.kt:3:17: error: the floating-point literal does not conform to the expected type Int
    someValue = 5.56
                ^
```

Kotlin vous indique de manière spécifique qu'il ne peut pas affecter une valeur décimale à une variable de type `Int`.

Mais du coup, quel type par défaut est donné à une variable quand on ne spécifie pas son type ?

On peut le savoir en écrivant un petit programme en Kotlin :

VarTpes.kt

```
fun main(args: Array<String>) {
    val someValue = 4
    val someValue2 = 4.56
    val someValue3 = 'c'

    println(someValue.javaClass.kotlin.qualifiedName)
    println(someValue2.javaClass.kotlin.qualifiedName)
    println(someValue3.javaClass.kotlin.qualifiedName)
}
```

Ce programme fait appel à des notions que nous verrons plus loin dans ce livre. Contentez-vous de le recopier tel qu'il est, ce n'est pas important si vous ne comprenez pas tout maintenant.

En compilant puis en exécutant ce programme, on obtient le résultat suivant :

```
kotlin.Int
kotlin.Double
kotlin.Char
```

Ainsi, on peut savoir que, implicitement, lorsqu'on écrit un nombre entier, c'est le type `Int` par défaut qui lui est assigné, pour un nombre avec une décimale, le type `Double`, et pour un caractère écrit entre guillemets simples, le type `Char`.

Mais comment faire alors pour assigner à une variable le type `Float` ou bien `Long` sans avoir à spécifier son type ?

On peut utiliser un caractère, à la fin de la valeur, pour spécifier le type de cette valeur :

VarTpesFull.kt

```
fun main(args: Array<String>) {
    val someValue = 4
    val someValue2 = 4L
    val someValue3 = 4.56
    val someValue4 = 4.56F
    val someValue5 = 'c'

    println(someValue.javaClass.kotlin.qualifiedName)
    println(someValue2.javaClass.kotlin.qualifiedName)
    println(someValue3.javaClass.kotlin.qualifiedName)
    println(someValue4.javaClass.kotlin.qualifiedName)
    println(someValue5.javaClass.kotlin.qualifiedName)
}
```

En compilant puis en exécutant ce programme, on obtient le résultat suivant :

```
kotlin.Int
kotlin.Long
kotlin.Double
kotlin.Float
kotlin.Char
```

On voit qu'en ajoutant `L` à la fin d'un nombre entier, cela passe son type à `Long`, et qu'en ajoutant `F` à la fin d'un nombre décimal, cela passe son type à `Float`. Ce sont les deux seuls types pour lesquels on peut utiliser ce raccourci.

Pour déclarer une variable de type `Short` ou `Byte`, vous devrez utiliser la déclaration complète.

Notations des entiers

Pour rendre le code plus lisible et/ou plus compréhensible, il est possible d'utiliser des notations différentes pour les entiers.

Underscore

Pour les très grands nombres, il est possible d'utiliser le caractère underscore afin de séparer les chiffres par groupe, dans le but de rendre cela plus lisible :

```
val complexGigaByte = 1073741824
val readableGigaByte = 1_073_741_824
```

Ces deux déclarations sont identiques, et l'on peut aisément voir que la seconde est bien plus lisible.

Notation Binaire

Parfois, on peut souhaiter écrire un nombre dans sa notation binaire. C'est possible en préfixant ce nombre de `0b`. Là encore, les underscores sont utilisables :

```
val readableGigaByte = 1_073_741_824
val binaryGigaByte = 0b1000_0000_0000_0000_0000_0000_0000_000
```

Ces deux déclarations sont équivalentes.

Notation Hexadécimale

De la même façon qu'il est possible d'écrire un nombre en notation binaire en le préfixant de `0b`, il est possible d'écrire un nombre dans sa notation hexadécimale en le préfixant de `0x` :

```
val readableGigaByte = 1_073_741_824
val hexadecimalGigaByte = 0x4000_0000
```

Opérateurs

Affectation

C'est le seul opérateur que nous ayons déjà utilisé. Il s'agit du signe `=` pour affecter une valeur à une variable :

```
val three = 3
```

Opérateurs mathématiques classiques

Il est possible d'effectuer des opérations mathématiques entre deux valeurs ou variables numériques :

Operators.kt

```
fun main(args: Array<String>) {
   val add = 2_000_000_000+2_000_000_000
   val multiply = 3*7
   val substract = 7-4
   val divide = 7/3
   val modulo = 7%3

   println(add)
   println(multiply)
   println(substract)
   println(divide)
   println(modulo)
}
```

L'opération modulo retourne le reste de la division euclidienne.

En compilant puis en exécutant le programme, on obtient le résultat suivant :

```
-294967296
21
3
2
1
```

On peut voir ici que la somme de 2 milliards + 2 milliards produit un résultat négatif. De même, on peut voir que la division de 7 par 3 donne comme résultat 2. C'est simplement parce qu'une opération entre deux variables de même type renvoie un résultat de même type. Ici, une addition de deux `Int`, ou une division de deux `Int`, produit un `Int`.

Très bien, cela explique pourquoi la division de 7 par 3 renvoie 2, mais pas pourquoi notre somme renvoie un nombre négatif. On a déjà dit que le type `Int` était limité à 2^{31}-1, soit 2 147 483 647. Si l'on dépasse cette valeur, alors plus rien de fiable n'est à attendre d'une variable `Int`.

Mais qu'en est-il alors d'une opération entre deux types différents ? C'est le type qui est codé sur le plus d'octets qui est gardé. La seule exception concerne le type `Char`, qui restera un `Char` même si l'on y ajoute une valeur codée sur plus d'octets. Ainsi, pour obtenir le résultat attendu, on peut écrire le programme suivant :

OperatorsCorrect.kt

```
fun main(args: Array<String>) {
   val byte:Byte = 7
   val short:Short = 5

   val add = 2_000_000_000+2_000_000_000L
   val multiply = 3*7
   val substract = 7-4
   val divide = 7/3.0
   val modulo = 7%3
   val add2 = byte+2_000_000_000
   val add3 = short+2_000_000_000
   val add4 = 'c'+3

   println(add)
   println(multiply)
   println(substract)
   println(divide)
   println(modulo)
   println(add2)
   println(add3)
   println(add4)
}
```

En compilant puis en exécutant le programme, on obtient le résultat suivant :

```
4000000000
21
3
2.3333333333333335
1
2000000007
2000000005
f
```

Affectations avancées

Parfois, vous pourrez avoir besoin d'écrire le code suivant :

```
var value = 3
value = value + 10
value = value - 1
value = value / 3
value = value * 4
value = value % 5
```

Quand votre variable est à la fois la valeur affectée, et le premier élément d'une opération mathématique, alors, vous pouvez l'écrire comme suit :

```
var value = 3
value += 10
value -= 1
value /= 3
value *= 4
value %= 5
```

Les deux extraits de code sont équivalents.

Cas particulier des chaines de caractères

Vous pouvez utiliser le signe `+` dans une chaine de caractère pour concaténer le résultat :

```
val HelloWorld100 = "Hello "+"World! "+100;
```

La variable `HelloWorld100` vaut désormais `"Hello World! 100"`.

Place à la pratique

Dans votre éditeur de texte, ouvrez un nouveau fichier que vous nommerez Circumference.kt.

Circumference.kt

```
fun main(args: Array<String>) {
   var diameter: Double = 3.78
   var circumference = circumference(diameter)
   println(circumference)
   diameter = 1.72
   circumference = circumference(diameter)
   println(circumference)
}

fun circumference(diameter: Double): Double{
   return 3.14*diameter
}
```

Ce programme retourne la circonférence d'un cercle en fonction de son diamètre. Si vous compilez puis exécutez ce programme, cela donnera le résultat suivant :

```
47.4768
22.2312
```

Nous pourrions nous satisfaire de cela, mais il manque une dernière petite chose.

Nettoyage du code

Le contenu de notre fonction `main` est difficile à appréhender. Il faut la lire ligne par ligne pour bien comprendre ce que fait cette fonction. Mais nous pourrions rendre les choses plus claires.

Commentaires

Dans le code, nous pouvons ajouter des commentaires. Ce sont des parties du code que le compilateur va tout simplement ignorer, mais qui peuvent aider à la lecture.

Commentaire sur une ligne

Pour écrire un commentaire sur une ligne, il suffit d'ajouter `//`. Tout ce qui suit sera ignoré par le compilateur, à l'exception des `//` dans une chaine de caractère :

```
var diameter: Double = 3.78 // diameter vaut 3.78
println("Je suis affiché // et moi aussi") // mais pas moi
```

Bloc de commentaires

Nous avons également la possibilité d'écrire un commentaire sur plusieurs lignes. Le commentaire doit commencer par `/*` et se terminer par `*/` :

```
/*Calcule la circonférence d'un cercle.
La formule utilisée et pi*diamètre
C'est quand même beaucoup plus clair*/
circumference = circumference(diameter)
```

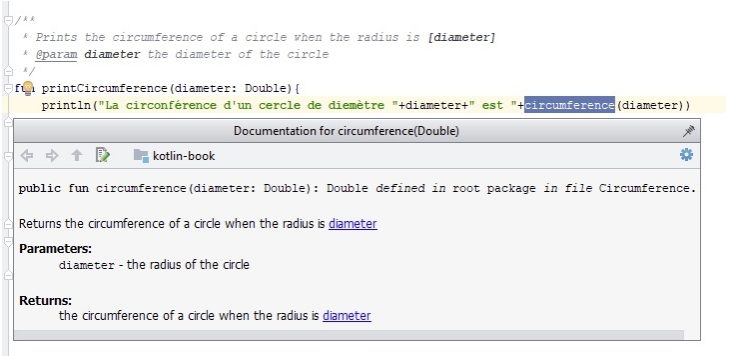
KDoc

KDoc est une forme particulière de commentaire, que l'on dispose avant un bloc, et qui permet d'expliquer ce bloc. L'avantage de KDoc est qu'étant donné qu'il est connu et utilisé par tous, sa lecture, son accès, et même la génération de documentation sont simplifiés.

On pourrait consacrer un chapitre entier à KDoc, mais je préfère simplement vous en montrer un exemple, et que vous vous imprégniez de son usage au fur et à mesure de la lecture de ce livre.

```
/**
 * Retourne la circonférence d'un cercle de diamètre [diameter].
 * @param diameter le diamètre du cercle
 * @return la circonférence d'un cercle de diamètre [diameter]
 */
fun circumference(diameter: Double): Double{
   return 3.14*diameter
}
```

Le bloc KDoc ici permet d'indiquer clairement ce que fait la fonction, et comment sont utilisés les arguments.



Exemple d'affichage d'une boîte d'aide sur le rôle d'une fonction pour laquelle une documentation au format KDoc a été rédigée dans Android Studio.

Langue anglaise

Comme vous pouvez le remarquer, tous les codes jusqu'ici ont été écrits en anglais. Non, il ne s'agit pas d'une traduction incomplète de ce livre, puisque ce dernier a d'abord été écrit en français, avant d'être traduit en anglais.

Mais l'immense majorité des développeurs s'accordent à tout écrire en anglais pour plusieurs raisons :

- Toutes les fonctions Kotlin sont en anglais, comme par exemple la fonction `println` que l'on a utilisée. Écrire son code en anglais permet donc de ne pas mélanger deux langues différentes dans son code, et par conséquent de garder une certaine homogénéité qui facilite la compréhension.
- De la même façon, vous vous servirez de nombreux sites anglophones à la fois pour poster des questions avec des extraits de votre code, mais aussi pour copier/coller du code disponible. Là encore, garder tout en anglais permet de conserver l'homogénéité, et de mieux se faire comprendre de ceux qui vont nous aider.
- Enfin, la dernière raison, c'est que les identifiants de variables et de fonctions n'aiment pas les caractères spéciaux. Vous ne pouvez pas utiliser de «é», de «è» ou de «ç» dans les noms de fonctions. Plutôt que d'écrire avec des erreurs, autant tout écrire correctement.

Factorisation

Vous l'avez vu, dans notre code, nous répétons à deux reprises la séquence suivante :

- diameter = valeur
- circumference = retourDeFonction
- affichage de circonférence

Le problème avec le code qui se répète, c'est que si l'on souhaite le modifier, pour corriger des bugs ou le faire évoluer, cela implique de devoir modifier son code deux fois.

Constantes

Afin de rendre le code plus lisible, il est recommandé de sortir toutes les variables brutes du code des fonctions pour les mettre dans des constantes.

En effet, si vous faites une faute de frappe avec un nombre ou une chaine de caractère dans votre code, cela peut avoir pour conséquence de rendre le code très complexe à corriger (dans des projets de grand volume).

Passer par des constantes simplifiera la chose, puisque si vous faites une faute de frappe dans l'écriture du nom d'une constante, le compilateur refusera de compiler, en vous indiquant précisément où se situe votre erreur.

Pour distinguer les constantes des variables, il est fréquent d'écrire le nom des constantes en majuscules.

Place à la pratique… encore

Du coup, en appliquant les conseils ci-dessus, on obtient le code suivant :

Circumference.kt

```kt
/** The value of π for calculating the circumference */
const val PI: Double = 3.14

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printCircumference(1.72)
}

/**
 * Prints the circumference of circle when diameter is [diameter]
 * @param diameter the diameter of circle
 */
fun printCircumference(diameter: Double){
   println(circumference(diameter))
}

/**
 * Returns the circumference of circle when diameter is [diameter]
 * @param diameter the diameter of circle
 * @return the circumference of a circle when diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

Cela rend le code beaucoup plus facile à lire et à maintenir. Imaginons maintenant que vous souhaitiez afficher `"circumference("+diameter+")="+circumference(diameter)`, vous n'aurez à effectuer cette modification qu'à un seul endroit :

Circumference.kt

```kotlin
/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
```

La factorisation vous a permis de ne modifier qu'un seul endroit pour rectifier les deux affichages.

Exercices

Exercice 1 - Facile

Développez un programme similaire au précédent, mais qui calcule le périmètre d'un carré en fonction de la mesure de ses côtés.

Exercice 2 - Facile

Développez un programme, qui, à partir des valeurs 13 et 5, renvoie `"13/5 = 2 et il reste 3"`.

Chapitre 3
Sources multiples, paquets et imports

Dans ce chapitre, certaines définitions seront surlignées, étant incomplètes ou trop imprécises quant à la notion de programmation orientée objets, abordée dans le chapitre suivant.

Jusque-là, nous avons vu comment développer des fonctions, avec des variables. Dès lors, vous pourriez avoir envie de développer de gros programmes. Rien ne vous empêche par exemple de développer un projet qui calcule la circonférence et l'aire de toutes les formes géométriques possibles.

Imaginez un peu, les quadrilatères, les cercles, les triangles, pourquoi pas les pentagones et les hexagones tant qu'on y est, jusqu'au dodécagone. Bref, le code source de votre programme deviendrait très volumineux, et aurait donc deux inconvénients principaux :

- Le premier, il serait complexe de maintenir un code source d'un énorme volume dans un seul fichier source ;
- Le second, si votre programme commence à devenir très important, cela pourrait occasionner des problèmes de chargement, le temps que l'ordinateur soit capable de charger l'intégralité des fonctions que vous avez développées.

C'est pour résoudre cette problématique que Kotlin offre l'avantage d'avoir un système de paquets et d'imports très pratique.

Sources multiples

Nous allons écrire un programme, semblable à celui développé dans le projet précédent, mais séparé en deux fichiers :

- L'un contiendra les fonctions d'affichage du programme ;
- L'autre contiendra les fonctions relatives au calcul sur un cercle.

C'est parti :

Geometry.kt

```
/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printCircumference(1.72)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
```

Circle.kt

```
/** The value of π for calculating the circumference of a circle */
const val PI: Double = 3.14

/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

Maintenant exécutez la commande suivante :

```
kotlinc Geometry.kt Circle.kt -d ./build
```

L'argument `-d` que vous connaissez surement, si vous avez résolu les exercices du chapitre 1, permet de créer un répertoire dans lequel seront stockés les fichiers exécutables. Cela permet de ne pas mélanger les fichiers sources et les fichiers exécutables, ce qui devient très important lorsque l'on travaille avec des projets sur plusieurs fichiers.

Pour exécuter le programme, il suffit d'exécuter la commande suivante :

```
kotlin -cp ./build GeometryKt
```

L'argument `-cp` permet d'indiquer au l'exécuteur kotlin dans quel répertoire chercher nos fichiers exécutables.

Vous pourrez observer dans le dossier build, que deux fichiers exécutables .class ont été créés. Ainsi, nous pouvons en déduire que le compilateur Kotlin a bien séparé nos fichiers source.

Paquets

Nous avons vu comment compiler un programme dans lequel les sources sont dans des fichiers séparés. Mais nous pourrions aller plus loin.

Ici, la constante `PI` a été ajoutée dans le fichier calculant la circonférence d'un cercle. Imaginons maintenant que nous souhaitions ajouter à notre programme la possibilité pour des angles exprimés en degrés de les convertir en radians, et vice versa. Nous aurons également besoin de cette valeur pour les angles.

Cela nous obligera à avoir quatre fichiers, afin de ne pas dupliquer de codes comme vu dans le chapitre précédent :

Geometry.kt

```
/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(PI)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double){
   println("toDegrees("+angRad+")="+toDegrees(angRad))
}
```

Circle.kt

```kotlin
/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return PI*diameter
}
```

Angle.kt

```kotlin
/** Flat angle measured in degrees */
const val DEGREES_FLAT_ANGLE = 180
/** Flat angle measured in radians */
const val RADIANS_FLAT_ANGLE = PI

/**
 * Converts an angle measured in radians to an equivalent angle measured in degrees.
 * @param angrad an angle, in radians
 * @return the measurement of the angle [angrad] in degrees
 */
fun toDegrees(angrad: Double): Double{
   return angrad/RADIANS_FLAT_ANGLE*DEGREES_FLAT_ANGLE
}

/**
 * Converts an angle measured in degrees to an equivalent angle measured in radians.
 * @param angdeg an angle, in degrees
 * @return the measurement of the angle [angdeg] in radians
 */
fun toRadians(angdeg: Double): Double{
   return angdeg/DEGREES_FLAT_ANGLE*RADIANS_FLAT_ANGLE
}
```

Constants.kt

```kotlin
/** The value of π for calculating the circumference of a circle */
const val PI: Double = 3.14
```

Là, nous n'avons pris que l'exemple de 4 fichiers, mais parfois, un projet peut contenir des centaines, voire des milliers de fichiers. Du coup, on ne va pas s'amuser à ajouter chacun de nos fichiers dans la commande de compilation :

```
kotlinc ./* -d build
```

Le programme compile ; on peut l'exécuter, tout va pour le mieux.

Maintenant, imaginons que nous ajoutions encore deux fichiers à notre projet, l'un pour calculer le périmètre d'un carré, l'autre pour calculer celui d'un triangle équilatéral :

Square.kt

```kotlin
/**
 * Returns the perimeter of a square when the length of a side is [sideLength]
 * @param sideLength the length of a side of the square
 * @return the perimeter of a square when the length of a side is [sideLength]
 */
fun perimeter(sideLength: Double): Double{
   return 4*sideLength
}
```

EquilateralTriangle.kt

```kotlin
/**
 * Returns the perimeter of an equilateral triangle when the length of a side is [sideLength]
 * @param sideLength the length of a side of the equilateral triangle
 * @return the perimeter of an equilateral triangle when the length of a side is [sideLength]
 */
fun perimeter(sideLength: Double): Double{
   return 3*sideLength
}
```

Encore une fois, nous allons compiler notre programme :

macOS et Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

Et là nous obtenons une erreur :

```
EquilateralTriangle.kt:6:1: error: conflicting overloads: public fun perimeter(sideLength: Dou
ble): Double defined in root package in file EquilateralTriangle.kt, public fun perimeter(side
Length: Double): Double defined in root package in file Square.kt
fun perimeter(sideLength: Double): Double{
^
Square.kt:6:1: error: conflicting overloads: public fun perimeter(sideLength: Double): Double
defined in root package in file EquilateralTriangle.kt, public fun perimeter(sideLength: Doubl
e): Double defined in root package in file Square.kt
fun perimeter(sideLength: Double): Double{
^
```

En fait, cette erreur nous indique que la fonction `perimeter(Double)` est définie deux fois. Et ce serait une erreur que de vouloir appeler nos méthodes différemment, elles comportent un nom clair.

Pour résoudre cette problématique, Kotlin nous permet de séparer nos différents fichiers sources en paquets. Et si une même fonction est définie dans deux paquets différents, Kotlin va les considérer comme deux fonctions différentes.

Pour définir un fichier source comme appartenant à un paquet, il suffit d'indiquer le nom du paquet que l'on souhaite affecter préfixé par le mot clé `package` sur la première ligne du fichier.

De plus, pour permettre de hiérarchiser les paquets, chaque paquet peut être défini dans un sous-paquet, il suffit juste de les séparer par un point. Ainsi, `geometry.angles` et `geometry.circle` seront deux paquets, `angles` et `circle`, appartenant chacun au paquet `geometry`.

Ainsi, les premières lignes de nos fichiers deviennent donc :

Geometry.kt

```
package geometry
```

Circle.kt

```
package geometry.circle
```

Angle.kt

```
package geometry.angles
```

Constants.kt

```
package geometry.constants
```

Square.kt

```
package geometry.quadrilateral.square
```

EquilateralTriangle.kt

```
package geometry.triangle.equilateral
```

Maintenant, il ne nous reste plus qu'à compiler notre programme. Essayons pour voir :

macOS et Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

Nous obtenons plusieurs erreurs. En fait le souci, c'est que le compilateur Kotlin ne sait plus trouver la fonction `circumference`, ni même toutes les autres références. Mais pourquoi ne les trouve-t-il plus ?

En fait, si nous ne précisons pas de paquet, Kotlin va automatiquement ajouter nos fichiers dans un même paquet, un paquet racine. Ainsi, avant que l'on spécifie nos différents paquets, Kotlin a considéré que tous nos fichiers faisaient partie du même paquet. Or, lorsqu'on fait appel à une fonction, Kotlin va automatiquement la rechercher dans le paquet du fichier qui effectue l'appel.

Ainsi, dans `Geometry.kt`, lorsqu'on appelle `printAngleDeg(PI)`, il va non plus le chercher dans le paquet racine, mais bien dans le paquet `geometry` que nous avons spécifié pour ce fichier. Or, la définition de la constante `PI` se trouve dans un paquet différent. Kotlin ne sait donc pas trouver la définition de cette constante.

Pour que Kotlin trouve notre constante `PI`, nous devons lui indiquer dans quel paquet trouver cette constante. Pour ce faire, nous devons faire précéder `PI` par le nom du paquet dans lequel il se trouve suivi d'un point. Ainsi, `PI` devient `geometry.constants.PI`.

De ce fait, le code de nos différents fichiers devient :

Geometry.kt

```
package geometry

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(geometry.constants.PI)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+geometry.circle.circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double){
   println("toDegrees("+angRad+")="+geometry.angles.toDegrees(angRad))
}
```

Angles.kt

```
package geometry.angles

/** Flat angle measured in degrees */
const val DEGREES_FLAT_ANGLE = 180
/** Flat angle measured in radians */
const val RADIANS_FLAT_ANGLE = geometry.constants.PI

/**
 * Converts an angle measured in radians to an equivalent angle measured in degrees.
 * @param angrad an angle, in radians
 * @return the measurement of the angle [angrad] in degrees
 */
fun toDegrees(angrad: Double): Double{
   return angrad/RADIANS_FLAT_ANGLE*DEGREES_FLAT_ANGLE
}

/**
 * Converts an angle measured in degrees to an equivalent angle measured in radians.
 * @param angdeg an angle, in degrees
 * @return the measurement of the angle [angdeg] in radians
 */
fun toRadians(angdeg: Double): Double{
   return angdeg/DEGREES_FLAT_ANGLE*RADIANS_FLAT_ANGLE
}
```

Circle.kt

```
package geometry.circle

/**
 * Returns the circumference of a circle when the diameter is [diameter]
 * @param diameter the diameter of the circle
 * @return the circumference of a circle when the diameter is [diameter]
 */
fun circumference(diameter: Double): Double{
   return geometry.constants.PI*diameter
}
```

Nous pouvons désormais compiler notre programme (je vous épargne le suspens puisque cela fonctionne), et l'exécuter :

```
kotlin -cp build GeometryKt
```

Cette fois, c'est l'exécution qui provoque une erreur :

```
error: could not find or load main class GeometryKt
```

En fait, comme c'était le cas dans nos fichiers sources, lorsque nous appelons `GeometryKt`, Kotlin cherche le fichier exécutable `GeometryKt` situé dans le paquet racine. Or, nous avons spécifié un paquet pour notre fichier : `geometry`. Il suffit simplement d'indiquer à Kotlin le paquet dans lequel se trouve l'exécutable `GeometryKt` pour que cela fonctionne :

```
kotlin -cp build geometry.GeometryKt
```

Il ne nous reste plus qu'à modifier notre fichier `Geometry.kt` pour intégrer les deux fonctions que nous avons développées :

Geometry.kt

```
package geometry

/**
 * Main method, displaying the circumference of two circles
 */
fun main(args: Array<String>) {
   printCircumference(3.78)
   printAngleDeg(geometry.constants.PI)
   printEquilateralTrianglePerimeter(3.0)
   printSquarePerimeter(3.0)
}

/**
 * Prints the circumference of a circle when the diameter is [diameter]
 * @param diameter the circumference of the circle
 */
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+geometry.circle.circumference(diameter))
}

/**
 * Prints the conversion of an angle measured in radian to
 */
fun printAngleDeg(angRad: Double) {
   println("toDegrees(" + angRad + ")=" + geometry.angles.toDegrees(angRad))
}

/**
 * Prints the perimeter of an equilateral triangle when the length of a side is [sideLength]
 * @param sideLength the length of a side of the equilateral triangle
 */
fun printEquilateralTrianglePerimeter(sideLength: Double){
   println("Equilateral triangle perimeter(" + sideLength + ")=" + geometry.triangle.equilater
al.perimeter(sideLength))
}

/**
 * Prints the perimeter of an square when the length of a side is [sideLength]
 * @param sideLength the length of a side of the square
 */
fun printSquarePerimeter(sideLength: Double){
   println("Square perimeter(" + sideLength + ")=" + geometry.quadrilateral.square.perimeter(s
ideLength))
}
```

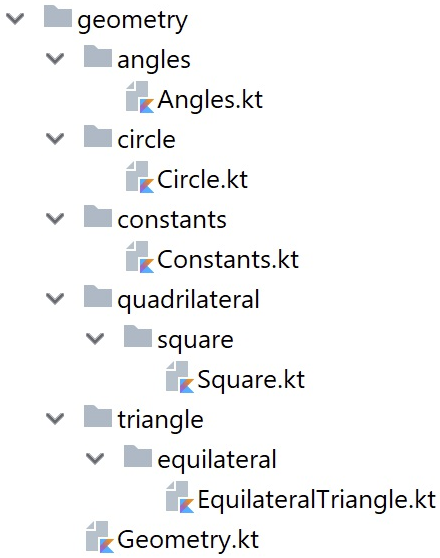
Convention d’arborescence

Avec tous nos fichiers dans un même répertoire, il est difficile de savoir quel fichier appartient à quel package d’un simple coup d’œil.

De plus, on peut facilement établir un parallèle entre les paquets qui peuvent s'imbriquer les uns dans les autres, et les dossiers avec lesquels on peut faire de même.

Du coup, la convention très répandue et appliquée par tous les développeurs est d'organiser les fichiers sources d'un projet, de sorte qu'à chaque paquet corresponde un dossier, et que chaque fichier source soit dans le dossier correspondant à son paquet.

Ce qui donne pour nos fichiers sources l'arborescence suivante :



La commande reste la même pour compiler le projet :

macOS et Linux :

```
rm -rf ./build/* && kotlinc ./* -d build
```

Windows :

```
del /S/Q build && kotlinc ./* -d build
```

Imports

Nous l'avons vu dans le fichier `Geometry.kt`, désormais, pour calculer la circonférence d'un cercle, nous devons appeler la fonction `geometry.circle.circumference()`.

D'une manière générale, on essaie de garder le code aussi simple et concis que possible. Imaginez qu'à plusieurs endroits, vous ayez besoin d'appeler cette fonction, écrire `geometry.circle.circumference()` à chaque fois n'est ni simple, ni concis.

Pour palier ce problème, en Kotlin, nous pouvons utiliser les imports. Un import nous permet de spécifier le chemin complet d'une fonction que nous appellerons dans un fichier. Voici un exemple avec Geometry.kt.

Geometry.kt

```
package geometry

import geometry.circle.circumference


// ...
fun printCircumference(diameter: Double){
   println("circumference("+diameter+")="+circumference(diameter))
}
// ...
```

La notation `// ...` indique ici qu'il y a des extraits de code que l'on omet. Cela permet de se focaliser uniquement sur une partie du code. Ici, seul ce qui change dans Geometry.kt a été affiché, le reste demeurant identique a été simplifié avec la notation `// ...`

On voit bien ici comment l'import nous simplifie les choses.

Étoile

Maintenant, prenons l'exemple des angles, et des deux fonctions qui y sont associées. On pourrait, avec les imports, écrire ceci :

```
import geometry.angles.toDegrees
import geometry.angles.toRadians
```

Mais il existe en Kotlin un moyen pour importer la totalité d'un paquet, il s'agit de la notation `*`. Ainsi, pour importer la totalité des fonctions contenues dans le paquet `geometry.angles`, on peut noter :

```
import geometry.angles.*
```

Ainsi, les deux méthodes pourront être appelées sous leur forme courte dans le code.

Bien que cela soit possible, nous vous déconseillons fortement d'utiliser cette notation, pour deux raisons principales :

Même si vous vous servez de l'intégralité du paquet aujourd'hui, il se peut que vous ajoutiez des fonctions au paquet par la suite. Cela impliquerait donc un import inutile, donc une perte en termes de performances.

Il est très compliqué de savoir tout de suite la liste de ce qui est importé et pourquoi, ce qui rend le code moins lisible.

Ainsi, dans la majorité des cas, il est préférable d'utiliser un import par utilisation et d'éviter les imports avec étoile.

Limitations

Toutefois, il n'est pas possible d'importer deux paquets contenant une méthode ayant le même nom. Ainsi, il ne sera pas permis d'effectuer les deux imports suivants :

```
import geometry.quadrilateral.square
import geometry.triangle.equilateral
```

Et ce, pour la même raison que celle qui nous a poussée à nous intéresser aux paquets. En utilisant ces deux imports, lorsque vous utiliserez la fonction `perimeter`, le compilateur ne saura pas déterminer laquelle des deux fonctions il doit utiliser.

Exercices

Exercice 1 - Facile

Simplifiez tous les appels aux fonctions de l'ensemble des fichiers, de sorte que toutes les fonctions soient appelées uniquement par leur nom au lieu de leur chemin complet. Comme vu, seules les fonctions `perimeter` ne pourront être simplifiées.

Exercice 2 - Moyen

En relisant bien ce qui a été dit sur cette partie, il est possible de simplifier quelque peu un appel à `perimeter`. Simplifiez donc un appel à cette fonction.

Chapitre 4
Programmation orientée objet

La programmation orientée objet est le fondement de tout développement haut niveau moderne. On pourrait consacrer un livre entier à la programmation orientée objet tellement le sujet est vaste. Le but de ce chapitre n'est donc pas de vous présenter l'intégralité de ce concept, mais seulement de vous donner les connaissances indispensables à la poursuite de cet ouvrage.

Attention, ce chapitre peut paraitre long, et rempli de définitions ; cependant, assurez-vous de bien comprendre l'intégralité de ce qui est dit ici, car c'est la base de toute la suite de ce livre.

Définitions

Objets, classes et instances

Un objet, c'est une entité régie par des propriétés et des comportements. Si l'on a choisi le terme objet, c'est bien parce que ce concept peut parfaitement représenter les objets physiques qui nous entourent. Ainsi, prenons l'exemple d'une voiture. Imaginons une voiture de sport rouge et une petite voiture citadine bleue. Si on les représente comme des entités régies par des propriétés et des comportements, on peut définir ces objets voitures comme suit :

- Voiture de sport rouge
  - Propriétés
    - **Couleur** : rouge
    - **Prix** : 120 000 €
    - **État** : comme neuve
    - **Propriétaire** : Monsieur Dupont
    - **Essence restante** : 33 L
  - Comportement
    - **Démarrer en trombe** : Appuyer sur l'accélérateur à fond
    - **Acheter** : Changer le nom du propriétaire
    - **Faire le plein** : L'essence restante passe à 50 L
- Voiture citadine bleue
  - Propriétés
    - **Couleur** : bleue
    - **Prix** : 1 500 €
    - **État** : cabossée
    - **Propriétaire** : Monsieur Durant
    - **Essence restante** : 43 L
  - Comportement

- **Démarrer en trombe** : Appuyer sur l'accélérateur à fond
- **Acheter** : Changer le nom du propriétaire
- **Faire le plein** : L'essence restante passe à 50 L

Ainsi, on peut voir que la liste des comportements et des propriétés est commune à nos deux voitures. Chacune des deux a une couleur, un prix, un état, et a la possibilité de démarrer en trombe ou d'être achetée. De cette manière, la liste des propriétés et des comportements va définir ce que l'on appelle une **classe**. Ainsi, on peut définir la **classe** `Voiture` comme étant un objet avec une couleur, un prix, un état et pouvant démarrer en trombe.

Par contre, les deux voitures ne sont pas les mêmes. Il s'agit de deux objets distincts de la classe `Voiture`, ayant chacune des valeurs distinctes pour ses propriétés. Si j'active le comportement `Acheter` de la voiture de sport rouge, seule celle-ci doit voir le nom de son propriétaire changer.

Par conséquent, chacune de ces voitures est une **instance** de la classe `Voiture`.

Propriétés et Méthodes

Nous avons déjà défini ce qu'étaient les **propriétés** d'une classe. Les **méthodes** d'une classe ne sont que ce que nous avons appelé les comportements de cette classe. Nous utiliserons donc les termes **méthodes** et **propriétés** dans la suite de ce livre.

Null, Nullables et Non Nullables

Imaginons maintenant une classe `Personne`. Cette classe définit une personne, et pour faire simple, cette classe ne possède que deux propriétés : `voiture`, qui contient l'instance de la classe `Voiture` que possède cette personne, et `date de naissance`, qui contient l'instance de la classe `Date` correspondant à la date de naissance de la personne.

Ainsi, il m'est possible pour une personne donnée de connaitre la couleur de sa voiture. Malheureusement, tout n'est pas aussi simple, car il se peut qu'une personne ne possède pas de voiture. Dans ce cas, on dira que la valeur de la propriété `voiture` est **null**. Comme on peut affecter **null** à la propriété `voiture`, on dira que cette dernière est **nullable**.

Par contre, toute personne ayant une date de naissance, il n'est pas possible pour une personne donnée que la valeur de `date de naissance` soit **null**. Ainsi, on dira que la propriété `date de naissance` est **non nullable**.

Héritage

Imaginons maintenant une classe `Véhicule` regroupant tous les types de véhicules. Cette classe va avoir des propriétés, comme par exemple une couleur, un prix, etc., et des méthodes comme Démarrer en trombe, etc. Par contre, cette classe n'aura pas de propriété Essence restante ou de méthode Faire le plein. Cela n'aurait pas de sens par exemple pour un vélo.

Par contre, la classe `Voiture` possède bien toutes les propriétés et toutes les méthodes de la classe `Véhicule`, mais possède en plus des propriétés et des méthodes qui sont propres aux voitures, comme Faire le plein. On dit alors que la classe `Voiture` **hérite** de la classe `Véhicule`, ou qu'elle l'**étend**. Si bien qu'une instance de la classe `Voiture` pourra très bien être considérée comme une instance de la classe `Véhicule` *(La réciproque est fausse, car si l'on a une instance de la classe `Véhicule`, alors il n'est pas possible de savoir de quel type de véhicule il s'agit, on ne peut donc pas considérer une instance de la classe `Véhicule` comme une instance de la classe `Voiture`)*.

De plus, il est possible d'imbriquer plusieurs niveaux de classe. Ainsi, on pourra avoir une classe `Véhicule Motorisé` qui **hérite** de `Véhicule`. On peut ensuite imaginer que `Voiture` **hérite** de `Véhicule Motorisé`.

Par contre, une classe ne peut avoir qu'une seule classe parente directe. Si `Voiture` **hérite** de `Véhicule Motorisé`, alors elle ne peut **hériter** que de cette classe.

Interfaces et implémentations

Une **interface**, c'est un groupe contenant plusieurs méthodes, sans définir ce que font ces méthodes.

Pour reprendre l'exemple utilisé précédemment, on peut définir une **interface** `Objet Roulant`. La seule chose que l'on puisse dire d'un `Objet Roulant`, c'est qu'il possède la méthode `rouler`. Par contre, il est impossible de définir ce que `rouler` veut dire pour un `Objet Roulant`. Imaginons deux objets aussi variés qu'une bille et une voiture…

Par contre, on peut dire de l'objet `Voiture` qu'elle hérite de `Véhicule Motorisé`, et qu'elle **implémente** `Objet Roulant`. L'objet `Voiture` devra donc définir ce que la méthode `rouler` fait exactement (faire tourner les quatre roues par exemple).

Un objet peut **implémenter** autant d'**interfaces** qu'on le souhaite.

Place à la pratique

Nous allons désormais illustrer tous ces concepts théoriques à travers du code Kotlin. Pour ce faire, nous allons développer un programme en se basant sur les exemples des chapitres précédents, avec des calculs sur les formes géométriques.

Classe

En Kotlin, il est d'usage de définir une classe par fichier, et un fichier par classe. Ainsi, on donne au fichier le nom de la classe. Commençons par définir une classe `Triangle` dans un fichier `Triangle.kt`.

En Kotlin, pour définir une classe, on utilise le mot clé `class`, suivi du nom de la classe, et enfin d'accolades permettant de délimiter le contenu de la classe.

Triangle.kt

```
package geometry.triangle

/**
 * This class represent what a triangle is
 */
class Triangle{
   // Content of the class triangle
}
```

Propriétés

Nous pouvons maintenant définir les propriétés de notre classe. Les propriétés sont des variables appartenant à la classe. Nous allons définir trois propriétés, une pour chaque côté du triangle :

Triangle.kt

```
package geometry.triangle

/**
 * This class represent what a triangle is
 */
class Triangle{
   /** First side of the triangle */
   val side1:Int
   /** Second side of the triangle */
   val side2:Int
   /** Third side of the triangle */
   val side3:Int
}
```

Méthodes

Nous pouvons maintenant définir une méthode. Les méthodes ne sont rien d'autre que des fonctions de la classe. Nous allons donc définir une méthode pour calculer le périmètre du triangle :

Triangle.kt

```
package geometry.triangle

/**
 * This class represent what a triangle is
 */
class Triangle{
   /** Length of the first side of the triangle */
   val side1Length:Double
   /** Length of the second side of the triangle */
   val side2Length:Double
   /** Length of the third side of the triangle */
   val side3Length:Double

   /**
    * Returns the perimeter of the triangle.
    * @return the perimeter of the triangle
    */
   fun perimeter():Double{
       return side1Length+side2Length+side3Length
   }

   /**
    * Returns the area of the triangle.
    * @return the area of the triangle
    */
   fun area():Double{
      val halfPerimeter = perimeter()/2.0
      // Using area formula square of area is p(p-a)(p-b)(p-c) when p is half of the perimeter
      val squareArea = halfPerimeter*
                       (halfPerimeter-side1Length)*
                       (halfPerimeter-side2Length)*
                       (halfPerimeter-side3Length)
      return Math.sqrt(squareArea)
   }
}
```

Instances et appel aux méthodes

Nous allons maintenant créer notre programme principal dans un fichier nommé `Geometry.kt`. Ce programme se contentera de créer une instance de la classe `Triangle` et d'afficher le périmètre d'un triangle.

En Kotlin, pour créer une instance d'une classe, on utilise le nom de la classe suivi de parenthèses.

Pour appeler une méthode, on utilise le point `.` entre le nom de la variable, et le nom de la méthode que l'on souhaite appeler.

Ainsi, le code de notre fichier `Geometry.kt` est :

Geometry.kt

```
package geometry

import geometry.triangle.Triangle

/**
 * Main function of the program.
 */
fun main(args: Array<String>) {
   val triangle = Triangle()
   println("Perimeter of the triangle is: "+triangle.perimeter())
   println("Area of the triangle is: "+triangle.area())
}
```

Essayez de compiler ce programme, vous obtiendrez une erreur :

```
geometry\triangle\Triangle.kt:8:5: error: property must be initialized or be abstract
    val side1Length:Double
    ^
geometry\triangle\Triangle.kt:10:5: error: property must be initialized or be abstract
    val side2Length:Double
    ^
geometry\triangle\Triangle.kt:12:5: error: property must be initialized or be abstract
    val side3Length:Double
    ^
```

En fait, Kotlin nous dit que les valeurs de nos trois côtés n'ont pas été spécifiées. En fait, ces valeurs sont non nullables par défaut, il nous est donc impératif de leur donner une valeur *(Nous verrons plus tard dans ce chapitre comment spécifier qu'une valeur est nullable en Kotlin)*.

Constructeur

Observez à nouveau comment nous avons créé une instance de `Triangle` : en écrivant `Triangle()`. Cette notation ne vous rappelle rien ? Elle ressemble étrangement à un appel à une fonction. C'est justement parce que la création d'une instance est une fonction ; une fonction qu'il n'est pas indispensable de définir, c'est pourquoi cet appel est correct. Mais nous pouvons, si nous le souhaitons, définir cette fonction.

En fait, cette fonction s'appelle un constructeur.

Ce que nous souhaitons, c'est pouvoir donner au constructeur trois arguments, à savoir les longueurs des trois côtés du triangle. Pour ce faire, nous allons définir ces trois arguments juste après le nom de la classe :

Triangle.kt

```
package geometry.triangle

/**
 * This class represent what a triangle is
 * @param side1Length the length of the first side of the triangle to set
 * @param side2Length the length of the second side of the triangle to set
 * @param side3Length the length of the third side of the triangle to set
 */
class Triangle(side1Length:Double, side2Length:Double, side3Length:Double){
    // ...
}
```

Maintenant, il nous faut définir le contenu de la méthode constructeur. Cela se fait en utilisant un bloc `init` délimité par des accolades :

Triangle.kt

```
package geometry.triangle

/**
 * This class represent what a triangle is
 * @param side1Length the length of the first side of the triangle to set
 * @param side2Length the length of the second side of the triangle to set
 * @param side3Length the length of the third side of the triangle to set
 */
class Triangle(side1Length:Double, side2Length:Double, side3Length:Double){
    // ...
    init{
       // content of constructor method
    }
}
```

Mot clé `this`

En fait, ce que nous voulons pour notre constructeur, c'est que la valeur de l'argument `side1Length` du constructeur soit affectée à la propriété `side1Length` de la classe `Triangle`.

Dans une méthode, si l'on utilise `side1Length`, la méthode va, dans l'ordre :

- Utiliser l'argument `side1Length` de la méthode ;
- Si la méthode ne possède pas d'argument de ce nom, comme dans la fonction `perimeter`, la méthode utilisera la propriété `side1Length` ;
- Si `side1Length` n'est ni le nom d'un argument, ni le nom d'une propriété, alors le compilateur refusera de compiler le programme.

Mais comment faire alors, dans le constructeur, pour utiliser la référence à la propriété `side1Length` ? Dans ce cas, nous pouvons utiliser le mot clé `this`, qui fait référence à l'instance de l'objet actuel. Ainsi, `this.side1Length` fera référence à la propriété du même nom, et non à l'argument de la méthode.

Nous savons désormais comment définir notre constructeur :

Triangle.kt

```
package geometry.triangle

class Triangle(side1Length:Double, side2Length:Double, side3Length:Double){
   // ...
   init{
       this.side1Length = side1Length
       this.side2Length = side2Length
       this.side3Length = side3Length
   }
}
```

Nous n'avons plus qu'à ajouter ces trois arguments lors de l'instanciation de la classe `Triangle` dans le fichier `Geometry.kt` :

Triangle.kt

```
package geometry

import geometry.triangle.Triangle

/**
 * Main function of the program.
 */
fun main(args: Array<String>) {
   val triangle = Triangle(3.0, 4.0, 5.0)
   println("Perimeter of the triangle is: "+triangle.perimeter())
   println("Area of the triangle is: "+triangle.area())
}
```

Vous pouvez désormais compiler le programme, et l'exécuter pour observer un résultat semblable à nos attentes :

```
Perimeter of the triangle is: 12.0
Area of the triangle is: 6.0
```

Héritage

Par défaut, en Kotlin, il n'est pas possible d'étendre une classe. Ainsi, vous ne pouvez pas créer de classe qui étende `Triangle` par défaut. Pour modifier ce comportement par défaut, il suffit d'ajouter le mot clé `open` avant le mot clé `class` de la classe que l'on souhaite voir étendue. Ainsi, notre classe `Triangle` devient :

Triangle.kt

```
package geometry.triangle

open class Triangle(side1Length:Double, side2Length:Double, side3Length:Double){
   // ...
}
```

Désormais, nous pouvons définir une classe qui étende la classe `Triangle`. C'est ce que nous allons voir tout de suite en créant une classe `RightTriangle` (triangle rectangle). Pour spécifier qu'une classe étend une autre classe, on fait suivre le nom de la classe de deux points `:`, suivi du nom de la classe parente :

RightTriangle.kt

```
package geometry.triangle

/**
 * This class represent what a right triangle is
 */
class RightTriangle:Triangle{

}
```

Pour le moment, la classe `RightTriangle` a exactement les mêmes propriétés et les mêmes méthodes que `Triangle`. Mais nous allons la modifier.

Surcharge du constructeur

La surcharge, en objet, consiste à redéfinir une méthode déjà présente dans la classe parente. Nous allons surcharger le constructeur, afin qu'il ne prenne que deux arguments (la longueur des deux côtés formant un angle droit). Nous calculerons ensuite la longueur du troisième côté (l'hypoténuse) à partir de la longueur des deux autres.

Pour ce faire, nous allons définir les arguments du constructeur de `RightTriangle` comme nous l'avons fait pour `Triangle`, puis nous allons faire l'appel au constructeur de `Triangle` (donc avec trois arguments) à partir des deux valeurs du constructeur de `RightTriangle`.

RightTriangle.kt

```
package geometry.triangle

/**
 * This class represent what a right triangle is
 */
class RightTriangle constructor(side1Length:Double, side2Length:Double):
       Triangle(side1Length,
                side2Length,
                // Third length is square root of the sum of square of lengths
                // of both other sides
                Math.sqrt(Math.pow(side1Length, 2.0)+Math.pow(side2Length, 2.0))
               ){

}
```

Retenez que Math.sqrt(x) calcule la racine carrée de x, et Math.pow(x, 2.0) calcule le carré du nombre x.
Notez bien l'intérêt d'aérer votre code et d'ajouter des commentaires dès que celui-ci devient complexe.

Surcharge de méthode

Nous avons vu comment surcharger le constructeur, mais nous pouvons tout aussi bien surcharger des méthodes.

Prenons par exemple la méthode `area()` de la classe `Triangle`, nous pourrions avoir envie de la surcharger pour utiliser un calcul beaucoup plus simple dans `RightTriangle`.

Pour cela, comme nous l'avons fait pour la classe `Triangle`, nous devons préciser que la méthode `area()` peut être surchargée. Pour ce faire, on ajoute toujours le mot clé `open` devant :

Triangle.kt

```
open fun area():Double{
   // ...
}
```

Dans la classe «enfant», pour surcharger une méthode, on se contente de définir une méthode comportant le même identifiant, la même valeur de retour et des arguments de même type dans le même ordre, en faisant précéder la définition du mot clé `override` :

RightTriangle.kt

```
/**
 * Returns the area of the right triangle.
 * @return the area of the right triangle
 */
override fun area():Double{
    return side1Length * side2Length / 2.0
}
```

Principe d'héritage

L'un des grands avantages de l'héritage, c'est qu'un objet de type `RightTriangle` peut être considéré comme un objet de type `Triangle`.

Considérons alors que nous écrivons la fonction suivante dans `Geometry.kt` :

Geometry.kt

```
/**
 * Displays the perimeter and the area of [triangle]
 * @param triangle the triangle for which to display the perimeter and the area
 */
fun displayTriangleData(triangle:Triangle){
   println("Perimeter of the triangle is: "+triangle.perimeter())
   println("Area of the triangle is: "+triangle.area())
}
```

En argument de cette fonction, je peux passer aussi bien un argument de type `Triangle`, qu'un argument d'un type qui hérite de triangle, comme `RightTriangle`.

Ainsi, écrire la fonction main suivante est tout à fait correct :

Geometry.kt

```
/**
 * Main function of the program.
 */
fun main(args: Array<String>) {
   val triangle = Triangle(3.0, 4.0, 6.0)
   val rightTriangle = RightTriangle(3.0, 4.0)

   displayTriangleData(triangle)
   // Passing RightTriangle, which will be considered as Triangle
   displayTriangleData(rightTriangle)
}
```

On peut compiler ce programme, et le résultat produit lors de l'exécution sera celui attendu :

```
Perimeter of the triangle is: 13.0
Area of the triangle is: 5.332682251925386
Perimeter of the triangle is: 12.0
Area of the triangle is: 6.0
```

Interface

Toutes les formes planes (considérons le cas général) ont un périmètre et une aire. Nous pouvons donc définir une interface `PlaneShape` contenant deux méthodes, `perimeter()` et `area()`.

Par contre, il n'existe pas de formule pour calculer l'aire ou le périmètre d'une forme plane. On ne peut donc pas définir le contenu de ces méthodes, mais on peut définir qu'elles existent. On appelle cela des méthodes abstraites. La définition d'une méthode abstraite est la même que celle d'une méthode classique, à la différence près qu'elle ne comportera pas d'accolades puisqu'il n'y a pas de contenu.

Aussi, nous pouvons écrire une interface `PlaneShape` comme suit :

PlaneShape.kt

```kotlin
package geometry

/**
 * Interface describing a geometrical plane shape.
 */
interface PlaneShape{
    /**
     * Returns the perimeter of the shape.
     * @return the perimeter of the shape
     */
    fun perimeter():Double

    /**
     * Returns the area of the shape.
     * @return the area of the shape
     */
    fun area():Double
}
```

Nous pouvons maintenant déclarer que la classe `Triangle` implémente l'interface `PlaneShape`, tout comme pour l'héritage, en ajoutant : suivi du nom de l'interface qu'il implémente.

Triangle.kt

```
package geometry.triangle

import geometry.PlaneShape

/**
 * This class represent what a triangle is
 * @param side1Length the length of the first side of the triangle to set
 * @param side2Length the length of the second side of the triangle to set
 * @param side3Length the length of the third side of the triangle to set
 */
open class Triangle(side1Length:Double, side2Length:Double, side3Length:Double):PlaneShape{
// ...
}
```

Toutes les méthodes d'une interface sont surchargeables. Nous pouvons donc supprimer le mot clé `open` de la méthode `area()` de la classe `Triangle`, puisque cette méthode est déjà considérée par `open`, étant donné qu'elle est définie dans l'interface `PlaneShape`.

Il ne nous reste plus qu'à ajouter le mot clé `override` devant les méthodes `perimeter()` et `area()` pour préciser qu'elles surchargent les méthodes de l'interface, et l'implémentation est terminée.

Petite parenthèse technique

Très bien, notre classe `Triangle` implémente `PlaneShape`, mais quelle est l'utilité d'une telle implémentation ? Ajoutons une classe `Circle` implémentant `PlaneShape` et définissant la méthode `perimeter()` pour en comprendre l'utilité.

Circle.kt

```
package geometry.circle

import geometry.PlaneShape

/**
 * This class represent what a circle is
 * @param diameter the length of the diameter of the circle to set
 */
open class Circle(diameter:Double):PlaneShape{
   /** Length of the diameter of the circle */
   val diameter:Double

   init{
       this.diameter = diameter
   }

   /**
    * Returns the circumference of the circle.
    * @return the circumference of the circle
    */
   override fun perimeter():Double{
       return diameter*Math.PI
   }
}
```

Essayons de compiler ce programme, le résultat sera le suivant :

```
geometry\circle\Circle.kt:9:6: error: class 'Circle' is not abstract and does not implement ab
stract member public abstract fun area(): Double defined in geometry.PlaneShape
open class Circle(diameter:Double):PlaneShape{
```

Personnellement, je ne comprends pas d'où vient cette erreur (en réalité, si, mais faisons comme si je ne savais pas d'où venait cette erreur).

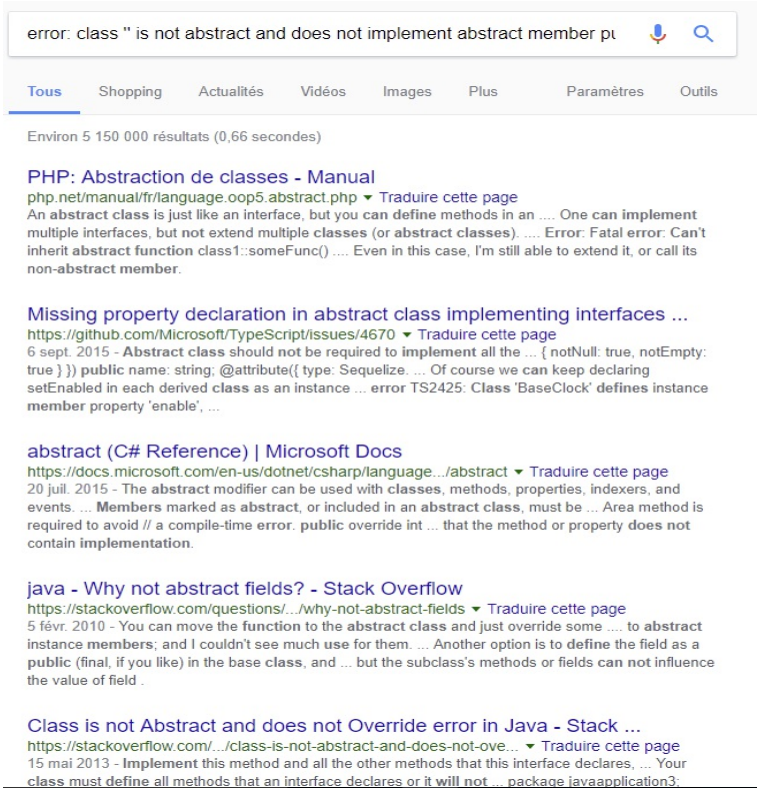
Il va donc falloir que j'aille sur Internet chercher ce que cette erreur peut bien vouloir dire. Je vais donc copier le message d'erreur, j'obtiens :

error: class 'Circle' is not abstract and does not implement abstract member public abstract fun area(): Double defined in geometry.PlaneShape

À partir de là, je vais supprimer de ce message d'erreur tout ce qui est propre au programme que j'ai développé, comme les noms des classes, des méthodes et des interfaces :

error: class '' is not abstract and does not implement abstract member public abstract fun(): defined in

Très bien, maintenant, il ne reste plus qu'à copier/coller ce message dans Google :

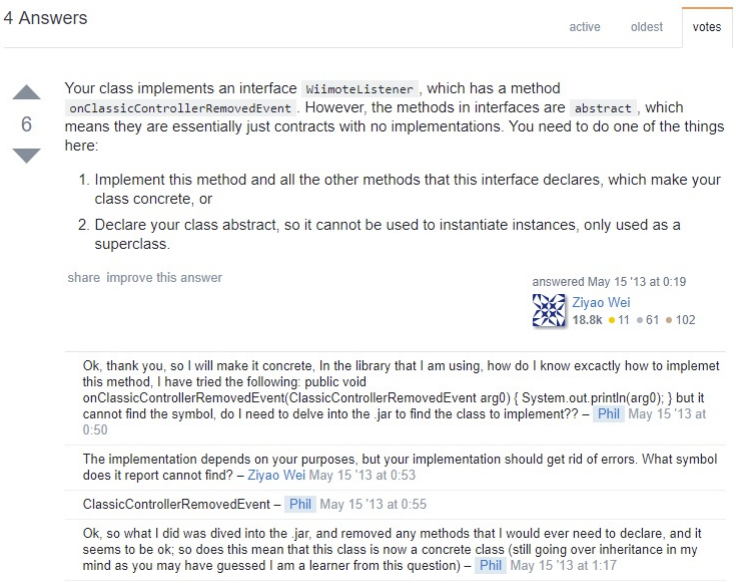


Regardons dans l'ordre :

- Le premier résultat concerne du PHP, donc peu de chances que j'y trouve mon bonheur ;
- Le second résultat ne me dit rien, en plus dans l'URL, je vois du Microsoft, faible probabilité que j'y décèle une réponse ;
- Le troisième résultat concerne du C#, éventualité fragile qu'il m'apporte une solution ;
- Le quatrième résultat est à propos de Java, or, nous avons vu au début de ce livre que Kotlin était basé sur du Java, cela peut m'intéresser ;
- Le cinquième résultat concerne aussi du Java, et je retrouve dans le titre un message vraiment similaire à mon message d'erreur. Il y a de très fortes chances que j'y apprenne quelque chose d'utile.

Je clique donc sur le cinquième lien. Il s'agit d'une question sur StackOverflow. Si vous vous mettez sérieusement au développement, le site StackOverflow va devenir très rapidement votre meilleur ami. Il s'agit d'un site de questions/réponses à propos de développement.

On va passer sur la question, car rien que le titre m'indique que la personne ayant posé la question a exactement le même problème que moi, je vais donc directement passer à la réponse :



4 Answers

active oldest votes

6

Your class implements an interface `WiimoteListener`, which has a method `onClassicControllerRemovedEvent`. However, the methods in interfaces are `abstract`, which means they are essentially just contracts with no implementations. You need to do one of the things here:

1. Implement this method and all the other methods that this interface declares, which make your class concrete, or
2. Declare your class abstract, so it cannot be used to instantiate instances, only used as a superclass.

share improve this answer

answered May 15 '13 at 0:19
Ziyao Wei
18.8k ● 11 ● 61 ● 102

Ok, thank you, so I will make it concrete, In the library that I am using, how do I know excactly how to implemet this method, I have tried the following: public void onClassicControllerRemovedEvent(ClassicControllerRemovedEvent arg0) { System.out.println(arg0); } but it cannot find the symbol, do I need to delve into the .jar to find the class to implement?? – Phil May 15 '13 at 0:50

The implementation depends on your purposes, but your implementation should get rid of errors. What symbol does it report cannot find? – Ziyao Wei May 15 '13 at 0:53

ClassicControllerRemovedEvent – Phil May 15 '13 at 0:55

Ok, so what I did was dived into the .jar, and removed any methods that I would ever need to declare, and it seems to be ok; so does this mean that this class is now a concrete class (still going over inheritance in my mind as you may have guessed I am a learner from this question) – Phil May 15 '13 at 1:17

On va s'intéresser à une partie seulement de la réponse :

> You need to do one of the things here:
>
> 1. Implement this method and all the other methods that this interface declares

Ainsi, nous avons déjà une piste pour savoir d'où vient notre erreur. Il semblerait qu'il faille que nous implémentions toutes les méthodes de notre interface. Or, dans la classe `Circle`, nous avons bien implémenté la méthode `perimeter()`, mais pas la méthode `area()`.

D'ailleurs, à y regarder de plus près, c'est bien ce que nous disait exactement le message d'erreur :

```
geometry\circle\Circle.kt:9:6: error: class 'Circle' is not abstract and does not implement ab
stract member public abstract fun area(): Double defined in geometry.PlaneShape
open class Circle(diameter:Double):PlaneShape{
```

Trois choses à retenir de cette petite parenthèse :

- Vous ne trouverez pas tout sur tout dans ce livre. C'est la raison pour laquelle j'ai trouvé indispensable de vous montrer comment résoudre seul un problème que l'on rencontre. Par exemple, vous pourriez essayer de chercher d'où vient le `Math.PI` que j'utilise dans le code sans l'avoir expliqué.
- Certains logiciels de développement proposent une configuration en français, ou bien adaptent les raccourcis clavier à un clavier français. Fuyez ces options, garder tout en anglais permet de trouver de meilleures réponses lorsque l'on cherche une information.
- Sur StackOverflow, vous pouvez vous aussi poser des questions si vous ne trouvez pas votre bonheur après avoir cherché sur Internet. En vous habituant à développer en anglais, vous pourrez partager votre code sur ce site, ce qui accélèrera les réponses que vous obtiendrez.

Nous pouvons donc maintenant corriger notre classe Circle :

Circle.kt

```
package geometry.circle

import geometry.PlaneShape

/**
 * This class represent what a circle is
 * @param diameter the length of the diameter of the circle to set
 */
open class Circle(diameter:Double):PlaneShape{
   /** Length of the diameter of the circle */
   val diameter:Double

   init{
       this.diameter = diameter
   }

   /**
    * Returns the circumference of the circle.
    * @return the circumference of the circle
    */
   override fun perimeter():Double{
       return diameter*Math.PI
   }

   /**
    * Returns the area of the circle.
    * @return the area of the circle
    */
   override fun area():Double{
       return diameter*Math.pow(Math.PI, 2.0)/2.0
   }
}
```

Utilité des interfaces

Nous avons donc implémenter notre interface dans deux classes distinctes. Mais quelle en est l'utilité ?

De la même façon que nous avons pu utiliser une variable de type `Triangle` dans `Geometry.kt` pour calculer l'aire et le périmètre aussi bien d'un `RightTriangle` que d'un `Triangle`, nous pourrions avoir envie de faire la même chose avec `PlaneShape`.

Dans `Geometry.kt`, remplacez la fonction `displayTriangleData` par la fonction suivante :

Geometry.kt

```
/**
 * Displays the perimeter and the area of [planeShape]
 * @param planeShape the plane shape for which to display the perimeter and the area
 */
fun displayPlaneShapeData(planeShape:PlaneShape){
   println("Perimeter of the plane shape is: "+planeShape.perimeter())
   println("Area of the plane shape is: "+planeShape.area())
}
```

On voit ici que l'on peut passer n'importe quelle variable d'une classe implémentant `PlaneShape` à cette fonction :

Geometry.kt

```
package geometry

import geometry.PlaneShape
import geometry.circle.Circle
import geometry.triangle.Triangle
import geometry.triangle.RightTriangle
import java.util.Date

/**
 * Main function of the program.
 */
fun main(args: Array<String>) {
   val triangle = Triangle(3.0, 4.0, 6.0)
   val rightTriangle = RightTriangle(3.0, 4.0)
   val circle = Circle(5.0)

   displayPlaneShapeData(triangle)
   displayPlaneShapeData(rightTriangle)
   displayPlaneShapeData(circle)
}

/**
 * Displays the perimeter and the area of [planeShape]
 * @param planeShape the plane shape for which to display the perimeter and the area
 */
fun displayPlaneShapeData(planeShape:PlaneShape){
   println("Perimeter of the plane shape is: "+planeShape.perimeter())
   println("Area of the plane shape is: "+planeShape.area())
}
```

Null et Nullables

Imaginons maintenant que la fonction `displayPlaneShapeData()` soit appelée à partir d'une forme dessinée sur une feuille blanche. Il se peut alors qu'il n'y ait rien de dessiné sur cette feuille, auquel cas, on aurait une «absence de forme».

En cas d'absence de forme, il semble logique que l'on ne puisse ni calculer le périmètre, ni calculer l'aire.

En Kotlin, pour définir qu'une variable contient une «absence d'instance de classe», on utilise le mot `null`. Étant donné que `null` s'utilise pour n'importe quel type de variable, nous serons obligé d'utiliser le typage de variable, et ne pourrons pas nous contenter d'un type implicite.

```
val noShape:PlaneShape = null
```

Toutefois, cette déclaration est erronée. Le compilateur refusera de compiler une telle instruction. Comme nous l'avons dit plus haut, en Kotlin, par défaut, les variables sont non nullables. Ainsi, nous ne pouvons pas définir `null` comme valeur d'une variable par défaut.

Pour rendre une variable nullable, donc pour qu'il soit possible de définir sa valeur comme étant `null`, nous devons ajouter un `?` à la suite de son type.

```
val noShape:PlaneShape? = null
```

De la même façon, pour pouvoir passer cette variable à la fonction `displayPlaneShapeData()`, il va falloir définir que l'argument est nullable. De la même façon, il suffit d'ajouter un `?` après le type de l'argument :

Geometry.kt

```
/**
 * Displays the perimeter and the area of [planeShape]
 * @param planeShape the plane shape for which to display the perimeter and the area
 */
fun displayPlaneShapeData(planeShape:PlaneShape?){
   println("Perimeter of the plane shape is: "+planeShape.perimeter())
   println("Area of the plane shape is: "+planeShape.area())
}
```

Mais si vous faites cela, là encore, le compilateur vous renverra une erreur. Le souci, c'est que vous essayez d'appeler `planeShape.perimeter()`, alors que vous avez spécifié que `planeShape` peut être null. Les conditions feront l'objet d'un prochain chapitre, mais nous pouvons déjà voir un petit exemple, sans entrer dans les détails de son explication :

Geometry.kt

```
/**
 * Displays the perimeter and the area of [planeShape]
 * @param planeShape the plane shape for which to display the perimeter and the area
 */
fun displayPlaneShapeData(planeShape:PlaneShape?){
   if(planeShape == null){
       println("No shape, so no perimeter and no area");
   }
   else {
       println("Perimeter of the plane shape is: " + planeShape.perimeter())
       println("Area of the plane shape is: " + planeShape.area())
   }
}
```

Ici, nous indiquons que si `planeShape` est null, alors nous affichons `"No shape, so no perimeter and no area"` , sinon, nous gardons le même comportement qu'avant. Encore une fois, cette notation `if ... else` sera vue au cours d'un prochain chapitre. Ne vous en faites pas si vous ne comprenez pas tout pour le moment.

Ainsi, voici le code final du fichier `Geometry.kt` :

Geometry.kt

```
package geometry

import geometry.PlaneShape
import geometry.circle.Circle
import geometry.triangle.Triangle
import geometry.triangle.RightTriangle
import java.util.Date

/**
 * Main function of the program.
 */
fun main(args: Array<String>) {
   val triangle = Triangle(3.0, 4.0, 6.0)
   val rightTriangle = RightTriangle(3.0, 4.0)
   val circle = Circle(5.0)
   val noShape:PlaneShape? = null

   displayPlaneShapeData(triangle)
   displayPlaneShapeData(rightTriangle)
   displayPlaneShapeData(circle)
   displayPlaneShapeData(noShape)
}

/**
 * Displays the perimeter and the area of [planeShape]
 * @param planeShape the plane shape for which to display the perimeter and the area
 */
fun displayPlaneShapeData(planeShape:PlaneShape?){
   if(planeShape == null){
       println("No shape, so no perimeter and no area");
   }
   else {
       println("Perimeter of the plane shape is: " + planeShape.perimeter())
       println("Area of the plane shape is: " + planeShape.area())
   }
}
```

Une fois compilé, puis exécuté, on obtient bien le résultat attendu :

```
Perimeter of the plane shape is: 13.0
Area of the plane shape is: 5.332682251925386
Perimeter of the plane shape is: 12.0
Area of the plane shape is: 6.0
Perimeter of the plane shape is: 15.707963267948966
Area of the plane shape is: 24.674011002723397
No shape, so no perimeter and no area
```

Conclusion

Nous en savons désormais assez pour pouvoir démarrer le développement Android, qui sera l'objet du prochain chapitre.

Toutefois, nous n'avons vu qu'une très mince partie de la notion d'objets, et de Kotlin. Nous continuerons à aborder de nouvelles notions dans la suite de ce livre.

En Kotlin, tout est objet

Dans le second chapitre, nous avons mis en opposition les types primitifs aux instances d'objets. Cette simplification a été faite dans le but de présenter les variables, les constantes, les opérateurs, etc.

Mais en Kotlin, tout est objet, et chaque variable n'est qu'une instance d'un objet. Ainsi, par exemple, ce que nous avons présenté comme le type primitif `Int` n'est rien d'autre qu'une variable contenant une instance de de la classe `Int`.

Exercices

Exercice 1 : Facile

Ajoutez une classe `Square` à votre application, qui calculera le périmètre et l'aire d'un carré, et pourra l'afficher dans le programme.

Exercice 2 : Difficile

Modifiez l'interface `Shape` de sorte qu'elle contienne une propriété `name`, qui permettra d'afficher le nom de la forme plutôt que simplement `"the plane shape"`.