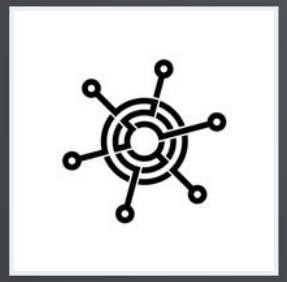
KOTLIN, LES BASES

- Qui je suis
- Qui vous êtes/ce que vous attendez
- Présentation du plan

QUI JE SUIS

- Développeur d'applications mobiles Android
- Formateur Android/Java et Kotlin (scolaires et pros)
- Co-fondateur de Startup Marseille 

QUI ÊTES VOUS ?

Tour de table :

- Prénom
- Expérience (Java, Web, Mobile)
- Attentes

KOTLIN, LES BASES

- Introduction
- Bases de Kotlin

- Les fonctions - Partie 1
- Classes en Kotlin
- Les fonctions - Partie 2
- Délégation
- Generics
- Autres fonctionnalités
- Interopérabilité
- Standard Library
- Programmation asynchrone

INTRODUCTION

- Pourquoi le Kotlin ?
- Introduction à la JVM (Java Virtual Machine)
- Installation des outils REPL de Kotlin (Read Eval Print Loop)
- La structure d'une application Kotlin
- Kotlin et IntelliJ IDEA
- Les conventions utilisées avec Kotlin

POURQUOI LE KOTLIN ?

- Kotlin et Java sont 100 % interopérables.

- 2010 : idée (JetBrains).
- 2015 : naissance, disponible sur GitHub.
- Origine du nom : l'île de Kotlin.
- Concis, sûr, pragmatique et 100 % interopérable avec Java.
- Statiquement typé, mais inférence de types.
- C'est aussi un langage fonctionnel.

POURQUOI LE KOTLIN ?

DATES IMPORTANTES

- 2010 : Lancement du projet Kotlin.
- Juillet 2011 : communication officielle de JetBrain sur le projet Kotlin.
- Février 2012 : mise à disposition du projet en opent source.
- 15 Février 2016 : Kotlin 1.0. Première version stable.
- 28 Novembre 2017 : Kotlin 1.2. Partage du code entre la JVM et la plafeforme Javascript.
- 29 Octobre 2018 : Kotlin 1.3. Coroutines.
- 7 Mai 2019 : Google annonce que Kotlin est le langage préféré pour développer des applications Android.

POURQUOI LE KOTLIN ?

INFÉRENCE DE TYPE

```
1 val number = 123
2 val message = "Hello world !"
3 fun sayHello() = "Hello world !"
```

POURQUOI LE KOTLIN ?

AVANTAGES

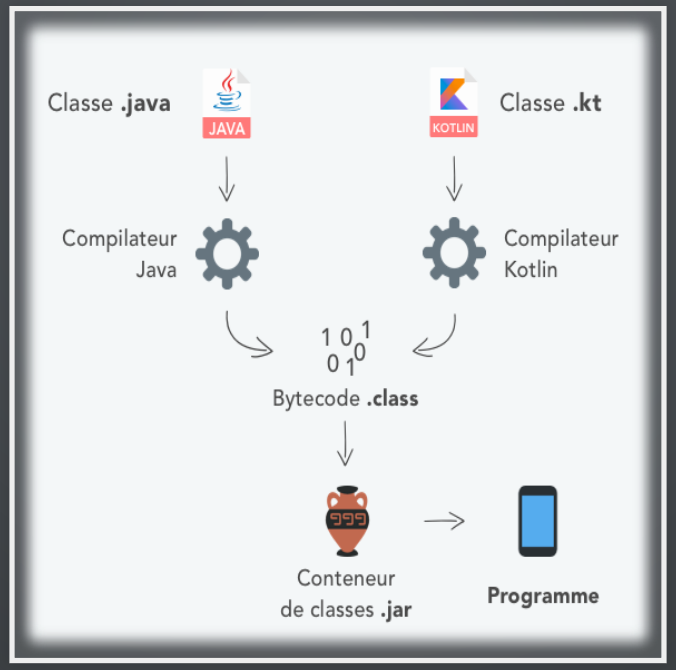
- Performance.
- Fiabilité.
- Efficacité.

POURQUOI LE KOTLIN ?

LANGAGE FONCTIONNEL

- First-class function.
- Immuabilité.
- Aucun effet de bord.

INTRODUCTION À LA JVM (JAVA VIRTUAL MACHINE)



INSTALLATION DES OUTILS REPL DE KOTLIN (READ EVAL PRINT LOOP)

- En ligne de commande : https://kotlinlang.org/docs/tutorials/command-line.html

```
kotlinc-jvm
```

- En ligne : https://play.kotlinlang.org/
- Nombreux exercices en ligne : https://try.kotlinlang.org

LA STRUCTURE D'UNE APPLICATION KOTLIN

LES RÉPERTOIRES

La structure des répetoires suit la structure des packages. Le package racine sera ignoré, par exemple : si le projet est dans le package `org.example.kotlin`, alors les fichiers seront placés directement dans le répertoire racine contenant les sources. Les fichiers dans le package `org.example.kotlin.network.socket` seront placés dans le sous répetoire : `network/socket`.

LA STRUCTURE D'UNE APPLICATION KOTLIN

LES FICHIERS

Un fichier ne contenant qu'une seule classe, sera nomé du nom de celle-ci (en utilisant le camel case), suivit de l'extention `.kt`.
Si le fichier contient plusieurs classes, ou seulement des déclarations top niveau (top level declarations), alors, choisir un nom qui correspond le mieux au contenu du fichier.

LA STRUCTURE D'UNE APPLICATION KOTLIN

DANS LE FICHIER SOURCE

Généralement le contenu d'une classe est organisé de la manière suivante :

- Déclaration des propriétés et des blocs d'initialiseurs
- Les constructeurs secondaires
- Les déclarations des méthodes
- L'objet compagnon

Regrouper les méthodes (classiques et d'extention) ensembles. Gardez une organisation cohérente sur tout le projet.
L'implémentation d'une interface gardera l'ordre de déclaration dans celle-ci.

KOTLIN ET INTELLIJ IDEA

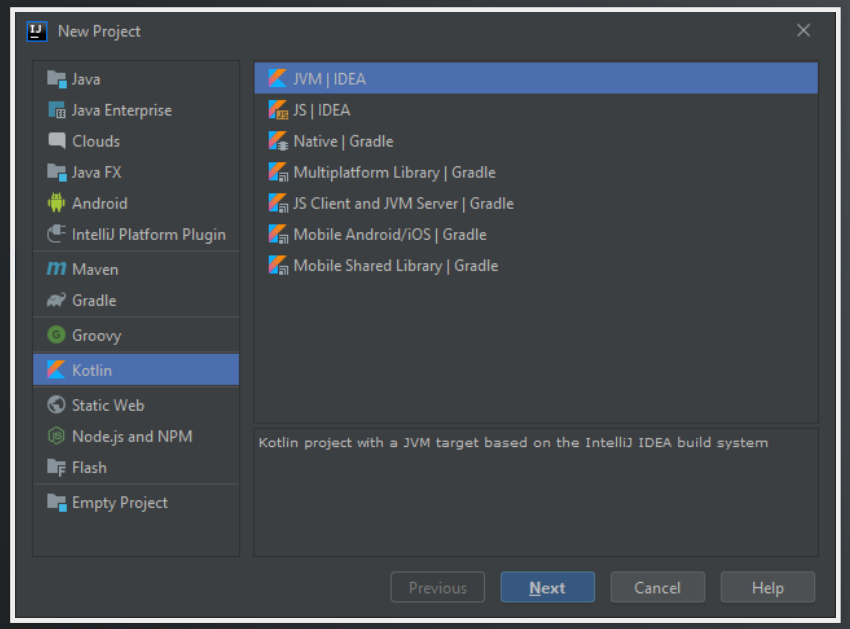
INSTALLATION

Commençons par installer InteliJ : Télécharger

KOTLIN ET INTELLIJ IDEA

CRÉATION DU PROJET

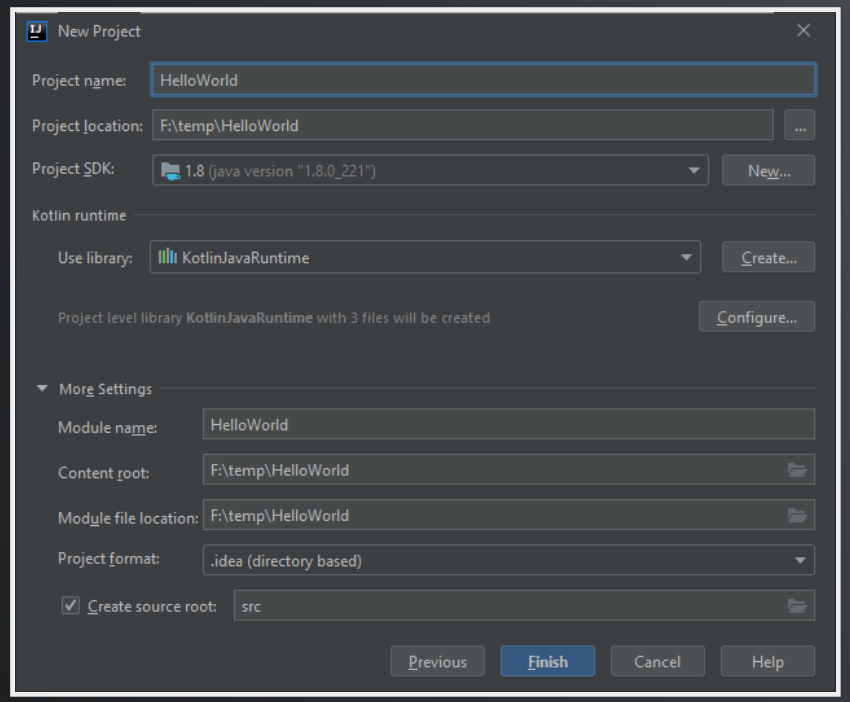
Il est temps de créer notre première application : **File / New / Project**. Sélectionner **Kotlin / JVM | IDEA**.



KOTLIN ET INTELLIJ IDEA

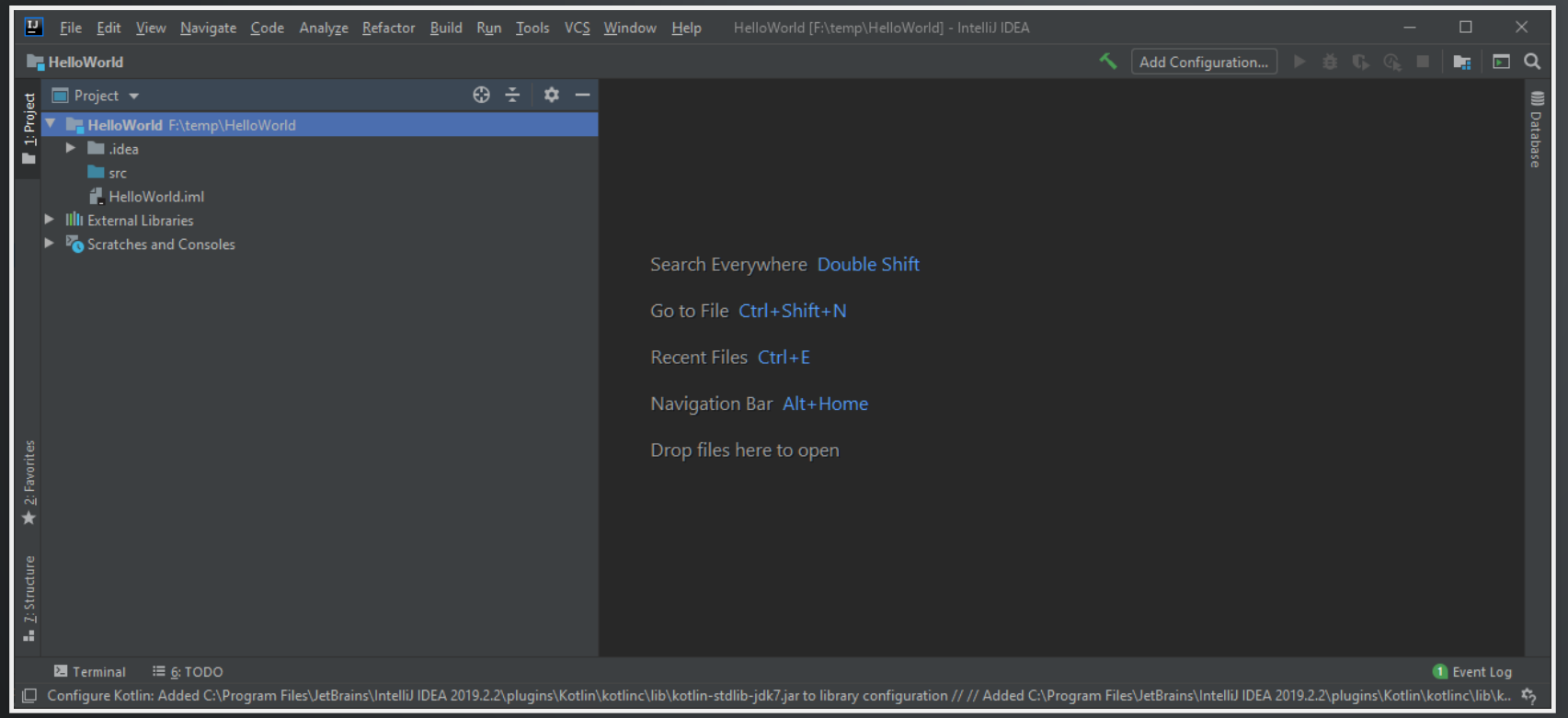
NOMMAGE

Nommons notre projet, par exemple **HelloWorld** :



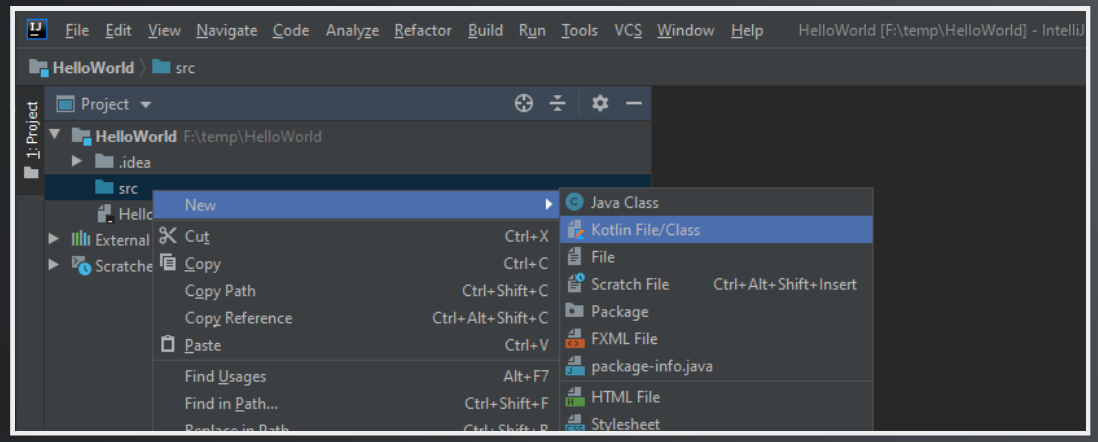
KOTLIN ET INTELLIJ IDEA

Nous devrions obtenir le résultat suivant :



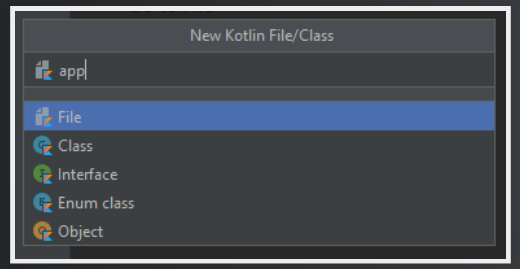
KOTLIN ET INTELLIJ IDEA

Créons un nouveau fichier dans le répertoire source. Click droit, **New / Kotlin File/Class** :



KOTLIN ET INTELLIJ IDEA

Nommons le **app** :



KOTLIN ET INTELLIJ IDEA

Ajoutons la fonction principale main :

Taper `main`, puis la touche **TAB**, pour lancer l'autocomplétion.

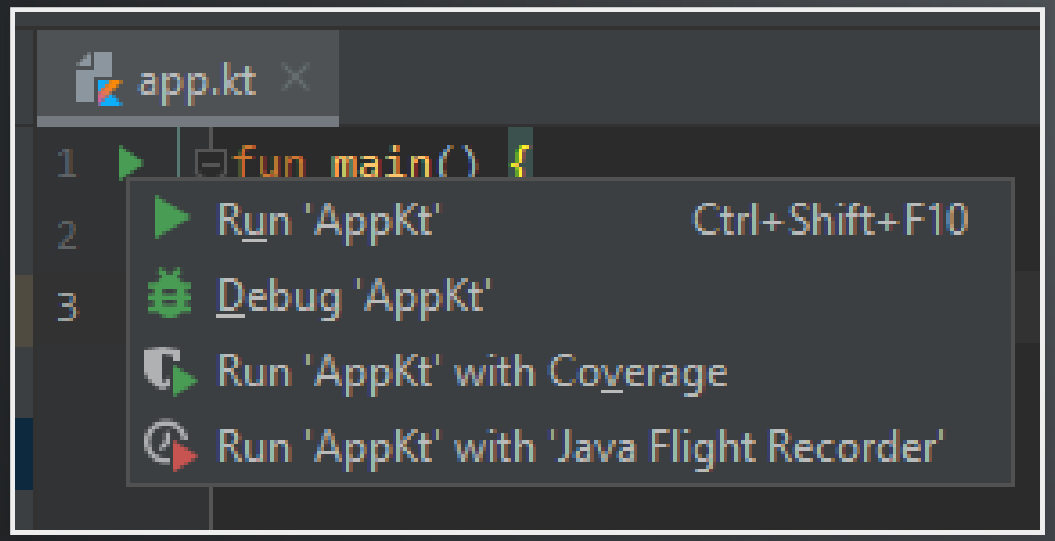
Il reste juste à compléter le corps de la méthode pour obtenir le résultat suivant :

```
1 fun main() {
2     println("Bonjour le monde")
3 }
```

KOTLIN ET INTELLIJ IDEA

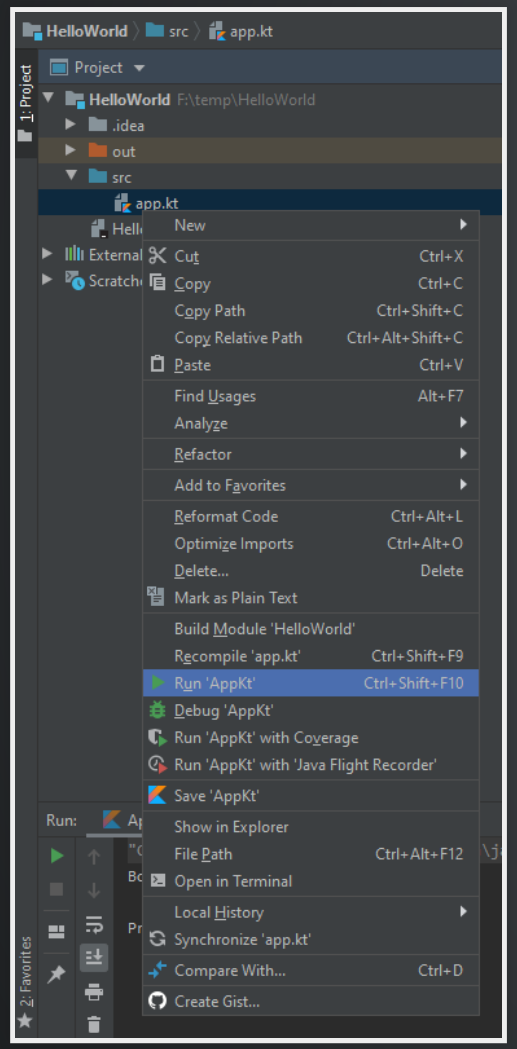
EXÉCUTER LE CODE

Il y a plusieurs façons de lancer le code, la plus rapide est de cliquer sur le bouton vert **Run** :



EXÉCUTER LE CODE

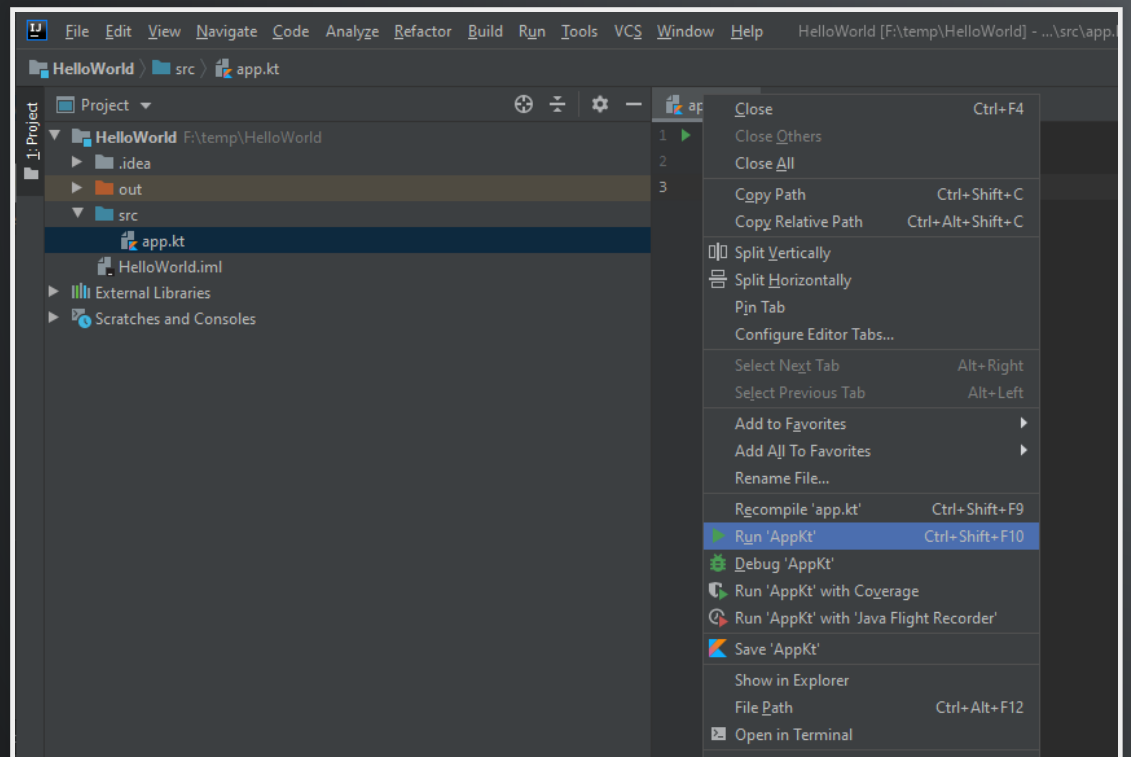
Ou encore :



EXÉCUTER LE CODE

KOTLIN ET INTELLIJ IDEA

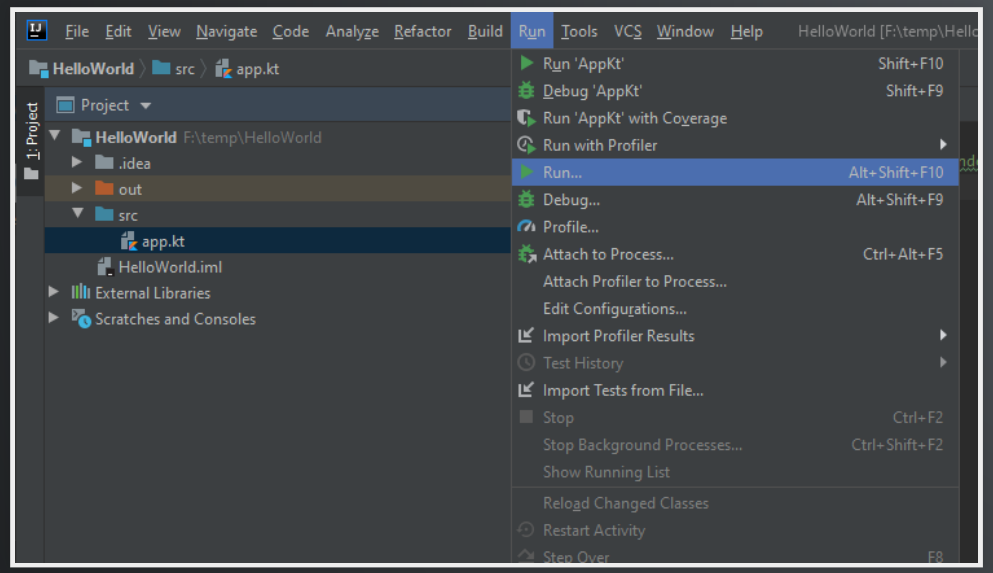
Ou bien :



KOTLIN ET INTELLIJ IDEA

EXÉCUTER LE CODE

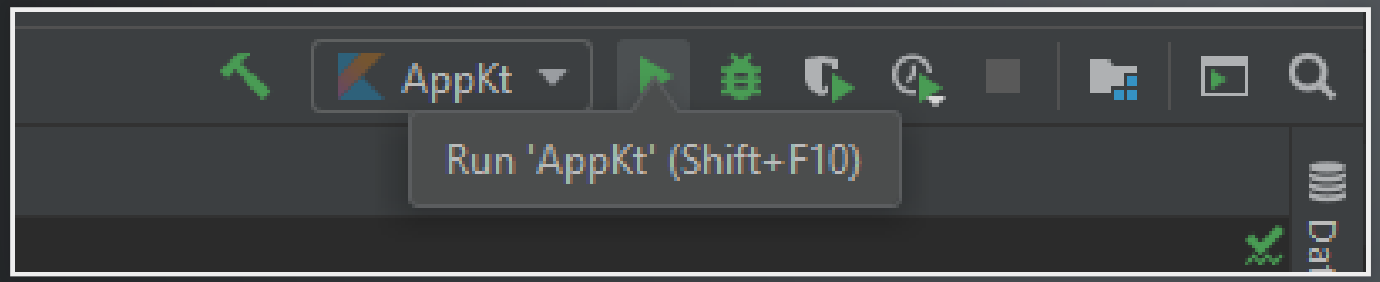
Ou bien encore :



KOTLIN ET INTELLIJ IDEA

EXÉCUTER LE CODE

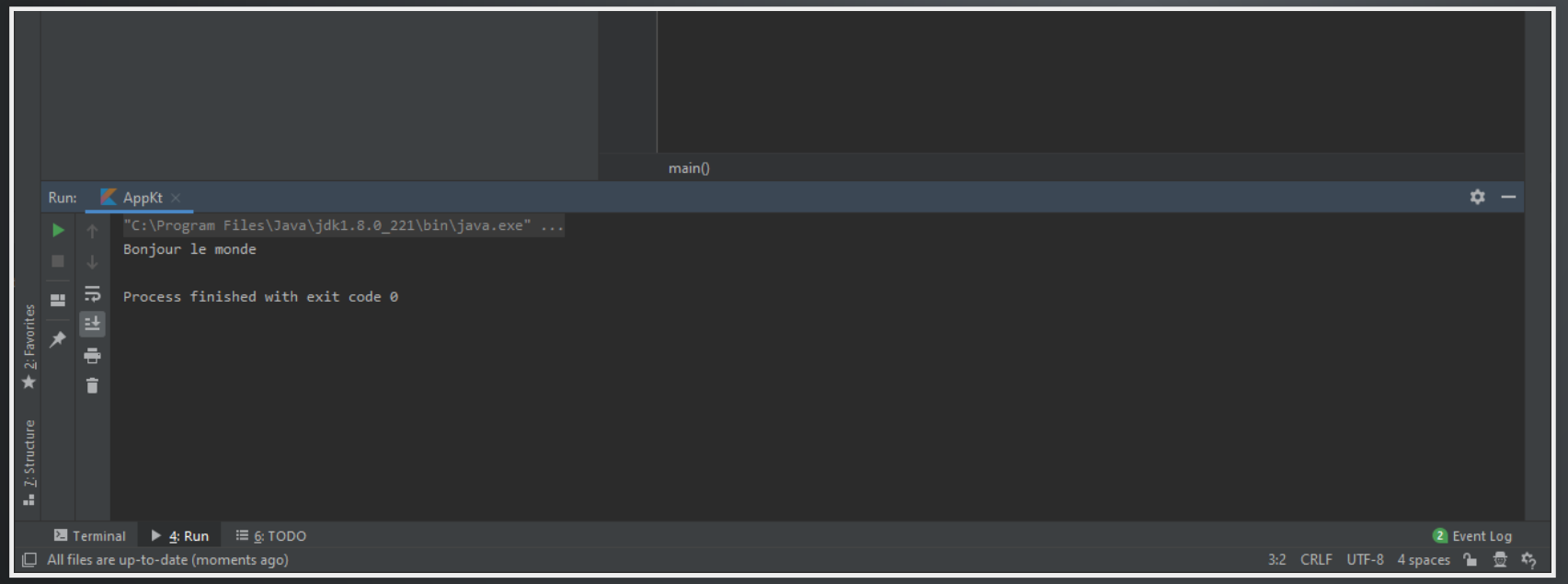
Ce qui permet d'avoir un nouveau raccourcit :



KOTLIN ET INTELLIJ IDEA

EXÉCUTER LE CODE

Ce qui nous donnera le résultat :



KOTLIN ET INTELLIJ IDEA

Nous pouvons aussi directement lancer notre code dans :

- des Scratches : File | New | Scratch file et sélectionner le type Kotlin.
- une fenêtre REPL : Tools | Kotlin | Kotlin REPL (Control + Return pour valider)

LES CONVENTIONS UTILISÉES AVEC KOTLIN

RÈGLES DE NOMMAGE

Elles sont identiques à celles en Java (nom des packages et des méthodes). A une différence prête : le nom des méthodes de fabrique est identique à celui de la classe :

```
1 abstract class Foo { ... }
2
3 class FooImpl : Foo { ... }
4
5 fun Foo(): Foo { return FooImpl(...) }
```

LES CONVENTIONS UTILISÉES AVEC KOTLIN

RÈGLES DES ESPACES

Quelque unes de règles, qui sont nombreuses, on utilise un espace :

- Pour indenter (4 espaces).

- Pour séparer un mot clé et une "(" (par exemple `if`, `for` ou `catch`.
- Pour séparer un mot clé et une "{" (par exemple `else`.
- Avant toute "{".
- Avant et après tout opérateur binaire.
- Avant et après la flèche `->`.
- Avant et après l'opérateur d'intervalle `..`.
- Après une virgule `,`.
- Avant et après le signe de commentaire ligne simple `//`.

LES CONVENTIONS UTILISÉES AVEC KOTLIN

NOMMAGE DES PROPRIÉTÉS

Les constantes (propriétés marquées avec un `const`, les propriétés top level ou propriétés `val` d'un objet sans fonction `get` custom, qui contient une valeur profondément immuable), doit utiliser des majuscules, et _ comme séparateur :

```
1 const val MAX_COUNT = 8
2 val USER_NAME_FIELD = "UserName"
```

Les fonctions top level ou les propriétés d'un objet dont les valeurs peuvent évoluer, utiliserons la notation camel-case :

```
1 val mutableCollection: MutableSet<String> = HashSet()
```

Le nom des propriétés qui contiennent une référence à un objet Singleton, peuvent utiliser la même règle de nommage :

```
1 val PersonComparator: Comparator<Person> = /*...*/
```

Pour les valeurs des enums, il est possible d'utiliser les majuscules séparés par des _, ou l'écriture camel-case, en commençant par une majuscule, selon l'usage.

LES CONVENTIONS UTILISÉES AVEC KOTLIN

CHOISIR LE BON NOM

Le nom d'une classe est souvent un nom, qui définit la nature de la classe : `List`, `PersonReader`.
Le nom des méthodes est plus souvent un verbe, ou une phrase, décrivant son action : `close`, `readPersons`.

Le nom doit aussi faire comprendre s'il modifie l'objet ou s'il en retourne un nouveau. Par exemple : `sort` modifie la collection, alors que `sorted` retournera une copie de la collection triée.

Les noms doivent être clairs, sur leur fonctionnement, ils doivent donc éviter de contenir des noms génériques tels que `Manager`, `Wrapper` , etc.

Quand vous utilisez des acronymes dans un nom, mettre en majuscules si sa taille est de 2 lettres (`IOStream`), mais ne mettez que la première lettre en majuscule s'il est plus long (`XmlFormatter`, `HttpInputStream`).

BASES DE KOTLIN

- Déclaration de variables en Kotlin
- Utilisation de variables "Basic Types" en Kotlin
- Boucles et ranges en Kotlin
- Structures conditionnelles If et When
- Collections en Kotlin
- Packages et imports en Kotlin

DÉCLARATION DE VARIABLES EN KOTLIN

- val : valeur
- var : variable

```
1 val message = "Hello world !"
2 val message: String = "Hello world !"
3 var message: String = "Hello world !"
```

DÉCLARATION DE VARIABLES EN KOTLIN

Le code :

```kotlin
val name: String = "Tristan"
val age: Int = 41
val isDeveloper: Boolean = true
```

Est équivalent à :

```kotlin
val name = "Tristan"
val age = 41
val isDeveloper = true
```

DÉCLARATION DE VARIABLES EN KOTLIN

Une valeur peut être assignée dans le même bloc que sa déclaration :

```kotlin
import kotlin.random.Random

fun isUserHappy() = Random.nextInt(0, 100) % 2 == 0
fun main() {
    val message: String
    if (isUserHappy())
        message = "That's great"
    else
        message = "What's going on?"

    println(message)
}
```

DÉCLARATION DE VARIABLES EN KOTLIN

NULL SAFETY

En Kotlin, c'est à nous de préciser qu'une variable peut prendre une valeur nulle.
Le code suivant, ne compilera pas.

```
1 var name: String = "Tristan"
2 name = null
```

Alors que le code suivant est correct.
Nous précisons que la variable peut prendre une valeur nulle avec le ?.

```
1 var name: String? = "Tristan"
2 name = null
```

DÉCLARATION DE VARIABLES EN KOTLIN

NULL SAFETY (SUITE)

Pour utiliser une variable possiblement nulle nous ne pouvons pas l'utiliser simplement.
L'exemple suivant ne compilera pas :

```
1 var name: String? = "Tristan"
2 name.toUpperCase()
```

Il faut donc prendre une précaution en utilisant cette variable, en utilisant `?` :

```
1 var name: String? = "Tristan"
2 name?.toUpperCase()
```

Si la valeur de la variable contient null, alors la méthode ne sera pas appellée.

DÉCLARATION DE VARIABLES EN KOTLIN

UTILISATION D'UNE VARIABLE DANS UNE CHAINE DE CARACTÈRE

Nous pouvons faire référence à une variable avec le symbole `$`, par exemple :

```
1 val name = "Tristan"
2 println("Hello $name")
```

DÉCLARATION DE VARIABLES EN KOTLIN

LES CONSTANTES

Le mot clé static n'existe pas en Kotlin, donc pour déclarer une constante on utilise le mot clé `const`.
Le code Java suivant :

```java
public static final String SERVER_URL = "http://my.api.com/";
```

Deviendra en Kotlin :

```kotlin
const val SERVER_URL = "http://my.api.com/"
```

UTILISATION DE VARIABLES "BASIC TYPES" EN KOTLIN

Plusieurs types basiques sont disponibles en Kotlin : les nombres, les lettres, les booléens, les tableaux et les chaînes de caractères.

NOMBRES

Type	Size (bits)	Min value	Max value
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
Long	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

```
1 val one = 1 // Int
2 val threeBillion = 3000000000 // Long
3 val oneLong = 1L // Long
4 val oneByte: Byte = 1
```

NOMBRES

NOMBRES À VIRGULE

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
Float	32	24	8	6-7
Double	64	53	11	15-16

```
1 val pi = 3.14 // Double
2 val e = 2.7182818284 // Double
3 val eFloat = 2.7182818284f // Float, actual value is 2.7182817
```

NOMBRES

CONVERSION AUTOMATIQUE DES TYPES.

```
1  fun main() {
2      fun printDouble(d: Double) { println(d) }
3
4      val i = 1
5      val d = 1.1
6      val f = 1.1f
7
8      printDouble(d)
9  //    printDouble(i) // Error: Type mismatch
10 //    printDouble(f) // Error: Type mismatch
11 }
```

NOMBRES

CONVERSION EXPLICITE.

Pour convertir les types numériques, il fauddra utiliser une des méthodes suivantes :

- toByte(): Byte
- toShort(): Short
- toInt(): Int
- toLong(): Long
- toFloat(): Float
- toDouble(): Double
- toChar(): Char

NOMBRES

EXERCICE

Exercice : modifiez l'exemple, pour que l'on puisse afficher sa valeur, avec la fonction `printDouble` (sans toucher au code de la fonction).

```
fun main() {
    fun printDouble(d: Double) { println(d) }

    val i = 1
    val d = 1.1
    val f = 1.1f

    printDouble(d)
//    printDouble(i) // Error: Type mismatch
//    printDouble(f) // Error: Type mismatch
}
```

NOMBRES

SOLUTION

```
fun main() {
    fun printDouble(d: Double) { println(d) }

    val i = 1
    val d = 1.1
    val f = 1.1f

    printDouble(d)
    printDouble(i.toDouble())
    printDouble(f.toDouble())
}
```

CARACTÈRES

Les caractères sont représentés par le type Char

```
1 fun check(c: Char) {
2     if (c == 1) { // ERROR: incompatible types
3         // ...
4     }
5 }
```

Pour écrire un caractère, on utilise la simple "quote" '.
Exemple : 't'.
Les caractères spéciaux sont préfixés par \.
Exemple : \t, \b, \n, \r, \', \", \\ and \$.
Pour les autres caractères, on peut utiliser la syntaxe Unicode : '\uFF00'.

CARACTÈRES

Le charactère peut être convertis explicitement :

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // Explicit conversions to numbers
}
```

LES BOOLÉENS

Les booléens sont représentés par le type Boolean qui peut prendre 2 valeurs : true et false.
Les opérations sur les booléens sont les suivantes :

- || – opération logique OU
- && – opération logique ET
- ! - négation

LES TABLEAUX

Les tableaux sont représentés par la classe Array qui dispose des méthodes `get` et `set`
(qui se transforment en `[]` grâce à la convention de surcharge des opérateurs),
et de la propriété size, entre autres.

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
        // ...
}
```

LES TABLEAUX

La méthode `arrayOf(1, 2, 3)` permet de créer des tableaux, `arrayOfNulls()` va créer un tableau de

valeurs nulles.
Le constructeur `Array` prend en paramètre le nombre d'éléments, et la fonction pour remplir le tableau.

```
1 var array = arrayOf(1, 2, 3)
2 array.forEach { println(it) }
3 var arrayOfNull: Array<Int?> = arrayOfNulls(5)
4 arrayOfNull.forEach { println(it) }
5 val asc = Array(5) { i -> (i * i).toString() }
6 asc.forEach { println(it) }
```

LES TABLEAUX

L'opérateur `[]` est équivalent à l'appel des méthodes `get()` et `set()`.

```
1 var array = arrayOf(1, 2, 3)
2 println(array[2])
3 array[0] = 4
4 array.forEach { println(it) }
```

LES TABLEAUX

EXERCICE

Déclarez un tableau contenant les chiffres pair de 0 à 20.

```
1 val oddArray = TODO()
```

LES TABLEAUX

SOLUTION

```
fun main() {
//    val oddArray = TODO()
    val oddArray = Array(11) { i -> (i * 2) }

    for(i in oddArray) {
        print("$i ")
    }
}
```

LES TABLEAUX

EXERCICE

Déclarez un tableau contenant toutes les lettres de l'alphabet.

```
1 val alphabetArray = TODO()
```

LES TABLEAUX

SOLUTION

```
fun main() {
    val alphabetArray = Array(26) { i -> (i+'a'.toInt()).toChar() }

    for (i in alphabetArray) {
        print("$i ")
    }
}
```

LES CHAÎNES DE CARACTÈRES (STRING)

Les tableaux sont représentés par la classe String qui sont immutable. Les éléments d'une chaîne de caractère peuvent être accessibles avec l'opétateur `[]`.
Il est possible de concaténer des chaînes avec l'opérateur `+`, même si l'on péfèrera les templates ($ dans les chaînes de caractères).

```
1 val s = "abc" + 1
2 println(s + "def")
```

LES CHAÎNES DE CARACTÈRES (STRING)

MODÈLES (STRING TEMPLATES)

Le chaines de caractère peuvent contenir des expressions (des morceau de code), qui sont préxifés par `$`.

```
1 val i = 10
2 println("i = $i") // affiche "i = 10"
```

Il est possible d'inclure une expression entre ${}, et d'afficher le symbole $ lui même. Cela fonctionne aussi dans une chaîne brute (raw string).

```
1 println("$name.length is ${name.length}")
2 println("price : ${'$'}9.99")
3
4 val price = """
5 ${'$'}9.99
6 """
```

EXERCICE

Définir la fonction `hello` qui doit retourner : Hello, <NAME>

```
1 fun hello(name: String): String = TODO()
```

SOLUTION

STRUCTURES CONDITIONNELLES IF ET WHEN

L'EXPRESSION IF

En Kotlin `if` est une expression : il retourne une valeur. De ce fait l'opérateur ternaire `?` n'existe plus.

```kotlin
// Usage traditionnel
var max = a
if (a < b) max = b

// Avec else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// Comme une expression
val max = if (a > b) a else b
```

STRUCTURES CONDITIONNELLES IF ET WHEN

L'EXPRESSION IF (SUITE)

Les branches du `if`, peuvent être des blocs, dont la dernière expression est la valeur du bloc :

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

Quand le `if` est utilisé comme expression (pour retourner une valeur ou pour assigner une valeur), l'expression doit obligatoirement comporter la branche `else`.

STRUCTURES CONDITIONNELLES IF ET WHEN

SELON LE CAS (WHEN)

Le `when` est l'équivalent du `switch` en Java.

```
1 var apiReponse = 404
2 when (apiReponse) {
3     200 -> "OK"
4     404 -> "NOT FOUND"
5     401 -> "UNAUTHORIZED"
6     403 -> "FORBIDDEN"
7     else -> "UNKNOWN"
8 }
```

Quand le when est utilisé comme expression (pour retourner une valeur ou pour assigner une valeur), l'expression doit obligatoirement comporter la branche else, sauf si tous les cas sont couverts (d'un enum par exemple).

BOUCLES ET RANGES EN KOTLIN

TANT QUE FAIRE SE PEUT (WHILE)

En Kotlin, la syntaxe du `while` est exactement la même qu'en Java :

```
1 var isRaining = true
2 while (isRaining){
3     println("I don't like rain.")
4 }
5
6 do {
7     println("I don't like rain.")
8 } while (isRaining)
```

BOUCLES ET RANGES EN KOTLIN

BOUCLE POUR (FOR LOOP)

La boucle `for`, peut itérer sur tout ce qui fournit un itérateur, sa syntaxe est la suivante :

```
1 for (item in collection) print(item)
```

Par exemple sur une liste de chaînes de caractères, cela donne :

```
1 val names = listOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
2
3 for(name in names) {
4     println("This developer rocks : $name")
5 }
```

BOUCLES ET RANGES EN KOTLIN

LES INTERVALLES

Il est possible d'itérer sur des intervalles de valeur, qui sont définis avec la méthode `rangeTo()`, correspondant à l'opérateur `..`

Cet opérateur est souvent complété par les fonctions `in` ou `!in`. Exemple :

```
1 if (i in 1..4) {  // equivalent à 1 <= i && i <= 4
2     print(i)
3 }
```

Pour définir un intervalle, souvent utilisés dans les boucles for, la syntaxe est la suivante :

```
1 for (i in 1..4) print(i)
```

Pour utiliser l'ordre décroissant, la syntaxe sera :

```
1 for (i in 4 downTo 1) print(i)
```

BOUCLES ET RANGES EN KOTLIN

LES INTERVALLES (SUITE)

Il est possible d'itérer sur un des nombres dont l'intervalle n'est pas nécessairement 1,
en utilisant la méthode step :

```
1 for (i in 1..8 step 2) print(i)
2 println()
3 for (i in 8 downTo 1 step 2) print(i)
```

Pour ne pas inclure le dernier élément de la liste, utiliser la fonction until :

```
1 for (i in 1 until 10) {       // i in [1, 10), 10 is excluded
2     print(i)
3 }
```

BOUCLES ET RANGES EN KOTLIN

BREAK ET CONTINUE

Ils fonctionnent de la même manière qu'en Java.

BOUCLES ET RANGES EN KOTLIN

EXERCICE

Ecrivez une boucle qui affiche séparément toutes les lettres d'`Une chaîne de caractères`.

```
1 var stringValue = "Une chaîne de caractères"
```

BOUCLES ET RANGES EN KOTLIN

SOLUTION

```
var stringValue = "Une chaîne de caractères"
for (c in stringValue) {
    print("$c ")
}
```

BOUCLES ET RANGES EN KOTLIN

EXERCICE

Afficher les nombres de 0 à 10, mais afficher Fizz si le nombre est divisible par 3, Buzz s'il est divisible par 5 et FizzBuzz s'il est divisible à la fois par 3 et par 5.

BOUCLES ET RANGES EN KOTLIN

SOLUTION

```kotlin
fun main(args: Array<String>) {
    for (x in 0 until 10) {
        when {
            (x % 3 == 0 && x % 5 == 0) -> print("FizzBuzz")
            (x % 3 == 0) -> print("Fizz")
            (x % 5 == 0) -> print("Buzz")
            else -> print(x)
        }
    }
}
```

COLLECTIONS EN KOTLIN

Kotlin propose plusieurs structures pour gérer des groupes d'objets en nombre variables (possiblement 0).
Si vous êtes familiers de ces concepts, passons à la suite. Sinon continuons ...

Une collection est une structure qui regroupe des objets de même type. Ces objets sont appelés des éléments, ou des items.

Il exite plusieurs types de collection :

- Une liste (List), est une collection ordonnée d'objets auxquels nous pouvons accéder via leur position/index (un nombre entier).Un élement peut être présent une ou plusieurs fois dans la liste. Exemple d'une phrase qui comporte des mots, dont l'ordre est important, et qui peuvent se répéter.
- Les ensembles (Set), est une collection d'éléments uniques. L'ordre dans un ensemble n'a pas d'importance. Par exemple les lettres de l'alphabet.

COLLECTIONS EN KOTLIN

- Les dictionnaires (Map), est un ensemble d'élements composés d'une paire (clé-valeur). Les clés ont des valeurs uniques qui désignent un seul objet de la collection. Les valeurs peuvent apparaitre en plusieurs fois.

  Cette structure est employée pour stocker une connexion logique entre 2 objets, par exemple, un numéro d'employé et sa fiche descriptive.

Le comportement de chaque type de collection sera toujours le même, peu importe le type des objets stockés dans ces structures.

COLLECTIONS EN KOTLIN

En Kotlin, il y a deux types principaux de collections :

- En lecture seule (read-only/immutable).
- En lecture/écriture (mutable) (ajout, retrait, modification des éléments).

Note : Une collection mutable peut être stockée dans une valeur (val) :

```
1 val numbers = mutableListOf("one", "two", "three", "four")
2 numbers.add("five")   // this is OK
3 //numbers = mutableListOf("six", "seven")      // compilation error
```

COLLECTIONS EN KOTLIN

Plusieurs exemples de collections `List` et `Set` :

```
1  val listOfNames = listOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
2  listOfNames[0]
3  //listOfNames[0] = "Mathieu NEBRA" // Error: List is immutable
4
5  val mutableListOfNames = mutableListOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
6  mutableListOfNames[0]
7  mutableListOfNames[0] = "Mathieu NEBRA" // OK
8
9  val setOfNames = setOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
10 setOfNames.first()
11 //setOfNames.add("Mathieu NEBRA") // Error: Set is immutable
12
13 val mutableSetOfNames = mutableSetOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
14 mutableSetOfNames.first()
15 mutableSetOfNames.add("Mathieu NEBRA") // OK
```

Exemple de tableau et de `Map` :

```
1  var arrayOfNames = arrayOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
2  var mapOfNames = mapOf(0 to "Jake WHARTON", 1 to "Joe BIRCH", 2 to "Robert MARTIN")
```

PACKAGES ET IMPORTS EN KOTLIN

Un ficher de code source peut commencer par la déclaration d'un package :

```kotlin
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

Tout le contenu (tels que les classes et les fonctions) du fichier de code source appartiendront au package déclaré. Dans l'exemple ci-dessus, le nom complet de `printMessage()` est `org.example.printMessage()`, de la même manière, le nom complet de `Message` est `org.example.Message`.

Si le nom du package n'est pas précisé, le contenu du fichier appartient au package par défaut qui n'a pas de nom.

PACKAGES ET IMPORTS EN KOTLIN

LES BONNES PRATIQUES

- Ne pas utiliser le mot `Utils` pour nomer un fichier source, qui apporte peu d'informations sur son contenu.

- Regouper les classes qui ont un sens proche, dans un même fichier est recommandé, sous condition que le fichier ne soit pas trop gros (quelques centaines de lignes).
- De même il est recommandé de regrouper les extensions correspondant aux même client, dans un unique fichier.
- Utiliser des val ou des collections en lecture seule autant que possible.
- Utiliser de préférence `until` au lieu d'un intervalle ouvert :

```
1 for (i in 0..n - 1) { ... }  // bad
2 for (i in 0 until n) { ... }  // good
```

- Utiliser les "string templates" au lieu de concaténations.

LES FONCTIONS - PARTIE 1

- Fonctions en Kotlin
- Paramètres des fonctions en Kotlin
- Fonctions Infix en Kotlin
- Fonctions Anonyme en Kotlin
- Returns et Local Returns en Kotlin
- Tail recursion en Kotlin
- Bonnes et mauvaises pratiques

FONCTIONS EN KOTLIN

DÉCLARATION

On utilise le mot clé fun

```kotlin
fun double(x: Int): Int {
    return 2 * x
}
```

FONCTIONS EN KOTLIN

UTILISATION

Appel traditionnel :

```
1 val result = double(2)
```

Appel d'une fonction membre d'un objet :

```
1 Stream().read() // créé une instance de la class Stream et appelle la fonction read()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

LES PARAMÈTRES SIMPLES

Chaque paramètre (explicitement typé), est séparé par une virgule :

```
1 fun powerOf(number: Int, exponent: Int) { /*...*/ }
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

PARAMÈTRES PAR DÉFAUTS

Les paramètres peuvent avoir une valeur par défaut (exprimée avec =) quand un argument n'est pas précisé :

```
1 fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { /*...*/ }
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

EXERCICE

Ecrivez une fonction qui prend en paramètre une chaîne de caractères, et retourne sa valeur en majuscule. Si aucune valeur n'est passée en paramètre, alors utiliser la valeur par défaut `"default"` :

```
1 fun upper(): String = TODO()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

SOLUTION

```
1 fun upper(str:String? = "default") = str?.toUpperCase()
```

Ou :

```
1 fun upper(str:String = "default") = str.toUpperCase()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

EXERCICE

Écrivez la fonction `add` qui additionne 2 `Int` passés en paramètre :

```
1 fun add(a: Int, b: Int): Int = TODO()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

SOLUTION

```
1 fun add(a: Int, b: Int): Int = a + b
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

EXERCICE

Écrivez les fonctions qui comparent 2 chaînes de caractères, en ignorant la casse ou pas.

```
1 fun strEq(s1: String, s2: String): Boolean = TODO()
2 fun strEq(s1: String, s2: String, ignoreCase: Boolean): Boolean = TODO()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

SOLUTION

```kotlin
fun strEq(s1: String, s2: String): Boolean = s1.equals(s2)
fun strEq(s1: String, s2: String, ignoreCase: Boolean): Boolean = if(ignoreCase) s1.toUpperCase().equals(s2.toUpperCase()) else

fun main() {
    println(strEq("Tristan", "TRISTAN"))
    println(strEq("Tristan", "TRISTAN", true))
}
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

EXERCICE

Écrivez la fonction qui affiche la liste des langages présents dans le tableau.

```
1 val languages = arrayOf("Java", "JavaScript", "Go", "Kotlin")
2 fun printLanguages(): Unit = TODO()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

SOLUTION

```kotlin
val languages = arrayOf("Java", "JavaScript", "Go", "Kotlin")
fun printLanguages(): Unit { for (language in languages) println(language) }

fun main() {
    printLanguages()
}
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

EXERCICE

Écrivez les fonctions suivantes :

- tenFirstNumber() qui affiche les 10 premiers chiffres (0-9).
- countdown() qui affiche les nombres de 10 à 0.
- firstEvenNumbers() qui affiche les 10 premiers nombres pairs.
- firstOddNumbers() qui affiche les 10 premiers nombres impairs.

```
1 fun tenFirstNumber(): Unit = TODO()
2 fun countdown(): Unit = TODO()
3 fun firstEvenNumbers(): Unit = TODO()
4 fun firstOddNumbers(): Unit = TODO()
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

SOLUTION

```kotlin
fun tenFirstNumber(): Unit { for(i in 0 until 10) print("$i "); println()}
fun countdown() : Unit { for(i in 10 downTo 0) print("$i "); println()}
fun firstEvenNumbers(): Unit { for(i in 0 until 20 step 2) print("$i "); println()}
fun firstOddNumbers(): Unit { for(i in 1 .. 20 step 2) print("$i "); println()}

fun main() {
    tenFirstNumber()
    countdown()
    firstEvenNumbers()
    firstOddNumbers()
}
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

PARAMÈTRES NOMMÉS

Chaque paramètre peut être nommé explicitement. Cela est particulièrement pratique pour les fonctions qui ont beaucoup de paramètres, ou des paramètres par défauts :

```
1 fun reformat(str: String,
2              normalizeCase: Boolean = true,
3              upperCaseFirstLetter: Boolean = true,
4              divideByCamelHumps: Boolean = false,
5              wordSeparator: Char = ' ') {
6 /*...*/
7 }
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

PARAMÈTRES NOMMÉS UTILISATION

Nous pouvons utiliser les paramètres par défaut :

```
1 reformat(str)
```

Nous pouvons préciser tous les paramètres :

```
1 reformat(str, true, true, false, '_')
```

L'appel avec tous les paramètres només est bien plus clair :

```
1 reformat(str,
2     normalizeCase = true,
3     upperCaseFirstLetter = true,
4     divideByCamelHumps = false,
5     wordSeparator = '_'
6 )
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

PARAMÈTRES PAR DÉFAUTS EN PREMIER

Si le paramètre par défaut est en premier : il faudra appeler la fonction avec des paramètres nommés :

```
1 fun foo(bar: Int = 0, baz: Int) { /*...*/ }
2
3 foo(baz = 1) // La valeur par défaut bar = 0 est utilisée
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

PARAMÈTRES PAR DÉFAUTS : LAMBDA

Si le dernier paramètre est une lambda, elle peut être passée via un paramètre nommé, ou à l'extérieur des parenthèses :

```kotlin
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { /*...*/ }

foo(1) { println("hello") }     // Utilise la valeur par défaut baz = 1
foo(qux = { println("hello") }) // Utilise les valeurs par défaut bar = 0 et baz = 1
foo { println("hello") }        // Utilise les valeurs par défaut bar = 0 et baz = 1
```

PARAMÈTRES DES FONCTIONS EN KOTLIN

LES PARAMÈTRES VARIABLES

En Java 5 il est possible d'utiliser la notation `...` pour définir un nombre variable de paramètres. En Kotlin, c'est le mot clé vararg qui est utilisé (pour le dernier paramètre généralement, autrement il faudra nommer les paramètres).

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

Un seul paramètre peut être marqué variable.
Nous pouvons aussi utiliser le déstructuring (opérateur *), si nous avons déjà une variable contenant une liste.

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

FONCTIONS INFIX EN KOTLIN

Les fonctions marquées avec le mot clé infix peuvent être appelées sans le point ni les parenthèses.

Les conditions sont :

- La fonction doit être une fonction membre ou une fonction extension.
- La fonction doit avoir un paramètre unique.
- Le paramètre ne doit pas accepter un nombre variable de valeurs, et ne doit pas avoir de valeur par défaut.

```
1 infix fun Int.shl(x: Int): Int { ... }
2
3 // appel de la méthode en utilisant la notation infix
4 1 shl 2
5
6 // est la même chose que
7 1.shl(2)
```

FONCTIONS ANONYME EN KOTLIN

En Kotlin il est possible de définir des méthodes anonymes (sans leur donner de nom). Par exemple :

```
1 fun(x: Int, y: Int): Int = x + y
```

Une fonction anonyme ressemble à une fonction classique, sauf le que le nom est omis. Le corps de la fonction peut aussi être un bloc :

```
1 fun(x: Int, y: Int): Int {
2     return x + y
3 }
```

Si le type des paramètres peut être inféré par le compilateur, il n'est pas nécessaire de le préciser, comme dans l'exemple :

```
1 ints.filter(fun(item) = item > 0)
```

RETURNS ET LOCAL RETURNS EN KOTLIN

RETOUR D'UNE FONCTION

Une fonction qui ne retourne rien, retourne implicitement un type `Unit`.

Il n'est pas besoin de retourner explicitement la valeur :

```
1 fun printHello(name: String?): Unit {
2     if (name != null)
3         println("Hello ${name}")
4     else
5         println("Hi there!")
6     // "return Unit" or "return" is optional
7 }
```

Il n'est même pas obligatoire de préciser ce type retour :

```
1 fun printHello(name: String?): {
2     ...
3 }
```

RETURNS ET LOCAL RETURNS EN KOTLIN

FONCTIONS LOCALES

Kotlin gère les fonctions locales, c'est à dire des fonctions à l'intérieur d'autres fonctions :

```kotlin
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

RETURNS ET LOCAL RETURNS EN KOTLIN

Une fonction locale a accès aux variables locales définies à l'extérieur de la fonction, donc dans le cas, ci-dessus, la variable `visited` peut être définie comme variable locale :

```kotlin
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

RETURNS ET LOCAL RETURNS EN KOTLIN

EXERCICE

Écrivez une fonction qui permet de déterminer si une valeur est paire en utilisant une fonction locale `fun isMultiple(operand: Int): Boolean:`

```
1 fun isEven(n: Int): Boolean = TODO()
```

RETURNS ET LOCAL RETURNS EN KOTLIN

SOLUTION

```kotlin
fun isEven(n: Int): Boolean {
    fun isMultiple(operand: Int): Boolean = n % operand == 0
    return isMultiple(2)
}

fun main() {
    println(isEven(2))
    println(isEven(3))
}
```

LES FONCTIONS "EXPRESSION SEULE" (SINGLE-EXPRESSION FUNCTION)

Il n'est pas nécessaire d'utiliser les accolades, un simple = peut les remplacer :

```
1 fun double(x: Int): Int = x * 2
```

Il n'est même pas nécessaire de préciser le type de retour, qui est inféré par le compilateur :

```
1 fun double(x: Int) = x * 2
```

LES FONCTIONS RÉCURSIVES (TAIL RECURSIVE FUNCTIONS)

Kotlin gère une façon de programmer appellée : "tail recursion".
Cela permet à certains algorithmes qui seraient écrits normalement avec des boucles for, d'être écrits en utilisant une fonction récursive, mais sans le risque de dépassement de pile (stack overflow). Quand une fonction est marquée avec le modifieur `tailrec` et répond au format requis, le compilateur optimise la récursion, en générant une boucle rapide et optimisée à la place :

```
1 val eps = 1E-10 // "good enough", could be 10^-15
2
3 tailrec fun findFixPoint(x: Double = 1.0): Double
4         = if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

LES FONCTIONS RÉCURSIVES (TAIL RECURSIVE FUNCTIONS)

Ce code calcule le point invariant de cosinus, qui est une constante mathématique.
Le code appelle la fonction `Math.cos` plusieurs fois, en partant de 1.0 jusqu'à ce que le résultat ne change plus, en arrivant au résultat de : 0.7390851331706995 pour la précision `eps`.

Le code est équivalent à la forme plus classique :

```
1  val eps = 1E-10 // "good enough", could be 10^-15
2
3  private fun findFixPoint(): Double {
4      var x = 1.0
5      while (true) {
6          val y = Math.cos(x)
7          if (Math.abs(x - y) < eps) return x
8          x = Math.cos(x)
9      }
10 }
```

CLASSES EN KOTLIN

- Une classe
- Les attributs
- Méthodes (Functions Members)

- Visibilité des membres en Kotlin
- Héritage en Kotlin
- Abstract Classes en Kotlin
- Interface en Kotlin
- Polymorphisme en Kotlin
- Data Classes en Kotlin
- Enum Classes en Kotlin
- Nested Classes en Kotlin
- Sealed Classes en Kotlin
- Bonnes et mauvaises pratiques

UNE CLASSE

DÉCLARATION

On utilise le mot clé class

```
1 class Invoice { /*...*/ }
```

Une déclaration de classe consiste en : un nom, un entête (qui spécifie le type des paramètres, le constructeur primaire, etc.) et le corps de la classe, le tout entouré d'accolades. L'entête et le corps de la classe sont optionnels. Si la classe n'a pas de corps, les accolades sont optionnelles :

```
1 class Empty
```

UNE CLASSE

LE CONSTRUCTEUR

Une classe peut avoir un **constructeur primaire** et un ou plusieurs **constructeurs secondaires**. Le constructeur primaire fait partie intégrante de l'entête : il est placé juste après le nom de la classe.

```
1 class Person constructor(firstName: String) { /*...*/ }
```

Si le constructeur n'a pas d'annotations, ou de modificateurs de visibilité, le mot clé `constructor` n'est pas obligatoire :

```
1 class Person(firstName: String) { /*...*/ }
```

UNE CLASSE

LA FORME CONCISE DU CONSTRUCTEUR

C'est cette forme que l'on utilisera de préférence :

```
1 class Person(val firstName: String, val lastName: String, var age: Int) { /*...*/ }
```

Les propriétés peuvent être :

- En lecture seule : `val`
- En lecture ET écriture (mutable) : `var`

UNE CLASSE

EXEMPLE DE CLASSE USER

En Java une classe User serait écrite comme suit :

```java
public class User {

    // PROPERTIES

    private String email;
    private String password;
    private int age;

    // CONSTRUCTOR
    public User(String email, String password, int age) {
        this.email = email;
        this.password = password;
        this.age = age;
    }

    // GETTERS
```

En Kotlin son équivalent est :

```kotlin
class User(var email: String, var password: String, var age: Int)
```

UNE CLASSE

LES VALEURS PAR DÉFAUT

En Kotlin, pas besoin de définir plusieurs constructeurs pour gérer les valeurs des paramètres par défaut, il suffit de préciser leur valeur avec le signe = dans le constructeur :

```
1 class Customer(val customerName: String = "")
```

UNE CLASSE

MODIFICATION DES ACCESSEURS PAR DÉFAUT

Kotlin permet de modifier le comportement par défaut des getters et des setters générés pour les propriétés :

```
1 class User(email: String, var password: String, var age: Int) {
2
3     var email: String = email
4         get() { println("User email read access done."); return field}
5         set(value) { println("User email write access done."); field = value}
6 }
```

UNE CLASSE

PROPRIÉTÉ PRIVÉE

Par défaut en Kotlin les propriétés sont publiques, pour changer cela, il suffit d'ajouter le modifieur de visibilité :

```
1 class User(var email: String, private var password: String, var age: Int)
```

UNE CLASSE

UTILISATION (INSTANCIATION) D'UNE CLASSE

En Kotlin, il n'y a pas de `new`, on utilise directement le nom de la classe :

```
1 val user = User("hello@gmail.com", "azerty", 41)
```

Pour accéder aux champs, la notation à . est utilisée :

```
1 val user = User("hello@gmail.com", "azerty", 41)
2 println(user.email) // Getter
3 user.email = "my_new_email@gmail.com"
4 println(user.email) // Getter
```

UNE CLASSE

EXERCICE

Écrivez une classe `Person` avec les propriétés `firstName` et `lastName` dont les valeurs par défaut sont la chaîne vide :`""`. La propriété `lastName` est en lecture seule. Vous afficherez le nom complet (prénom nom) après avoir modifié le prénom et vérifiée que le nom est bien en lecture seule.

```
1 class Person
```

UNE CLASSE

SOLUTION

```
1  class Person (var firstName: String = "", val lastName: String = "")
2
3  fun main() {
4      var me = Person("Tristan", "SALAUN")
5      println(me)
6      println( "${me.firstName} ${me.lastName}")
7
8      me.firstName = "Mélody"
9  //     me.lastName = "ANDRÉ"
10
11     println( "${me.firstName} ${me.lastName}")
12 }
```

UNE CLASSE

EXERCICE

Testez la classe `User`, création d'une instance, changement de la valeur de l'email, et affichage de cette valeur.

UNE CLASSE

SOLUTION

```
1  class User(email: String, var password: String, var age: Int) {
2
3      var email: String = email
4          get() { println("User email read access done."); return field}
5          set(value) { println("User email write access done."); field = value}
6  }
7
8  fun main() {
9      var currentUser = User("tristan.salaun.pro@gmail.com", "azerty", 41)
10     currentUser.email = "test@test.com"
11     println(currentUser.email)
12 }
```

LES ATTRIBUTS

Les propriétés en Kotlin peuvent être déclarées en lecture/écriture (mutable) en utilisant le mot clé `var`, ou en lecture seule (immutable) en utilisant le mot clé `val` :

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "221b Baker Street"
    var city: String = "London"
    var state: String? = null
    var zip: String = "NW1 6XE"
}
```

Pour accéder aux propriétés, il suffit d'utiliser son nom :

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

MÉTHODES (FUNCTIONS MEMBERS)

Une méthode membre d'une classe, est définie comme suit :

```
1 class Sample() {
2     fun foo() { print("Foo") }
3 }
```

Comme déjà vu, pour appeler une telle méthode il suffit d'utiliser la notation à point :

```
1 Sample().foo() // creates instance of class Sample and calls foo
```

VISIBILITÉ DES MEMBRES EN KOTLIN

En Kotlin la visibilité par défaut est `public`, les modifieurs de visibilité disponibles sont :

- `private` : Un membre déclaré comme `private` sera visible uniquement dans la classe où il est déclaré.
- `protected` : Un membre déclaré comme `protected` sera visible uniquement dans la classe où il est déclaré ET dans ses sous-classes (via l'héritage).
- `internal` : Un membre déclaré comme `internal` sera visible par tous ceux du même module. Un module est un ensemble de fichiers compilés ensemble (comme une librairie Gradle ou Maven, par exemple).
- `public` : Un membre déclaré comme `public` sera visible partout et par tout le monde.

HÉRITAGE EN KOTLIN

Toutes les classes en Kotlin héritent de la super class `Any` (équivalent à `Object` en Java), qui est la superclasse par défaut de toutes les classes qui n'ont pas déclaré de super type :

```
1 class Example // Implicitly inherits from Any
```

La classe `Any` à 3 méthodes : `equals()`, `hashCode()` et `toString()`.
Pour déclarer un super type, la syntaxe est la suivante :

```
1 open class Base(p: Int)
2
3 class Derived(p: Int) : Base(p)
```

LA SURCHARGE

LA SURCHARGE DES MÉTHODES

En Kotlin, tout doit être explicite, une méthode qui peut être surchargée sera marquée avec le modifieur `open` :

```
1 open class Shape {
2     open fun draw() { /*...*/ }
3     fun fill() { /*...*/ }
4 }
5
6 class Circle() : Shape() {
7     override fun draw() { /*...*/ }
8 }
```

Une méthode qui redéfinit une autre de la classe parente est elle même redéfinissable, à moins de la marquer comme `final` :

```
1 open class Rectangle() : Shape() {
2     final override fun draw() { /*...*/ }
3 }
```

LA SURCHARGE DES PROPRIÉTÉS

La surcharge d'une propriété fonctionne de la même manière que la surcharge des méthodes :

```
1 open class Shape {
2     open val vertexCount: Int = 0
3 }
4
5 class Rectangle : Shape() {
6     override val vertexCount = 4
7 }
```

Il est possible de redéfinir une propriété de type `val` avec une autre de type `var`, mais pas l'inverse.
En effet la propriété `val` déclare une méthode `get`, en la redéfinissant par une `var`, on déclare une méthode `set` de plus dans la classe dérivée.

```
1 interface Shape {
2     val vertexCount: Int
3 }
4
5 class Polygon : Shape {
6     override var vertexCount: Int = 0  // Can be set to any number later
7 }
```

LA SURCHARGE

L'APPEL À LA CLASSE PARENTE

Comme en Java, le mot clé pour appeler le parent est `super` :

```
1  open class Rectangle {
2      open fun draw() { println("Drawing a rectangle") }
3      val borderColor: String get() = "black"
4  }
5
6  class FilledRectangle : Rectangle() {
7      override fun draw() {
8          super.draw()
9          println("Filling the rectangle")
10     }
11
12     val fillColor: String get() = super.borderColor
13 }
```

ABSTRACT CLASSES EN KOTLIN

Une classe et quelques membres peuvent êtres déclarés `abstract`. Un membre abstrait n'a pas d'implémentation dans la classe. Le mot clé `open` n'est pas nécessaire, dans le cas d'une classe ou méthode abstraite, car cela est évident.
Note : il est possible de surcharger un membre non abstrait par un abstrait :

```
1 open class Polygon {
2     open fun draw() {}
3 }
4
5 abstract class Rectangle : Polygon() {
6     override abstract fun draw()
7 }
```

INTERFACE EN KOTLIN

Les interfaces en Kotlin peuvent conteneur des méthodes abstraites, mais aussi des méthodes implémentées. Ce qui les différencies d'une classe abstraite, est le fait qu'elles ne contiennent pas d'état. Elles peuvent comporter des propriétés, mais elles doivent être abstraites ou fournir une implémentation pour les

accesseurs.
Une interface est définie en utilisant le mot clé `interface` :

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

Implémenter une interface :

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

POLYMORPHISME EN KOTLIN

Le principe est le même qu'en Java : nous pouvons utiliser par exemple une variable d'un type parent, contenant des sous types. Par exemple :

```kotlin
open class User (var name: String, var firstName: String, var age: Int) {
    open fun displayInformations() = println("$firstName $name is $age old")
}
class Admin(name: String, firstName: String, age: Int, var phoneNumber: String) : User(name, firstName, age) {
    override fun displayInformations() = println("$firstName $name is $age old, phone number : $phoneNumber")
}

fun main() {
    var currentUser = User(firstName = "Tristan", name = "SALAUN", age = 41)
    currentUser.displayInformations()
    currentUser = Admin(firstName = "Tristan", name = "SALAUN", age = 41, phoneNumber = "0123456789")
    currentUser.displayInformations()
}
```

POLYMORPHISME EN KOTLIN

EXERCICE

Écrivez les classes `Dog`, `Bird`, `Duck` et `Snake` avec les cris respectifs : "Waf", "Cui cui", "Coin coin" et "Sssssss". Validez en appellant la méthode `speak` sur les différentes instances d'annimaux référencées par un même objet de type `Animal`.

```
interface Animal {
  fun speak()
}

class Dog: Animal
class Bird: Animal
class Duck: Animal
class Snake: Animal
```

POLYMORPHISME EN KOTLIN

SOLUTION

```kotlin
interface Animal {
    fun speak()
}

class Dog: Animal { override fun speak() = println("Waf") }
class Bird: Animal { override fun speak() = println("Cui cui") }
class Duck: Animal { override fun speak() = println("Coin coin") }
class Snake: Animal { override fun speak() = println("Sssssssss") }

fun main() {
    var myPet: Animal = Dog()
    myPet.speak()
    myPet = Bird()
    myPet.speak()
    myPet = Duck()
```

POLYMORPHISME EN KOTLIN

SURCHARGE DE FONCTIONS

Les fonctions définies ci-dessous, diffèrent uniquement par leur signature :

```
1  fun printNumber(n : Number){
2      println("Using printNumber(n : Number)")
3      println(n.toString() + "\n")
4  }
5
6  fun printNumber(n : Int){
7      println("Using printNumber(n : Int)")
8      println(n.toString() + "\n")
9  }
10
11 fun printNumber(n : Double){
12     println("Using printNumber(n : Double)")
13     println(n.toString() + "\n")
14 }
```

POLYMORPHISME EN KOTLIN

SURCHARGE DE FONCTIONS

Quelles seront les méthodes appelées lors de l'exécution ?

```kotlin
fun main() {
    val a : Number = 99
    val b = 1
    val c = 3.1

    printNumber(a) //Which version of printNumber is getting used?
    printNumber(b) //Which version of printNumber is getting used?
    printNumber(c) //Which version of printNumber is getting used?
}
```

DATA CLASSES EN KOTLIN

Nous écrivons régulièrement des classes, dont le rôle est de contenir de l'information. Dans ce genre de classes, des fonctionnalités, et des fonctions utilitaires sont souvent déduites mécaniquement des données. En Kotlin, ces classes sont appelées data class, et sont donc marquées `data` :

```
1 data class User(val name: String, val age: Int)
```

Le compilateur va générer les membres suivant, en utilisant toutes les propriétés déclarées dans le constructeur principal :

- `equals()/hashCode()`.
- `toString()` sous la forme `"User(name=Tristan, age=40)"`.
- Les fonctions `componentN()` correspondant aux propriétés dans leur ordre de déclaration.
- La fonction `copy()`.

DATA CLASSES EN KOTLIN

EXERCICE

Reprenez la classe Person, ajoutez le modifieur `data` et utiliser la copie pour changer uniquement le prénom (pour déclarer un parent par exemple) :

```
1 data class Person (var firstName: String = "", var lastName: String = "")
```

DATA CLASSES EN KOTLIN

SOLUTION

```kotlin
data class Person (var firstName: String = "", var lastName: String = "")

fun main() {
    var personTristan = Person("Tristan", "SALAUN")
    var personMelody = personTristan.copy(firstName = "Mélody")
    println(personTristan)
    println(personMelody)
}
```

ENUM CLASSES EN KOTLIN

L'usage le plus simple d'une classe enum est d'implémenter une énumération sûre :

```
1 enum class Direction {
2     NORTH, SOUTH, WEST, EAST
3 }
```

Chaque constante de l'énumération est un objet. Les constantes sont séparées par une virgule.

ENUM CLASSES EN KOTLIN

INITIALISATION

Chaque valeur de l'énumération est une instance de la classe enum, elles peuvent êtres initialisées de la façon suivante :

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

ENUM CLASSES EN KOTLIN

EXERCICE

Utilisez la classe enumération `Direction` précédente, et utilisez un "switch" (when) pour afficher en clair la direction prise.

```kotlin
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
fun main() {
    val direction = Direction.NORTH
    when{
        true -> TODO()
    }
}
```

ENUM CLASSES EN KOTLIN

SOLUTION

```kotlin
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
fun main() {
    val direction = Direction.NORTH;
    when(direction){
        Direction.NORTH -> println("On va au Nord.")
        Direction.SOUTH -> println("On va au Sud.")
        Direction.EAST -> println("On va à l'Est.")
        Direction.WEST -> println("On va à l'Ouest.")
    }
}
```

NESTED CLASSES EN KOTLIN

Les classes peuvent être imbriquées dans d'autres classes :

```kotlin
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

NESTED CLASSES EN KOTLIN

CLASSES IMBRIQUÉES

Une classe peut être marquée comme `inner` pour avoir la possibilité d'accéder aux membres de la classe englobante (outer class). Une classe imbriquée peut référencer un objet de la classe englobante :

```
1 class Outer {
2     private val bar: Int = 1
3     inner class Inner {
4         fun foo() = bar
5     }
6 }
7
8 val demo = Outer().Inner().foo() // == 1
```

SEALED CLASSES EN KOTLIN

Les classes scellées sont utilisées pour représenter une hiérarchie restreinte. Quand une valeur peut avoir qu'un type parmi un nombre limité de types. C'est, dans un sens, une extension des classes enum : les différentes valeurs d'un enum sont aussi limitées ; il n'existe qu'une seule instance pour chaque constante de l'enum, alors qu'une sous-classe scellée peut avoir plusieurs instances qui peuvent contenir un état.

Pour déclarer une classe scellée, il suffit d'utiliser le modifieur `sealed` avant le nom de la classe. Une classe scellée, peut avoir des sous classes, mais toutes doivent être déclarées dans le même fichier où celle-ci est déclarée. (Avant Kotlin 1.1, la règle était encore plus stricte : les classes devaient être des classes imbriquées dans la déclaration de la classe scellée).

```
1 sealed class Expr
2 data class Const(val number: Double) : Expr()
3 data class Sum(val e1: Expr, val e2: Expr) : Expr()
4 object NotANumber : Expr()
```

(L'exemple ci-dessus utilise la possibilité d'une fonctionnalité de Kotlin 1.1 : la possibilité pour les classes data d'étendre une autre classe, incluant les classes scellées.)

LES FONCTIONS - PARTIE 2

- Operator Overloading en Kotlin
- Lambda expression en Kotlin
- Extensions de fonctions en Kotlin
- Extensions de propriétés en Kotlin
- Closures en Kotlin
- Bonnes et mauvaises pratiques

OPERATOR OVERLOADING EN KOTLIN

Kotlin permet de fournir le code pour une liste prédéfinie d'opérateurs sur notre propre type. Ces opérateurs ont une représentation figée (exemple `+` ou `*`) et des règles de priorité figées. Pour définir un opérateur, il suffit de coder la méthode membre, ou une fonction d'extension, correspondant au nom de l'opérateur à définir. Les fonctions doivent être marquées avec le modifieur `operator`.

OPERATOR OVERLOADING EN KOTLIN

LES OPÉRATIONS UNAIRES

Expression	Traduites en
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

OPERATOR OVERLOADING EN KOTLIN

Exemple d'implémentation pour la classe `Point` :

```kotlin
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
    println(-point)  // prints "Point(x=-10, y=-20)"
}
```

OPERATOR OVERLOADING EN KOTLIN

LES OPÉRATIONS INCRÉMENT ET DÉCRÉMENT

Expression	Traduites en
`a++`	`a.inc()`
`a--`	`a.dec()`

Les méthodes `a.inc()` et `a.dec()` doivent retourner une valeur, qui sera assignée à la variable sur laquelle l'opérateur `++` ou `--` à été utilisé. Il ne faut pas changer la valeur de l'objet original.

OPERATOR OVERLOADING EN KOTLIN

EXERCICE

Écrivez l'opérateur ++ et -- pour la classe `Point`.

```kotlin
data class Point(var x: Int, var y: Int)
```

Testez la différence entre point++ et ++point.

OPERATOR OVERLOADING EN KOTLIN

SOLUTION

```kotlin
data class Point(var x: Int, var y: Int) {
    operator fun inc() : Point {
        return Point(this.x + 1, this.y + 1)
    }
}
fun main() {
    var point1 = Point(1,1)
    println(point1)
    point1++
    println(point1)
}
```

OPERATOR OVERLOADING EN KOTLIN

LES OPÉRATIONS BINAIRES

On parle ici d'opérations qui nécessitent deux valeurs.

OPERATOR OVERLOADING EN KOTLIN

LES OPÉRATIONS ARITHMÉTIQUES

Expression	Traduites en
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`,`a.mod(b)` (deprecated)
`a..b`	`a.rangeTo(b)`

Pour les opérations présentes dans ce tableau, le compilateur résout l'expression en utilisant tout simplement la colonne de droite.

OPERATOR OVERLOADING EN KOTLIN

EXEMPLE

Implémentation de l'opérateur `+` pour la classe `Counter`.

```kotlin
data class Counter(val dayIndex: Int) {

    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

Exercice : proposer un exemple d'utilisation.

```kotlin
fun main() {
    var counter: Counter = Counter(0)

    println(counter)
    println(counter + 3)
}
```

OPERATOR OVERLOADING EN KOTLIN

EXERCICE

Écrivez l'opérateur + pour la classe `Point` qui permet d'additionner 2 points.

```kotlin
data class Point(var x: Int, var y: Int)
```

OPERATOR OVERLOADING EN KOTLIN

SOLUTION

```
1 data class Point(var x: Int, var y: Int) {
2     operator fun plus(other: Point) : Point {
3         return Point(this.x + other.x, this.y + other.y)
4     }
5 }
```

Ou de manière équivalente :

```
1 data class Point(var x: Int, var y: Int)
2 operator fun Point.plus(other: Point) : Point { return Point(this.x + other.x, this.y + other.y) }
```

OPERATOR OVERLOADING EN KOTLIN

L'OPÉRATEUR 'IN'

Expression	Traduites en
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

Notez que l'ordre des paramètres est inversé pour `in` et `!in` par rapport au précédent opérateurs.

OPERATOR OVERLOADING EN KOTLIN

OPÉRATEUR D'ACCÈS INDEXÉ

Expression	Traduites en
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`

Les crochets, sont traduits en appels aux fonctions `get` et `set` avec le nombre d'arguments correspondant.

OPERATOR OVERLOADING EN KOTLIN

OPÉRATEUR D'INVOCATION

Expression	Traduites en
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`

Les parenthèses sont traduites par l'appel aux méthodes `invoke` avec le nombre de paramètres approprié.

OPERATOR OVERLOADING EN KOTLIN

OPÉRATEUR D'AFFECTATION ÉTENDUS

Expression	Traduites en
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`,`a.modAssign(b)` (deprecated)

OPERATOR OVERLOADING EN KOTLIN

OPÉRATEURS D'ÉGALITÉ ET D'INÉGALITÉ

Expression	Traduites en
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

Ces opérateurs ne fonctionnent qu'avec la fonction `equals(other: Any?): Boolean` qui peut être surchargée pour fournir une fonction personnalisée de comparaison d'égalité. Toute autre fonction avec le même nom (par exemple `equals(other: Foo)`) ne sera pas appelée.

Note : Il n'est pas possible de surcharger les fonctions `===` et `!==` (tests d'identité), donc aucune convention n'existe pour ces opérateurs.

L'opération `==` est spéciale : elle est traduite en une expression complexe qui recherche les valeurs nulles. `null == null` est toujours vrai, et `x == null` est toujours faux, et n'invoque pas `x.equals()`.

OPERATOR OVERLOADING EN KOTLIN

Attention à la définition de l'égalité :

```kotlin
fun main() {
    val first = Integer(10)
    val second = Integer(10)

    println(first == second)        // 1
    println(first.equals(second))   // 2
    println(first === second)       // 3
}
```

A votre avis, quel sera l'affichage du code ci-dessus ?

- 1 => true
- 2 => true
- 3 => false

OPERATOR OVERLOADING EN KOTLIN

OPÉRATEURS DE COMPARAISON

Expression	Traduites en
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

Toutes les comparaisons sont traduites en appels à la fonction `compareTo`, qui doit nécessairement retourner un `Int`.

LAMBDA EXPRESSION EN KOTLIN

Les fonctions en Kotlin sont des fonctions "première classe" (first-class), c'est à dire qu'elles peuvent êtres stockées dans des variables, des structures de donnée, passées en argument et retournées depuis des fonctions d'ordre supérieur (higher-order functions). Vous pouvez utiliser les fonctions, comme n'importe quel autre type classique.

FONCTIONS D'ORDRE SUPÉRIEUR

Une fonction d'ordre supérieur est une fonction qui prend en paramètre, ou retourne une fonction.
Un très bon exemple est la fonctionnalité `fold` pour les collections, qui prend une valeur initiale pour l'accumulateur, et une fonction pour combiner les items. Le résultat est obtenu en appliquant la fonction sur l'item courant, et l'accumulateur, pour en faire changer la valeur. Exemple :

```kotlin
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
    ): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

Dans le code ci-dessus, le paramètre `combine` à un type de fonction `(R, T) -> R`, donc il accepte une fonction qui prend 2 arguments, de types `R` et `T` et retourne une valeur de type `R`. Cette fonction est appelée dans une boucle `for`, et la valeur de retour est ensuite assignée à l'accumulateur.

LAMBDA EXPRESSION EN KOTLIN

Pour appeler la méthode `fold`, nous devons passer une instance de type fonction en argument, et les expression lambdas sont courament utilisées à cet usage :

```kotlin
val items = listOf(1, 2, 3, 4, 5)

// Lambdas are code blocks enclosed in curly braces.
items.fold(0, {
    // When a lambda has parameters, they go first, followed by '->'
    acc: Int, i: Int ->
    print("acc = $acc, i = $i, ")
    val result = acc + i
    println("result = $result")
    // The last expression in a lambda is considered the return value:
    result
})
```

```kotlin
// Parameter types in a lambda are optional if they can be inferred:
val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

// Function references can also be used for higher-order function calls:
val product = items.fold(1, Int::times)
```

LAMBDA EXPRESSION EN KOTLIN

EXPRESSION LAMBDA ET FONCTIONS ANONYMES

Une expression lambda, ou une fonction anonyme, sont des fonctions littérales, c'est à dire des fonctions qui ne sont pas déclarées, mais qui sont passées directement comme expression. Dans l'exemple suivant :

```
1 max(strings, { a, b -> a.length < b.length })
```

La fonction `max` est une fonction d'ordre supérieur, qui prend une fonction en second paramètre. Cette argument est une expression, qui est elle même une fonction : une fonction littérale qui correspond à la fonction nommée suivante :

```
1 fun compare(a: String, b: String): Boolean = a.length < b.length
```

LAMBDA EXPRESSION EN KOTLIN

SYNTAXE DE L'EXPRESSION LAMBDA

La syntaxe complète d'une expression lambda est la suivante :

```
1 val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

Une expression lambda est toujours entourée d'accolades. Les paramètres dans la syntaxe complète, sont déclarés dans ces accolades, et leur typage est optionnel. Le corp est situé après la ->. Si le type de retour inféré de la lambda n'est pas `Unit`, la dernière (et possiblement seule) expression dans le corps de la lambda est considéré comme la valeur de retour.
En retirant toutes les annotations optionnelles, le code devient :

```
1 val sum = { x: Int, y: Int -> x + y }
```

LAMBDA EXPRESSION EN KOTLIN

PASSER UNE LAMBDA EN PARAMÈTRE

En Kotlin, il y a une convention : si le dernier paramètre d'une fonction est une fonction, alors l'expression lambda, passée en paramètre peut être placée à l'extérieur des parenthèses :

```
1 val product = items.fold(1) { acc, e -> acc * e }
```

Cette notation est appelée lambda de fin (trailing lambda).
Si la lambda, est le seul argument, alors les parenthèses peuvent être complètement omises :

```
1 run { println("...") }
```

LAMBDA EXPRESSION EN KOTLIN

IT : LE NOM IMPLICITE DU PARAMÈTRE UNIQUE

Il est courant qu'une expression lambda ait un unique paramètre. Si le compilateur peut déterminer la signature de lui même, alors il n'est pas obligatoire de déclarer le paramètre unique, et omètre par la même occasion –>. Le paramètre sera implicitement déclaré avec le mon `it` :

```
1 var ints = listOf(0, 1,-2,3,-1)
2 ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

LAMBDA EXPRESSION EN KOTLIN

EXERCICE

Écrivez la lambda qui permet d'aditionner deux `Int` et stockez la dans une variable. Appelez cette lambda stockée à partir de la variable avec la fonction `invoke()`.

```
1 val sum = TODO()
```

LAMBDA EXPRESSION EN KOTLIN

SOLUTION

```
1 fun main() {
2     val sum = { x: Int, y: Int -> x + y }
3     print(sum.invoke(3, 5))
4 }
```

Ou de manière plus concise :

```
1 fun main() {
2     val sum = { x: Int, y: Int -> x + y }
3     println(sum(3, 5))
4 }
```

LAMBDA EXPRESSION EN KOTLIN

EXERCICE

Supposons que vous ayez besoin de développer une calculatrice. Commencez par écrire les lambdas correspondant aux opérations de base (addition, soustraction, multiplication et division), stockez les dans des variables, et tester leur usage avec la fonction `executeOperation` écrite ci-dessous.

```
1 inline fun executeOperation(x: Int, y: Int, operation: (Int, Int) -> Int) = operation(x, y)
```

`inline` permet d'optimiser l'appel de la méthode (il est optionnel).

LAMBDA EXPRESSION EN KOTLIN

SOLUTION

```kotlin
val addition = { x: Int, y: Int -> x + y }

val subtraction = { x: Int, y: Int -> x - y }
val multiplication = { x: Int, y: Int -> x * y }
val division = { x: Int, y: Int -> x / y }

inline fun executeOperation(x: Int, y: Int, operation: (Int, Int) -> Int) = operation(x, y)

fun main() {
    println(executeOperation(10, 5, addition))
    println(executeOperation(10, 5, subtraction))
    println(executeOperation(10, 5, multiplication))
    println(executeOperation(10, 5, division))
}
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Exemple de code en Java :

```java
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d(TAG, "User clicked button");
    }
});
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Qui va afficher un message de log dans la console, de type DEBUG :

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d(TAG, "User clicked button");
    }
});
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

La version utilisant les lambdas Java :

```
1 button.setOnClickListener ( v -> {
2     Log.d(TAG, "User clicked button");
3 });
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

L'équivalent en Kotlin :

```
1 button.setOnClickListener( { v: View -> Log.d(TAG, "User clicked button"); })
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Le ; en fin de ligne n'est pas nécessaire :

```
1 button.setOnClickListener( { v: View -> Log.d(TAG, "User clicked button"); } )
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

La lambda peut être sortie des parenthèses de la méthode :

```
1 button.setOnClickListener( { v: View -> Log.d(TAG, "User clicked button")  } )
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Les parenthèses de la méthode ne sont pas nécessaires :

```
1 button.setOnClickListener() { v: View -> Log.d(TAG, "User clicked button")   }
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Le type du paramètre est inféré par le compilateur :

```
1 button.setOnClickListener   { v: View -> Log.d(TAG, "User clicked button")   }
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Le paramètre est unique, et n'est pas utilisé dans notre expression, donc inutile :

```
1 button.setOnClickListener    { v          -> Log.d(TAG, "User clicked button")   }
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Ce qui nous donne au final le code suivant ...

```
1 button.setOnClickListener   {            Log.d(TAG, "User clicked button")   }
```

SIMPLIFICATION D'UNE MÉTHODE JAVA EN KOTLIN

Avec les espaces superflu en moins, cela donne :

```
button.setOnClickListener{ Log.d( TAG, "User clicked button" ) }
```

Qui correspond au code Java :

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d(TAG, "User clicked button");
    }
});
```

Ou encore :

```
button.setOnClickListener ( v -> {
    Log.d(TAG, "User clicked button");
});
```

EXTENSIONS

Kotlin permet d'étendre les fonctionnalités d'une classe, sans avoir besoin d'en hériter ni d'utiliser le design pattern du décorateur. Il faut utiliser la déclaration spéciale appelée : extensions. Par exemple, il est possible d'ajouter des nouvelles fonctions à une classe d'une librairie externe, dont le code source n'est pas disponible. Il est possible d'utiliser ces fonctions comme si elles faisaient partie intégrante du code source original. Ce mécanisme est appelé *extension de fonctions*. Nous verrons par la suite, le mécanisme d'*extension de propriétés*, qui permet d'ajouter des propriétés à une classe existante.

EXTENSIONS

EXTENSIONS DE FONCTIONS EN KOTLIN

Pour déclarer une fonction d'extention, nous devons la préfixer par le nom du type receveur, c'est à dire le type qui va être étendu. L'exemple ci-dessous ajoute la fonction `swap` à `MutableList<Int>`

```
1 fun MutableList<Int>.swap(index1: Int, index2: Int) {
2     val tmp = this[index1] // 'this' corresponds to the list
3     this[index1] = this[index2]
4     this[index2] = tmp
5 }
```

Le mot clé `this`, dans la fonction d'extension, fait référence à l'objet receveur (qui est juste avant le point). Maintenant, nous pouvons appeler cette fonction sur tous les objets de type `MutableList<Int>`.

```
1 val list = mutableListOf(1, 2, 3)
2 list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

Testez ce code pour en comprendre la puissance.

EXTENSIONS

EXERCICE

Écrivez une extension à la classe `String` qui permet de mettre la première lettre d'une chaîne de caractère en majuscule et le reste en minuscule (pour afficher par exemple un prénom, non composé).

```
1 fun String.firstUpper(): Any = TODO()
```

EXTENSIONS

SOLUTION

```
1 fun String.firstUpper() = this.first().toUpperCase() + this.substring(1).toLowerCase()
2
3 fun main() {
4     println("tristan".firstUpper())
5     println("TRISTAN".firstUpper())
6 }
```

EXTENSIONS

EXTENSIONS DE PROPRIÉTÉS EN KOTLIN

De manière similaire, Kotlin permet l'extention de propriétés. Mais il faut garder en tête que nous ne pouvons pas vraiment ajouter des propriétés à un objet. En pratique c'est plus un moyen de récupérer et/ou modifier un élément de la classe que l'on souhaite étendre. Exemple

```
1 val <T> List<T>.lastIndex: Int
2     get() = size - 1
```

EXTENSIONS

EXERCICE

Écrivez une propriété d'extension `firstLetter` pour la classe `StringBuilder` qui permet d'accéder à la première lettre contenue dans l'objet.

```
var StringBuilder.firstLetter: Char
    get() = TODO()
    set(value) = TODO()
```

EXTENSIONS

SOLUTION

```
1  import java.lang.StringBuilder
2
3  var StringBuilder.firstLetter: Char
4      get() = get(0)
5      set(value) = this.setCharAt(0, value)
6
7  fun main() {
8      val message = StringBuilder("hello world !")
9      println("${message.firstLetter} is the first letter of $message")
10     message.firstLetter = 'H'
11     println("${message.firstLetter} is the first letter of $message")
12 }
```

CLOSURES EN KOTLIN

Une expression lambda, ou une fonction anonyme (ou une fonction locale, ou encore une expression object)

peuvent accéder à leur environnement (closure), c'est à dire aux variables définies en dehors de leur périmètre (scope). Exemple :

```
1 var sum = 0
2 ints.filter { it > 0 }.forEach {
3     sum += it
4 }
5 print(sum)
```

CLOSURES EN KOTLIN

EXERCICE

Reprenez l'exemple précédent, et tester l'accès de la lambda à la variable `sum` située en dehors des accolades. N'oubliez pas de déclarer la liste d'`Int`s contenus dans la variable `ints`.

```
1 var sum = 0
2 ints.filter { it > 0 }.forEach {
3     sum += it
4 }
5 print(sum)
```

CLOSURES EN KOTLIN

SOLUTION

```
fun main() {
    var ints = listOf(1, 2, 3, 4, 5, 6)
    var sum = 0
    ints.filter { it > 0 }.forEach {
        sum += it
    }
    print(sum)
}
```

DÉLÉGATION

- Concept de délégation en Kotlin
- Délégation de fonctions en Kotlin
- Délégation de propriétés en Kotlin
- Bonnes et mauvaises pratiques

CONCEPT DE DÉLÉGATION EN KOTLIN

Le principe de délégation en informatique est le fait d'assigner le traitement d'une action d'une instance à une

autre. La délégation peut être faite de manière definitive, ou temporaire. L'héritage est une délégation statique immuable. Par exemple, supposons que nous voulons que la classe `B` ait les même fonctionnalités que la classe `A` et que la classe `B` soit une classe `A`. Dans ce cas là, vous pouvez utiliser l'héritage. Cela donne une relation permanente entre les classes. En utilisant la délégation, vous pouvez passer en paramètre un autre objet d'un autre type, un sous type de la classe A, par exemple, à l'instance de B. Ce qui rend le mécanisme de délégation extrêmement puissant.

DÉLÉGATION DE FONCTIONS EN KOTLIN

EXEMPLE EN KOTLIN

```kotlin
interface Nameable {
    var name: String
}

class JackName : Nameable {

    override var name: String = "Jack"
}

class LongDistanceRunner: Runnable {
    override fun run() {
        println("long")
    }
}

class Person(name: Nameable, runner: Runnable): Nameable by name, Runnable by runner
```

```kotlin
fun main(args: Array<String>) {
    val person = Person(JackName(), LongDistanceRunner())
    println(person.name) //Jack
    person.run() //long
}
```

DÉLÉGATION DE FONCTIONS EN KOTLIN

Ce mécanisme permet d'étendre les fonctionnalités d'une classe, facilement. Pas besoin d'implémenter les méthodes de l'interface manuellement, c'est le compilateur qui s'en charge.
Il est alors possible de faire de l'héritage multiple, tel que dans l'exemple précédent.
Pour cela, il suffit d'utiliser le mot clé `by`.

DÉLÉGATION DE FONCTIONS EN KOTLIN

EXERCICE

Reprenez l'exemple précédent, implémentez une nouvelle classe `class TristanName : Nameable` et `class ShortDistanceRunner: Runnable` qui seront utilisées par la classe Person :

```
fun displayPerson(person: Person){
    println(person.name)
    person.run()
}
```

DÉLÉGATION DE FONCTIONS EN KOTLIN

SOLUTION

```kotlin
interface Nameable {
    var name: String

}

class JackName : Nameable {
    override var name: String = "Jack"
}
class TristanName : Nameable {
    override var name: String = "Tristan"
}

class LongDistanceRunner : Runnable {
    override fun run() {
        println("long")
    }
```

DÉLÉGATION DE FONCTIONS EN KOTLIN

EXERCICE

Reprenez l'exemple précédent, ajoutez des attributs à l'interface `Nameable`, par exemple `firstName`, `companyName` et `cityName`, il faut implémenter ces nouveaux attributs dans les classes dérivées, et tester que notre objet `Person` à bien hérité de ces nouvelles propriétés .

DÉLÉGATION DE FONCTIONS EN KOTLIN

SOLUTION

```kotlin
interface Nameable {
    var name: String

    var firstName: String
    var companyName: String
    var cityName: String
}

class JackName : Nameable {
    override var name: String = "Jack"
    override var firstName: String = "Terence"
    override var companyName: String = "IBM"
    override var cityName: String = "London"
}
class TristanName : Nameable {
    override var name: String = "SALAUN"
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

Comme vous le savez maintenant, pour utiliser une propriété en Kotlin, on utilise simplement son nom, préfixé d'un `.` :

```
class Foo {
    var prop: String? = null

}

val foo = Foo()
foo.prop = "something"
val another = foo.prop
```

Vous pouvez aussi écrire des getters et des setters sur mesure :

```
var str: String
    get() = this.toString()
    set(value) {
        println(value)
        field = value
    }
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

Parfois nos méthodes getters et setters contiennent le même code. Pour éviter la duplication du code, ou simplement encapsuler la logique de ces fonctions, vous pouvez utiliser la délégation de propriétés :

```kotlin
import kotlin.reflect.KProperty

class Example {
    val someName by NameDelegate()
    val otherName by NameDelegate()
}

class NameDelegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return property.name
    }
}

fun main() {
    val example = Example()
```

Que va afficher le code ci-dessus ?
Est il possible d'affecter une valeur à l'attribut `someName`, et pourquoi ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

Autre exemple de délégation :

```kotlin
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

Reprenez l'exemple précédent, sans changer le code de la fonction, main, et remplacer la délégation `NameDelegate` par la délégation `Delegate` ci-dessus. Que constatez vous ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

DELEGATE STANDARDS

Kotlin fournit plusieurs classes de délégations standards :

- "lazy properties" : les valeurs sont calculées lors du premier accès.
- "observable properties" : les listeners sont notifiés lors des changements de la propriété.
- "vetoable properties" : il est possible de mettre un veto sur le changement d'une valeur.
- "notNull properties" : permet de vérifier que la valeur est définie avant un premier accès.

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

DELEGATE `lazy`

Exemple de mise en oeuvre de la délégation `lazy` :

```
1 val myVar: String by lazy {
2     println("Lazy init")
3     "Hello"
4 }
5 println("myVar is not initialized yet")
6 println(myVar + " My dear friend")
```

De manière plus classique/concise, nous utiliserons :

```
1 val myString by lazy { "Some Value" }
```

Que va afficher le code ci-dessus ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

DELEGATE `observable`

Comme toutes les délégations standards, la classe de délégation `observable` se trouve dans la classe `Delegates`. Elle prend en paramètre une valeur initiale et une lambda qui sera exécutée à chaque fois que la valeur du champ sera modifiée, sa signature est la suivante :

```
1 inline fun <T> observable(
2     initialValue: T,
3     crossinline onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Unit
4 ): ReadWriteProperty<Any?, T>
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXEMPLE D'UTILISATION

```
1  var observed = false
2  var max: Int by Delegates.observable(0) { property, oldValue, newValue ->
3      observed = true
4  }
5
6  println(max) // 0
7  println("observed is ${observed}")
8
9  max = 10
10 println(max) // 10
11 println("observed is ${observed}")
```

Que va afficher le code ci-dessus ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXERCICE

Écrivez une délégation `by observable` pour la variable `maxCount` qui affichera le nom de la propriété modifiée, son ancienne valeur et la nouvelle.

```
1 fun main() {
2     var maxCount: Int = 0
3
4     maxCount = 5
5     maxCount = 10
6     maxCount = 20
7 }
```

Qu'affiche le code ci-dessus ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

SOLUTION

```kotlin
import kotlin.properties.Delegates

fun main() {
    var maxCount: Int by Delegates.observable(initialValue = 0) { property, oldValue, newValue ->
        println("${property.name} is being changed from $oldValue to $newValue")
    }

    maxCount = 5
    maxCount = 10
    maxCount = 20
}
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

DELEGATE `vetoable`

Le syntaxe est pratiquement la même que `observable`, sauf que la lambda, doit retourner un `Boolean`, qui indique si la valeur doit être modifiée, on non.
Cette délégation est parfaite pour garantir qu'une valeur est comprise dans un interval cohérent, ou pour implémenter un framework de validation simplement.

```
1  var age: Int by vetoable(initialValue = 0) { property, oldValue, newValue ->
2    newValue > 0
3  }
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXERCICE

Essayez d'affecter les valeurs suivantes à la variable `age` déclarée comme vu précédemment :

- 10
- -1
- 30
- 0

Affichez la valeur de la variable à chaque affectation.

```
1  var age: Int by vetoable(initialValue = 0) { property, oldValue, newValue ->
2    newValue > 0
3  }
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

SOLUTION

```kotlin
import kotlin.properties.Delegates.vetoable

fun main() {
    var age: Int by vetoable(initialValue = 0) { property, oldValue, newValue ->
        newValue > 0
    }

    age = 10
    println(age)
    age = -1
    println(age)
    age = 30
    println(age)
    age = 0
    println(age)
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXERCICE

Modifiez la condition de test pour afficher un message quand la valeur est rejetée.

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

SOLUTION

```
1  import kotlin.properties.Delegates
2
3  fun main() {
4      var age: Int by Delegates.vetoable(initialValue = 0) { property, oldValue, newValue ->
5          if (newValue > 0) true else {println("${property.name} rejected value $newValue staying at value $oldValue"); false}
6      }
7
8      age = 10
9      println(age)
10     age = -1
11     println(age)
12     age = 30
13     println(age)
14     age = 0
15     println(age)
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

DELEGATE `notNull`

C'est le plus simple des délégations standards. Il fonctionne comme `lateinit`, dans le sens ou il lève une `IllegalStateException` si la variable est accédé avant d'être initialisée.

```
1 var age by notNull<Int>()
2 fun main() = println(age)
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXERCICE

Modifiez le code pour le rendre valide (indiquez que nous allons initialiser la variable tardivement :

```
var person1:Person

fun main(args: Array<String>) {
    // initializing variable lately
    person1 = Person("Ted",28)
    print(person1.name + " is " + person1.age.toString())
}

data class Person(var name:String, var age:Int)
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

SOLUTION

```
1 lateinit var person1:Person
2
3 fun main(args: Array<String>) {
4     // initializing variable lately
5     person1 = Person("Ted",28)
6     print(person1.name + " is " + person1.age.toString())
7 }
8
9 data class Person(var name:String, var age:Int)
```

Une remarque sur les bonnes pratiques utilisées (ou pas) dans cet exemple ?

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

EXERCICE

Écrivez une classe de délégation, qui permet d'afficher un message, quand une propriété est accédé, ou modifiée. Héritez de la classe `ReadWriteProperty`. Pour tester son usage nous écrirons une classe `Demo` contenant un attribut `var name` qui sera déléguée à notre `class LogDelegate<T>` :

```
1 class LogDelegate<T> : ReadWriteProperty<Any, T?> {}
```

DÉLÉGATION DE PROPRIÉTÉS EN KOTLIN

SOLUTION

```kotlin
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty

class LogDelegate<T> : ReadWriteProperty<Any, T?> {
    private var value: T? = null
    override fun getValue(thisRef: Any, property: KProperty<*>): T? {
        println("LOG get $value")
        return value
    }
    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        println("LOG set : $value")
        this.value = value
    }
}

```

BONNES ET MAUVAISES PRATIQUES

- utilisez la délégation pour factoriser le code pour adhérer à la bonne pratique "DRY" (Don't Repeat Yourself).
- Utiliser la délégation notNull pour les types primitifs, car lateinit ne peut pas s'appliquer (essayez `lateinit var age: Int`)

GENERICS

- Generics en Kotlin
- Generics et invariance en Kotlin
- Covariance en Kotlin
- Contravariance en Kotlin
- Bonnes et mauvaises pratiques

GENERICS EN KOTLIN

Tout comme en Java, en Kotlin les classes peuvent avoir des paramètres typés :

```
1 class Box<T>(t: T) {
2     var value = t

3 }
```

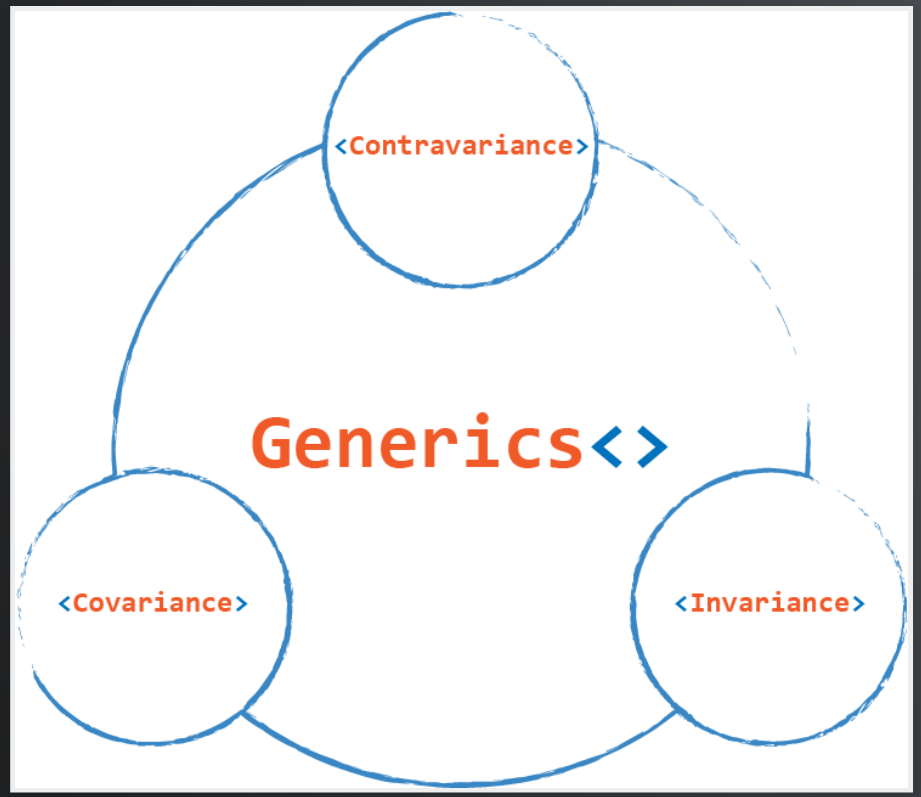
En général pour créer une instance de ce genre de classe, nous devons fournir le type de l'argument :

```
1 val box: Box<Int> = Box<Int>(1)
```

Mais si le type du paramètre peut être inféré, par exemple depuis le type des paramètres du constructeur, ou par un autre moyen, alors il est possible d'omètre le type des arguments :

```
1 val box = Box(1) // 1 est de type Int, donc le compilateur peut en déduire que nous utilisons un type : Box<Int>
```

GENERICS ET INVARIANCE EN KOTLIN



GENERICS ET INVARIANCE EN KOTLIN

INVARIANCE

Quand on utilise des classes simples, étendre de ces classes est simple. Mais quand on commence à utiliser des classes génériques, alors les règles se compliquent un peu.
L'invariance exprime le fait que bien que 2 classes aient une relation de hiérarchie, un type complexe (un généric) ne suit pas cette même hiérarchie. Pour que cela soit plus clair, prenons un exemple :

```
1 open class A
2 open class B : A()
```

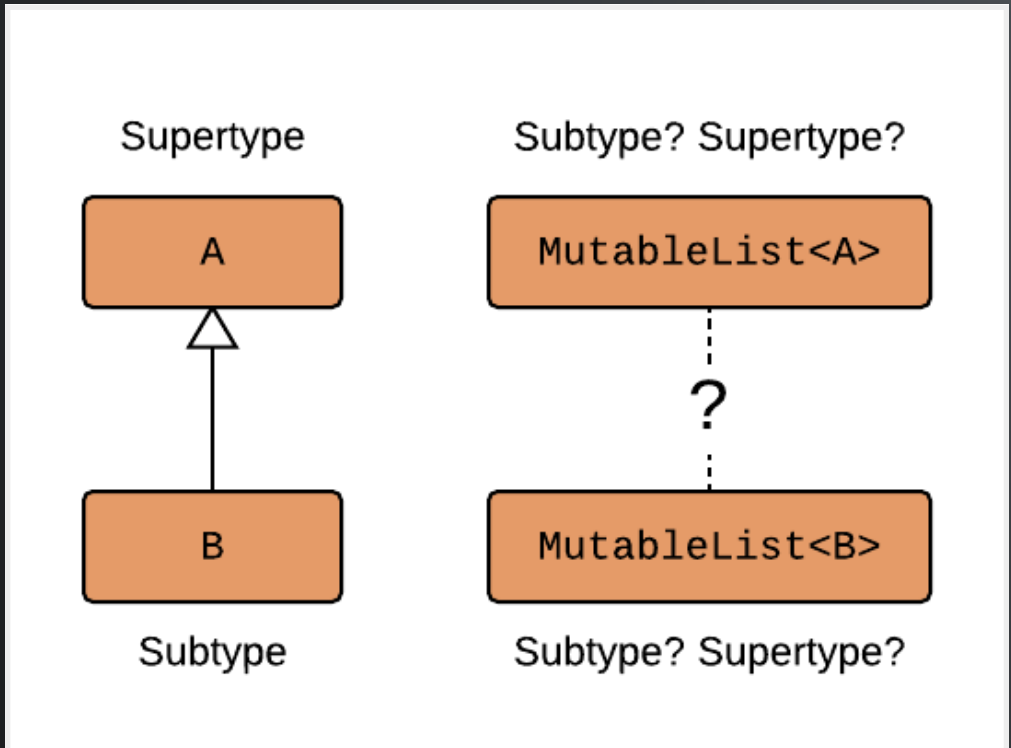
Considérons les types suivants :

```
1 MutableList<A>
2 MutableList<B>
```

GENERICS ET INVARIANCE EN KOTLIN

RELATION ?

Quelle est la relation entre les deux `MutableList` ?



GENERICS ET INVARIANCE EN KOTLIN

TESTONS

```
1 open class A
2 open class B : A()
```

```
 1 fun main() {
 2     var objectA = A()
 3     var objectB = B()
 4     var listofA: MutableList<A> = mutableListOf<A>()
 5     var listofB: MutableList<B> = mutableListOf<B>()
 6     println("objectA is A " + (objectA is A))
 7     println("objectB is B " + (objectB is B))
 8     println("objectA is B " + (objectA is B))
 9     println("objectB is A " + (objectB is A))
10     println("listofA is MutableList<A> " + (listofA is MutableList<A>))
11     println("listofB is MutableList<B> " + (listofB is MutableList<B>))
12     println("listofA is MutableList<B> " + (listofA is MutableList<B>))
13     println("listofB is MutableList<A> " + (listofB is MutableList<A>))
14 }
```

GENERICS ET INVARIANCE EN KOTLIN

RÉPONSE

Comme nous avons pu le constater, les 2 listes n'ont aucune liaison entre elles, la `MutableList` est invariante.

COVARIANCE EN KOTLIN

La covariance exprime la relation entre deux jeux de données dont les deux sous-types vont dans la même direction, reprenons notre exemple :

```
1 open class A
2 open class B : A()
```

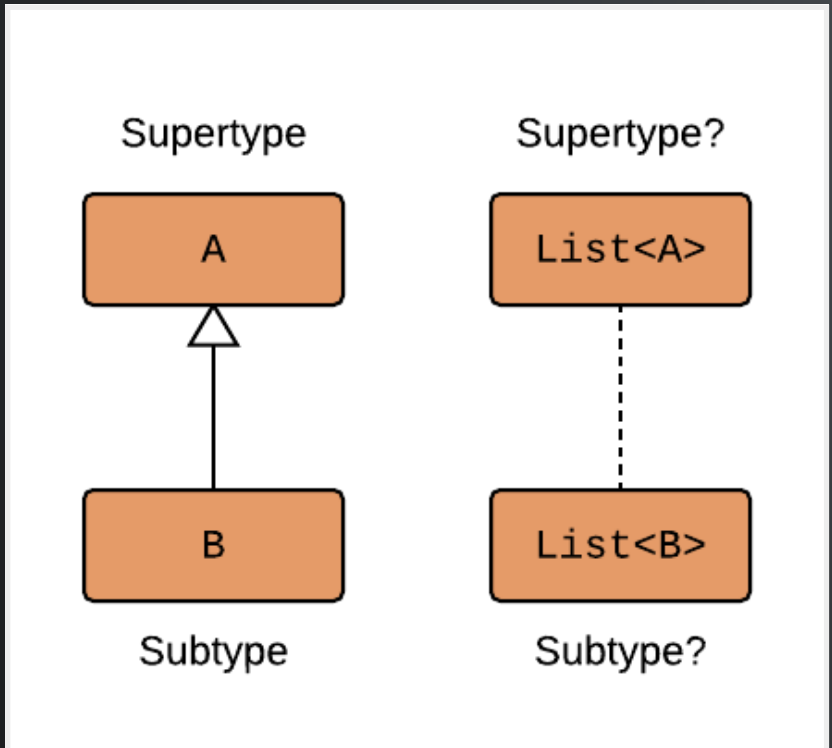
Considérons les types en lecture seule suivants :

```
1 List<A>
2 List<B>
```

COVARIANCE EN KOTLIN

RELATION ?

Quelle est la relation entre les deux `List` ?



COVARIANCE EN KOTLIN

TESTONS

```
1 open class A
2 open class B : A()
```

```
 1 fun main() {
 2     var objectA = A()
 3     var objectB = B()
 4     var listofA: List<A> = listOf<A>()
 5     var listofB: List<B> = listOf<B>()
 6     println("objectA is A " + (objectA is A))
 7     println("objectB is B " + (objectB is B))
 8     println("objectA is B " + (objectA is B))
 9     println("objectB is A " + (objectB is A))
10     println("listofA is List<A> " + (listofA is List<A>))
11     println("listofB is List<B> " + (listofB is List<B>))
12     println("listofA is List<B> " + (listofA is List<B>))
13     println("listofB is List<A> " + (listofB is List<A>))
14 }
```

COVARIANCE EN KOTLIN

RÉPONSE

Comme nous avons pu le constater, le type `List` est covariant : `List<B>` est un sous-type de `List<A>`. Cela fonctionne car la `List<T>` étant immutable, lors de l'affectation de `List<B>` dans une variable de type `List<A>`, une copie de la liste est effectuée.

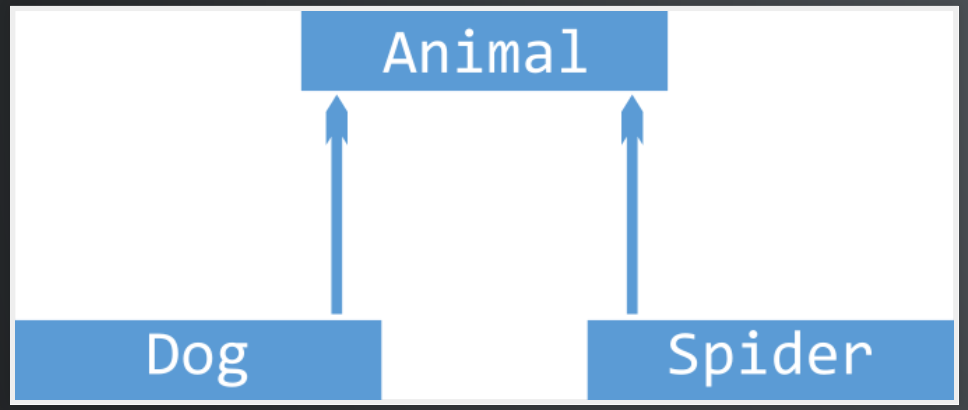
COVARIANCE EN KOTLIN

EXERCICE

Prenons les classes suivantes :

```kotlin
abstract class Animal(val size: Int)

class Dog(val cuteness: Int): Animal(100)
class Spider(val terrorFactor: Int): Animal(1)
```



COVARIANCE EN KOTLIN

VALIDONS LA HIÉRARCHIE

Vérifions que `Dog` et `Spider` sont bien des sous types de `Animal` :

```
1 val dog: Dog = Dog(10)
2 val spider: Spider = Spider(9000)
3 var animal: Animal = dog
4 animal = spider
```

Tout est OK.

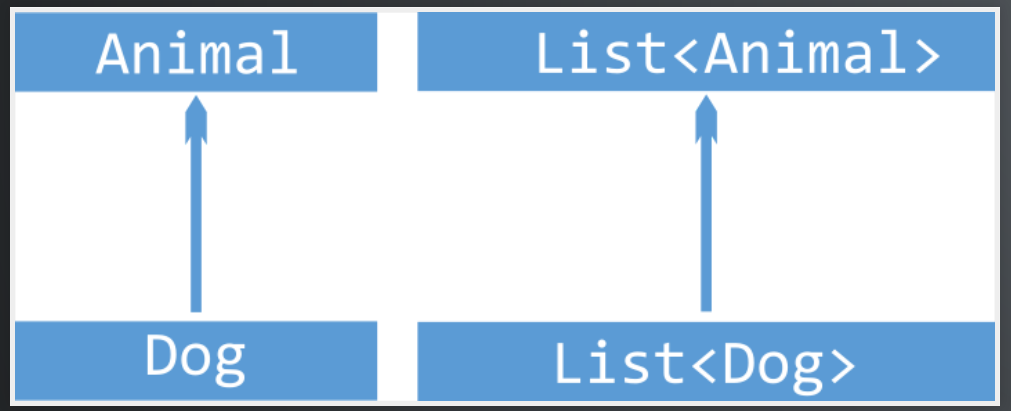
COVARIANCE EN KOTLIN

TEST AVEC LES `List`

Faisons le même test que précédemment, en utilisant des `List` :

```
1 val dogList: List<Dog> = listOf(Dog(10), Dog(20))
2 val animalList: List<Animal> = dogList
```

La liste étant immuable (impossible de changer sa valeur après initialisation), cela fonctionne bien, car une copie de la liste est réalisée.



CONTRAVARIANCE EN KOTLIN

Reprenons l'exemple des animaux, et supposons que nous souhaitions comparer ces animaux. Créons pour cela une `interface Compare<T>` avec une méthode `compare(T item1, T item2)`, qui permet de

comparer deux items.
Comparons les chiens du plus mignon au moins mignon. Le code du comparateur sera :

```
1 val dogCompare: Compare<Dog> = object: Compare<Dog> {
2   override fun compare(first: Dog, second: Dog): Int {
3     return first.cuteness - second.cuteness
4   }
5 }
```

Essayons de l'affecter à une variable qui permet de comparer les `Animal` :

```
1 val animalCompare: Compare<Animal> = dogCompare // Compiler error
```

CONTRAVARIANCE EN KOTLIN

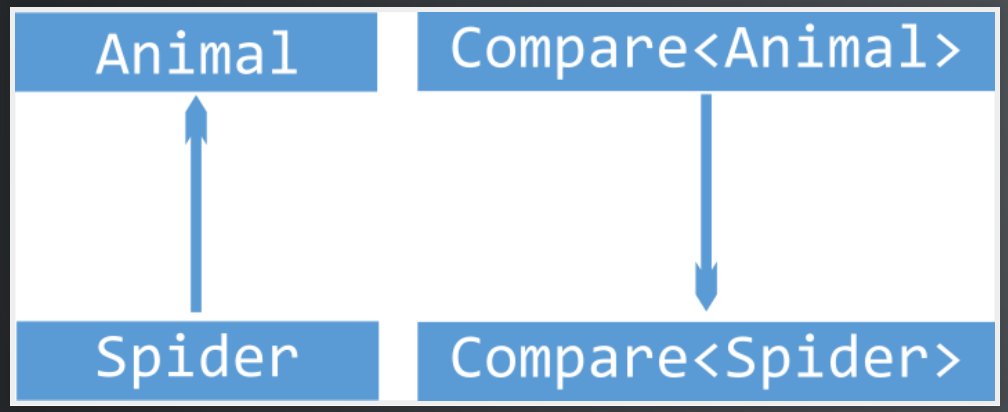
COMPARAISON D'ANIMAUX

Si nous voulons comparer tous les animaux, le mécanisme doit fonctionner pour tous les chiens et les araignées :

```
1 val animalCompare: Compare<Animal> = object: Compare<Animal> {
2   override fun compare(first: Animal, second: Animal): Int {
3     return first.size - second.size
4   }
5 }
6 val spiderCompare: Compare<Spider> = animalCompare // Works nicely!
```

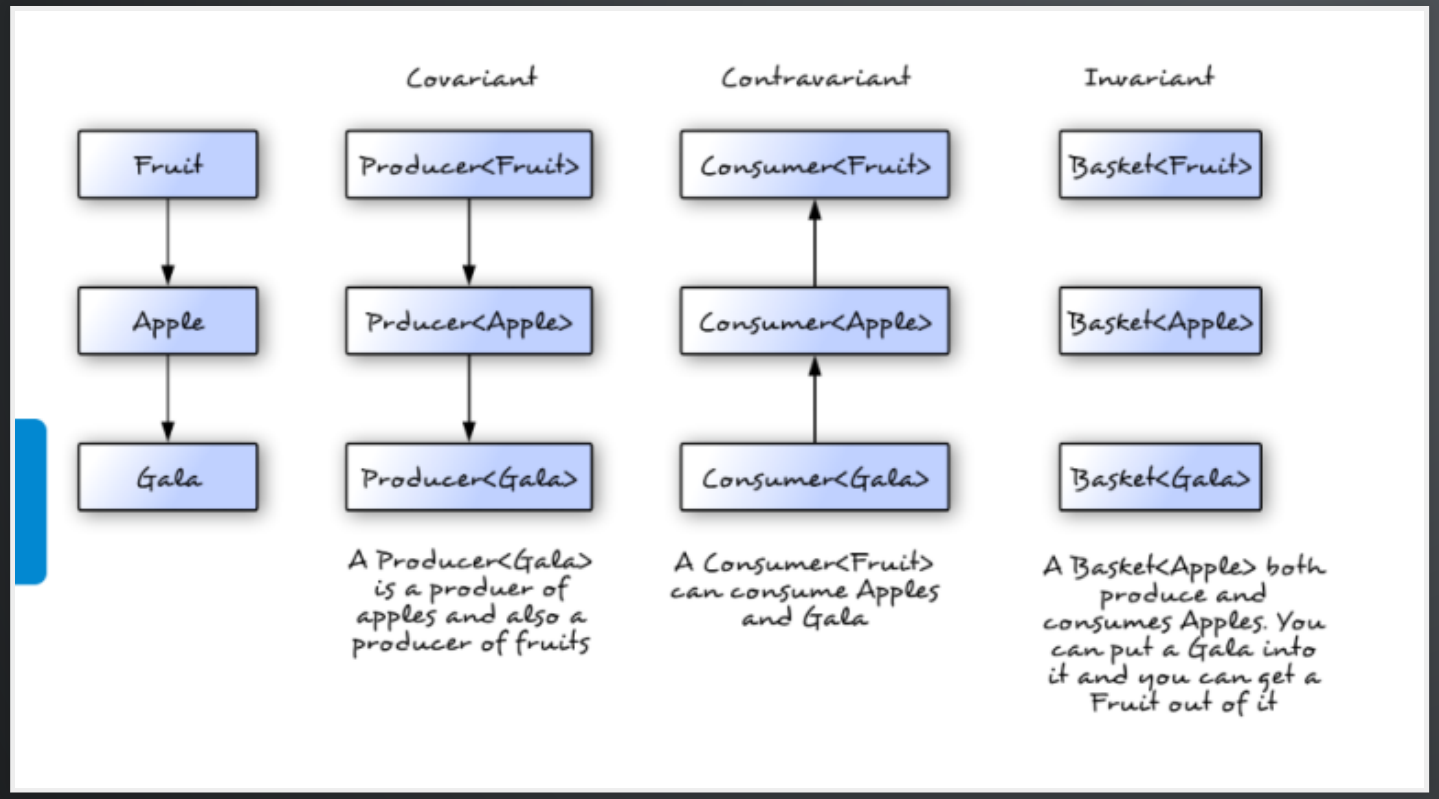
CONTRAVARIANCE EN KOTLIN

La relation entre le type et le type complexe sont inversés, on parle de contravariance.



AUTRE EXEMPLE AVEC DES FRUITS

PRENONS UN AUTRE EXEMPLE AVEC DES FRUITS



AUTRE EXEMPLE AVEC DES FRUITS

DÉFINITION DES FRUITS

```
1 open class Fruit
2 open class Apple: Fruit() // Apple extends Fruit
3 class Gala: Apple() // Gala extends Apple
```

AUTRE EXEMPLE AVEC DES FRUITS

EXPRESSION DES VARIANCES

```
class Variance{
    val fruitProducer: () -> Fruit = ::Fruit
    val appleProducer: () -> Apple = ::Apple
    val galaProducer: () -> Gala = ::Gala
    val fruitConsumer: (Fruit) -> Unit = ::eatFruit
    val appleConsumer: (Apple) -> Unit = ::eatApple
    val galaConsumer: (Gala) -> Unit = ::eatGala
    val newFruitProducer1: () -> Fruit = appleProducer
    val newFruitProducer2: () -> Fruit = galaProducer
    val newAppleProducer: () -> Apple = galaProducer
    val newGalaConsumer1: (Gala) -> Unit = appleConsumer
    val newGalaConsumer2: (Gala) -> Unit = fruitConsumer
    val newAppleConsumer: (Fruit) -> Unit = fruitConsumer
}
```

AUTRE EXEMPLE AVEC DES FRUITS

FONCTIONS UTILITAIRES

```
1 fun eatFruit(fruit: Fruit) {}
2 fun eatApple(apple: Apple) {}
3 fun eatGala(fruit: Gala) {}
```

AUTRES FONCTIONNALITÉS

- Casting de types en Kotlin
- Tuples
- Deconstructing Values
- Gestion des exceptions
- Déclaration de constantes
- Annotation en Kotlin
- Bonnes et mauvaises pratiques

CASTING DE TYPES EN KOTLIN

Il est possible de tester le type d'une variable :

```
1 fun foo(x: Any) {
2     if (x is Person) {
3         println("${x.name}") // This wouldn't compile outside the if
4     }
5 }
```

Notez que nous n'avons pas besoin de caster l'objet `x`, pour quelle raison, à votre avis ?

Pour changer le type d'une variable, il est possible de le faire explicitement :

```
1 val p = x as Person
```

Si l'objet n'est pas vraiment de type `Person` (ou d'une sous classe), alors l'exception `ClassCastException` sera levée.
Si vous n'êtes pas sûr du type, mais que vous pouvez vous satisfaire d'un null, si l'instance n'est pas de type `Person`, vous pouvez utiliser `as?`. Notez que le type de retour sera `Person?` :

```
1 val p = x as? Person
```

CASTING DE TYPES EN KOTLIN

EXERCICE

Tester le cast de la variable `x` avec les 2 méthodes (`as` et `as?`.
Une première fois avec `x` de type `Any` contenant une instance de `Person`, et ensuite contenant une instance de `Other`.

```
1 class Person(var name: String)
2 class Other()
```

Vous testerez ensuite la différence entre les 2 autres types de cast (`String` en `Int`) :

```
1 val value = "Message"
2 println(value as? Int)
3 println(value as Int?)
```

CASTING DE TYPES EN KOTLIN

SOLUTION

```kotlin
class Person(var name: String)
class Other()

fun main() {
    fun foo(x: Any) {
        if (x is Person) {
            println("${x.name}") // This wouldn't compile outside the if
        }
    }

    var x: Any = Person("SALAUN")
    var p = x as Person
    var pNull = x as? Person

    x = Other()
```

TUPLES

LES TYPES STANDARDS

Kotlin propose dans la librairie standard des Tuples :

- `Pair` : qui contient 2 éléments.
- `Triple` : qui contient 3 éléments.

TUPLES

`Pair`

Tous les détails sur la page officielle.

La classe peut contenir 2 éléments. Pratique lorsque l'on doit retourner deux valeurs dans une fonction.

- `first` contient la première valeur.
- `second` contient la seconde valeur.

TUPLES

`Triple`

Tous les détails sur la page officielle.

La classe peut contenir 3 éléments. Pratique lorsque l'on doit retourner trois valeurs dans une fonction.

- `first` contient la première valeur.
- `second` contient la seconde valeur.
- `third` contient la troisième valeur.

TUPLES

EXERCICE

Écrivez une variable référençant une `Pair` de valeur, et les afficher séparément.

Procédez de la même manière avec `Triple`.

Les signatures des classes sont rappelées ci-dessous :

```
1 data class Pair<out A, out B> : Serializable
2 data class Triple<out A, out B, out C> : Serializable
```

TUPLES

SOLUTION

```
fun main() {

    val pairValue= Pair(1, "x")
    val tripleValue = Triple("Tristan","SALAUN", 40)

    println(pairValue.first)
    println(pairValue.second)
    println(tripleValue.first)
    println(tripleValue.second)
    println(tripleValue.third)
}
```

DECONSTRUCTING (DESTRUCTURING) VALUES

Parfois il est pratique de déstructurer un objet en plusieurs variables, par exemple :

```
1 val (name, age) = person
```

Cette syntaxe est appelée "destructuring declaration". Cette déclaration crée plusieurs variables en une seule fois. Dans notre exemple, les 2 variables déclarées, peuvent être utilisées de manière indépendantes :

```
1 println(name)
2 println(age)
```

Le code généré correspond à :

```
1 val name = person.component1()
2 val age = person.component2()
```

Les fonctions `component1()` et `component2()` est un nouvel exemple des conventions utilisées dans Kotlin (tout comme +, *, ...).

Il est aussi possible d'utiliser le "destructuring" dans une boucle for :

```
1 for ((a, b) in collection) { ... }
```

DECONSTRUCTING (DESTRUCTURING) VALUES

EXEMPLE : RETOURNER DEUX VALEURS DEPUIS UNE FONCTION

Vous avez deux valeurs à retourner depuis une fonction, par exemple un objet `result` et un `status`. Une façon pratique de le faire, en Kotlin, est de déclarer une classe `data`, et retourner une instance de cet objet.

```
1 data class Result(val result: Int, val status: Status)
2 fun function(...): Result {
3     // computations
4
5     return Result(result, status)
6 }
7
8 // Now, to use this function:
9 val (result, status) = function(...)
```

Les classes `data` déclarent automatiquement les fonctions `componentN()`, donc la déstructuration fonctionne directement dans notre cas.

DECONSTRUCTING (DESTRUCTURING) VALUES

EXERCICE

Reprenez l'exemple avec les tuples et destructurez les `Pair` dans des variables `a` et `b`. Respectivement pour `Triple` et `c`, `d` et `e`.

DECONSTRUCTING (DESTRUCTURING) VALUES

SOLUTION

```
1  fun main() {
2
3      val (a, b)= Pair(1, "x")
4      val (c, d, e) = Triple("Tristan","SALAUN", 40)
5
6      println(a)
7      println(b)
8      println(c)
9      println(d)
10     println(e)
11
12     println("a = $a, b = $b, c = $c, d = $d and e = $e")
13 }
```

GESTION DES EXCEPTIONS

LES CLASSES D'EXCEPTION

Toutes les classes d'exceptions sont des descendantes de la classe `Throwable`. Toutes les exceptions ont un message, une pile d'exécution (stack trace), et une cause optionnelle.
Pour lever une exception, il faut utiliser l'expression `throw` :

```
1 throw Exception("Hi There!")
```

Pour attraper une exception, il faut utiliser l'expression `try` :

```
1 try {
2     // some code
3 }
4 catch (e: SomeException) {
5     // handler
6 }
7 finally {
8     // optional finally block
9 }
```

Il peut y avoir 0 ou plus blocs de `catch`, le block `finally` est optionnel. Toutefois il doit y avoir au moins un block catch ou un block finally.

GESTION DES EXCEPTIONS

TRY EST UNE EXPRESSION

Try est une expression, et en tant que telle, elle doit avoir une valeur de retour.

```
1 val numValue: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

La valeur retournée par l'expression est, soit la dernière expression du block try, ou la dernière expression du bloc catch. Le contenu du block finally n'affecte pas le résultat de l'expression.

GESTION DES EXCEPTIONS

EXERCICE

Testez différentes valeurs d'`input` (`"2005"` et `"azerty"`) pour affecter la valeur à `numValue` :

```
1 val numValue: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

GESTION DES EXCEPTIONS

SOLUTION

```
fun main() {
    var input = "2005"
    var numValue: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
    println(numValue)
    input = "azerty"
    numValue = try { parseInt(input) } catch (e: NumberFormatException) { null }
    println(numValue)
}
```

DÉCLARATION DE CONSTANTES

Une propriété qui est connue au moment de la compilation est annotée avec le mot clé `const`.
Elle peut être utilisée dans les annotations :

```
1 const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"
2
3 @Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

ANNOTATION EN KOTLIN

DÉFINITION

Les annotations sont utilisées pour attacher des méta-données auc classes, interfaces, paramètres, etc., lors de la compilation. Ces annotations peuvent avoir un impact sur l'exécution du code.

ANNOTATION EN KOTLIN

MÉTA ANNOTATION EN KOTLIN

Lors de la déclaration d'une annotation, il est possible d'ajouter des méta-informations. Ci-dessous en voici quelques unes :

Nom de l'annotation	Usage
@Target	Liste de tous les types d'éléments possibles qui peuvent être annotés avec cette annotation.
@Retention	Définie si l'annotation est stockée dans la fichier de la classe compilée, et s'il est visible via la réflexion lors de l'éxécution du programme.
@Repeatable	Définie si l'annotation est applicable plusieurs fois sur un même bloc de code.
@MustBeDocumented	Spécifie que l'annotation fait partis d'une API publique et doit donc être inclue dans la classe ou méthode.

ANNOTATION EN KOTLIN

EXEMPLE D'ANNOTATION

```
1 @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,

2 AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
3 @Retention(AnnotationRetention.SOURCE)
4 @MustBeDocumented
5 annotation class MyClass
```

DÉCLARATION D'UNE ANNOTATION

L'annotation est déclarée en utilisant le modifieur `annotation` avant la class.

```
1 annotation class MyClass
```

ANNOTATION EN KOTLIN

ANNOTER UN CONSTRUCTEUR

Il est possible d'annoter un constructeur d'une classe, pour cela il faut préciser le mot clé `constructor` lors de la déclaration, et placer l'annotation avant ce mot clé.

```
1 class MyClass @Inject constructor( dependency: MyDependency){
2     //. . .
3 }
```

ANNOTATION EN KOTLIN

EXEMPLE D'UNE ANNOTATION

Déclaration de l'annotation :

```kotlin
import java.lang.annotation.ElementType
import java.lang.annotation.RetentionPolicy

@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class Ann(val value: Int)
```

Utilisation de l'annotation :

```kotlin
@Ann(value = 10)
class MyClass

fun main(args: Array<String>) {
    val c = MyClass()
    val x = c.javaClass.getAnnotation(Ann::class.java)
    println("Value:" + x?.value)
}
```

Que va afficher le code ci-dessus ?

BONNES ET MAUVAISES PRATIQUES

Au lieu d'utiliser des tuples standards, il est recommandé d'utiliser des objets qui nommerons les champs pour leur donner plus de sens.

INTEROPÉRABILITÉ

- Interopérabilité avec Java
- De Kotlin au Java
- Nulls de Java
- Le Kotlin dans Java
- Extensions de fonctions à partir du Java
- Interopérabilité avec Java 7 et Java 8
- Java Réflexion avec Kotlin
- Kotlin Réflexion

INTEROPÉRABILITÉ AVEC JAVA

Kotlin est conçu pour être interopérable avec le Java. Le code existant Java peut être appelé en Kotlin, naturellement, et du code Kotlin peut être appelé assez facilement en Java. Par exemple :

```kotlin
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }

    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

```kotlin
fun main() {
    demo(listOf(1,3,5,74,1,-2,5,98))
}
```

INTEROPÉRABILITÉ AVEC JAVA

GETTERS ET SETTERS

Les méthodes qui suivent la convention Java de nomage pour les getters et les setters (pas d'argument, avec un nom commençant par `get` et un seul argument avec un nom commençant par `set`) sont représentées comme des propriétés en Kotlin. Les accesseurs pour les valeurs de type `Boolean` (le nom du getter

commence par `is` et le nom du setter commence par `set`) sont représentés aussi comme des propriétés. Par exemple :

```
1  import java.util.Calendar
2
3  fun calendarDemo() {
4      val calendar = Calendar.getInstance()
5      if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // call getFirstDayOfWeek()
6          calendar.firstDayOfWeek = Calendar.MONDAY      // call setFirstDayOfWeek()
7      }
8      if (!calendar.isLenient) {                         // call isLenient()
9          calendar.isLenient = true                      // call setLenient()
10     }
11 }
```

```
1  fun main() {
2      calendarDemo()
3  }
```

INTEROPÉRABILITÉ AVEC JAVA

UN AUTRE EXEMPLE

Écrivez la classe JAVA suivante (en ajoutant les getters/setters avec le menu) :

```
1 public class Customer {
2
3     private String firstName;
4     private String lastName;
5     private int age;
6
7     //standard setters and getters
8 }
```

INTEROPÉRABILITÉ AVEC JAVA

UTILISATION EN KOTLIN

Nous pouvons utiliser la classe `Customer` directement dans notre code en Kotlin :

```kotlin
fun main() {
    val customer = Customer()

    customer.firstName = "Frodo"
    customer.lastName = "Baggins"

    println("${customer.firstName} ${customer.lastName}")
}
```

INTEROPÉRABILITÉ AVEC JAVA

REMARQUES

Nous devons nous rappeler que si une classe Java ne comporte que des méthodes setter, la propriété ne sera pas accessible car Kotlin ne prend pas en charge les propriétés en écriture seule (set-only).
Si une méthode retourne `void`, alors quand elle est appelée depuis Kotlin elle retournera `Unit`.

DE KOTLIN AU JAVA

APPELER DU KOTLIN EN JAVA

Il est assez facile d'appeler du code écrit en Kotlin depuis du code Java. Il y a cependant certains points qui méritent une attention particulière, nous allons développer certains points ci-dessous.

DE KOTLIN AU JAVA

LES PROPRIÉTÉS

Une propriété Kotlin sera compilée en Java, de la manière suivante :

- Une méthode getter, sera formatée en préfixant le nom de la propriété par `get`.

- Une méthode setter, sera formatée en préfixant le nom de la propriété par `set`(pour les propriétés de type `var`).
- Un champ privé aura le même nom que le nom de la propriété.

Par exemple : `var firstName: String` sera compilée en Java de la manière suivante :

```java
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

DE KOTLIN AU JAVA

PROPRIÉTÉS (SUITE)

Si le nom de la propriété commence par `is`, alors la règle diffère : le nom du getter sera le même que la propriété, et le nom du setter sera obtenu en replaçant le `is` par `set`. Par exemple, une propriété `isOpen`, le getter sera `isOpen()` et le setter : `setOpen`. Cette règle est valable pour toutes les propriétés, peu importe leur type, pas seulement pour les propriétés de type `Boolean`.

DE KOTLIN AU JAVA

EXERCICE

Déclarer les 2 variables suivantes en Kotlin, dans un fichier séparé.

```kotlin
var firstName = ""
var isOpen = false
```

Obtenez le bytecode Kotlin avec le menu : Tools, Kotlin, Show Kotlin Bytecode.
Cliquez ensuite sur le bouton `Decompile`, pour obtenir le code Java équivalent.

Que constatez vous ?

DE KOTLIN AU JAVA

LES FONCTIONS AU NIVEAU DU PACKAGE

Toutes les fonctions et propriétés déclarées dans un fichier `app.kt` dans un package `org.example`, incluant toutes les fonctions d'extention, sont compilées dans des méthodes statiques d'une classe nomée `org.example.AppKt`. Exemple :

```kotlin
// app.kt
package org.example
class Util

fun getTime(): Int { println("10h21"); return 10 }
```

```java
// Java
import org.example.AppKt;
import org.example.Util;

public class Test {

    Util utilVar = new Util();
    int timeValue = AppKt.getTime();
}
```

DE KOTLIN AU JAVA

LES CHAMPS D'INSTANCES

Si vous voulez exposer une propriété Kotlin en tant que champs d'instance en Java, il faut l'annoter avec `@JvmField`. Le champ aura alors la même visibilité que la propriété d'origine.

```kotlin
class User(id: String) {
    @JvmField val ID = id
}
```

```java
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

Une propriété modifiée avec `lateinit` sera aussi exposée en tant que champs d'instance. La visibilité du champ sera la même que la visibilité du setter de la propriété.

DE KOTLIN AU JAVA

CHAMPS STATIQUES (STATICS)

Les propriétés en Kotlin, déclarées dans un objet nommé, ou un objet compagnon sont par défaut privées, mais peuvent être rendues publiques en Java en utilisant une de ces méthodes :

- L'annotation `@JvmField`
- le modifieur `lateinit`
- le modifieur `const`

DE KOTLIN AU JAVA

STATICS / `@JvmField`

En annotant la propriété avec `@JvmField`, cela donnera un champ "static" avec la même visibilité que la propriété elle même :

```kotlin
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```java
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

DE KOTLIN AU JAVA

STATICS / `lateinit`

En modifiant la propriété avec `lateinit`, cela donnera un champ "static" avec la même visibilité que la propriété elle même :

```kotlin
object Singleton {
    lateinit var provider: Provider
}
```

```kotlin
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

DE KOTLIN AU JAVA

STATICS / `const`

Une propriété déclarée avec `const` (dans une classe ou au top niveau (top level)), seront traduites en champs statiques en Java :

```kotlin
// file example.kt
object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

```java
// Java
int const = Obj.CONST;
int max = ExampleKt.MAX;
int version = C.VERSION;
```

DE KOTLIN AU JAVA

SURCHARGES

Normalement, si vous écrivez une fonction Kotlin avec des paramètres par défauts, en Java, c'est la version avec tous les paramètres qui sera disponible. Si vous voulez exposer de multiples méthodes (surcharge) avec des signatures différentes, vous pouvez utiliser l'annotation `@JvmOverloads`.

Cette annotation fonctionne de partout : constructeurs, méthodes statiques, etc. Toutefois elle ne peut pas être utilisée sur des méthodes abstraites, ni dans dans interfaces.

Exemple :

```kotlin
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*...*/ }
}
```

Pour chaque paramètre avec une valeur par défaut, une méthode de surcharge sera générée. Dans notre exemple, cela donnera :

```java
// Constructors:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// Methods
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

NULLS DE JAVA

Kotlin est bien connu pour sa fonctionnalité de sécurité `null`, mais comme nous le savons, ce n'est pas le cas pour Java, ce qui le rend peu pratique pour les objets qui en proviennent. Un exemple très simple permet de mettre cela en relief. Prenez le code suivant :

```java
// Java
public class Nullable {

    public String get(){
        return null;
    }
}
```

```kotlin
fun main() {
    Nullable().get().length
}
```

Que se passe t'il quand on lance le code ?
Que pouvons nous faire pour éviter l'erreur ?

NULLS DE JAVA

SOLUTION

```
1 fun main() {
2     Nullable().get()?.length
3 }
```

LE KOTLIN DANS JAVA

LA CLASSE KOTLIN

Dans cette section nous allons voir comment appeler du Kotlin en Java. Créons une class `Shape`, en Kotlin, avec des propriétés : `height`, `width` et `area`, et deux fonctions : `shapeMessage` et `draw` :

```kotlin
// Shape.kt
class Shape(var width: Int, var height: Int, val shape: String) {
    var area: Int = 0
    fun shapeMessage() {
        println("Hi i am $shape, how are you doing")
    }
    fun draw() {
        println("$shape is drawn")
    }
    fun calculateArea(): Int {
        area = width * height
        return area
    }
}
```

LE KOTLIN DANS JAVA

L'APPEL EN JAVA

Vous pouvez instancier la classe Kotlin de la même manière que si vous instanciez une classe en Java. Par exemple :

```java
public class FromKotlinClass {
    public static void callShapeInstance() {
        Shape shape = new Shape(5,5,"Square");
        shape.shapeMessage();
        shape.setHeight(10);
        System.out.println(shape.getShape() + " width " + shape.getWidth());
        System.out.println(shape.getShape() + " height " + shape.getHeight());
        System.out.println(shape.getShape() + " area " + shape.calculateArea());
        shape.draw();
    }
    public static void main(String[] args) {
        callShapeInstance();
    }
}
```

LE KOTLIN DANS JAVA

APPEL D'UN SINGLETON KOTLIN

Il est possible d'appeler une classe Singleton Kotlin en Java, en utilisant le mot clé `object` :

```kotlin
// Kotlin
object Singleton {
    fun happy() {
        println("I am Happy")
    }
}
```

Pour appeler le singleton en Java, il faudra utiliser le mot clé `INSTANCE` :

```java
// Java
public static void main(String args[]) {
    Singleton.INSTANCE.happy();
}
```

LE KOTLIN DANS JAVA

APPEL D'UN SINGLETON (SUITE)

Il est possible d'éviter l'utilisation du mot clé `INSTANCE` en utilisant l'annotation `@JvmStatic` :

```kotlin
object Singleton {

    fun happy() {
        println("I am Happy")
    }

    @JvmStatic fun excited() {
        println("I am very Excited")
    }
}
```

Ce qui donnera l'appel en Java :

```java
public static void main(String args[]) {
    Singleton.INSTANCE.happy();
    Singleton.excited();
}
```

LE KOTLIN DANS JAVA

APPEL DE FONCTIONS TOP LEVEL

Les fonctions en Kotlin, n'ont pas besoin d'être déclarées dans une Classe, comme c'est le cas en Java. Nous allons voir en détail comment appeler ce genre de fonctions.

Prenons par exemple un fichier `utils.kt` :

```kotlin
fun logD(message: String) {
    Log.d("", message)
}
fun logE(message: String) {
    Log.e("", message)
}
```

En Java, il sera possible d'appeler simplement ces fonctions comme suit :

```java
UtilsKt.logD("Debug");
UtilsKt.logE("Error");
```

EXTENSIONS DE FONCTIONS À PARTIR DU JAVA

EXEMPLE D'EXTENSION

Supposons que nous avons le code suivant :

```
1 fun String.firstUpper() = this.first().toUpperCase() + this.substring(1).toLowerCase()
```

Que l'on pourrait appeler en Kotlin :

```
1 println("tristan".firstUpper())
2 println("TRISTAN".firstUpper())
```

En Java cela deviendra :

```
1 System.out.println(ExtensionKt.firstUpper("Tristan"));
```

Comme vous pouvez le voir, l'objet sur lequel est appliqué la fonction (le receveur/"reveiver") est ajouté en parmètre de la fonction. De plus les paramètres optionnels deviennent obligatoires, car Java ne gère pas cette fonctionnalité.

INTEROPÉRABILITÉ AVEC JAVA 7 ET JAVA 8

Kotlin et un langage récent et le langage Java comporte de nombreuses fonctionnalités et est en constante évolution. De ce fait, toutes les fonctionnalités de Java ne sont pas encore supportés en Kotlin. Par exemple les méthodes `default` dans les interfaces, ne sont disponibles que pour la JVM 1.8, et l'annotation `@JvmDefault` correspondante est expérimentale en Kotlin 1.3.

JAVA RÉFLEXION AVEC KOTLIN

EXERCICE

Reprenons notre classe Java `Customer` :

```
1 public class Customer {
2
3     private String firstName;

4     private String lastName;
5     private int age;
6
7     //standard setters and getters
8 }
```

La reflexion fonctionne à la fois sur les classes Kotlin et Java.
Testons la classe `Customer` avec les méthodes de réflexion Kotlin :

- <ClassName>::class.java permet de récupérer la classe.
- la propriété `constructors` de `Class<T>` contient le tableau des constructeurs.
- la propriété `name` du `Constructor` contient le non du constructeur.

Afficher le nombre et le nom du/des constructeur(s).

JAVA RÉFLEXION AVEC KOTLIN

SOLUTION

```kotlin
fun main() {
    val type = Customer::class.java
    val constructors = type.constructors

    println(constructors.size)
    println(constructors[0].name)
}
```

KOTLIN RÉFLEXION

REFLEXION JAVA EN KOTLIN

La réflexion fonctionne de manière équivalente en Java et en Kotlin. Prenons un exemple :

```
1 MyClass::class.java.methods
```

Qui permet de lister les méthodes d'une classe. Décomposons cette construction :

- `MyClass::class` nous donne une représentation de la class `MyClass`
- `.java` nous donne l'équivalent de `java.lang.Class`
- `.methods` appelle la méthode `java.lang.Class.getMethods()`

Prenons un exemple concret :

```
1 data class ExampleDataClass(
2     val name: String, var enabled: Boolean)
3
4 fun main() {
5     ExampleDataClass::class.java.methods.forEach(::println)
6 }
```

STANDARD LIBRARY

- Kotlin Standard Library et collections dans Kotlin
- Filtering, Mapping et Flatmapping en Kotlin
- Kotlin lazy evaluation

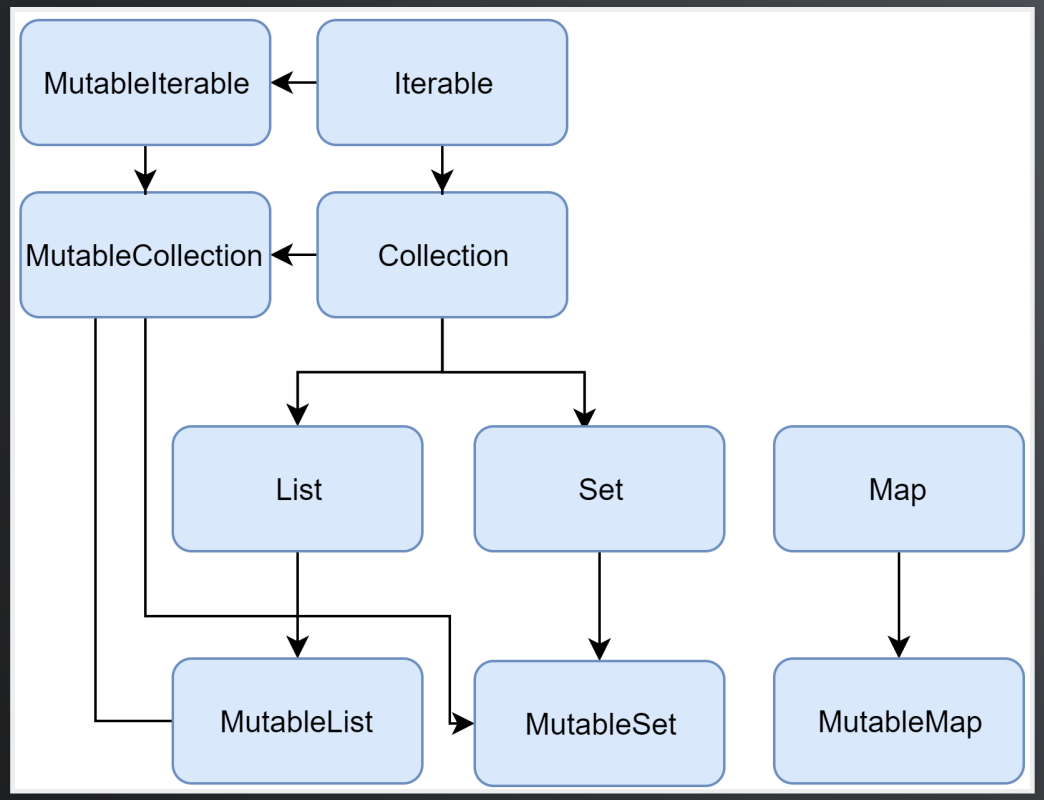
KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

La librairie standard de Kotlin fournit l'essentiel des méthodes pour développer tels que :

- Les fonctions d'ordre supérieur, pour gérer les cas classiques (let, apply, use, synchronized, ...).
- Des fonctions d'extension, qui fournissent des opérations pour interroger des collections et des séquences.
- Des outils pour travailler avec les chaines de caractères et les caractères.
- Des extensions pour les classes du JDK pour que cela soit plus pratique de travailler avec des fichiers, les Entrés/Sorties (IO), et les fils d'exécutions (threading).

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES COLLECTIONS



KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

`Collection<T>` est parente de toute la hiérarchie des collections. Elle définie le comportement commun d'une collection en lecture seule : la récupération de la taille de la liste, la vérification d'appartenance d'un objet à la collection, etc.

`Collection` hérite d'`Iterable<T>` qui définie les opérations pour itérer sur les éléments. C'est le type à utiliser pour gérer les différents types de collections. Dans les cas plus précis, préférer `List` ou `Set`.

```
1  fun printAll(strings: Collection<String>) {
2      for(s in strings) print("$s ")
3      println()
4  }
5
6  fun main() {
7      val stringList = listOf("one", "two", "one")
8      printAll(stringList)
9
10     val stringSet = setOf("one", "two", "three")
11     printAll(stringSet)
12 }
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

MutableCollection est une Collection avec les opérateurs d'écriture tels que add et remove.

```
1  fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
2      this.filterTo(shortWords) { it.length <= maxLength }
3      // throwing away the articles
4      val articles = setOf("a", "A", "an", "An", "the", "The")
5      shortWords -= articles
6  }
7
8  fun main() {
9      val words = "A long time ago in a galaxy far far away".split(" ")
10     val shortWords = mutableListOf<String>()
11     words.getShortWordsTo(shortWords, 3)
12     println(shortWords)
13 }
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

DIFFÉRENCES ET SIMILITUDES ENTRE `List<T>` ET `Array<T>`

Exemples d'utilisation :

```
1 // Initializing array and list
2 val array = arrayOf(1, 2, 3)
3 val list = listOf("apple", "ball", "cow")
4 val mixedArray = arrayOf(true, 2.5, 1, 1.3f, 12000L, 'a') // mixed Array
5 val mixedList = listOf(false, 3.5, 2, 1.4f, 13000L, 'b') // mixed List
```

Les deux semblent similaires, ...

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

DÉFINITION DE `List<T>`

Une liste est une interface. C'est une collection générique et ordonnée d'éléments. Les méthodes dans cette interface permettent un accès en lecture seule aux éléments.
Les opérations de lecture et écriture sont définies dans l'interface `MutableList`.

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

DÉFINITION DE `Array<T>`

Les tableaux sont un conteneur d'objets qui regroupe un nombre fixe d'éléments d'un seul et même type.
La taille d'un tableau est établie de manière définitive lors de sa création.

Les tableaux peuvent être concaténés pour donner un nouveau tableau :

```
1 val array1 = arrayOf(1, 2, 3, 4, 5, 6)
2 val array2 = arrayOf(7, 8, 9, 10, 11, 12)
3 val newArray = array1 + array2
4 for(element in newArray) {
5     println("$element - ")
6 } // 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES SIMILITUDES

1. Les 2 comportent un nombre finit d'éléments. Il n'est pas possible d'ajouter des éléments après initialisation. Leur taille est fixe et ne peut pas augmenter ni diminuer :

```
1 array.add(1) // cannot be expanded or shrank
2 list.add("dog") // cannot be expanded or shrank
```

   Notons que les listes mutables peuvent changer de taille.
2. Les 2 permettent de stocker plusieurs éléments et font partis de la famille des collections :

```
1 val array = arrayOf(1, 2, 3)
2 val list = listOf("apple", "ball", "cow")
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES DIFFÉRENCES

- Les `Array` sont modifiables (mutable), leur contenu peut changer de valeur, contrairement aux `List` qui sont en lecture seule (immutable) :

```kotlin
val array = arrayOf(1, 2, 3)
array[2] = 4 // OK
for (element in array){
  println("$element, ") // 1, 2, 4
}
val list = listOf("apple", "ball", "cow")
list[2] = "cat" //will not compile, lists are immutable
// println(list) = {apple, ball, cow}

val m = mutableListOf(1, 2, 3)
m[0] = m[1] // OK
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES DIFFÉRENCES (SUITE)

- Les tableaux sont optimisés pour les types primitifs (`IntArray`, `DoubleArray`, `CharArray`, etc) : qui utilisent directement les tableaux Java primitifs (int[], double[], char[], etc.)
  Les listes en général n'ont pas d'implémentations optimisées pour les sypes primitifs.

```
1 val optimisedIntegerArray = intArrayOf(1, 2, 3, 4, 5, 6) // optimised for Integer Array
2 val optimisedDoubleArray = doubleArrayOf(1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8) // Optimised for Double Array
3 val optimisedCharacterArray = charArrayOf('a', 'b', 'c', 'd') // Optimised for Character Array
4 //List does not have intListOf, charListOf as such it is not optimised for primitive arrays
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES DIFFÉRENCES (SUITE)

- `Array<T>` est une classe dont l'implémentation est connue : c'est une région de mémoire séquentielle de taille fixe qui stocke les éléments. Dans la JVM elle est représenté par `Java array`
  `List<T>` et `MutableList<T>` sont des interfaces qui peuvent avoir plusieurs implémentations : `ArrayList<T>`, `LinkedList<T>`, etc.
- `Array<T>` est invariant (`Array<Int>` n'est pas un `Array<Number>`, idem pour `MutableList<T>`
  Par contre `List<T>` est covariant (`List<Int>` est une `List<Number>`).

```
1 val a: Array<Number> = Array<Int>(0) { 0 } // won't compile
2 val l: List<Number> = listOf(1, 2, 3) // OK
```

KOTLIN STANDARD LIBRARY ET COLLECTIONS DANS KOTLIN

LES DIFFÉRENCES (SUITE)

- Au niveau de l'interopérabilité JAVA, les 2 ne sont pas gérés de la même manière.
- Certains types de tableaux sont utilisés dans les annotations, alors que pour les listes et autres collections cela n'est pas possible.
- L'usage veut que l'on utilise de préférence les liste au lieu des tableaux, sauf dans le cas ou la performance est un critère critique.

FILTERING, MAPPING ET FLATMAPPING EN KOTLIN

ENTRAÎNEMENT EN LIGNE

Plutôt que de réinventer la roue, je vous propose de vous entrainer sur les collections directement en ligne ICI.

KOTLIN LAZY EVALUATION

Ci-dessous, un exemple d'appel d'une fonction en mode "lazy".

```
1  fun main() {
2      printValue(getValue())
3  }
4
5  private fun printValue(value: Lazy<Int>) = println("Nothing")
6
7  private fun getValue() = lazy {
8      println("Returning 5")
9      return@lazy 5
10 }
```

Que remarquez vous ?
Changez la valeur du `println` en :

```
1  "Nothing ${value.value}"
```

Quel changement observez vous, et pourquoi ?

PROGRAMMATION ASYNCHRONE

- Le problème de la programmation asynchrone
- Coroutines en Kotlin et l'implémentation des coroutines
- Async et Await en Kotlin
- Yield en Kotlin
- Reactive extension en Kotlin
- Bonnes et mauvaises pratiques

LE PROBLÈME DE LA PROGRAMMATION ASYNCHRONE

Depuis des décennies, les développeurs sont confrontés à un problème à résoudre : comment faire pour que les applications ne se bloquent pas. Que l'on développe pour un ordinateur, un mobile ou un serveur, nous voulons éviter que l'utilisateur attende, ou encore pire, des goulots d'étranglements qui empêcherait à

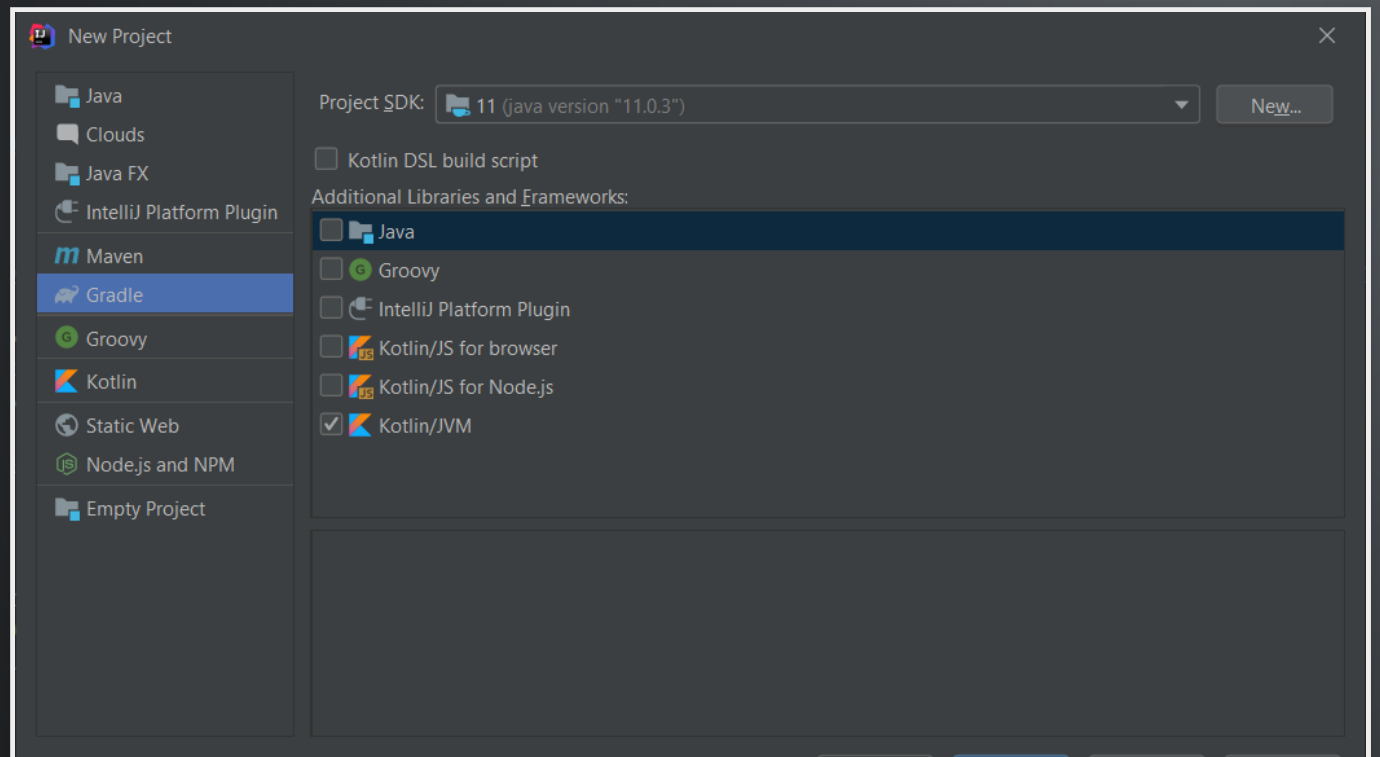
l'application de passer à l'échelle.

Plusieurs approches sont possibles pour résoudre ce problème :

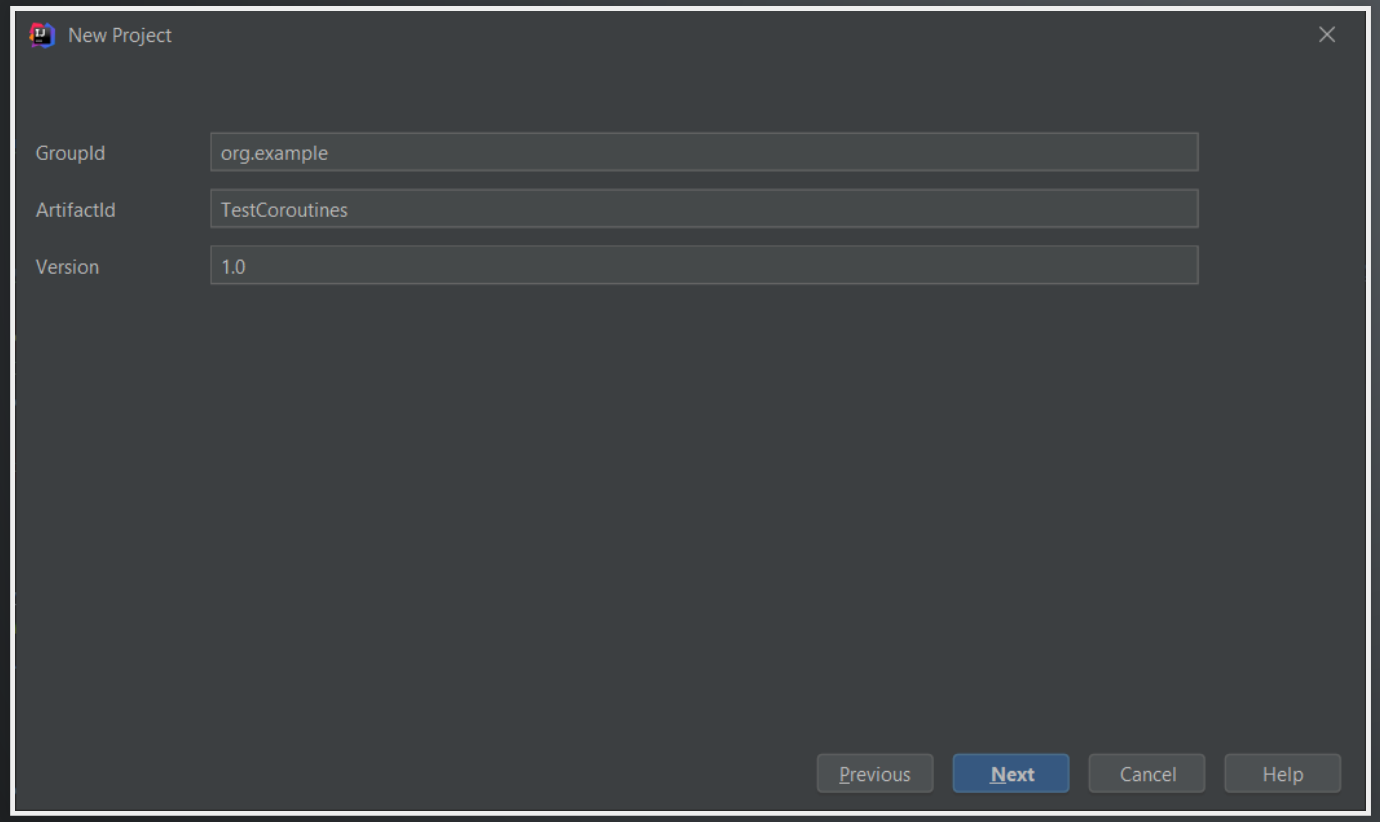
- Traitements en parallèle (Threading)
- Les "callbacks"
- Futures, Promises et al.
- Les extensions réactives
- Coroutines

MISE EN PLACE DU PROJET

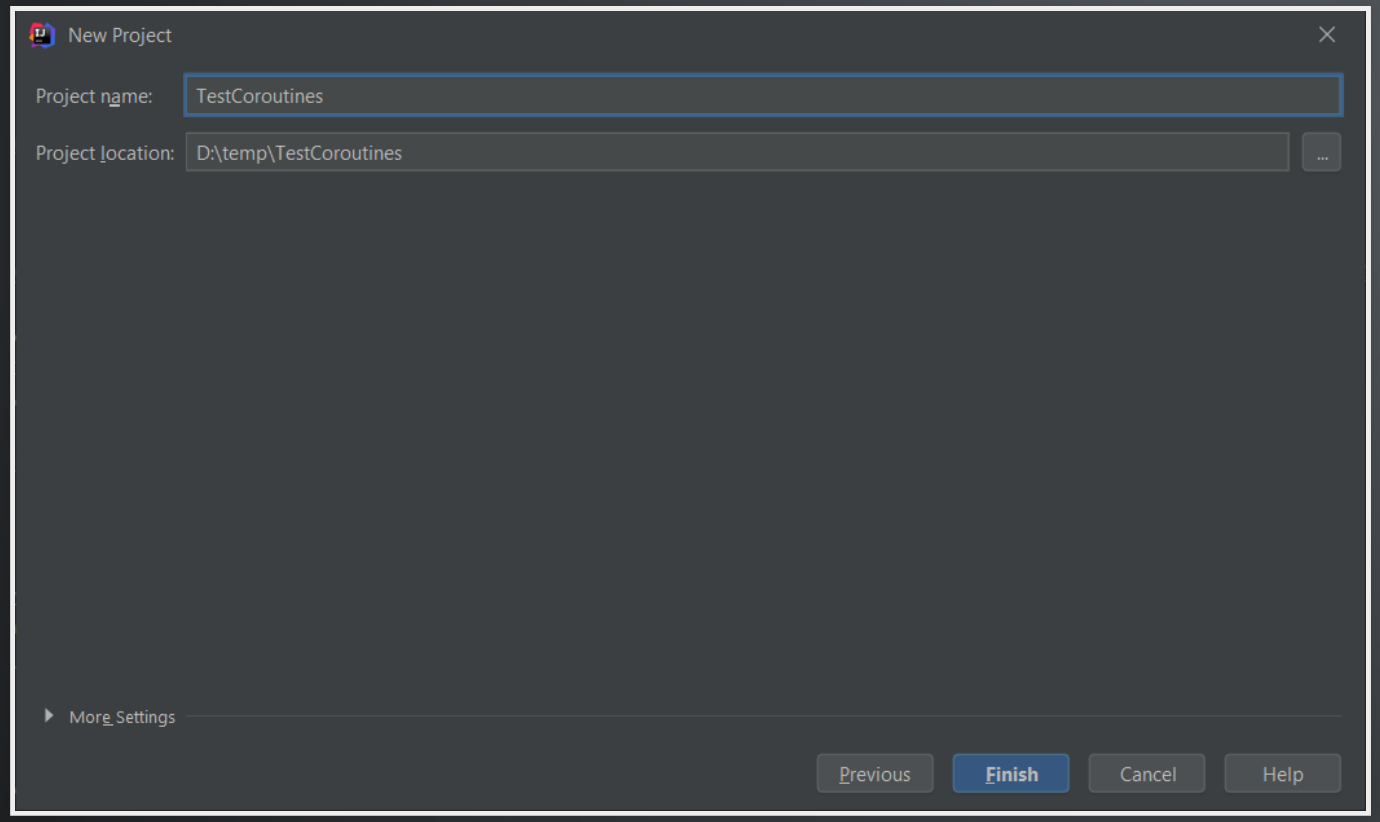
Créez un nouveau projet : **File / New / Project**. Sélectionner **Gradle / Kotlin / JVM**.



NOMMAGE DU MODULE

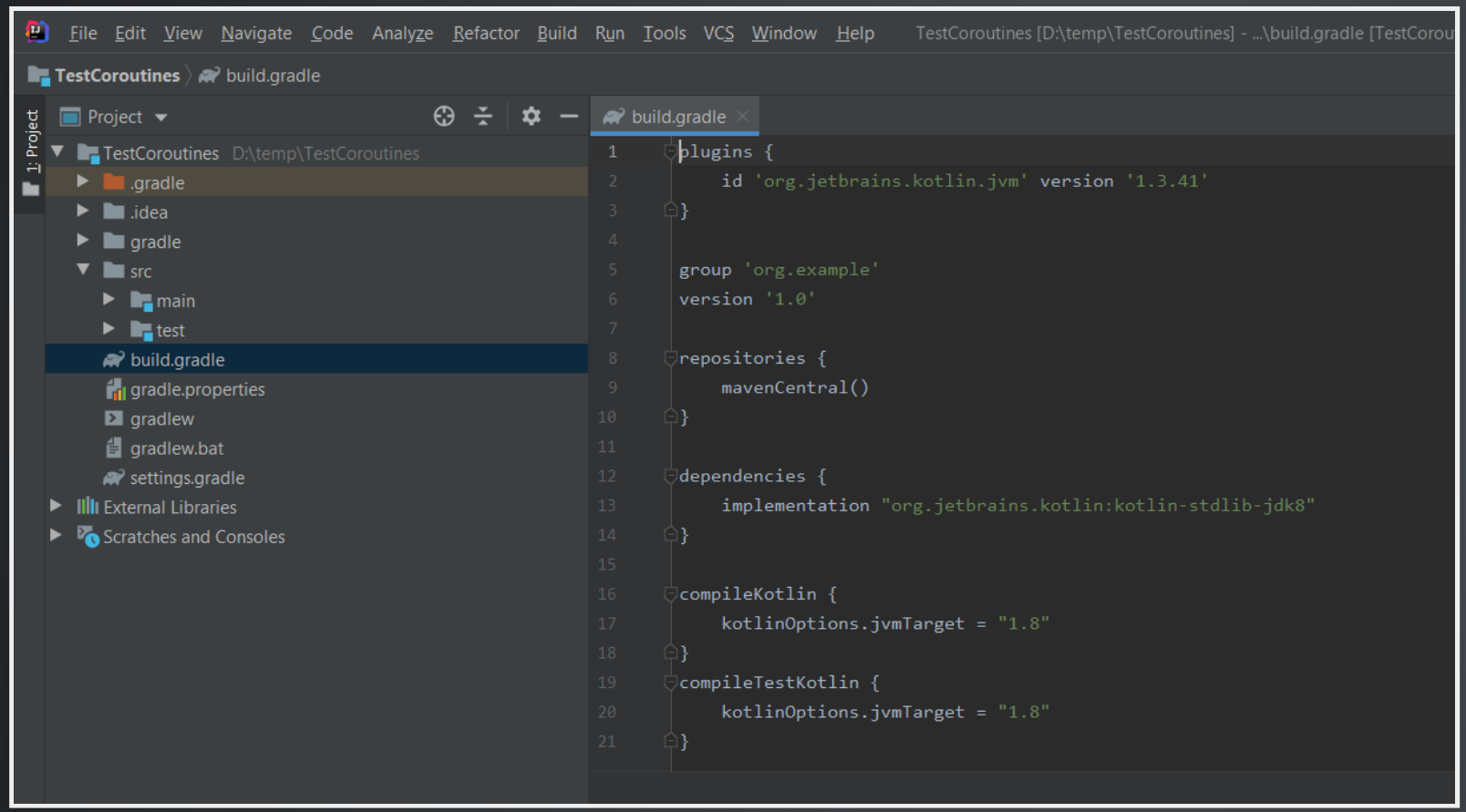


NOMMAGE DU PROJET



COROUTINES EN KOTLIN

Ouvrez le fichier `build.gradle`



COROUTINES EN KOTLIN

AJOUTEZ LA DÉPENDANCE

```
dependencies {
    ...
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.1.1"
}
```

COROUTINES EN KOTLIN

Une coroutine pourrait être vue comme un thread léger, car une coroutine peut être lancée en parallèle, s'attendre les unes les autres et communiquer ensemble. Mais la grosse différence est le coût de celles-ci : pratiquement rien. Il est possible d'en créer des centaines, sans impacter les performances, contrairement aux thread classiques.

Pour lancer une coroutine, le point de départ est la fonction `launch {}` :

```
1 launch {
2 //    ...
3 }
```

Les coroutines, utilisent un pool de threads pour fonctionner, mais un thread peut faire tourner plusieurs coroutines, donc il n'est pas nécessaire d'avoir beaucoup de threads lancés.

COROUTINES EN KOTLIN

Lançons notre première coroutine :

```
println("Start")

// Start a coroutine
GlobalScope.launch {
    delay(1000)

    println("Hello")
}

Thread.sleep(2000) // wait for 2 seconds
println("Stop")
```

La fonction `delay()` fonctionne comme la fonctions `Thread.sleep()`, mais elle ne bloque pas le thread, elle suspend simplement la coroutine. Le thread retourne dans le pool de thread. Et la coroutine, reprendra son fonctionnement sur un tread disponible.
Que se passe-t-il si l'on supprime la ligne `Thread.sleep(2000)` ?
Que se passe-t-il si l'on remplace `Thread.sleep(2000)` par `delay(2000)` ?

BLOCAGE DU THREAD PRINCIPAL

Pour pouvoir utiliser la fonction `delay(...)`, nous allons l'appeler dans une fonction `runBlocking {}` :

```
runBlocking {
    delay(2000)
}
```

Ce qui nous donnera au final :

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    println("Start")

// Start a coroutine
    GlobalScope.launch {
        delay(1000)
        println("Hello")
    }

    // Step 1
    //Thread.sleep(2000) // wait for 2 seconds
```

COROUTINES EN KOTLIN

LANÇONS BEAUCOUP DE THREAD

Nous allons comparer les thread et les coroutines, en lançant, disons 1 million de processus en parallèle
Commençons avec les `threads` :

```
1 val c = AtomicLong()
2
3 for (i in 1..1_000_000L)
4     thread(start = true) {
5         c.addAndGet(i)
6     }
7
8 println(c.get())
```

Que remarquez vous, concernant la charge de la machine ?

COROUTINES EN KOTLIN

LANÇONS BEAUCOUP DE THREAD

Écrivez le même code avec des coroutines& :

```
1 val c = AtomicLong()
2
3 for (i in 1..1_000_000L)
4     GlobalScope.launch {
5         c.addAndGet(i)
6     }
7
8 println(c.get())
```

Que constate-t-ons ?
Toutefois, le résultat n'est pas correct, toutes les coroutines n'ont pas terminé leur traitement avant que la fonction `main()` affiche le résultat. Nous allons corriger cela.

ASYNC ET AWAIT EN KOTLIN

Une autre manière de lancer les coroutines, et d'utiliser `async {}`. C'est comme `launch {}` mais cela retourne une instance de `Deferred<T>` qui comporte une fonction `await()` qui retourne le résultat de la coroutine. `Deferred<T>` est une sorte de *future* très basique.

Nous allons donc maintenant lancer de nouveau le million de coroutines, et attendre leur retour. La variable de type `AtomicLong` n'est plus nécessaire

```
1 val deferred = (1..1_000_000).map { n ->
2     GlobalScope.async {
3         n
4     }
5 }
6 val sum = deferred.sumBy { it.await() }
7 println("Sum: $sum")
```

Quel est le problème ici ?

ASYNC ET AWAIT EN KOTLIN

Il faut donc lancer la fonction `await()` dans un contexte de coroutine, nous utilisons de nouveau `runBlocking {}`&nbps;

```
val deferred = (1..1_000_000).map { n ->
    GlobalScope.async {
        n
    }
}
runBlocking {
    val sum = deferred.sumBy { it.await() }
    println("Sum: $sum")
}
```

Nous devrions obtenir le résultat suivant :

```
Sum: 1784293664
```

ASYNC ET AWAIT EN KOTLIN

PARALLÈLE

Cela fonctionne vraiment en parallèle ? Pour en être convaincu, ajoutons un délai à notre traitement

```
val deferred = (1..1_000_000).map { n ->
    GlobalScope.async {
        delay(1000)
        n
    }
}
runBlocking {
    val sum = deferred.sumBy { it.await() }
    println("Sum: $sum")
}
```

Allons nous devoir attendre 1 million de secondes (11,5 jours) pour obtenir notre résultat ?

ASYNC ET AWAIT EN KOTLIN

FONCTIONS SUSPENDUES

Nous voulons extraire le code fonctionnel dans une fonction :

```
1 fun workload(n: Int): Int {
2     delay(1000)
3     return n
4 }
```

Le compilateur, n'est pas content, car `delay` ne peut être utilisé que dans une cadre de coroutine. Marquons la fonctions avec le mot clé `suspend` :

```
1 suspend fun workload(n: Int): Int {
2     delay(1000)
3     return n
4 }
```

ASYNC ET AWAIT EN KOTLIN

CODE FINAL

Nous obtenons le code final, suivant :

```
1  import kotlinx.coroutines.GlobalScope
2  import kotlinx.coroutines.async
3  import kotlinx.coroutines.runBlocking
4
5  fun main() {
6
7      suspend fun workload(n: Int): Int {
8          //delay(3000)
9          return n
10     }
11
12     val deferred = (1..1_000_000).map { n ->
13         GlobalScope.async {
14             workload(n)
15         }
```

YIELD EN KOTLIN

CONSTRUCTION DE SÉQUENCES

Nous allons construire une séquence, de manière paresseuse, par exemple la suite de finonacci en utilisant la fonction `sequence` :

```
1  fun main(args: Array<String>) {
2     val fibonacciSeq = sequence {

3         var a = 0
4         var b = 1
5
6         yield(1)
7
8         while (true) {
9             yield(a + b)
10
11            val tmp = a + b
12            a = b
13            b = tmp
14        }
15    }
```

Ce qui devrait nous afficher :

```
1 [1, 1, 2, 3, 5, 8, 13, 21]
```

YIELD EN KOTLIN

Nous venons de définir une séquence, potentiellement infinie, générée par une coroutine. Validons ensemble que cette génération est bien paresseuse :

```kotlin
fun main(args: Array<String>) {
    val lazySeq = sequence {
        print("START ")
        for (i in 1..5) {
            yield(i)
            print("STEP ")
        }
        print("END")
    }

    // Print the first three elements of the sequence
    lazySeq.take(3).forEach { print("$it ") }
}
```

Combien de fois est affiché le message `END` ?

Que faudrait-il faire pour qu'il s'affiche ?

Que se passe-t'il si l'on demande par exemple 10 éléments de la séquence ?

YIELD EN KOTLIN

GENERATION COMPLÈTE DE LA SÉQUENCE

Si nous voulons générer la séquence d'un coup, nous utiliserons alors `yieldAll`, par exemple :

```kotlin
fun main(args: Array<String>) {
    val lazySeq = sequence {
        yield(0)
        yieldAll(1..10)
    }

    lazySeq.forEach { print("$it ") }
}
```

YIELD EN KOTLIN

GÉNÉRATION FILTRÉE DE LA SÉQUENCE

Il est possible de décomposer notre génération de séquence :

```
1  suspend fun SequenceScope.yieldIfOdd(x: Int) {
2      if (x % 2 != 0) yield(x)
3  }
4
5  val lazySeq = sequence {
6      for (i in 1..10) yieldIfOdd(i)
7  }
8
9  fun main(args: Array) {
10     lazySeq.forEach { print("$it ") }
11 }
12
```

REACTIVE EXTENSION EN KOTLIN

LIBRAIRIES

Il existe plusieurs projets permettant de gérer l'asynchronisme plus facilement :

- Présentation de RxKotlin
- Présentation du projet Reactor

BONNES ET MAUVAISES PRATIQUES

- N'oubliez pas de stopper toute les coroutines en quittant votre programme. Exemple en Android :

```
override fun onDestroy() {
    super.onDestroy()
    async.cancelAll()
}
```

- Utilisez les coroutines, uniquement quand cela est nécessaire.

ANNEXES

LA PRÉSENTATION EN PDF

CODES SOURCE

Projet InteliJ des exercices
Projet InteliJ des exercices coroutines

SCRIPTS EN KOTLIN

Kotlin peut être utilisé pour écrire des scripts. Exemple :

```
1 // list_folders.kts
2 import java.io.File
3 val folders = File(args[0]).listFiles { file -> file.isDirectory() }
4 folders?.forEach { folder -> println(folder) }
```

Pour lancer le script, il faut ajouter l'option `-script` et passer le chemin du script à lancer :

```
1 $ kotlinc -script list_folders.kts "path_to_folder_to_inspect"
```

LES FONCTIONS INLINE

Utiliser des fonctions d'ordre supérieur, imposent des contraintes lors de l'exécution du programme : chaque fonction est un objet, qui comporte un contexte (closure), c'est à dire des variables accessibles depuis le cors de la fonction. L'allocation de mémoire (pour les objets fonction et les classes) et les appels virtuels engendrent une surcharge.

Dans plusieurs cas, il et possible d'éliminer cette contrainte en mettant en ligne (inlining) une expresion lambda. Voici un très bon exemple de fonction à "inliner" : la fonction `lock()`

```
1 lock(l) { foo() }
```

Au lieu de créer un objet de type fonction en paramètre, et de générer un appel à celle-ci, le compilateur pourrait produire le code suivant :

```
1 l.lock()
2 try {
3     foo()
4 }
5 finally {
6     l.unlock()
7 }
```

LES FONCTIONS INLINE

Pour indiquer au compilateur de produire ce code, nous devons marquer la fonction `lock()` avec le modifieur `inline`:

```
1 inline fun <T> lock(lock: Lock, body: () -> T): T { ... }
```

Ce modifieur va impacter la fonction, en elle même, mais aussi la lambda passée en paramètre : tout sera mis en ligne à l'endroit de l'appel.
Utiliser cette technique pourra augmenter la taille du code compilé, il faut donc faire attention à son usage (éviter d'inliner des grosses fonctions), mais l'on gagnera en performances, particulièrement lors d'appel dans des boucles.

RÉFÉRENCES

LES RÉFÉRENCES

- La référence : https://kotlinlang.org/docs/reference
- La source : https://github.com/JetBrains/kotlin
- Ressource très complète en FR : https://code.i-harness .com/fr/docs/kotlin/index
- Rappels des bases : https://kotlinlang .org/docs/reference/basic-syntax.html

RÉFÉRENCES

COURS EN LIGNE

- Sur Openclassrooms : https://openclassrooms.com/fr/courses/5353106-initiez-vous-a-kotlin
- Sur CodinGame : https://www.codingame.com/playgrounds/28826/formation-kotlin/les-bases-de-kotlin
- Kotlin Koans : https://play.kotlinlang.org/koans/overview

RÉFÉRENCES

TESTS/CHALLENGES

- https://github.com/Kotlin/kotlin-koans-edu
- https://tech.io/explore
- https://github.com/SK1dev/KotlinChallenges/blob/master/README.md
- https://github.com/SK1dev/KotlinChallengesPart2
- https://github.com/igorwojda/kotlin-coding-puzzle

RÉFÉRENCES

ARTICLES SUR DES POINTS PRÉCIS

- https://typealias.com/
- http://zetcode.com/all/
- Pourquoi adopter Kotlin : https://medium.com/videdressing-engineering/pourquoi-je-suis-pass%C3%A9-%C3%A0-kotlin-7d40d79054a4

RÉFÉRENCES

LIVRES

- https://kotlinlang.org/docs/books.html
- https://dzone.com/articles/2018s-top-6-book-recommendations-to-learn-kotlin

RÉFÉRENCES

LISTE DES MOTS CLÉS

- https://kotlinlang.org/docs/reference/keyword-reference.html

RÉFÉRENCES

KOTLIN POUR LES APPLICATIONS BACKEND

- https://soat.developpez.com/tutoriels/kotlin-application-backend/

RÉFÉRENCES

KOTLIN POUR ANDROID

- https://www.udacity.com/course/ud9012
- https://blog.ippon.fr/2017/12/11/introduction-a-kotlin-pour-android/
- https://kotlinlang.org/docs/reference/android-overview.html
- https://developer.android.com/kotlin
- https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin/#0
- https://codelabs.developers.google.com/codelabs/kotlin-coroutines/#0