Formations
Tristan SALAÜN

Pour Kotlin, Android (Java et Kotlin)
premier niveau et avancé

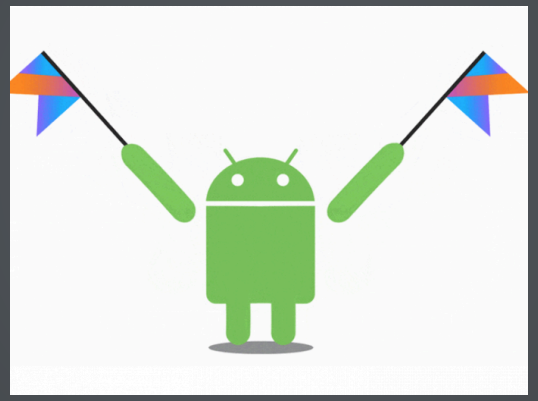
Supports disponibles

- Android premier niveau.
- Android, perfectionnement (Réf. IOD).
- Kotlin.
- Kotlin, développer des applications pour Android (Réf. OTA).
- Kotlin mise en oeuvre (Réf. OTB).

Utilisation de cette présentation

Appuyons sur la touche ?

Kotlin, les bases et mise en œuvre pour Android



Le support en PDF.

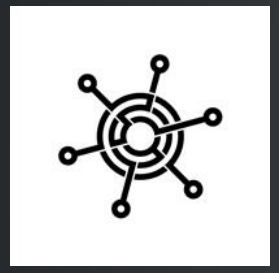
Présentations

- Qui suis-je.
- Qui êtes-vous ? Quelles sont vos attentes.
- Présentation du plan.

Présentations

Qui suis-je

- Développeur d'applications mobiles Android (depuis 2009).
- Formateur Android/Java et Kotlin, et Javacard (scolaires et pros).
- Co-fondateur (1/7) de Startup Marseille.



- Fondateur de Light4Events.

Présentations

Qui êtes-vous ?

Tour de table :

- Prénom.
- Expérience (Java, Kotlin, Mobile Android/iOS).
- Attentes.

Kotlin pour Android

Introduction

- Pourquoi le Kotlin ?
- Introduction à la JVM (Java Virtual Machine).
- Interpréteur en ligne.
- La structure d'une application Kotlin.
- Kotlin et IntelliJ IDEA.
- Les conventions de nommage en Kotlin.

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 de nommage en 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 et pragmatique.
- Statiquement typé, mais inférence de types.
- C'est aussi un langage fonctionnel.

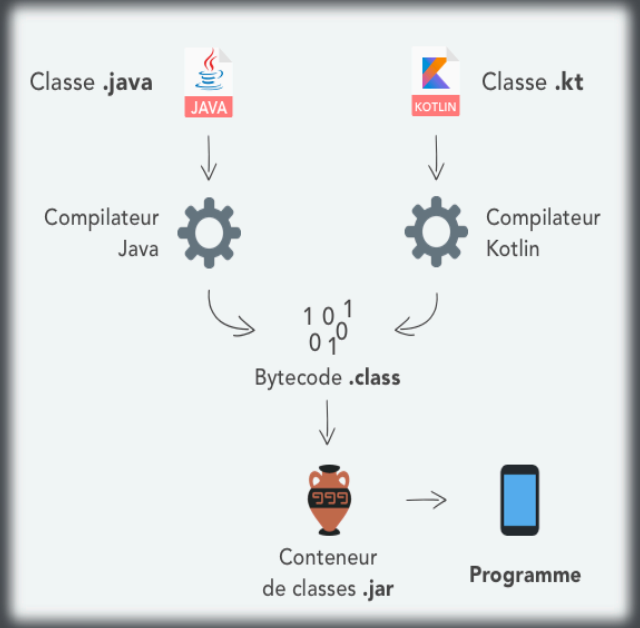
Pourquoi le Kotlin ?

Inférence de type

```kotlin
val number = 123
val message = "Hello world !"
fun sayHello() = "Hello world !"
```

Copier

Introduction à la JVM (Java Virtual Machine)



Lancer notre premier script

Nous avons différentes solutions pour lancer du code Kotlin :

- En ligne de commande.
- En ligne (sur le web).
- Avec IntelliJ (plus tard).

Lancer notre premier script

En ligne de commande

Récupérons le compilateur sur GitHub, plus bas dans la catégorie assets, par exemple : `kotlin-compiler-1.8.21.zip`.
Décompressions le fichier.
Nous pouvons ajouter le répertoire `/bin` dans le path pour que cela soit plus pratique.
Écrivons maintenant notre premier programme : `hello.kt` :

```
notepad hello.kt
```

```
1 fun main() {
2     println("Hello, World!")
3 }
```

Copier

Compilons notre code :

```
1 kotlinc hello.kt -include-runtime -d hello.jar
```

Copier

Nous pouvons le lancer maintenant :

```
1 java -jar hello.jar
```

Copier

Lancer notre premier script

En ligne de commande

Nous pouvons aussi récupérer le compilateur natif windows, toujours sur GitHub, par exemple `kotlin-native-windows-1.8.21.zip`.
Nous pouvons ajouter le répertoire `/bin` dans le path pour que cela soit plus pratique.
Reprenons notre premier programme : `hello.kt` :
`notepad hello.kt`

```
1 fun main() {
2     println("Hello, World!")
3 }
```

Copier

Compilons notre code :

```
1 kotlinc hello.kt
```

Copier

Nous pouvons le lancer un executable maintenant :

```
1 program.exe
```

Copier

Le premier lancement est long, mais les suivants sont rapides.

Lancer notre premier script

En ligne de commande : REPL

Nous allons maintenant tester REPL (Read–Eval–Print Loop) qui permet de tester du code facilement.
Pour cela lançons la commande : `kotlinc`. Testons quelques commandes, par exemple :

```
40+2
"Hello, World!"
println("Hello, World!")
1
1.0
```

Copier

Lancer notre premier script

En ligne

Nous pouvons utiliser le compilateur en ligne à l'adresse : https://play.kotlinlang.org.

```
fun main() {
    println("Hello, World!")
}
```

Copier

Lancer notre premier script

Nombreux exercices en ligne : https://play.kotlinlang.org/koans.

Lancer notre premier script

Pour aller plus loin, vous pouvez utiliser Gradle pour compiler votre projet Kotlin.

La structure d'une application Kotlin

Les répertoires

La structure des répertoires 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 répertoire : `network/socket`.

La structure d'une application Kotlin

Les fichiers

Un fichier ne contenant qu'une seule classe, sera nommé du nom de celle-ci (en utilisant le pascal case), suivit de l'extension `.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'extension) 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

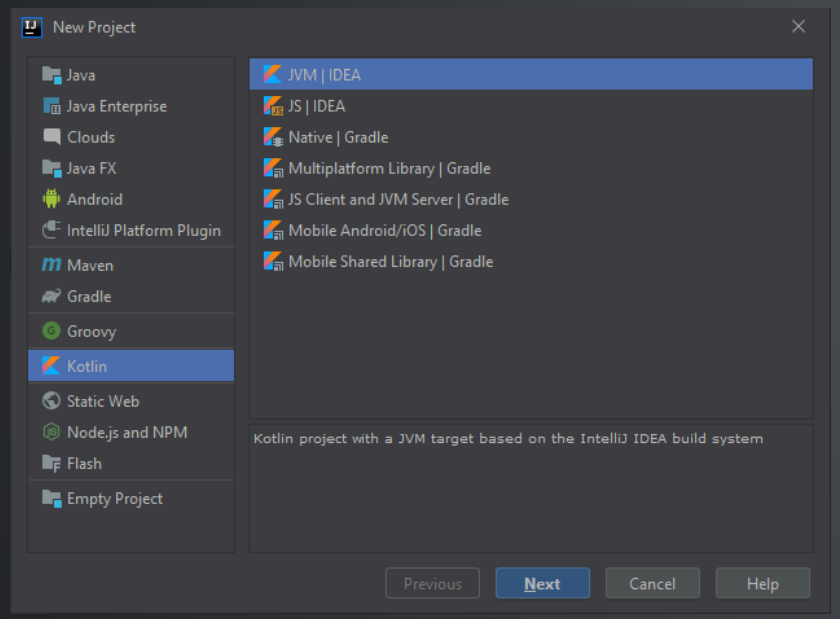
Installation

Commençons par installer InteliJ : Télécharger.
Optons pour la version **Community**.

Kotlin et IntelliJ IDEA

Création du projet

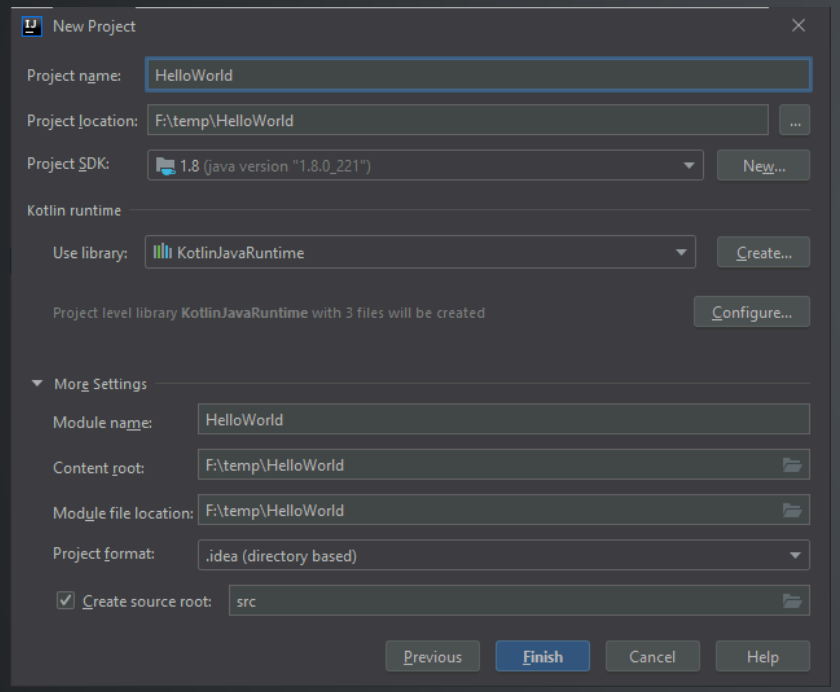
Il est temps de créer notre première application : **File / New / Project**. Sélectionner **Kotlin / JVM | IDEA**.
Nouvelle version ici.



Kotlin et IntelliJ IDEA

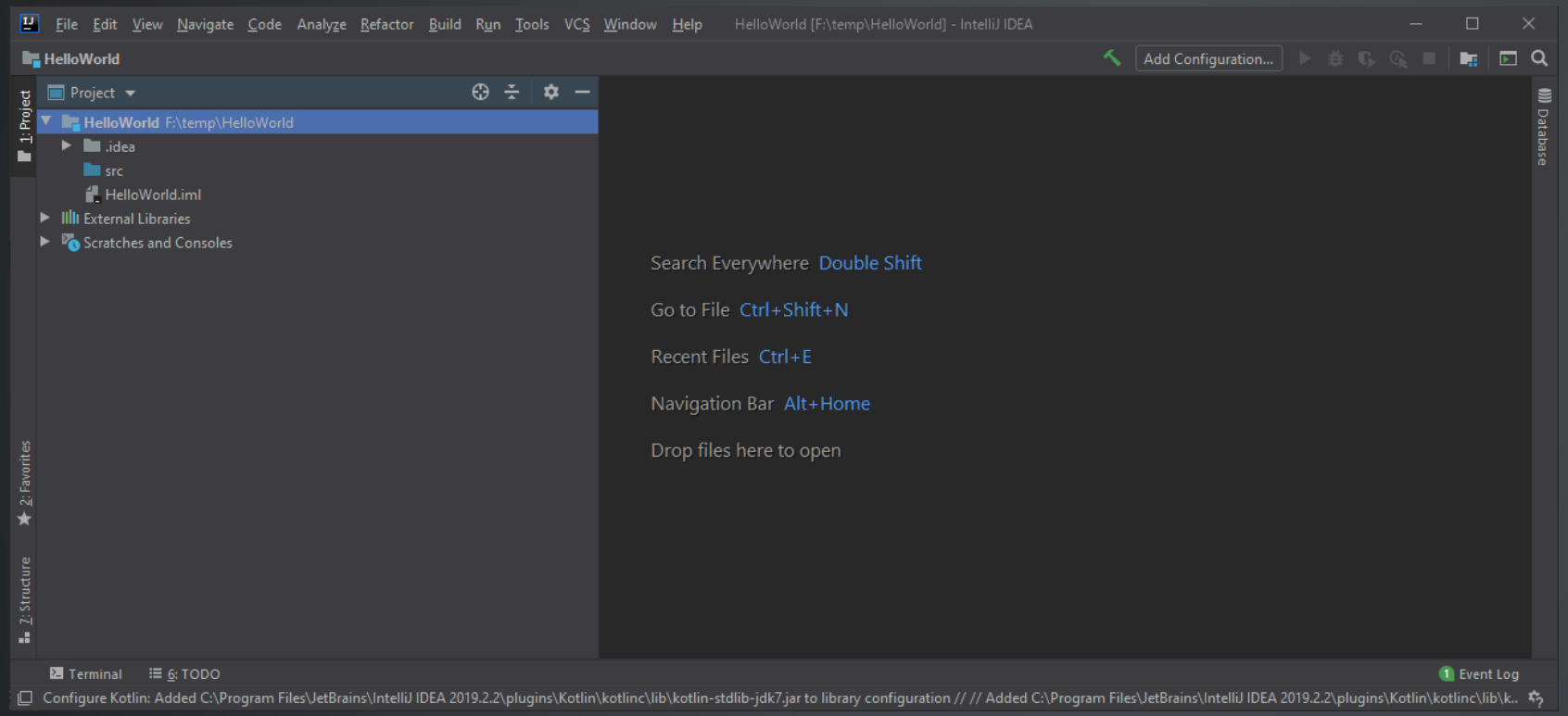
Nommage

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



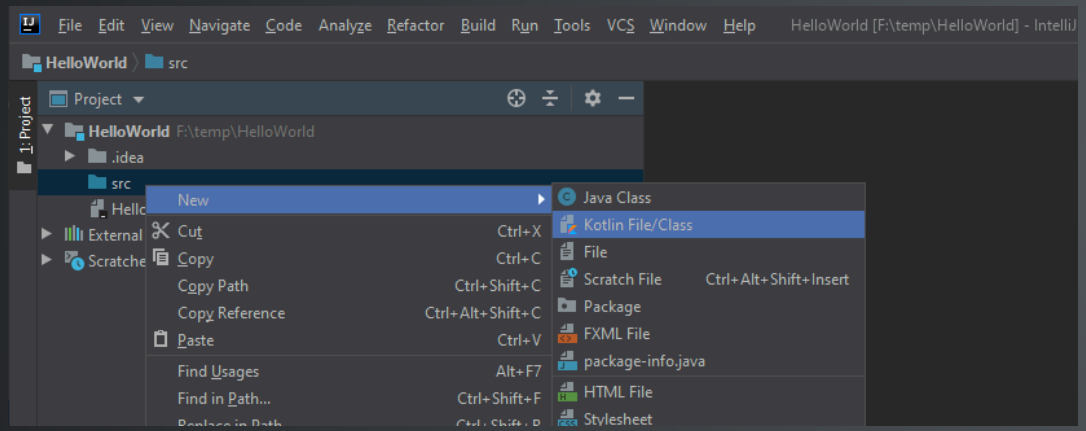
Kotlin et IntelliJ IDEA

Nous devrions obtenir le résultat suivant :



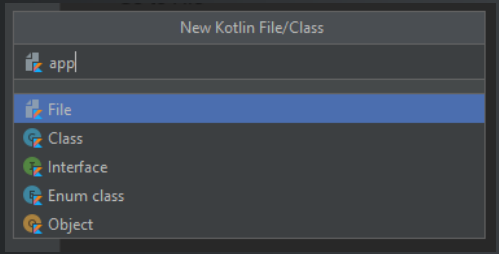
Kotlin et IntelliJ IDEA

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

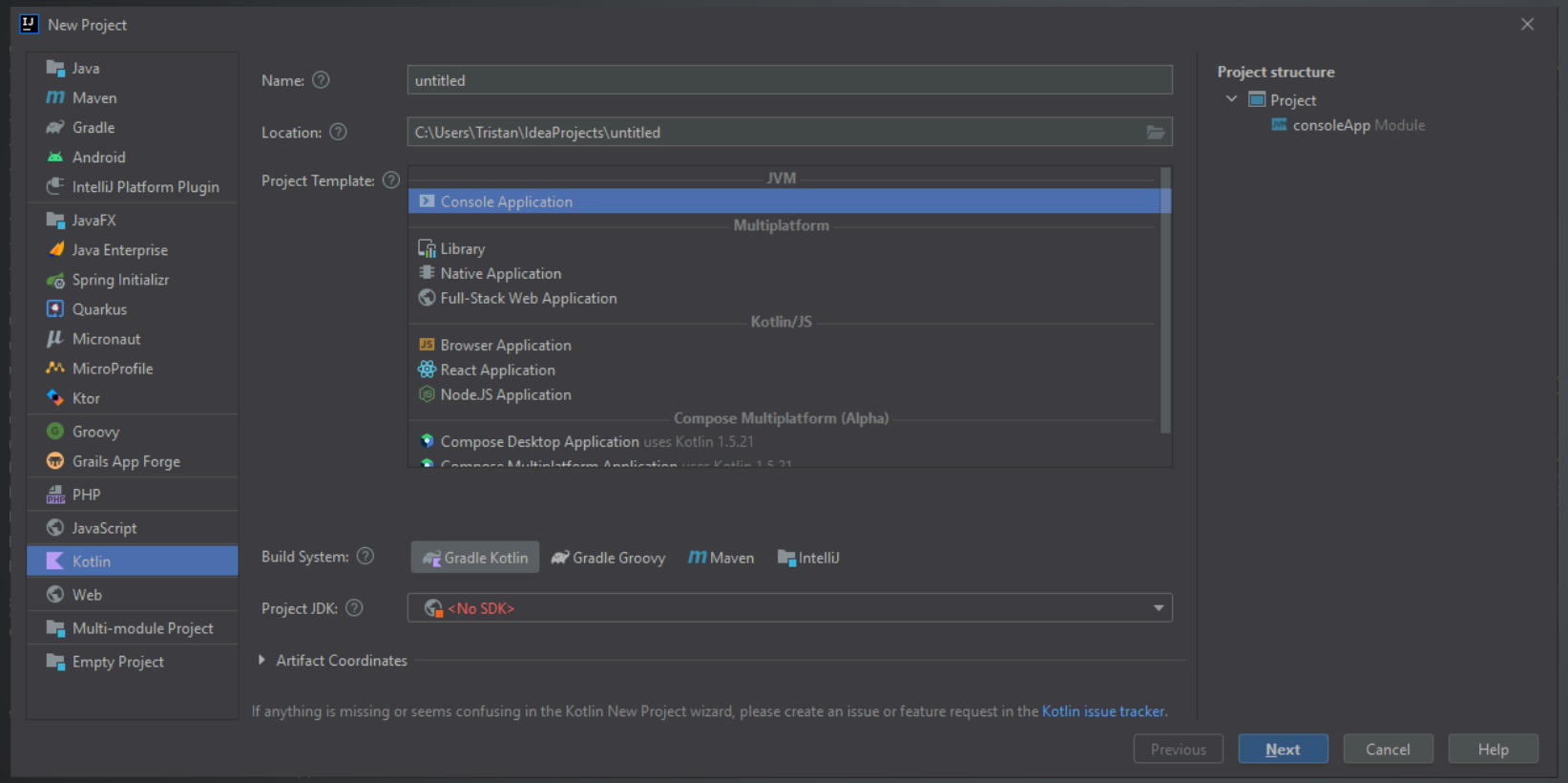
Ajoutons la fonction principale main :
Taper `main`, puis la touche **TAB**, pour lancer l'autocomplétion.
Il reste juste à compléter le corps de la méthode pour obtenir le résultat suivant :

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

Copier

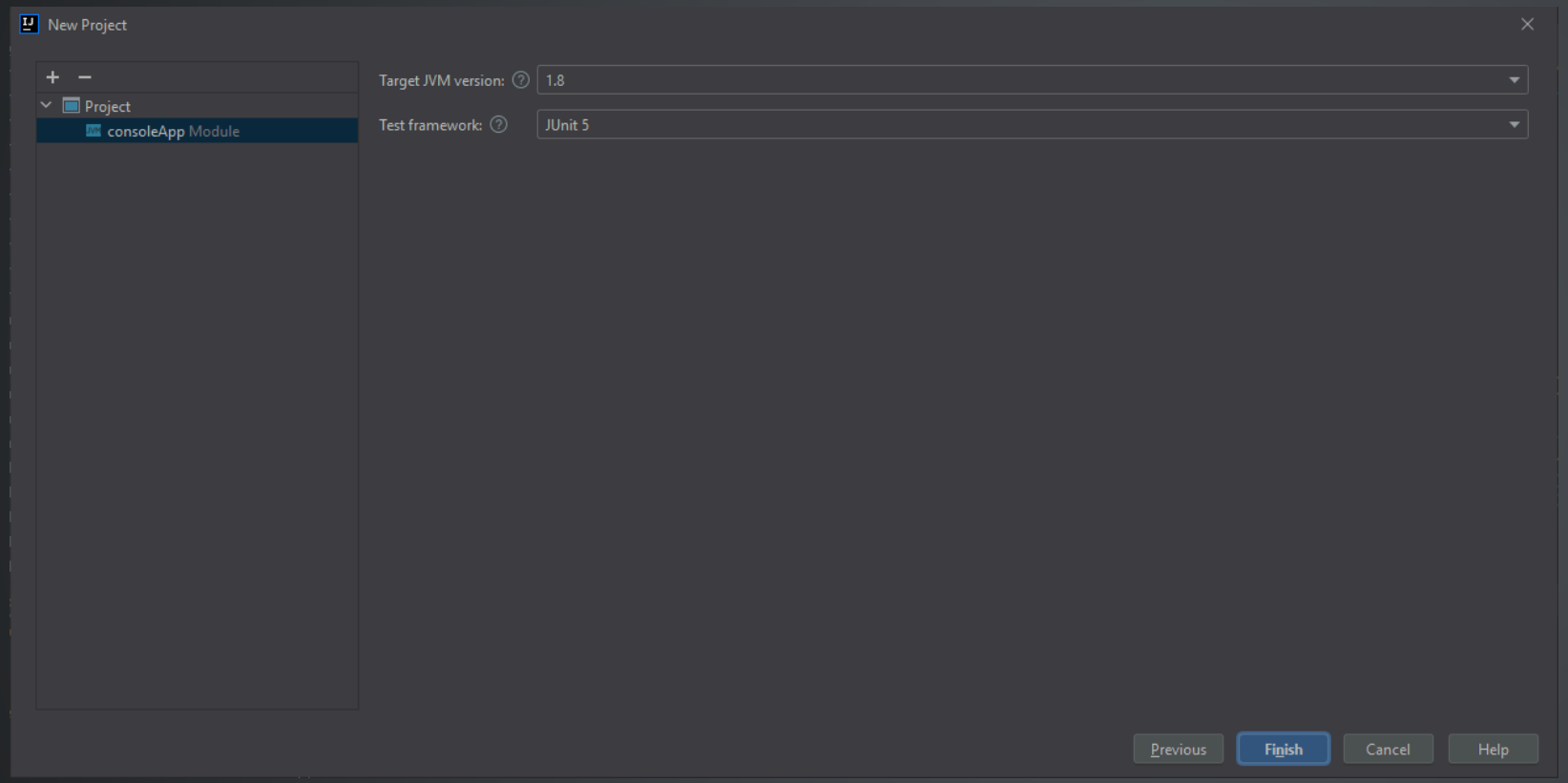
Kotlin et IntelliJ IDEA

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



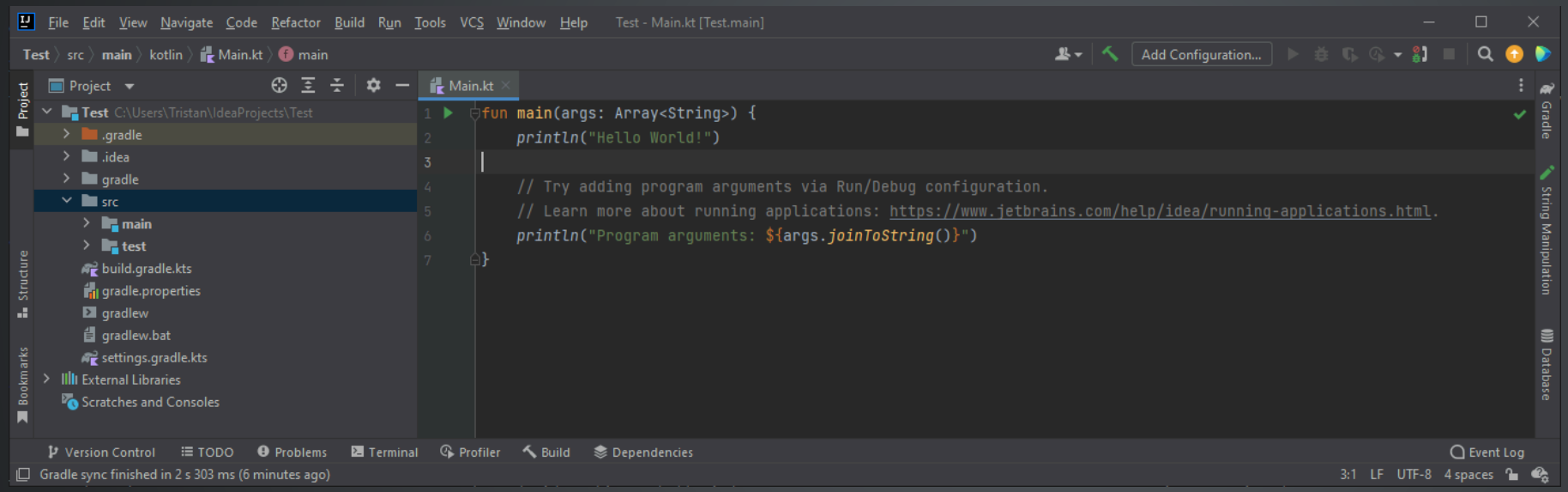
Kotlin et IntelliJ IDEA

Gardons les paramètres par défaut :



Kotlin et IntelliJ IDEA

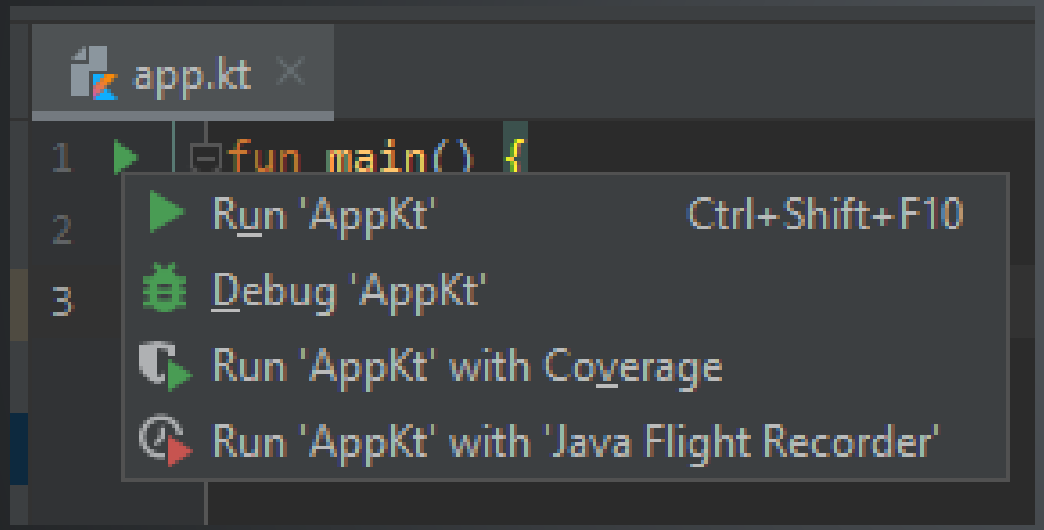
Nous obtenons :



Kotlin et IntelliJ IDEA

Exécuter le code

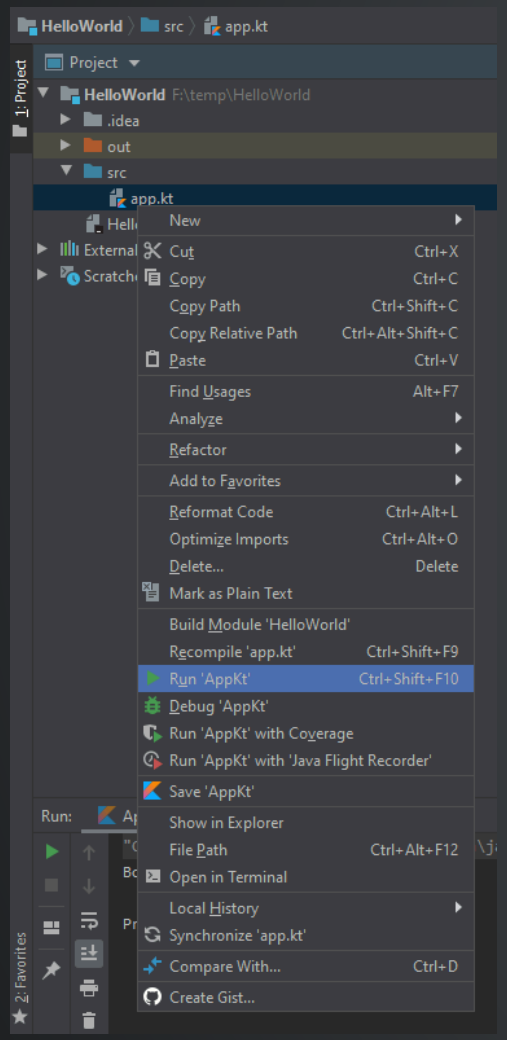
Il y a plusieurs façons de lancer le code, la plus rapide est de cliquer sur le bouton vert **Run** :



Kotlin et IntelliJ IDEA

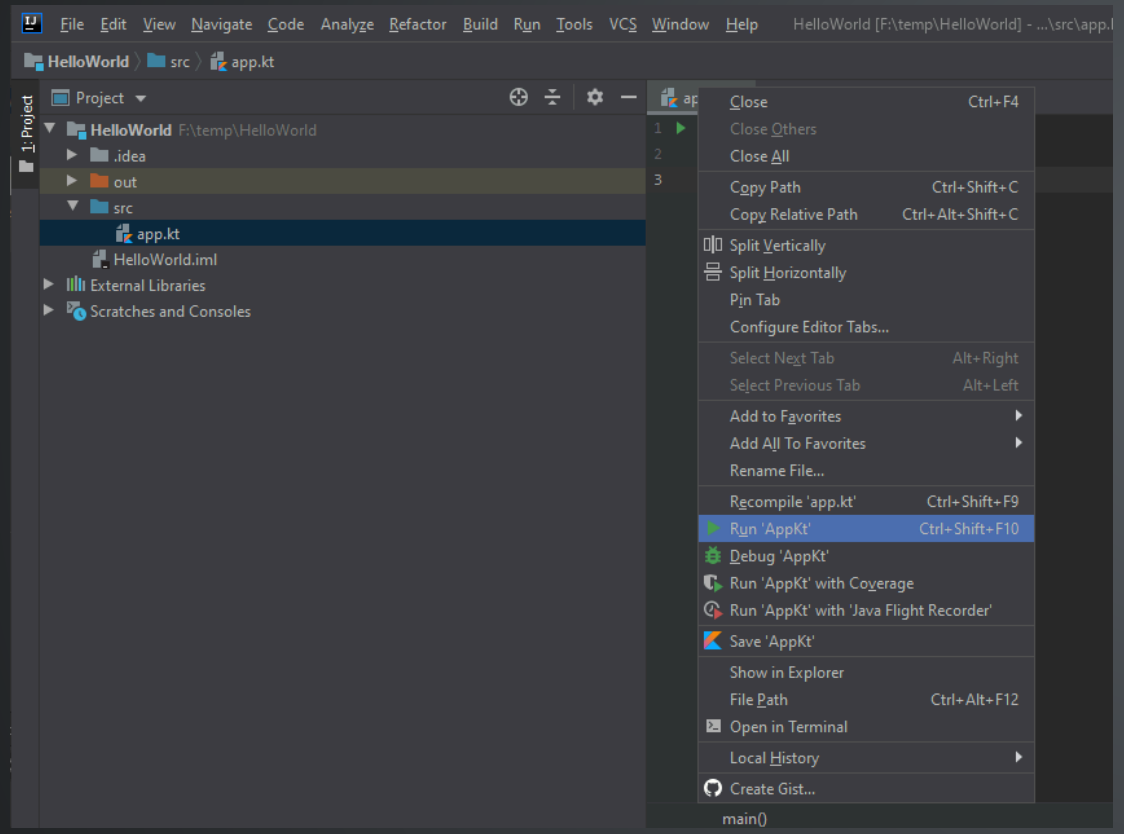
Exécuter le code

Ou encore :



Exécuter le code

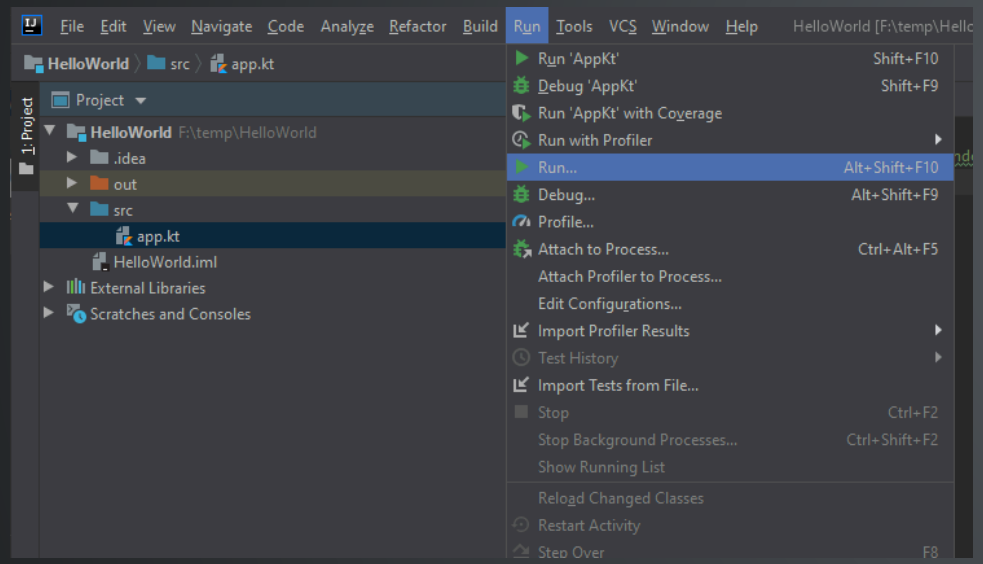
Ou bien :



Kotlin et IntelliJ IDEA

Exécuter le code

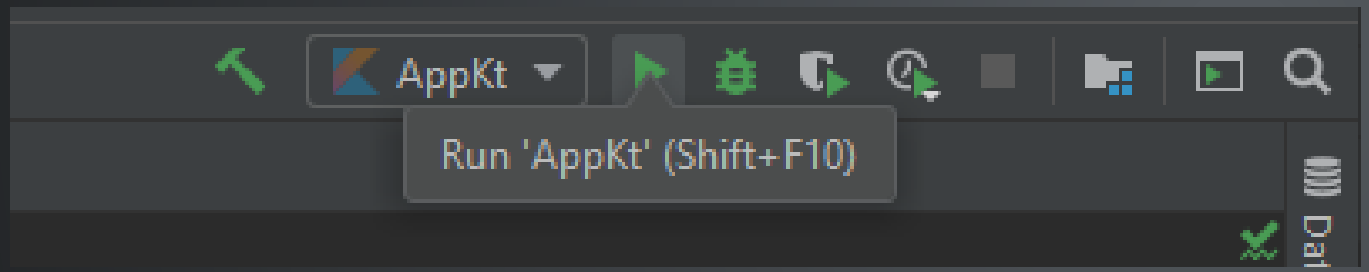
Ou bien encore :



Kotlin et IntelliJ IDEA

Exécuter le code

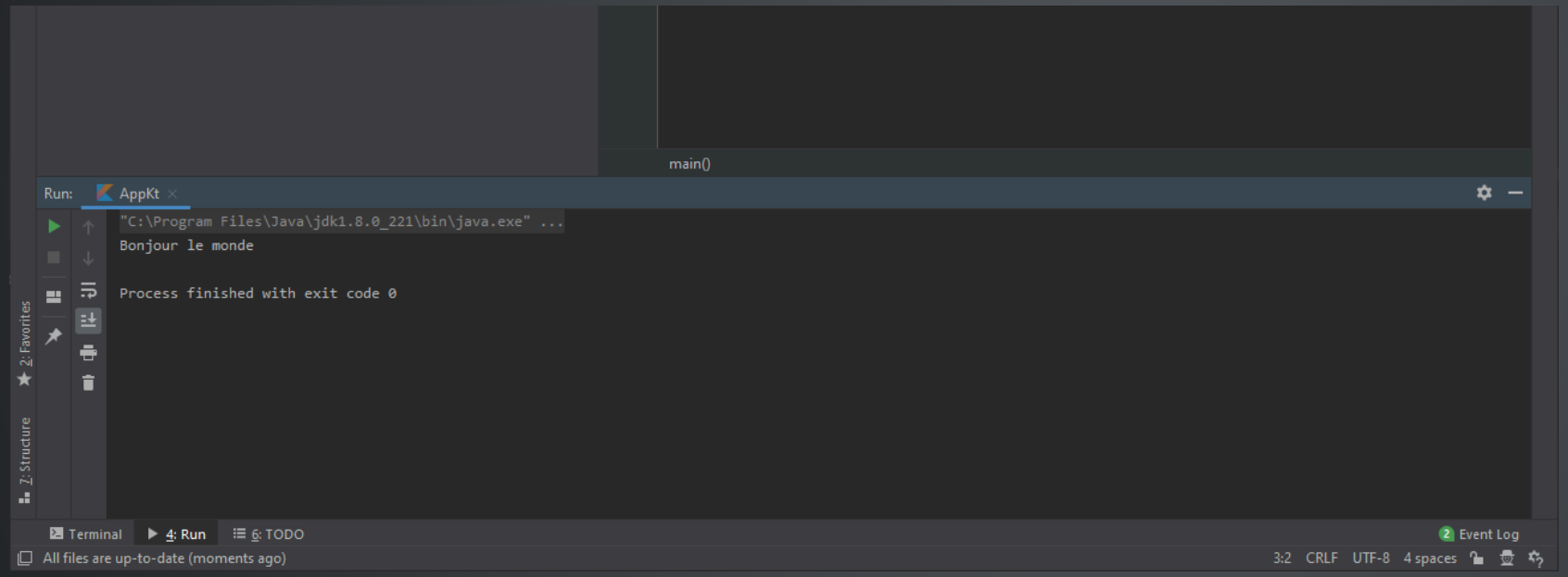
Ce qui permet d'avoir un nouveau raccourci :



Kotlin et IntelliJ IDEA

Exécuter le code

Ce qui nous donnera le résultat :



Kotlin et IntelliJ IDEA

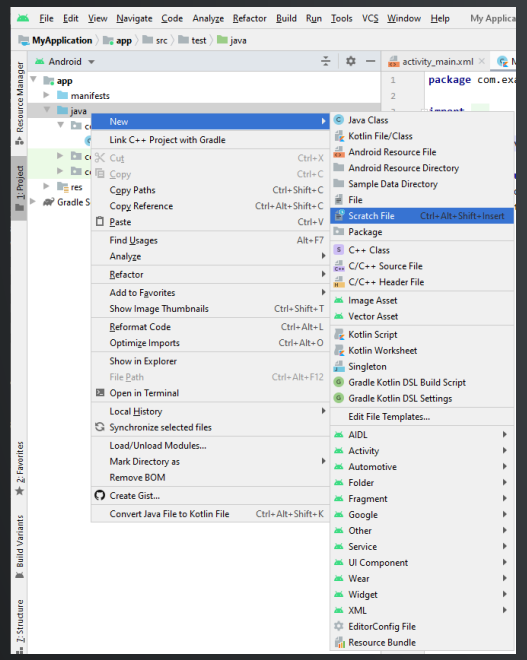
Nous pouvons aussi directement lancer notre code dans :

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

Le scratch

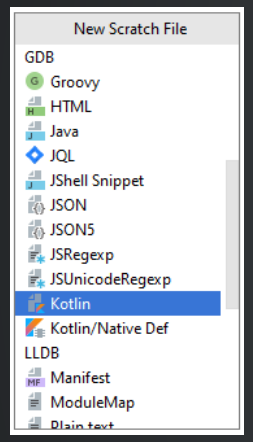
Commençons par ouvrir un espace pour faire nos tests : un scratch.
Au choix :

- Menu : File | New | Scratch file.
- Ctrl + Alt + Shift + Inser.
- Sur Android Studio : Click droit sur app dans notre projet (par exemple), puis : New/Scratch File.



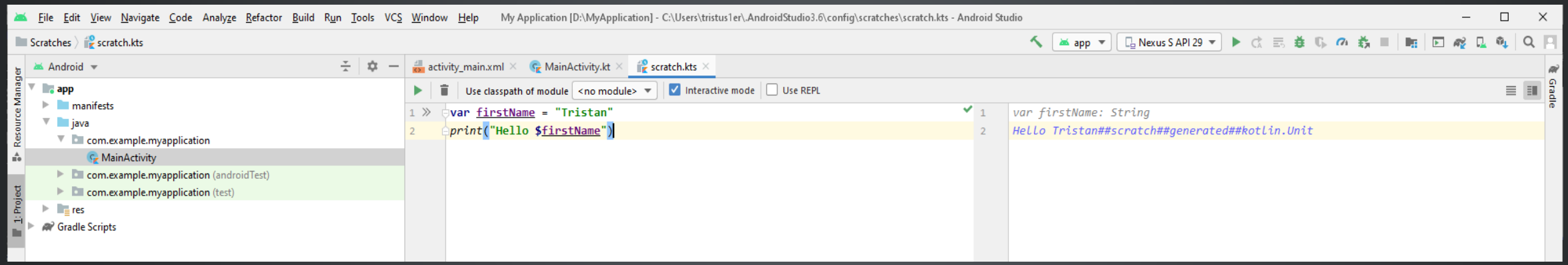
Kotlin et IntelliJ IDEA

Choisissons le type de fichier : Kotlin :



Kotlin et IntelliJ IDEA

Pour obtenir le résultat suivant :



Référence vers les chaînes de caractère.

Les conventions utilisées avec Kotlin

Règles de nommage

Elles sont identiques à celles en Java (nom des packages et des méthodes). À 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(...) }
```

Copier

Les conventions utilisées avec Kotlin

Règles des espaces

Quelque unes des 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"
```

Copier

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()
```

Copier

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

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

Copier

Pour les valeurs des enums, il est possible d'utiliser les majuscules séparées 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.
- Les commentaires.
- Structures conditionnelles If et When.
- Boucles et ranges en Kotlin.
- Collections en Kotlin.
- Packages et imports en Kotlin.

Déclaration de variables en Kotlin

- val : valeur
- var : variable

```kotlin
val message = "Hello world !"
val message: String = "Hello world !"
var message: String = "Hello world !"
```

Copier

Déclaration de variables en Kotlin

Le code :

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

Copier

Est équivalent à :

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

Copier

Déclaration de variables en Kotlin

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

Copier

```
1  import kotlin.random.Random
2
3  fun isUserHappy() = Random.nextInt(0, 100) % 2 == 0
4  fun main() {
5      val message: String
6      if (isUserHappy())
7          message = "That's great"
8      else
9          message = "What's going on?"
10
11     println(message)
12 }
```

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
```

Copier

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
```

Copier

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()
```

Copier

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

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

Copier

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

Déclaration de variables en Kotlin

Utilisation d'une variable dans une chaîne de caractère

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

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

Copier

Référence vers les chaînes de caractère.

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/";
```

Copier

Deviendra en Kotlin :

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

Copier

Utilisation de variables "Basic Types" en Kotlin

Plusieurs types basiques sont disponibles en Kotlin :

- Les nombres.
- Les caractères.
- Les booléens.
- Les tableaux.
- Les chaînes de caractères (String).
- Les types spéciaux.

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
```

Copier

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
```

Copier

Nombres

Conversion automatique des types.

Copier

```
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

Conversion explicite.

Pour convertir les types numériques, il faudra 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).

Copier

```
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

Copier

```
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 }
```

Copier

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 :

Copier

```
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>
        // ...
}
```

Copier

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) }
```

Copier

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) }
```

Copier

Les tableaux

Exercice

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

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

Copier

Les tableaux

Solution

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

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

Copier

Les tableaux

Exercice

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

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

Copier

Les tableaux

Solution

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

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

Copier

Les chaînes de caractères (String)

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

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

Copier

Les chaînes de caractères (String)

La valeur d'une chaîne de caractère.

Kotlin dispose de deux méthodes pour définir les valeurs des chaînes de caractères :

- Avec échappement (avec `"`)
- Brutes (avec `"""`)

```
1 val stringWithEscape = "Hello, world!\n"
2 println(stringWithEscape)
3
4 val stringRaw = """
5     for (c in "foo")
6         print(c)
7 """
8 println(stringRaw)
```

Copier

Les chaînes de caractères (String)

Modèles (String templates)

Les chaînes de caractère peuvent contenir des expressions (des morceaux de code), qui sont préfixés par $.

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

Copier

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 """
```

Copier

Les chaînes de caractères (String)

Exercice

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

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

Copier

Les chaînes de caractères (String)

Solution

```
fun main() {
    fun hello(name: String): String = "Hello, ${name.toUpperCase()}"
    print(hello("Tristan"))
}
```

Copier

Les types "spéciaux"

Kotlin comporte deux types "spéciaux" :

- Any qui correspond à Object en Java (n'importe quel type d'objet).
- Unit qui correspond à `void` en Java. Autrement dit : rien.

Les commentaires

Il existe deux types de commentaires, comme en Java :

```
/*
Les commentaires sur plusieurs
lignes, pour pouvoir bien s'exprimer.
Vous pouvez écrire autant que vous voulez.
*/
// ceci est un commentaire uniligne.
```

Copier

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.

Copier

```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
}
```

Copier

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.

```
var apiReponse = 404
when (apiReponse) {
    200 -> "OK"
    404 -> "NOT FOUND"
    401 -> "UNAUTHORIZED"
    403 -> "FORBIDDEN"
    else -> "UNKNOWN"
}
```

Copier

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).

Structures conditionnelles If et When

Selon le cas (when) (aller plus loin)

Le when peut aussi être utilisé comme une expression

Copier

```
var apiReponse = 404
fun printResponse(apiReponse: Int) = when (apiReponse) {
    200 -> print("OK")
    404 -> print("NOT FOUND")
    401 -> print("UNAUTHORIZED")
    403 -> print("FORBIDDEN")
    else -> print("UNKNOWN")
}
printResponse(apiReponse)
```

Boucles et ranges en Kotlin

Tant que faire se peut (while)

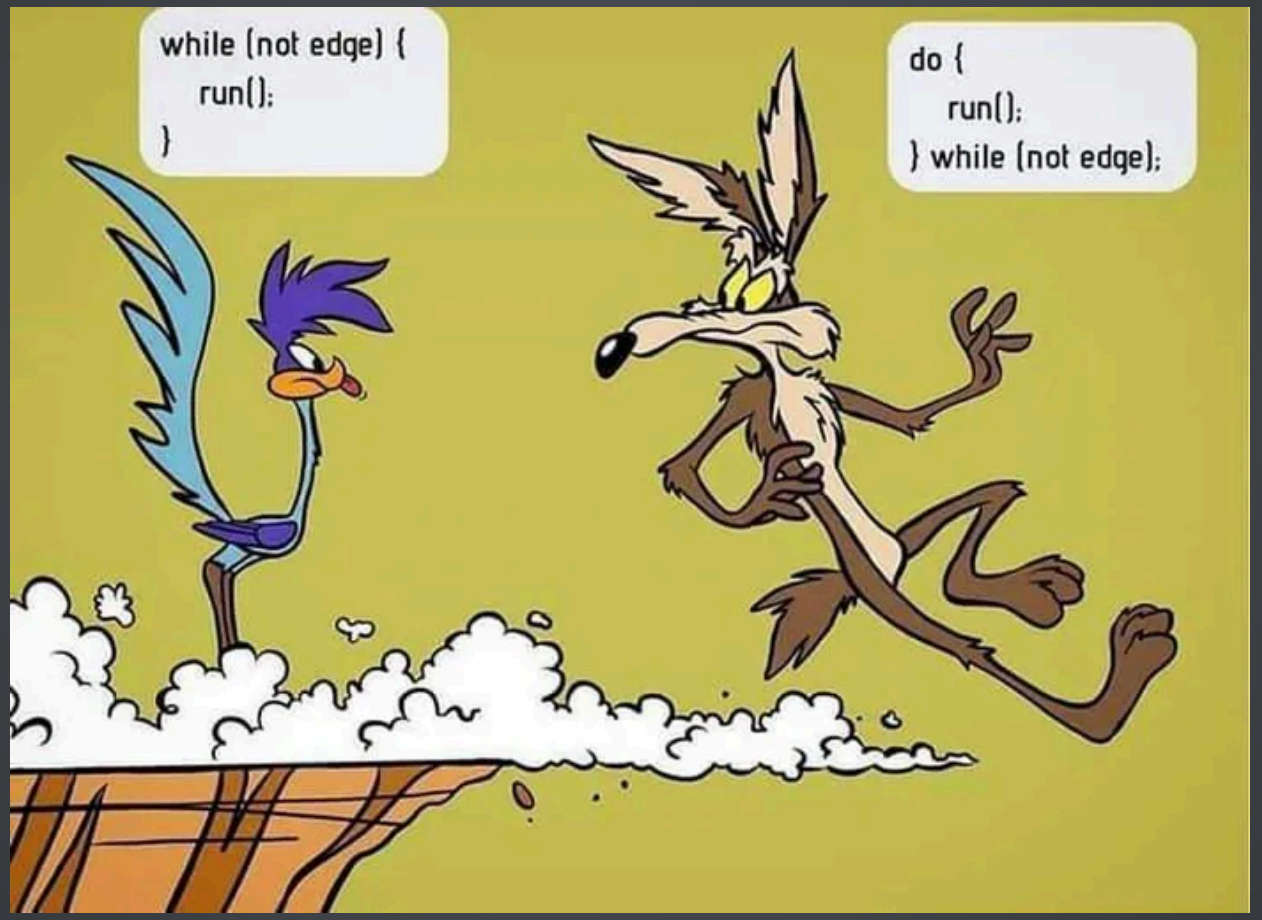
En Kotlin, la syntaxe du `while` est exactement la même qu'en Java :

```kotlin
var isRaining = true
while (isRaining){
    println("I don't like rain.")
}

do {
    println("I don't like rain.")
} while (isRaining)
```

Copier

While illustration



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 :

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

Copier

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

```
val names = listOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")

for(name in names) {
    println("This developer rocks: $name")
}
```

Copier

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     println(i)
3 }
```

Copier

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

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

Copier

Pour utiliser l'ordre décroissant, la syntaxe sera :

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

Copier

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)
```

Copier

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 }
```

Copier

Boucles et ranges en Kotlin

Boucle pour (for loop) (suite)

Il est possible d'itérer sur une liste en utilisant les indexes :

```
1 for (i in array.indices) {
2     println(array[i])
3 }
```

Copier

Ou alors en utilisant le format suivant qui met en œuvre la déstructuration (destructuring) :

```
1 for ((index, value) in array.withIndex()) {
2     println("the element at $index is $value")
3 }
```

Copier

Boucles et ranges en Kotlin

Break et continue

Ils fonctionnent de la même manière qu'en Java.

Boucles et ranges en Kotlin

Exercice

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

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

Copier

Le résultat attendu est :

```
U n e   c h a î n e   d e   c a r a c t è r e s
```

Boucles et ranges en Kotlin

Solution

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

Copier

Solution bis :

```
var stringValue = "Une chaîne de caractères"
stringValue.forEach { print("$it ")}
```

Copier

Solution ter :

```
var stringValue = "Une chaîne de caractères"
stringValue.toCharArray().joinToString(" ")
```

Copier

Référence joinToString.

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

Copier

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

Boucles et ranges en Kotlin

Solution bis

Copier

```
fun main(args: Array<String>) {
    for (x in 0 until 10) {
        println( when {
            (x % 3 == 0 && x % 5 == 0) -> "FizzBuzz"
            (x % 3 == 0) -> "Fizz"
            (x % 5 == 0) -> "Buzz"
            else -> 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 existe 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 élément 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.
- L'ensemble (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

- Le dictionnaire (Map), est un ensemble d'éléments composés d'une paire (clé-valeur). Les clés ont des valeurs uniques qui désignent un seul objet de la collection. Les valeurs peuvent apparaître 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
```

Copier

Collections en Kotlin

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

```
val listOfNames = listOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
listOfNames[0]
//listOfNames[0] = "Mathieu NEBRA" // Error: List is immutable

val mutableListOfNames = mutableListOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
mutableListOfNames[0]
mutableListOfNames[0] = "Mathieu NEBRA" // OK

val setOfNames = setOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
setOfNames.first()
//setOfNames.add("Mathieu NEBRA") // Error: Set is immutable

val mutableSetOfNames = mutableSetOf("Jake WHARTON", "Joe BIRCH", "Robert MARTIN")
mutableSetOfNames.first()
mutableSetOfNames.add("Mathieu NEBRA") // OK
```

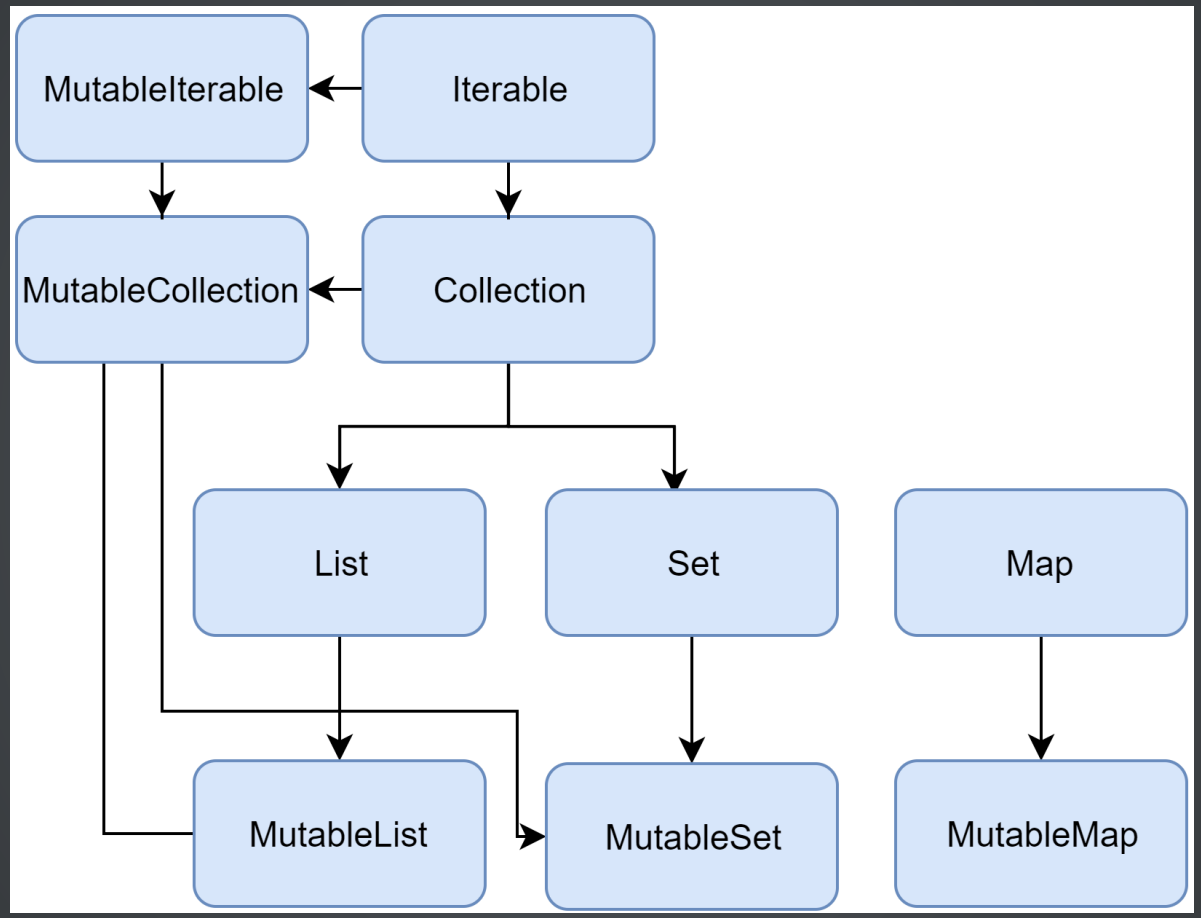
Copier

Exemple de tableau et de `Map` :

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

Copier

Collections en Kotlin



Packages et imports en Kotlin

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

```kotlin
package org.example

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

// ...
```

Copier

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

Imports

Chaque fichier peut faire appel à des classes externes, pour cela, on va faire appel à des `import` :

```
1 import org.example.Message // Message is now accessible without qualification
```

Copier

Il est aussi possible d'importer tout un groupe de classes avec l'utilisation de * :

```
1 import org.example.* // everything in 'org.example' becomes accessible
```

Copier

Packages et imports en Kotlin

Imports divers

Le mot clé `import` est aussi utilisé pour importer d'autres types que des classes, telles que :

- Fonctions et propriétés top level
- Fonctions et propriétés déclarées dans des déclarations object
- Fonctions d'extensions
- Constantes d'un `ENUM`

Exercices : faisons les deux premiers exercices :
Practice: Kotlin Fundamentals

Les fonctions - Partie 1

- Fonctions en Kotlin.
- Fonctions "expression seule".
- Paramètres des fonctions en Kotlin.
- Returns et Local Returns en Kotlin.

Fonctions en Kotlin

Déclaration

On utilise le mot clé fun

```
1 fun double(x: Int): Int {
2     return 2 * x
3 }
```

Copier

Fonctions en Kotlin

Utilisation

Appel traditionnel :

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

Copier

Appel d'une fonction membre d'un objet :

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

Copier

Fonctions "expression seule"

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
```

Copier

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
```

Copier

Paramètres des fonctions en Kotlin

Les paramètres simples

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

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

Copier

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) { /*...*/ }
```

Copier

Paramètres des fonctions en Kotlin

Exercice

Écrivez 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()
```

Copier

Paramètres des fonctions en Kotlin

Solution

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

Copier

Dans ce cas, si l'on précise une valeur null, c'est cette valeur qui est utilisée et la méthode n'affichera rien.
Ou :

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

Copier

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()
```

Copier

Paramètres des fonctions en Kotlin

Solution

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

Copier

Paramètres des fonctions en Kotlin

Exercice

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

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

Copier

Paramètres des fonctions en Kotlin

Solution

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

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

Copier

Solution bis :

```kotlin
fun strEq(s1: String, s2: String, ignoreCase: Boolean): Boolean = s.equals(p, ignoreCase)
```

Copier

Référence : equals.

Paramètres des fonctions en Kotlin

Exercice

Écrivez la fonction qui affiche la liste des langages présents dans le tableau.
Note : le mot clé `Unit` indique que la méthode ne retourne pas de valeur, c'est l'équivalent à `void` en Java.

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

Copier

Paramètres des fonctions en Kotlin

Solution

```
1 val languages = arrayOf("Java", "JavaScript", "Go", "Kotlin")
2 fun printLanguages(): Unit { for (language in languages) println(language) }
3
4 fun main() {
5     printLanguages()
6 }
```

Copier

Solution bis :

```
1 fun printLanguages() { languages.forEach { println(it) } }
```

Copier

Solution ter :

```
1 fun printLanguages() = languages.forEach { println(it) }
```

Copier

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.

```kotlin
fun tenFirstNumber(): Unit = TODO()
fun countdown(): Unit = TODO()
fun firstEvenNumbers(): Unit = TODO()
fun firstOddNumbers(): Unit = TODO()
```

Copier

Paramètres des fonctions en Kotlin

Solution

Copier

```
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

Solution bis

Copier

```
fun tenFirstNumber() = (0..9).forEach { println(it) }
fun countdown() = (10 downTo 0).forEach { println(it) }
fun firstEvenNumbers() = (1..20 step 2).forEach { println(it) }
fun firstOddNumbers() = (0..9).forEach { println(it * 2) }

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 :

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
/*...*/
}
```

Copier

Paramètres des fonctions en Kotlin

Paramètres nommés utilisation

Nous pouvons utiliser les paramètres par défaut :

```
reformat(str)
```

Copier

Nous pouvons préciser tous les paramètres :

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

Copier

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

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

Copier

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
```

Copier

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
}
```

Copier

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)
```

Copier

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 :

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

Copier

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

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

Copier

Returns en Kotlin

Fonctions locales

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

Copier

```
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())
}
```

Classes en Kotlin

- La POO.
- Une classe.
- Les attributs.
- Méthodes (Functions Members).
- Visibilité des membres en Kotlin.
- Héritage en Kotlin.
- La surcharge.
- Abstract Classes en Kotlin.
- Interface en Kotlin.
- Data Classes en Kotlin.
- Enum Classes en Kotlin.

Classes en Kotlin

- La POO.
- Une classe.
- Les attributs.
- Méthodes (Functions Members).
- Visibilité des membres en Kotlin.
- Héritage en Kotlin.
- La surcharge.
- 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.

La POO : Programmation Orientée Objet

L'objectif de la Programmation Orientée Objet est d'organiser le code afin de mieux pouvoir le réutiliser.
Pour cela, nous utilisons l'encapsulation qui permet de :

- Rassembler dans une même structure :
  - Attributs (données) ou "variables membres".
  - Méthodes (fonctions).
- Garantir l’intégrité des données en :
  - Ne laissant visible que ce qui doit être réellement utilisé (visibilité).
  - N’accédant aux données que par des méthodes.

La POO : Programmation Orientée Objet

Classe = description abstraite d'un objet.
Instancier une classe = créer un objet sur son modèle (grâce au constructeur).
Propriété = attribut accessible par un getter et/ou setter.

Une classe

Déclaration

On utilise le mot clé class

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

Copier

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
```

Copier

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) { /*...*/ }
```

Copier

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) { /*...*/ }
```

Copier

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) { /*...*/ }
```

Copier

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
    public String getEmail() { return email; }
    public String getPassword() { return password; }
```

Copier

En Kotlin son équivalent est :

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

Copier

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 = "")
```

Copier

Une classe

Modification des accesseurs par défaut

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

Copier

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

    var email: String = email
        get() { println("User email read access done."); return field}
        set(value) { println("User email write access done."); field = value}
}
```

Une classe

Utilisation (instantiation) 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)
```

Copier

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" // Setter
4 println(user.email) // Getter
```

Copier

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 (affectation d'une nouvelle valeur pour le prénom de notre objet créé) et vérifiée que le nom est bien en lecture seule (en essayant de modifier sa valeur).

```
1 class Person
```

Copier

Une classe

Solution

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

fun main() {
    var father = Person("Anakin", "Skywalker")
    println(father)
    println( "${father.firstName} ${father.lastName}")

    me.firstName = "Luke"
//    me.lastName = "Organa d'Alderaan"

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

Copier

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

Copier

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

    var email: String = email
        get() { println("User email read access done."); return field}
        set(value) { println("User email write access done."); field = value}
}

fun main() {
    var currentUser = User("tristan.salaun.pro@gmail.com", "azerty", 41)
    currentUser.email = "test@test.com"
    println(currentUser.email)
}
```

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` :

```kotlin
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"
}
```

Copier

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

```kotlin
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
}
```

Copier

Méthodes (Functions Members)

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

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

Copier

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

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

Copier

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
```

Copier

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)
```

Copier

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 }
```

Copier

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 }
```

Copier

La surcharge

La surcharge des propriétés

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

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle: Shape() {
    override val vertexCount = 4
}
```

Copier

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.

```
interface Shape {
    val vertexCount: Int
}

class Polygon: Shape {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

Copier

La surcharge

L'appel à la classe parente

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

Copier

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle: Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

Classes abstraites en Kotlin

Une classe et quelques membres peuvent être 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 :

```kotlin
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle: Polygon() {
    override abstract fun draw()
}
```

Copier

L'objet compagnon

Nous utiliseront principalement l'objet compagnon comme déclaration de méthodes ou d'attributs statiques à la classe.

Copier

```
class MyClass1 {
    companion object {
        private const val TAG = "MyClass1"
        fun myMethod() { ... }
    }
}
```

Interface en Kotlin

Les interfaces en Kotlin peuvent contenir 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
    }
}
```

Copier

Implémenter une interface :

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

Copier

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 :

Copier

```
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'animaux référencées par un même objet, de type `Animal`.

Copier

```
interface Animal {
  fun speak()
}

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

Polymorphisme en Kotlin

Solution

Copier

```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("Ssssssss") }

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

Polymorphisme en Kotlin

Surcharge de fonctions

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

Copier

```
fun printNumber(n: Number){
    println("Using printNumber(n: Number)")
    println(n.toString() + "\n")
}

fun printNumber(n: Int){
    println("Using printNumber(n: Int)")
    println(n.toString() + "\n")
}

fun printNumber(n: Double){
    println("Using printNumber(n: Double)")
    println(n.toString() + "\n")
}
```

Polymorphisme en Kotlin

Surcharge de fonctions

Quelles seront les méthodes appelées lors de l'exécution ?

Copier

```
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?
}
```

Polymorphisme en Kotlin

- printNumber(a) => Using printNumber(n: Number).
- printNumber(b) => Using printNumber(n: Int).
- printNumber(c) => Using printNumber(n: Double).

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` :

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

Copier

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

La copie

Il arrive régulièrement de devoir copier un objet, et de modifier quelques-unes de ses propriétés, tout en gardant le reste intact. C'est le but de la fonction générée `copy()`. Pour la classe `User` ci-dessous, l'implémentation serait la suivante :

```
1 fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

Copier

Ce qui nous permet d'écrire :

```
1 val jack = User(name = "Jack", age = 1)
2 val olderJack = jack.copy(age = 2)
```

Copier

Data Classes en Kotlin

La destructuration

Les fonctions du composant générées permettent le "destructuring" :

```kotlin
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age"
```

Copier

Classes standards

La librairie standard met à disposition les classes `Pair` et `Triple`. Dans la majorité des cas, les classes data sont plus appropriées, car elles permettent de nommer les propriétés, ce qui rend le code beaucoup plus compréhensible.

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 = "")
```

Copier

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)
}
```

Copier

Data Classes en Kotlin

Les listes dans les data classes

Ajoutons un attribut de classe qui sera de type `List`, et regardons comment l'affichage va se comporter.

Copier

```
data class Person (var firstName: String = "", var lastName: String = "", var list: List<Int>)

fun main() {
    val listValue = listOf(1,2,3,4,5,6)
    var personTristan = Person("Tristan", "SALAUN", listValue)
    var personMelody = personTristan.copy(firstName = "Mélody")
    println(personTristan)
    println(personMelody)
}
```

Data Classes en Kotlin

L'affichage sera :

```
1 Person(firstName=Tristan, lastName=SALAUN, list=[1, 2, 3, 4, 5, 6])
2 Person(firstName=Mélody, lastName=SALAUN, list=[1, 2, 3, 4, 5, 6])
```

Copier

Enum Classes en Kotlin

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

```kotlin
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Copier

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 être initialisées de la façon suivante :

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

Copier

Enum Classes en Kotlin

Exercice

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

Copier

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

Enum Classes en Kotlin

Solution

Copier

```
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.")
    }
}
```

Enum Classes en Kotlin

Solution bis

Copier

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

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

Copier

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 :

Copier

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

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()
```

Copier

(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.)

Sealed Classes en Kotlin

Une classe scellée est abstraite, elle ne peut pas être instanciée directement et peut avoir des membres abstraits.

Les classes scellées ne peuvent pas avoir de constructeurs non privés (private) : leurs constructeurs sont privés par défaut).

Notez que les classes qui étendent d'une sous-classe d'une classe scellée (héritage indirect), peuvent être déclarés n'importe où : pas obligatoirement dans le même fichier.

Sealed Classes en Kotlin

L'avantage principal de l'utilisation de ces classes scellées, est lorsque nous utilisons l'expression `when`. S'il est possible de vérifier que les cas, couvrent toutes les possibilités, alors il n'est pas nécessaire d'ajouter la clause `else`. Toutefois, cela ne fonctionne que si vous utilisez `when` en tant qu'expression et non comme une déclaration :

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Copier

Sealed Classes en Kotlin

Exemple sans classe scellée

Résolvons l'exercice situé ici.
Que pouvons-nous remarquer concernant l'usage de l'interface Expr ?

Sealed Classes en Kotlin

Ajoutons maintenant de nouvelles expressions, par exemple

```
1 data class Substract(val e1: Expr, val e2: Expr): Expr
2 data class Multiply(val e1: Expr, val e2: Expr): Expr
3 data class Divide(val e1: Expr, val e2: Expr): Expr
```

Copier

Que constatons-nous, et que devons-nous corriger ?

Sealed Classes en Kotlin

Solution

Copier

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    is Substract -> eval(expr.e1) - eval(expr.e2)
    is Multiply -> eval(expr.e1) * eval(expr.e2)
    is Divide -> eval(expr.e1) / eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Les fonctions - Partie 2

- Expressions Lambda en Kotlin.
- Extensions de fonctions en Kotlin.

Lambda expression en Kotlin

Les fonctions en Kotlin sont des fonctions "première classe" (first-class), c'est-à-dire qu'elles peuvent être 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.

Lambda expression en Kotlin

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
}
```

Copier

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 expressions lambdas sont couramment utilisées à cet usage :

Copier

```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 ->
    println("acc = $acc, i = $i, ")
    val result = acc + i
    println("result = $result")
    // The last expression in a lambda is considered the return value:
    result
})
```

Copier

```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

Invoquer un type fonction

La valeur d'un type fonction peut être appelée en utilisant son opérateur `invoke(...)` : par exemple `f.invoke(x)` ou simplement `f(x)`.

Si la valeur à un type receveur, alors, l'objet receveur doit être passé en premier paramètre. Une autre façon d'invoquer la valeur de type fonction avec un type de receveur est de préfixer avec l'objet receveur, comme s'il s'agissait d'une fonction d'extension, exemple : `1.foo(2)`.

Copier

```
fun main() {
    val stringPlus: (String, String) -> String = String::plus
    val intPlus: Int.(Int) -> Int = Int::plus

    println(stringPlus.invoke("<-", "->"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // extension-like call
}
```

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 })
```

Copier

La fonction `max` est une fonction d'ordre supérieur, qui prend une fonction en second paramètre. Cet 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
```

Copier

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 }
```

Copier

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 corps 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 }
```

Copier

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 }
```

Copier

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("...") }
```

Copier

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 omettre 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'
```

Copier

Lambda expression en Kotlin

Exercice

Écrivez la lambda qui permet d'additionner 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()
```

Copier

Lambda expression en Kotlin

Solution

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

Copier

Ou de manière plus concise :

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

Copier

Lambda expression en Kotlin

Exercice

Supposons que nous ayons besoin de développer une calculatrice. Commençons par écrire les lambdas correspondant aux opérations de base (addition, soustraction, multiplication et division), stockons les dans des variables et testons leur usage avec la fonction `executeOperation` écrite ci-dessous.

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

Copier

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

Lambda expression en Kotlin

Exercice

Copier

```kotlin
val addition: (Int, Int) -> Int = TODO()
val subtraction: (Int, Int) -> Int  = TODO()
val multiplication: (Int, Int) -> Int  = TODO()
val division: (Int, Int) -> Int  = TODO()

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))
}
```

Lambda expression en Kotlin

Solution

Copier

```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");
    }
});
```

Copier

Simplification d'une méthode Java en Kotlin

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

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

Copier

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 });
```

Copier

Simplification d'une méthode Java en Kotlin

L'équivalent en Kotlin :

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

Copier

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"); } )
```

Copier

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")  } )
```

Copier

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")  }
```

Copier

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")  }
```

Copier

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")  }
```

Copier

Simplification d'une méthode Java en Kotlin

Ce qui nous donne finalement le code suivant ...

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

Copier

Simplification d'une méthode Java en Kotlin

Avec les espaces superflus en moins, cela donne :

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

Copier

Qui correspond au code Java :

```
1 button.setOnClickListener(new View.OnClickListener() {
2     @Override
3     public void onClick(View v) {
4         Log.d(TAG, "User clicked button");
5     }
6 });
```

Copier

Ou encore :

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

Copier

Exercices : reprenons nos execices :

Practice: Kotlin Fundamentals

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*. Il est aussi possible d'ajouter des propriétés à une classe existante en utilisant le mécanisme d'*extension de propriétés*.

Extensions

Extensions de fonctions en Kotlin

Pour déclarer une fonction d'extension, 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>`

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

Copier

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>`.

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

Copier

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).

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

// Exemple d'appel :
fun main() {
    println("tristan".firstUpper()) // => Tristan
    println("TRISTAN".firstUpper()) // => Tristan
}
```

Copier

Extensions

Solution

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

fun main() {
    println("tristan".firstUpper())
    println("TRISTAN".firstUpper())
}
```

Copier

Autre solution :

```
fun String.firstUpper() = this.toLowerCase().capitalize()
```

Copier

Encore plus concis (sans utiliser le this) :

```
fun String.firstUpper() = toLowerCase().capitalize()
```

Copier

Délégation

- Délégation de propriétés en Kotlin

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
```

Copier

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

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

Copier

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 :

Copier

```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()
    println(example.someName)
    println(example.otherName)
```

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 :

Copier

```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 œuvre de la délégation `lazy` :

```
val myVar: String by lazy {
    println("Lazy init")
    "Hello"
}
println("myVar is not initialized yet")
println("$myVar My dear friend")
```

Copier

De manière plus classique/concise, nous utiliserons :

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

Copier

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 :

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

Copier

Délégation de propriétés en Kotlin

Exemple d'utilisation

Copier

```kotlin
var updated = false
var max: Int by Delegates.observable(0) { property, oldValue, newValue ->
    updated = true
}

println(max) // 0
println("updated is ${updated}")

max = 10
println(max) // 10
println("updated is ${updated}")
```

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.

```
fun main() {
    var maxCount: Int = 0

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

Copier

Qu'affiche le code ci-dessus ?

Délégation de propriétés en Kotlin

Solution

Copier

```
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`

La syntaxe est pratiquement la même que `observable`, sauf que la lambda, doit retourner un `Boolean`, qui indique si la valeur doit être modifiée, ou 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  }
```

Copier

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.

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

Copier

Délégation de propriétés en Kotlin

Solution

Copier

```
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

Copier

```
import kotlin.properties.Delegates

fun main() {
    var age: Int by Delegates.vetoable(initialValue = 0) { property, oldValue, newValue ->
        if (newValue > 0) true else {println("${property.name} rejected value $newValue staying at value $oldValue"); false}
    }

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

Délégation de propriétés en Kotlin

Delegate `notNull`

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

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

Copier

Délégation de propriétés en Kotlin

Exercice

Modifions le code pour le rendre valide (indiquons au compilateur que nous allons initialiser la variable tardivement) :

Copier

```
var person1: Person // Erreur sur cette ligne. à corriger.

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

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

Délégation de propriétés en Kotlin

Solution

Copier

```
var person1 by notNull<Person>()

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

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?> {}
```

Copier

Délégation de propriétés en Kotlin

Solution

Copier

```
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
    }
}

class Demo {
    var name by LogDelegate<String>()
```

Generics

- Generics en Kotlin.

Generics en Kotlin

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

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

Copier

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

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

Copier

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'omettre le type des arguments :

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

Copier

Generics en Kotlin

Les types peuvent être génériques, mais les fonctions aussi, dans ce cas-là, le type générique est noté avant le nom de la fonction :

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String { // extension function
    // ...
}
```

Copier

Pour appeler les méthodes génériques, il faut préciser le type après le nom de la méthode :

```
val l = singletonList<Int>(1)
```

Copier

Mais si le type du paramètre peut être inféré, alors il est possible d'omettre le type :

```
val l = singletonList(1)
```

Copier

Autres fonctionnalités

- Casting de types en Kotlin.
- Gestion des exceptions.
- Déclaration de constantes.
- Initialisation tardive.

Autres fonctionnalités

- Casting de types en Kotlin.
- Gestion des exceptions.
- Déclaration de constantes.
- Initialisation tardive.
- Annotation en Kotlin.

Casting de types en Kotlin

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

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

Copier

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 :

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

Copier

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?` :

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

Copier

Casting de types en Kotlin

Vous pouvez utiliser `x as Person?` pour caster un type qui peut être null. La différence entre cette solution et la précédente `as?` est que celle-ci échouera si x est une instance non nulle d'un autre type que `Person` :

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

Copier

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()
```

Copier

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?)
```

Copier

Casting de types en Kotlin

Solution

Copier

```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()
    p = x as Person
    pNull = x as? Person
```

Casting de types en Kotlin

Exercice

Prenons le code en Java, et convertissons-le en Kotlin :

Copier

```java
import java.util.ArrayList;
import java.util.List;

public class Test {

    private static int getDefaultSize(Object object){
        if(object instanceof String) {
            return ((String) object).length();
        } else if(object instanceof List) {
            return ((List) object).size();
        }
        return 0;
    }

    public static void main(String[] args) {

        List list = new ArrayList();
```

Casting de types en Kotlin

Solution

```kotlin
private fun getDefaultSize(anyObject: Any): Int {
    if (anyObject is String) {
        return anyObject.length
    } else if (anyObject is List<*>) {
        return anyObject.size
    }
    return 0
}
```

Copier

Ou encore plus concis :

```kotlin
private fun getDefaultSize(anyObject: Any) = when (anyObject) {
    is String -> anyObject.length
    is List<*> -> anyObject.size
    else -> 0
}
```

Copier

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!")
```

Copier

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 }
```

Copier

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 }
```

Copier

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 ne modifie 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 }
```

Copier

Gestion des exceptions

Solution

Copier

```
1 import java.lang.Integer.parseInt
2 fun main() {
3     var input = "2005"
4     var numValue: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
5     println(numValue)
6     input = "azerty"
7     numValue = try { parseInt(input) } catch (e: NumberFormatException) { null }
8     println(numValue)
9 }
```

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() { ... }
```

Copier

Initialisation tardive

Les valeurs déclarées comme n'acceptant pas de valeur nulle, doivent être initialisées dans le constructeur, toutefois, cela est parfois peu pratique. Par exemple des propriétés qui peuvent être initialisés via une injection de dépendance, ou dans une méthode `setup` dans un test unitaire. Nous ne voulons toutefois pas que la variable puisse avoir une valeur nulle, pour ce faire, nous pouvons utiliser le modifieur `lateinit` :

Copier

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

Initialisation tardive

Ce modifieur peut être utilisé sur des variables de type `var`, déclarées dans le corps de la classe (pas dans le constructeur primaire, et seulement si la propriété n'a pas d'accesseurs sur mesures (custom getter or setter). Accéder à cette propriété avant son initialisation, lève une exception :

```
1 lateinit var allByDefault: String // error: explicit initializer required, default getter and setter implied
2 print(allByDefault) // KO, Exception : kotlin.UninitializedPropertyAccessException: lateinit property allByDefault has not been initialized
```

Copier

Initialisation tardive

Exercice : utilisation de lateinit

Étant donné le code suivant, que ce passe-t-il quand nous l'exécutons, pourquoi, et comment corriger le problème ?

Copier

```
class MyTest() {
    lateinit var subject: String

    fun displaySubject() {
        println(subject)
    }
}

fun main() {
    var myTest = MyTest()
    myTest.displaySubject()
}
```

Initialisation tardive

Solution

Respectons notre contrat : initialisons la variable, comme promis :

```
fun main() {
    var myTest = MyTest()
    myTest.subject = "The best subject"
    myTest.displaySubject()
}
```

Copier

Exercice récapitulatif

Nous allons reprendre et mettre en pratique les notions que nous avons apprises jusqu'à présent. Pour cela, nous allons développer un système de gestion d'utilisateurs.

- 1 - Définition des structures de données.
- 2 - Gestion avancée du numéro de téléphone.
- 3 - Gestion avancée de l'age.
- 4 - Affichage.
- 5 - Affichage liste.
- 6 - Sécurisation du code.

Exercice récapitulatif

- 7 - Gestion des valeurs nulles.
- 8 - La fabrique.
- 9 - Interface.
- 10 - Fold.

Exercice récapitulatif

1 - Définition des structures de données

Dans un premier temps, nous allons déclarer nos structures de données :
Une classe parente `Person` :

- Un champ texte, `firstName` de type chaîne de caractère en lecture/écriture.
- Un champ texte, `lastName` de type chaîne de caractère en lecture.

Exercice récapitulatif

Classes filles

Une classe fille `Friend` qui hérite de `Person` qui contient en plus :

- Un champ de type `Short` : `age` en lecture/écriture.

Une classe fille `Colleague` qui hérite de `Person` qui contient en plus :

- Un champ `role` qui ne peut prendre que les valeurs suivantes : MANAGER, CEO, WORKER.

Une classe fille `Contact` qui hérite de `Person` qui contient en plus :

- Un champ texte, `phoneNumber` de type chaîne de caractère en lecture seule.
- Un champ texte, `email` de type chaîne de caractère en lecture seule.

Exercice récapitulatif

Solution proposée

Copier

```
open class Person(var firstName: String, val lastName: String)

class Friend(firstName: String, lastName: String, var age: Short): Person(firstName, lastName)
class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName)
class Contact(firstName: String, lastName: String, var phoneNumber: String, var email: String): Person(firstName, lastName)

enum class Role {
    MANAGER, CEO, WORKER
}
```

Exercice récapitulatif

2 - Gestion avancée du numéro de téléphone

Ajoutons un peu plus de détails pour la gestion du numéro de téléphone : il sera composé maintenant deux champs :

- Le `type` du numéro, qui ne peut prendre que les valeurs suivantes : MOBILE, FIX, FAX.
- Un champ texte, `value` de type chaîne de caractère, avec une valeur initiale de "0000000000", qui ne peut prendre comme valeur qu'une chaîne de caractère, composée de 10 chiffres (utilisation de vetoable).

Exercice récapitulatif

Solution proposée

Copier

```
class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, var email: String): Person(firstName, lastName)

enum class Role {
    MANAGER, CEO, WORKER
}

class PhoneNumber(type: PhoneType, value: String = "0000000000") {
    var value: String by Delegates.vetoable(value) { property, oldValue, newValue ->
        newValue.length == 10 && isNumber(newValue)
    }
}

fun isNumber(s: String?): Boolean {
    return if (s.isNullOrEmpty()) false else s.all { Character.isDigit(it) }
}

enum class PhoneType {
```

Exercice récapitulatif

3 - Gestion avancée de l'age

Modifions notre classe `Friend` afin de ne pouvoir définir un age qu'avec une valeur valide :

- Un nombre positif.
- Inférieur à une constante `MAX_AGE` définie avec la valeur : 150.

Note, pour la gestion de la valeur, nous utiliserons un interval.

Exercice récapitulatif

Solution

Copier

```
5  class Friend(firstName: String, lastName: String, age: Short): Person(firstName, lastName) {
6      val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
7          newValue in 0..MAX_AGE
8      }
9  }
10
11 class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName)
12 class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, varemail: String): Person(firstName, lastName)
13
14 enum class Role {
15     MANAGER, CEO, WORKER
16 }
17
18 class PhoneNumber(type: PhoneType, value: String) {
19     var value: String by Delegates.vetoable("0000000000") { property, oldValue, newValue ->
20         newValue.length == 10 && isNumber(newValue)
21     }
```

Exercice récapitulatif

4 - Affichage

Afin d'afficher nos objets, redéfinissons l'affichage pour les 3 classes filles.
Pour cela :

- Ajoutons une méthode `toString` abstraite à notre classe `Person`.
- Implémentons la méthode dans les classes filles.

Au lieu d'afficher l'age, nous allons afficher le stade de la vie, correspondant à l'age :

- Bébé : de la naissance à 2 ans.
- Enfant : de 2 ans à 10 ans.
- Adolescent : de 10 ans à 18 ans.
- Adulte : de 18 ans à 70 ans.
- Personne âgée : à partir de 70 ans.

Exercice récapitulatif

Solution proposée

Copier

```
3  abstract class Person(var firstName: String, val lastName: String) {
4      abstract override fun toString(): String
5  }
6
7  class Friend(firstName: String, lastName: String, age: Short): Person(firstName, lastName) {
8      val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
9          newValue in 0..MAX_AGE
10     }
11     override fun toString() = "$firstName $lastName ${age.lifeStep()}"
12 }
13
14 class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName) {
15     override fun toString() = "$firstName $lastName $role"
16 }
17
18 class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, var email: String):
19     Person(firstName, lastName) {
```

Exercice récapitulatif

5 - Affichage liste

- Définissons une liste qui contiendra plusieurs `Person`.
- Affichons cette liste.
- Préfixons l'affichage de l'objet, par son type ("Amis, Collègue ou Contact") (Nous allons définir une méthode `fun getType(person:Person)` qui retournera le type en `String` de l'objet passé en paramètre.
- Préfixons l'affichage de l'objet par sa position dans la liste.

Exercice récapitulatif

Solution proposée

Copier

```
53 fun getType(person:Person) = when (person) {
54     is Friend -> "Amis"
55     is Colleague -> "Collègue"
56     is Contact -> "Contact"
57     else -> "Inconnu"
58 }
59
60 fun main() {
61
62     val person: Person = Friend("Tristan", "SALAUN", 42.toShort())
63     val person2: Person = Friend("Melody", "SALAUN", 25.toShort())
64     val person3: Person = Colleague("Nicolas", "DUPOND", Role.MANAGER)
65     val person4: Person = Colleague("Jean-Christophe", "ANDRE", Role.WORKER)
66     val person5: Person = Contact("John", "DOE", PhoneNumber(PhoneType.MOBILE, "0612345678"), "john.doe@test.com")
67
68     val personList = listOf(person, person2, person3, person4, person5)
69
```

Exercice récapitulatif

6 - Sécurisation du code

Nous remarquons que dans la méthode `getType`, qui nous permet d'afficher le type d'objet en français, nous avons dû utiliser le `else`. Si nous avons besoin d'implémenter une nouvelle classe fille, il est fort probable, que nous oublions d'ajouter ce cas dans cette méthode. Pour fiabiliser le code, nous allons mettre en place une classe scellée pour la classe `Person`.

Exercice récapitulatif

Solution proposée

Copier

```
3  sealed class Person(var firstName: String, val lastName: String) {
4      abstract override fun toString(): String
5  }
6
7  class Friend(firstName: String, lastName: String, age: Short): Person(firstName, lastName) {
8      val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
9          newValue in 0..MAX_AGE
10     }
11     override fun toString() = "$firstName $lastName ${age.lifeStep()}"
12 }
13
14 class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName) {
15     override fun toString() = "$firstName $lastName $role"
16 }
17
18 class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, var email: String):
19     Person(firstName, lastName) {
```

Exercice récapitulatif

7 - Gestion des valeurs nulles

Ajoutons une classe `Address` qui sera une propriété optionnelle de la classe `Friend`, avec les propriétés suivantes :

- Un champ texte `line1` de type chaîne de caractère, obligatoire.
- Un champ texte `line2` de type chaîne de caractère, qui peut prendre une valeur nulle.
- Un champ texte `cp` de type chaîne de caractère, au format similaire au numéro de téléphone : longueur de 5, ne contenant que des chiffres.
- Un champ texte `city` de type chaîne de caractère, obligatoire.
- Un champ texte `country` de type chaîne de caractère, prenant la valeur "France", par défaut.

Exercice récapitulatif

Solution proposée

Copier

```
7  class Friend(firstName: String, lastName: String, age: Short, var address: Address? = null): Person(firstName, lastName) {
8      val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
9          newValue in 0..MAX_AGE
10     }
11     override fun toString() = "$firstName $lastName ${age.lifeStep()}${if (address != null) "\n$address" else ""}"
12 }
13
14 class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName) {
15     override fun toString() = "$firstName $lastName $role"
16 }
17
18 class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, var email: String):
19     Person(firstName, lastName) {
20     override fun toString() = "$firstName $lastName $phoneNumber $email"
21 }
22
23 enum class Role {
```

8 - La fabrique

Exercice récapitulatif

Afin de créer des instances de personnes, plus facilement, nous allons mettre en place une fabrique :

- Nous allons ajouter une méthode "statique" à notre classe `Person`, qui retourne une instance d'une des classes fille.
- Nous allons passer le constructeur des classes filles en private.
- Nous allons ajouter une méthode statique, qui retourne une instance de la classe fille dans chaque classe fille.
- Nous allons utiliser une classe de génération de valeurs aléatoires pour générer des prénoms, noms, ...

Copier

```
import kotlin.random.Random

class RandomValues {
    companion object {
        val firstNameList = listOf("Jean", "Pierre", "Clément")
        val lastNameList = listOf("DUPOND", "DUPONT", "MARTIN")

        fun getFirstName(): String {
            return firstNameList[Random.nextInt(0, firstNameList.size)]
        }

        fun getLastName(): String {
            return lastNameList[Random.nextInt(0, lastNameList.size)]
        }

        fun getRole(): Role {
            return Role.values()[Random.nextInt(0, Role.values().size)]
```

Exercice récapitulatif

Solution proposée

Copier

```
    companion object {
        fun getPerson(): Person {
            val typeId = Random.nextInt(0, 3)
            return when (typeId) {
                0 -> Friend.getFriend()
                1 -> Colleague.geColleague()
                2 -> Contact.getContact()
                else -> Friend.getFriend()
            }
        }
    }
}

class Friend private constructor(firstName: String, lastName: String, age: Short, var address: Address? = null):
    Person(firstName, lastName) {
    val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
        newValue in 0..MAX_AGE
```

Exercice récapitulatif

9 - Interface

Ajoutons une interface `JsonExport` qui comporte une méthode `toJson` qui va nous permettre d'exporter nos contacts en JSON.

Copier

```
interface JsonExport {
    fun toJson(): String
}

sealed class Person(var firstName: String, val lastName: String): JsonExport {
....
}
```

Exercice récapitulatif

Solution proposée

Copier

```
interface JsonExport {
    fun toJson(): String
}

sealed class Person(var firstName: String, val lastName: String): JsonExport {
    abstract override fun toString(): String

    companion object {
        fun getPerson(): Person {
            val typeId = Random.nextInt(0, 3)
            return when (typeId) {
                0 -> Friend.getFriend()
                1 -> Colleague.geColleague()
                2 -> Contact.getContact()
                else -> Friend.getFriend()
            }
        }
```

Exercice récapitulatif

10 - Fold

Nous allons maintenant utiliser la méthode `fold`, pour obtenir des statistiques sur notre liste d'éléments :

- La taille moyenne du prénom (en nombre de lettres).
- L'age cumulé des `Friend`.

Exercice récapitulatif

Solution proposée

Copier

```
val firstNameCumulatedSize = randomPersonList.fold(0) { acc, person ->
    acc + person.firstName.length
}
println("$firstNameCumulatedSize / ${randomPersonList.size} = ${firstNameCumulatedSize.toFloat() / randomPersonList.size.toFloat()}")

val ageCumulated = randomPersonList.fold(0) { acc, person ->
    if(person is Friend) {
        acc + person.age
    } else {
        acc
    }
}
println("$ageCumulated")
```

Interopérabilité

- Interopérabilité avec Java.
- De Kotlin au Java.
- Nulls de Java.
- Le Kotlin dans Java.
- 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 :

Copier

```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)
    }
    print(list)

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

Copier

```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 nommage 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 :

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY      // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) {                         // call isLenient()
        calendar.isLenient = true                      // call setLenient()
    }
}
```

Copier

```
fun main() {
    calendarDemo()
}
```

Copier

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 }
```

Copier

Interopérabilité avec Java

Utilisation en Kotlin

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

Copier

```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 :

Copier

```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
```

Copier

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'extension, sont compilées dans des méthodes statiques d'une classe nommée `org.example.AppKt`. Exemple :

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

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

Copier

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

public class Test {

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

Copier

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`

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;
    }
}
```

Copier

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

Copier

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

Nulls de Java

Solution

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

Copier

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` :

Copier

```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 :

Copier

```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")
    }
}
```

Copier

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

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

Copier

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` :

```
object Singleton {

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

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

Copier

Ce qui donnera l'appel en Java :

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

Copier

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)
}
```

Copier

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

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

Copier

Java Réflexion avec Kotlin

Exercice

Reprenons notre classe Java `Customer` :

```java
public class Customer {

    private String firstName;
    private String lastName;
    private int age;

    //standard setters and getters
}
```

Copier

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 nom du constructeur.

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

Java Réflexion avec Kotlin

Solution

```
1 fun main() {
2     val type = Customer::class.java
3     val constructors = type.constructors
4 
5     println(constructors.size)
6     println(constructors[0].name)
7 }
```

Copier

Note, la méthode ci-dessous nécessite une librairie Kotlin

```
1 Customer::class.constructors.forEach { println(it) }
```

Copier

Kotlin Réflexion

reflexion Java en Kotlin

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

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

Copier

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 :

```
data class ExampleDataClass(
    val name: String, var enabled: Boolean)

fun main() {
    ExampleDataClass::class.java.methods.forEach(::println)
}
```

Copier

Standard Library

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

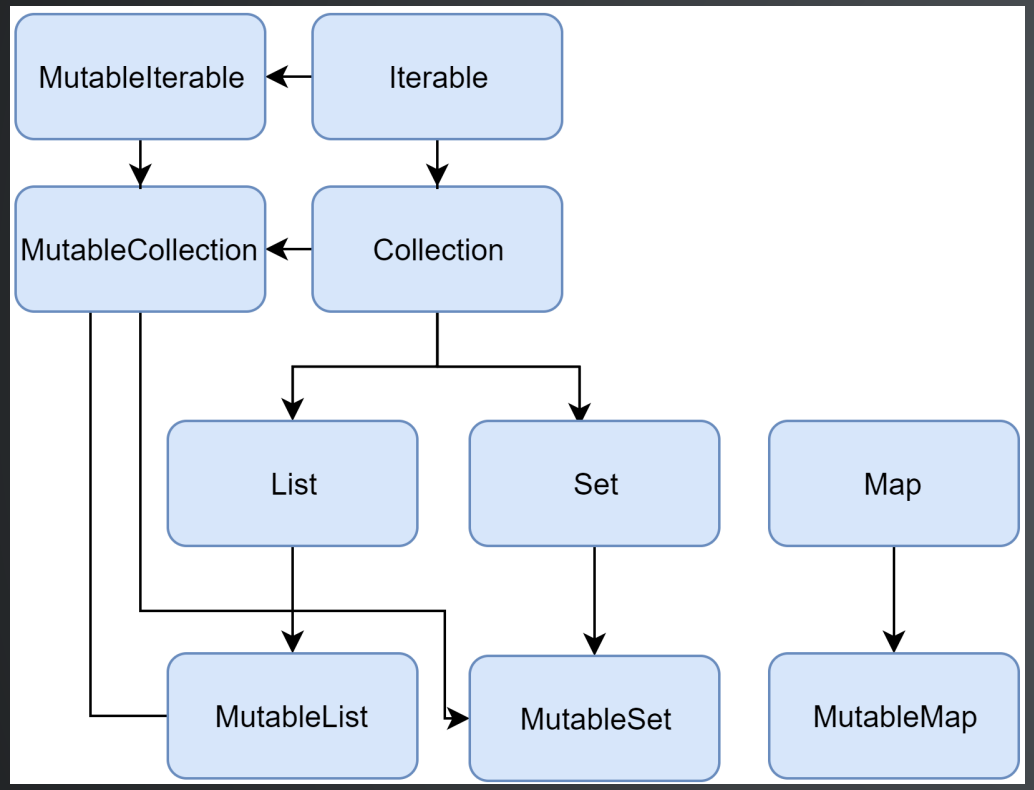
Kotlin Standard Library et collections dans Kotlin

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 chaînes de caractères et les caractères.
- Des extensions pour les classes du JDK pour que cela soit plus pratique de travailler avec des fichiers, les Entrées/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 définit 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`.

Copier

```
fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

Kotlin Standard Library et collections dans Kotlin

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

Copier

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

Filtering, Mapping et Flatmapping en Kotlin

Entraînement en ligne

Plutôt que de réinventer la roue, je vous propose de vous entraîner sur les collections directement en ligne Filter et FlatMap.

Programmation asynchrone

- Le problème de la programmation asynchrone.
- Coroutines en Kotlin et l'implémentation des coroutines.

Programmation asynchrone

- Le problème de la programmation asynchrone.
- Coroutines en Kotlin et l'implémentation des coroutines.
- Async et Await en Kotlin.

Le problème de la programmation asynchrone

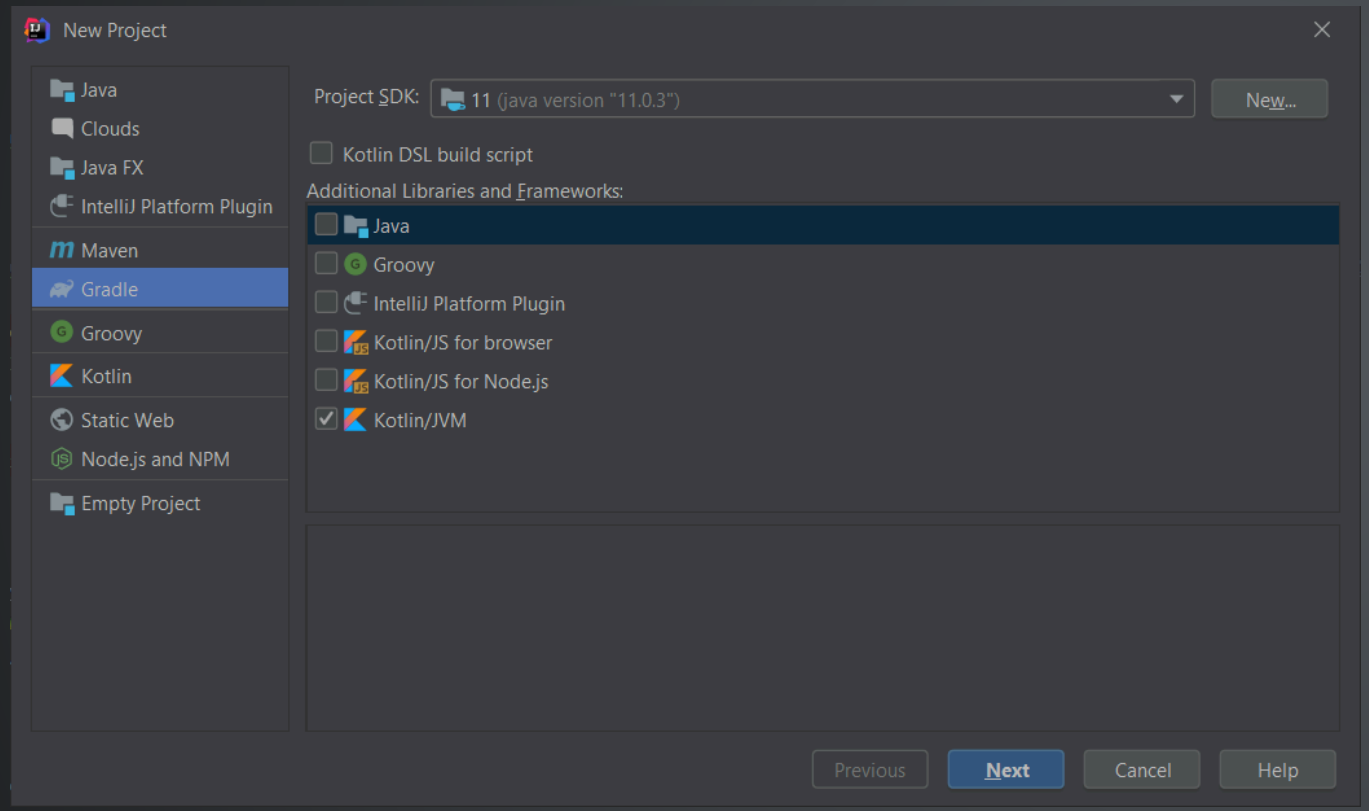
Depuis des décennies, les développeurs sont confrontés à un problème à résoudre : comment faire pour que les applications ne se bloquent pas. Que l'on développe pour un ordinateur, un mobile ou un serveur, nous voulons éviter que l'utilisateur attende, ou encore pire, des goulots d'étranglements qui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

Plusieurs approches sont possibles pour résoudre ce problème :

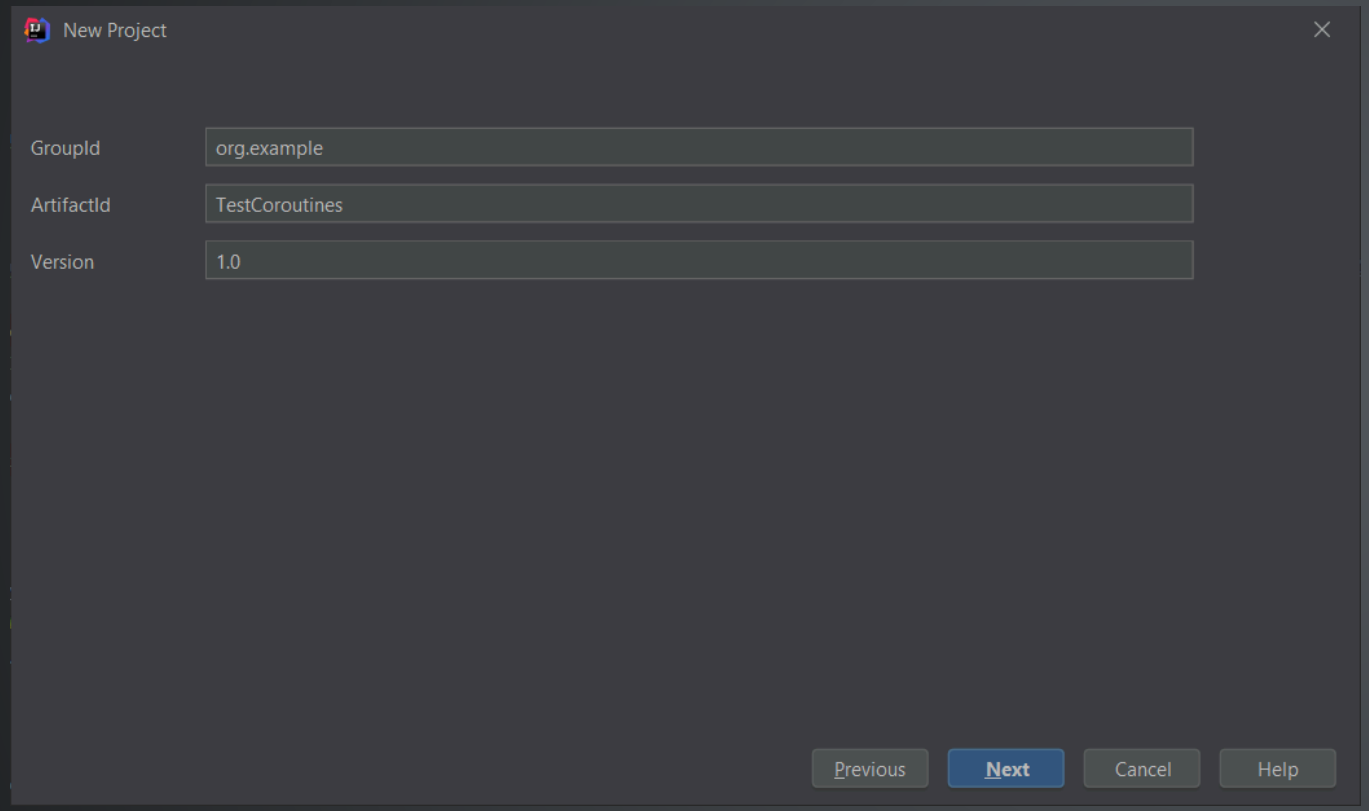
- Traitements en parallèle (Threading).
- Les "callbacks".
- Futures, Promises et al.
- Les extensions réactives.
- Coroutines.

Mise en place du projet

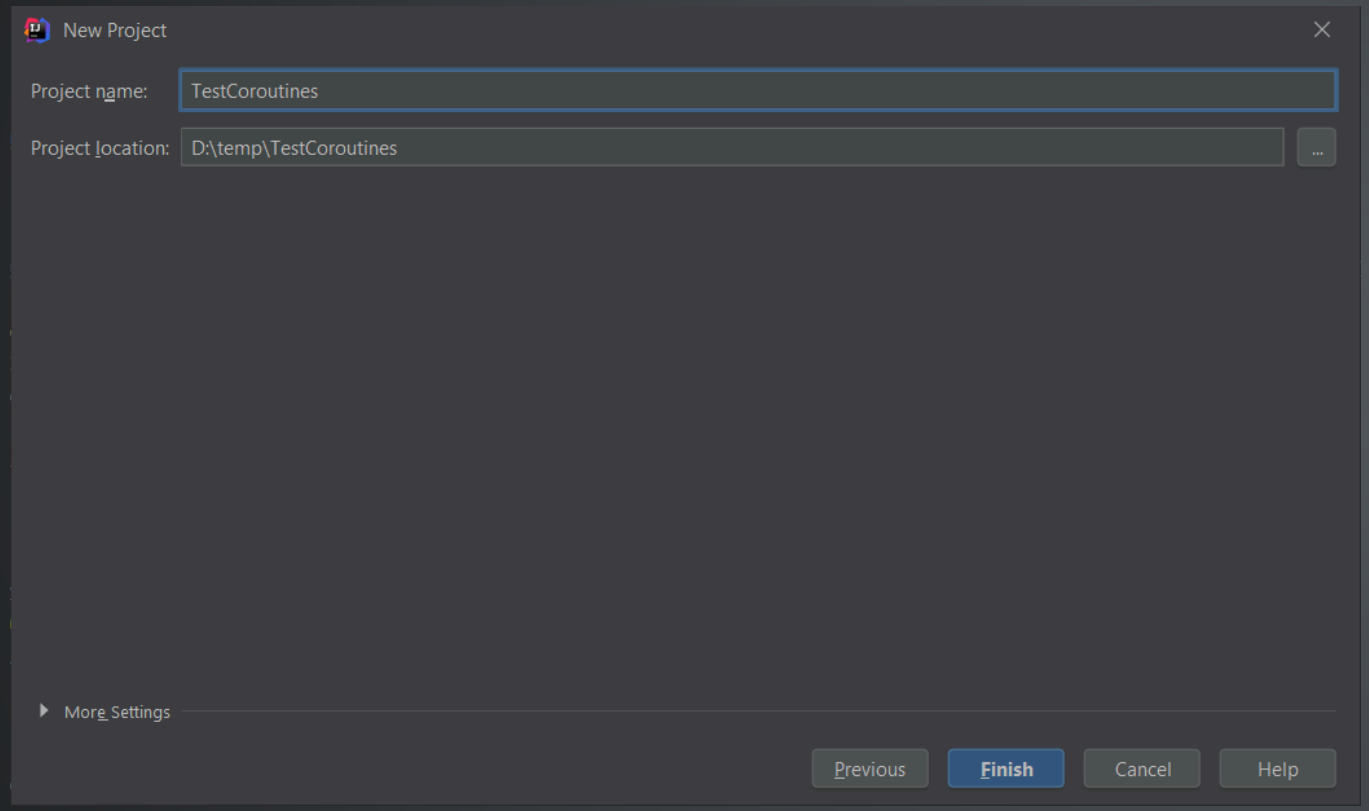
Créez un nouveau projet : **File / New / Project**. Sélectionner **Gradle / Kotlin / JVM**.



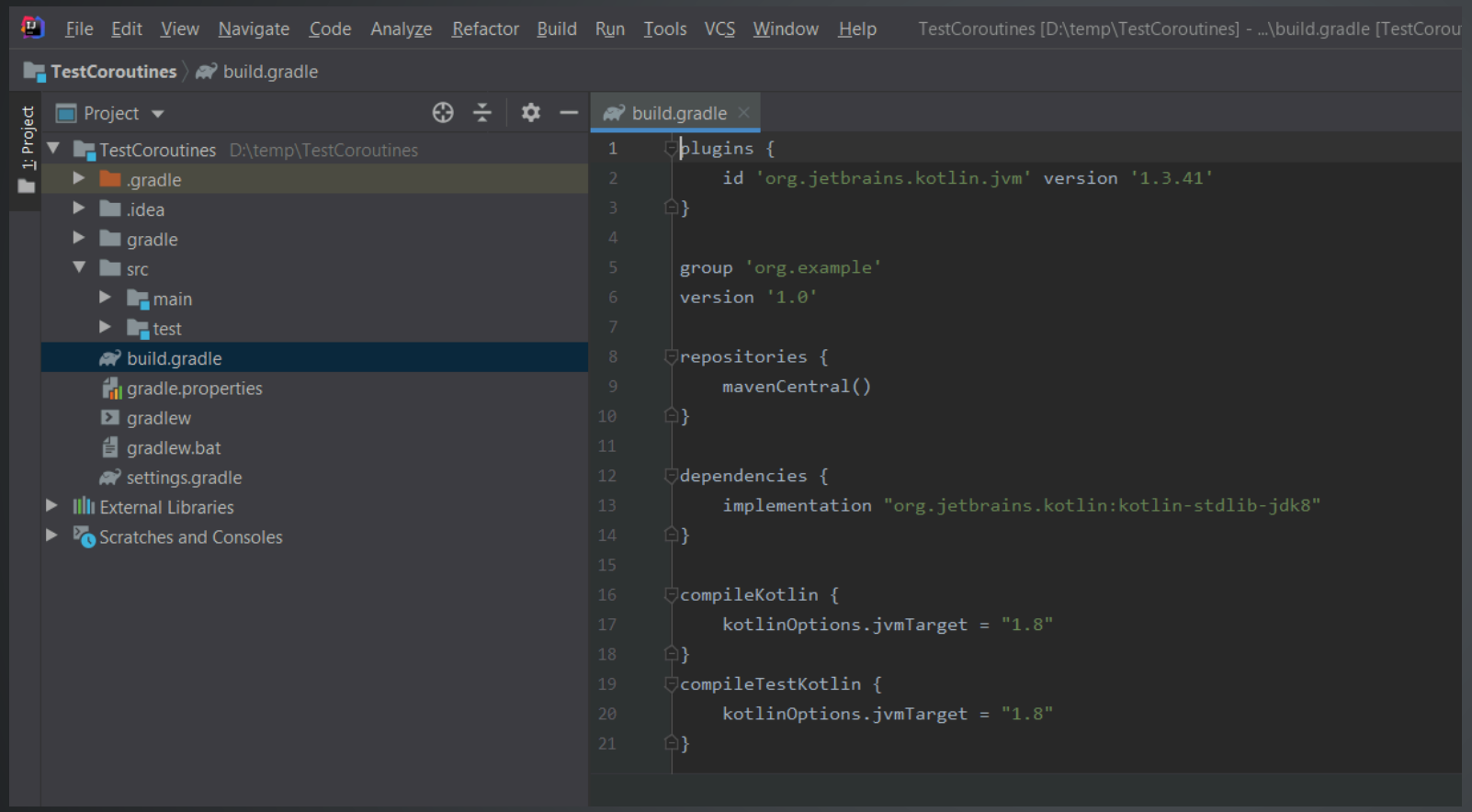
Nommage du module



Nommage du projet



Ouvrez le fichier `build.gradle`



Coroutines en Kotlin

Ajoutez la dépendance

Copier

```
dependencies {
    ...
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.1")
}
```

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 ensembles. 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 threads classiques.

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

```
launch {
//    ...
}
```

Copier

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")
```

Copier

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 thread disponible.

Q1 - Que se passe-t-il si l'on supprime la ligne `Thread.sleep(2000)` ?

Q2 - Que se passe-t-il si l'on remplace `Thread.sleep(2000)` par `delay(2000)` ?

Coroutines en Kotlin

Lint

Nous avons une note qui nous informe que nous n'avons pas le droit d'utiliser les coroutines de la sorte. Pour retirer le "warning" nous pouvons ajouter `@OptIn(DelicateCoroutinesApi::class)` :

```
1 @OptIn(DelicateCoroutinesApi::class)
2 fun main() {
3 ...
```

Copier

Blocage du thread principal

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

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

Copier

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

```

Copier

Coroutines en Kotlin

Lançons beaucoup d'opérations

Nous allons lancer 1 000 000 d'opérations
Commençons avec une simple boucle :

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

Copier

Que se passerait-il si l'opération prenait 1 seconde ?
Par exemple en mettant en place une pause avec `Thread.sleep(1000)`
Cela prendrait environ 11 jours, 13 heures, 46 minutes et 40 secondes.

Coroutines en Kotlin

Lançons beaucoup de thread

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

```
val c = AtomicLong()

for (i in 1..1_000_000L)
    thread(start = true) {
        c.addAndGet(i)
    }

println(c.get())
```

Copier

Que remarquez-vous, concernant la charge de la machine ?

Coroutines en Kotlin

Lançons beaucoup de thread

Écrivez le même code avec des coroutines :

```
val c = AtomicLong()

for (i in 1..1_000_000L)
    GlobalScope.launch {
        c.addAndGet(i)
    }

println(c.get())
```

Copier

Que constate-t-on ?
Si l'on ajoute une pause (en Coroutine on utilise `delay(1000)`), que ce passe-t-il ? Et pourquoi ?
Les coroutines n'ont pas terminé, quand la fonction `main` se termine.

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

```
val deferred = (1L..1_000_000L).map { n ->
    GlobalScope.async {
        n
    }
}
val sum = deferred.sumBy { it.await() }
println("Sum: $sum")
```

Copier

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 {}`

```
val deferred = (1L..1_000_000L).map { n ->
    GlobalScope.async {
        n
    }
}
runBlocking {
    val sum: Long = deferred.sumOf { it.await() }
    println("Sum: $sum")
}
```

Copier

Nous devrions obtenir le résultat suivant :

```
Sum: 1784293664
```

Copier

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.sumOf { it.await() }
    println("Sum: $sum")
}
```

Copier

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: Long): Long {
2     delay(1000)
3     return n
4 }
```

Copier

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

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

Copier

Async et Await en Kotlin

Code final

Nous obtenons le code final, suivant :

Copier

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.async
import kotlinx.coroutines.runBlocking

fun main() {

    suspend fun workload(n: Long): Long {
        //delay(3000)
        return n
    }

    val deferred = (1..1_000_000).map { n ->
        GlobalScope.async {
            workload(n)
        }
    }
    runBlocking {
```

Async et Await en Kotlin

Code final version bis

Nous obtenons le code final, suivant :

Copier

```kotlin
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.async
import kotlinx.coroutines.runBlocking

fun main() {

    val deferred: List<Deferred<Long>> = (1..1_000_000L).map { n ->
        GlobalScope.async { n }
    }

    runBlocking {
        val sum = deferred.sumOf { it.await() }
        println("Sum: $sum")
    }
}
```