

Table des matières

Introduction	1.1
1. Premier programme Kotlin	1.2
2. Variables, Types, Constantes et Opérateurs	1.3
3. Sources multiples, paquets et imports	1.4

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 chaîne 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

```
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 chaîne 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 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 paraître 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 connaître 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'écrivons 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 chaîne 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 connaître 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 :

VarTypes.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 subtract = 7-4  
    val divide = 7/3  
    val modulo = 7%3  
  
    println(add)  
    println(multiply)  
    println(subtract)  
    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 subtract = 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(subtract)
    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 chaîne 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 chaîne 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 est pi*diamètre
C'est quand même beaucoup plus clair*/
circumference = circumference(diameter)
```

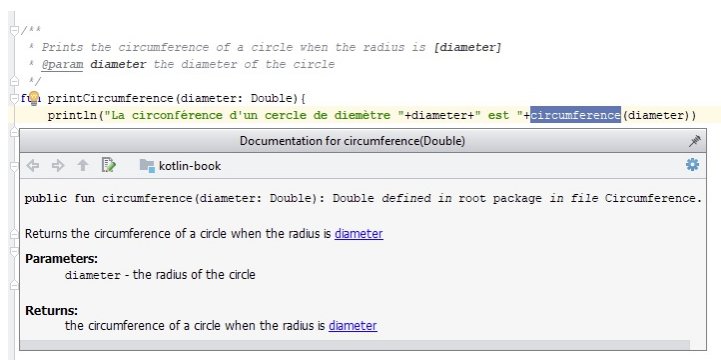
KDoc

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 chaîne 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

```
/** The value of  $\pi$  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

```

/**
 * 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  $\pi$  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 sûrement, 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 souhaitons 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

```
/**
 * 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

```
/** 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

```

/** The value of  $\pi$  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 ajoutons 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

```

/**
    * 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

```

/**
    * 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: Double): Double defined in root package in file EquilateralTriangle.kt, public fun perimeter(sideLength: 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: Double): 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 suspense 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.equilateral.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(sideLength))
}

```

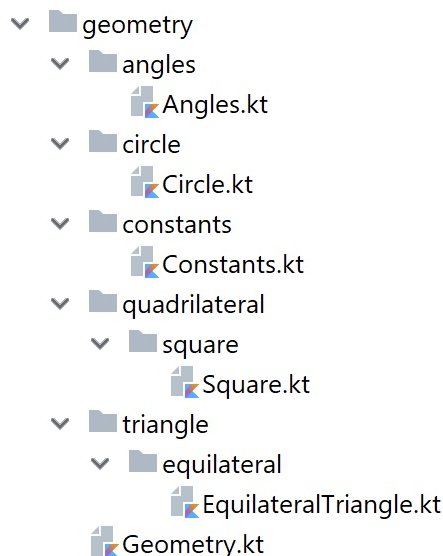
Convention d'arborescence

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.