

Formations

Tristan SALAÛN

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 de A à Z : les différents aspects du langage

[Le support en PDF.](#)

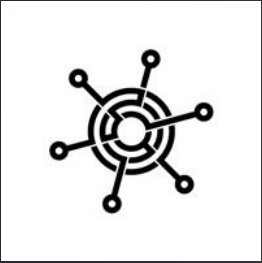
Présentations

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

Qui suis-je

Présentations

- 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 % interoperables.
- 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 ?

Dates importantes

- 2010 : Lancement du projet Kotlin.
- juillet 2011 : communication officielle de JetBrains sur le projet Kotlin.
- février 2012 : mise à disposition du projet en open source.
- 15 février 2016 : Kotlin 1.0. Première version stable.
- 28 novembre 2017 : Kotlin 1.2. Partage du code entre la JVM et la plate-forme Javascript.
- 29 octobre 2018 : Kotlin 1.3. Coroutines.

Pourquoi le Kotlin ?

Dates importantes (suite)

- 7 mai 2019 : Google annonce que Kotlin est le langage préféré pour développer des applications Android.
- 17 août 2020 : Kotlin 1.4. Mix des paramètres nommés et non nommés, virgule de fin, ...
- 5 mai 2021 : Kotlin 1.5. @JvmRecord, sealed interface, @JvmInline value class, ...
- 16 novembre 2021 : Kotlin 1.6. when exhaustif, ...
- 9 juin 2022 : Kotlin 1.7. Nouveau compilateur (K2 en Alpha), opérateur _ pour les types, ...
- 28 décembre 2022 : Kotlin 1.8. Meilleur support Swift et Objective-C, ajout `copyToRecursively()` et `deleteRecursively()`, ...
- 25 avril 2023 : Dernière version de Kotlin : 1.8.21.

Pourquoi le Kotlin ?

Inférence de type

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

[Copier](#)

Pourquoi le Kotlin ?

Avantages

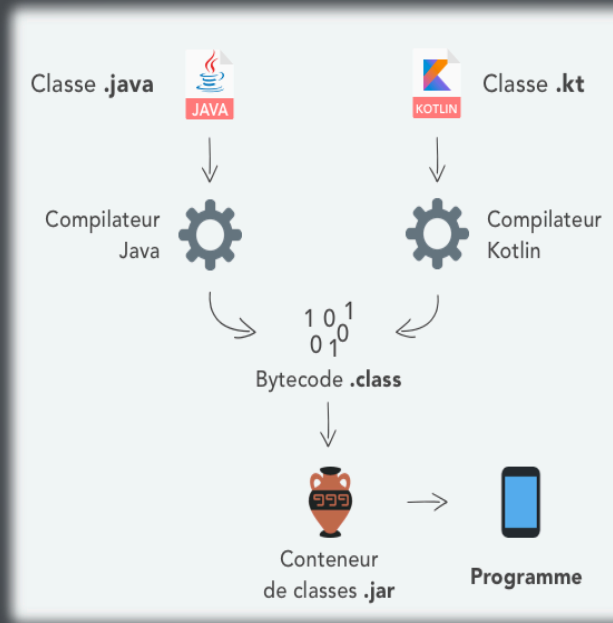
- Performance.
- Fiabilité.
- Efficacité.

Pourquoi le Kotlin ?

Langage fonctionnel

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

Introduction à la JVM (Java Virtual Machine)



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 :

```
1 40+2
2 "Hello, World!"
3 println("Hello, World!")
4 1
5 1.0
```

[Copier](#)

Lancer notre premier script

En ligne

Nous pouvons utiliser le compilateur en ligne à l'adresse :

<https://play.kotlinlang.org>.

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

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

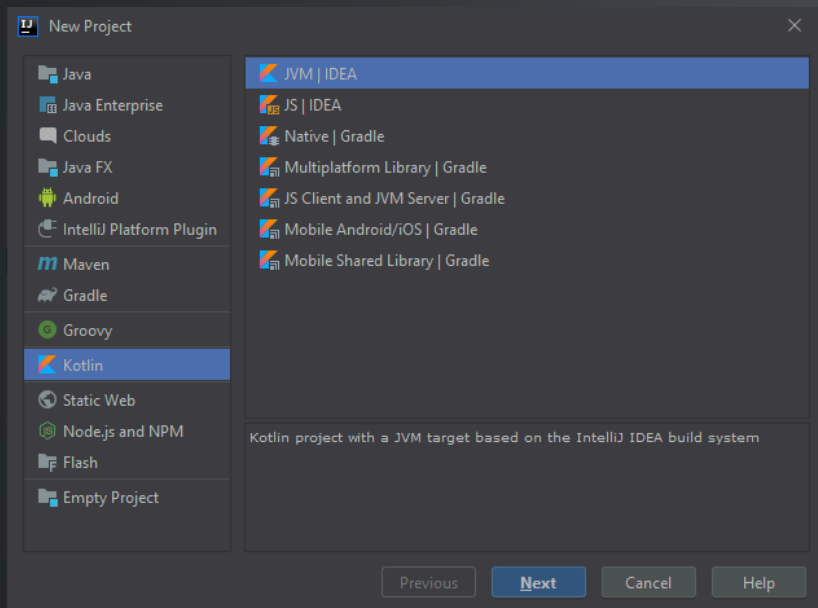
Installation

Commençons par installer IntelliJ : [Télécharger](#).
Optons pour la version **Community**.

Kotlin et IntelliJ IDEA

Création du projet

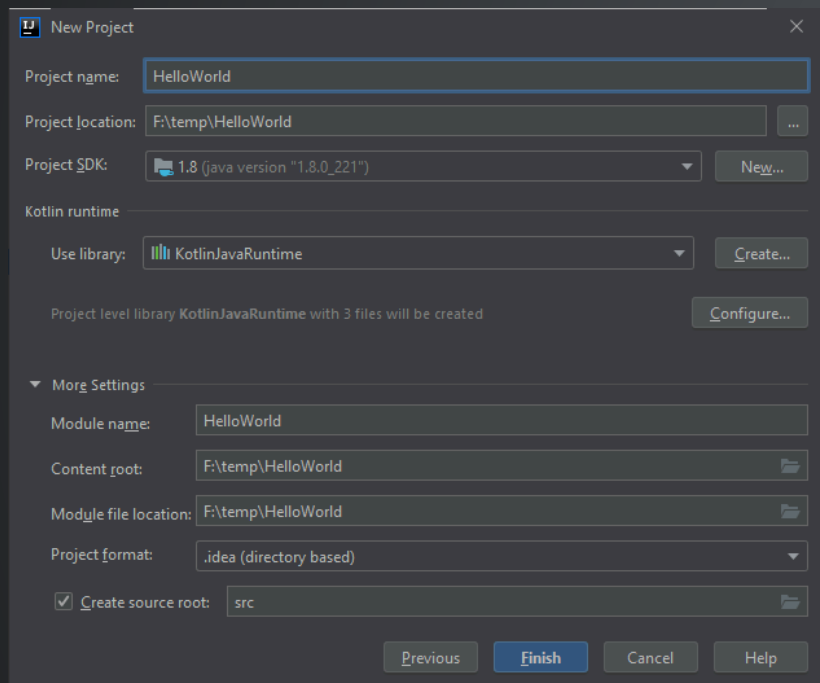
Il est temps de créer notre première application : **File / New / Project**. Sélectionner **Kotlin / JVM | IDEA**.
[Nouvelle version ici.](#)



Kotlin et IntelliJ IDEA

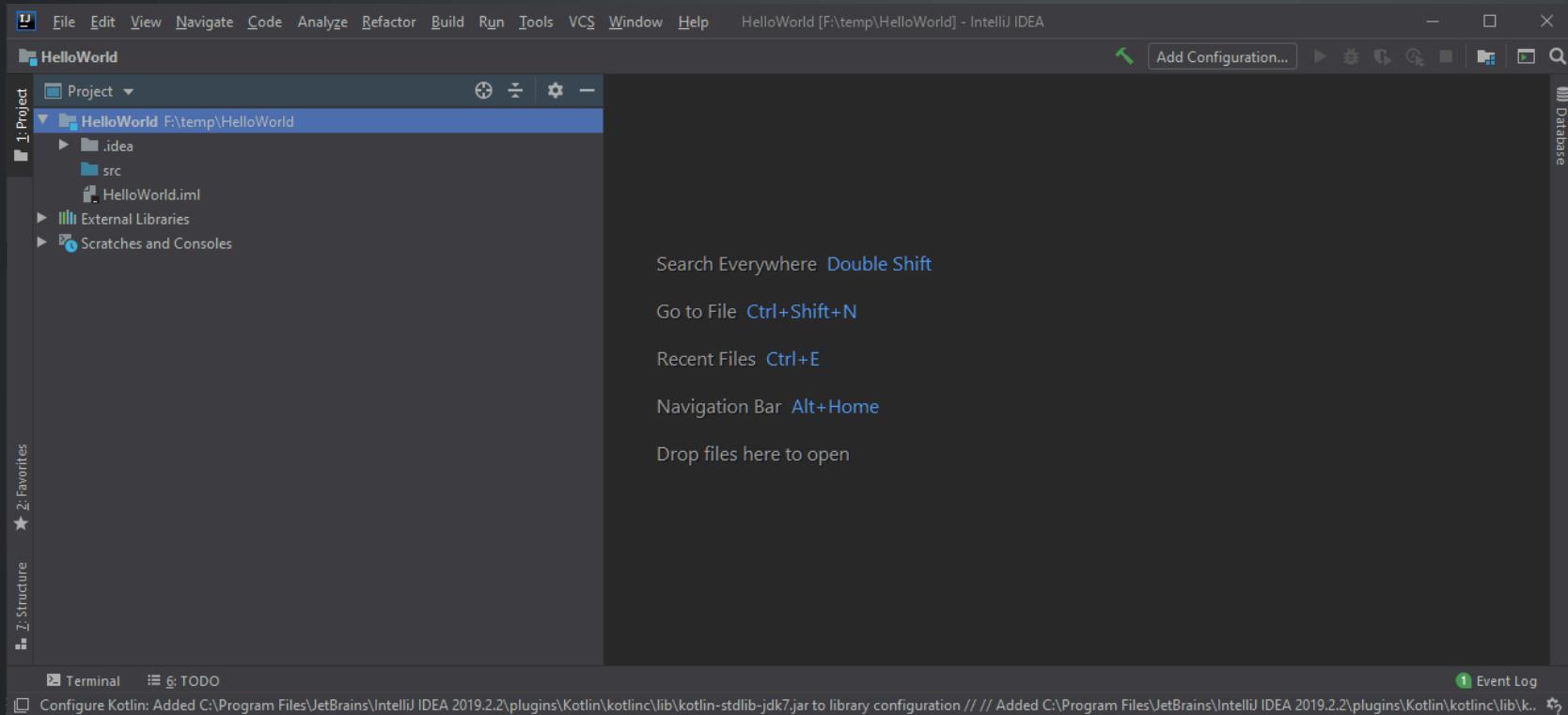
Nommage

Nommons notre projet, par exemple HelloWorld :



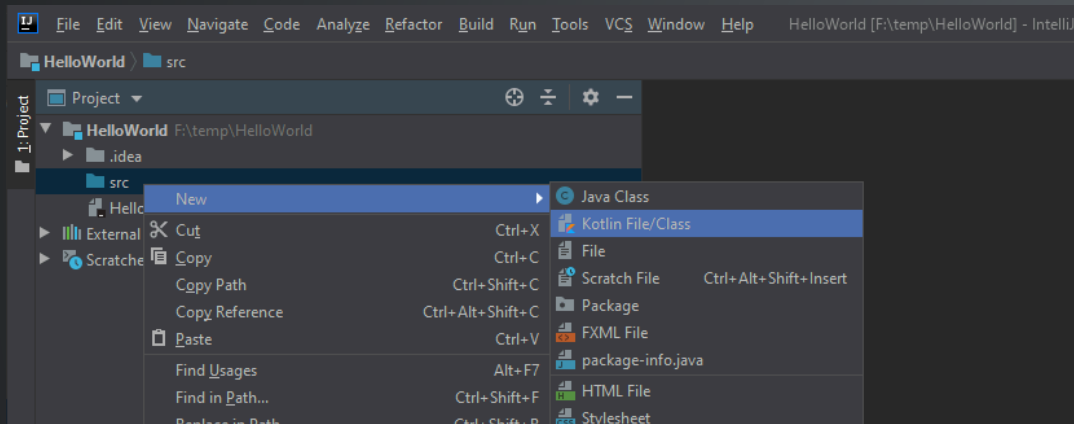
Kotlin et IntelliJ IDEA

Nous devrions obtenir le résultat suivant :



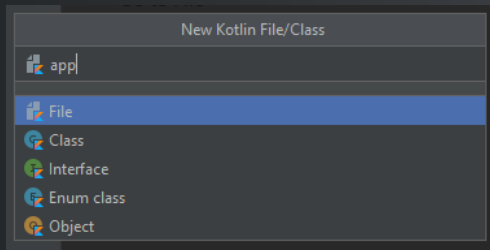
Kotlin et IntelliJ IDEA

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.

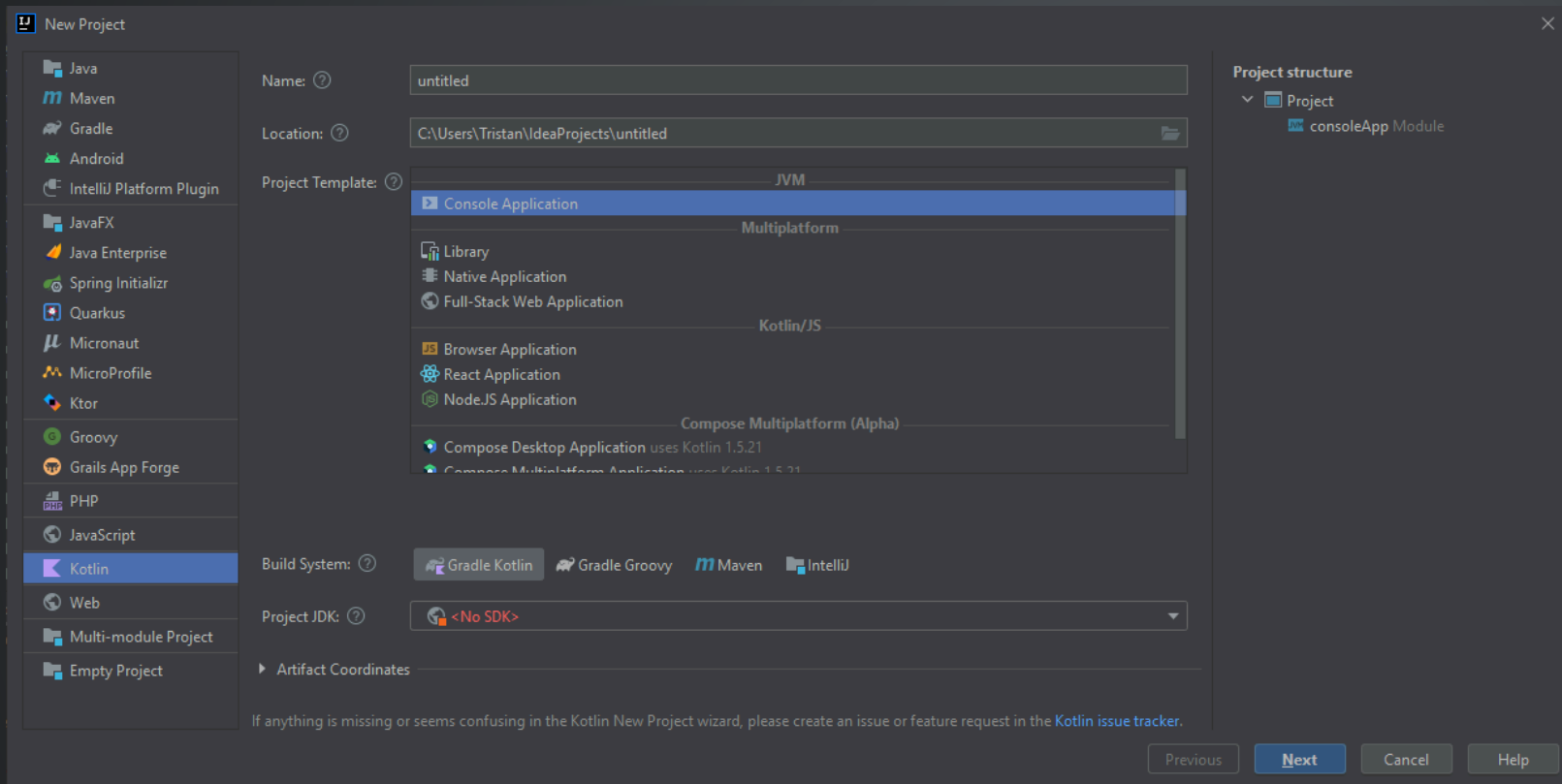
Il reste juste à compléter le corps de la méthode pour obtenir le résultat suivant :

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

Copier

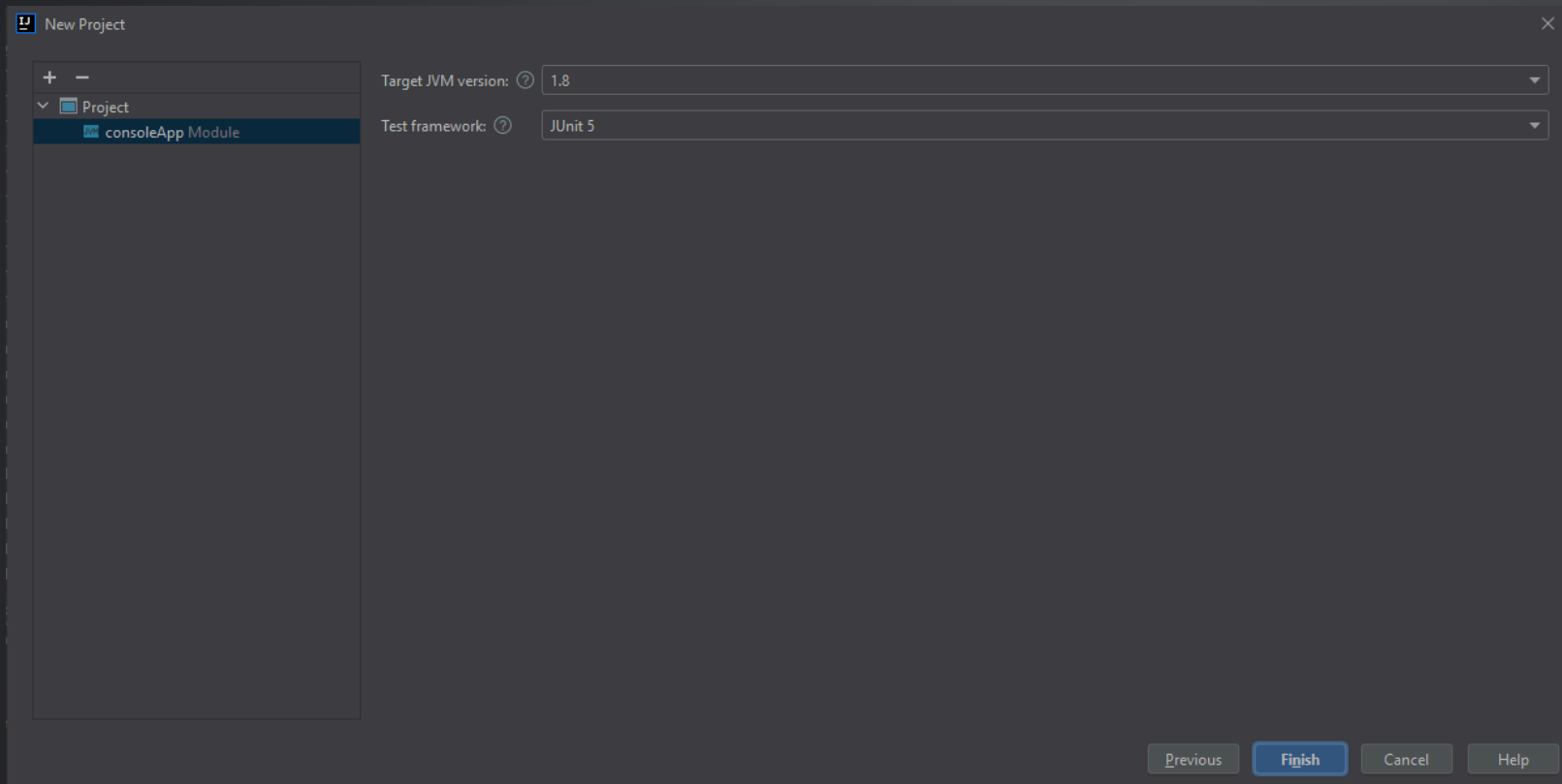
Kotlin et IntelliJ IDEA

Nommons notre projet, par exemple HelloWorld :



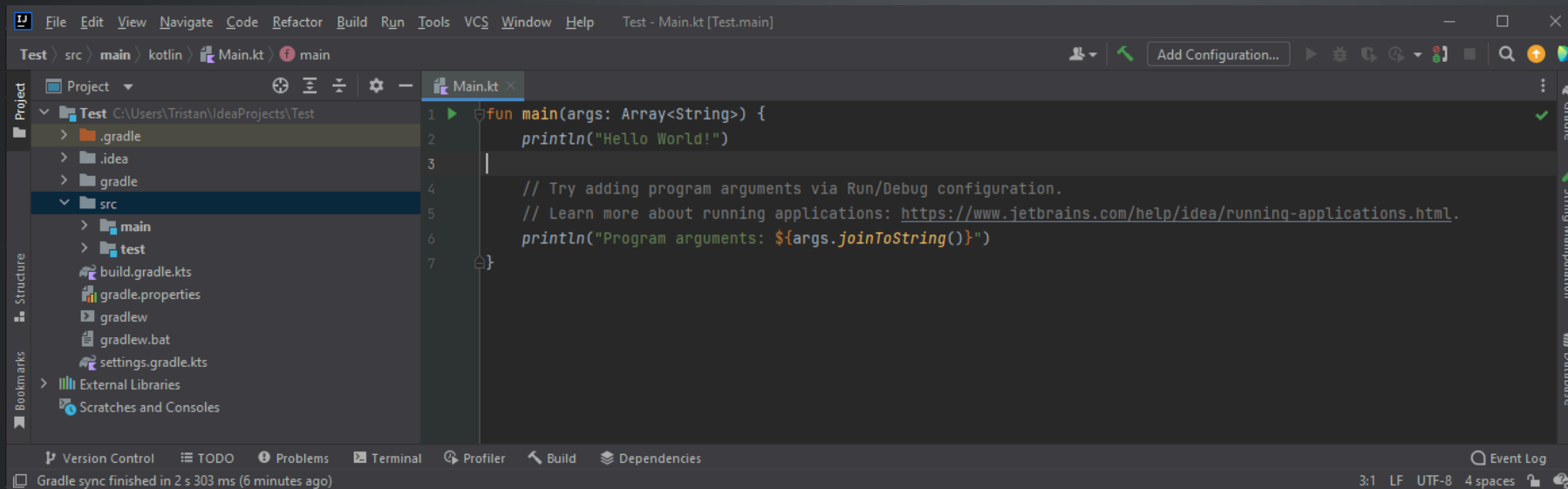
Kotlin et IntelliJ IDEA

Gardons les paramètres par défaut :



Kotlin et IntelliJ IDEA

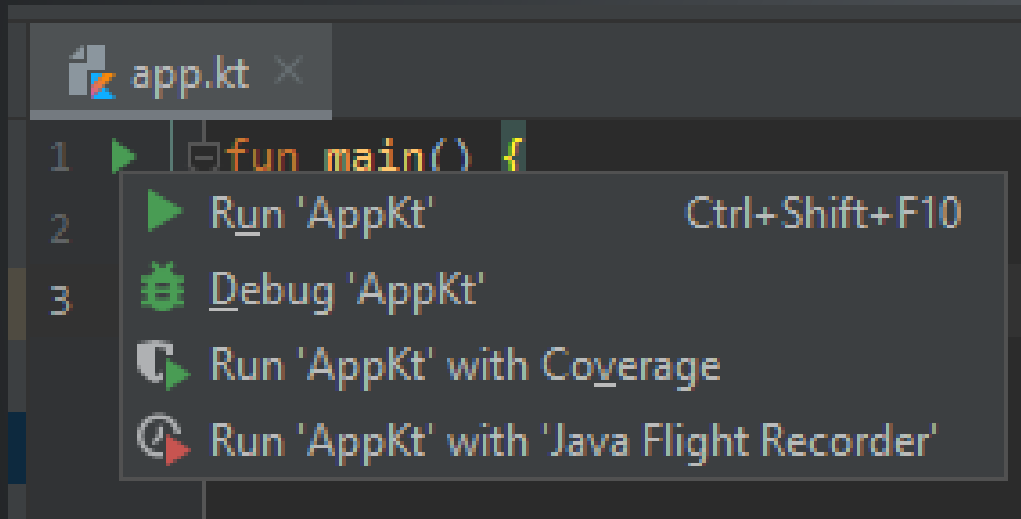
Nous obtenons :



Kotlin et IntelliJ IDEA

Exécuter le code

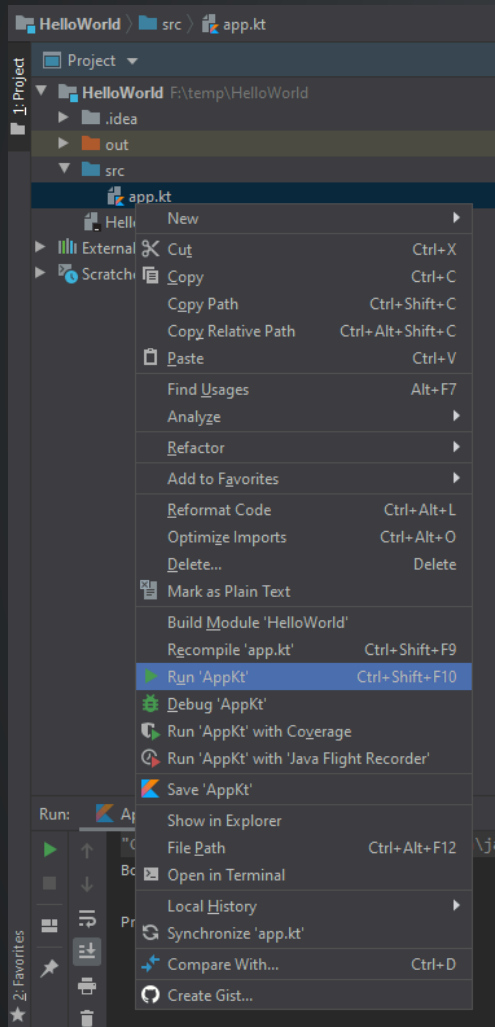
Il y a plusieurs façons de lancer le code, la plus rapide est de cliquer sur le bouton vert **Run** :



Exécuter le code

Ou encore :

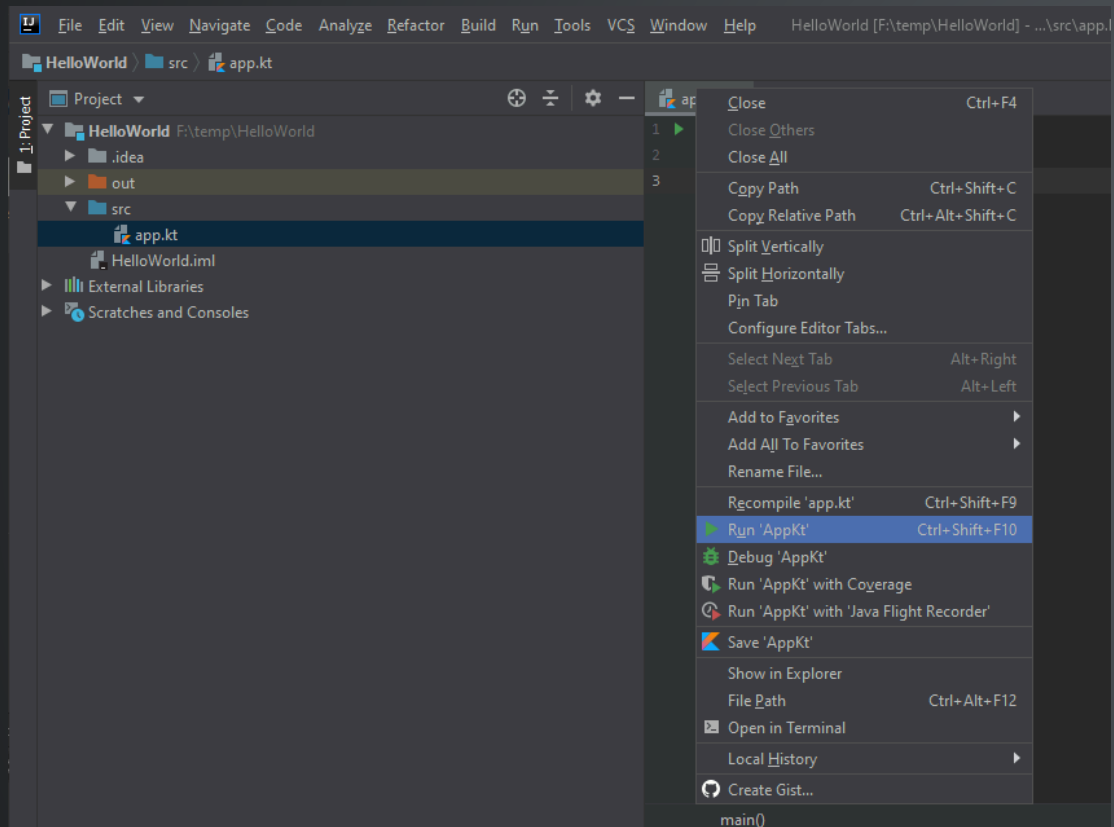
Kotlin et IntelliJ IDEA



Exécuter le code

Ou bien :

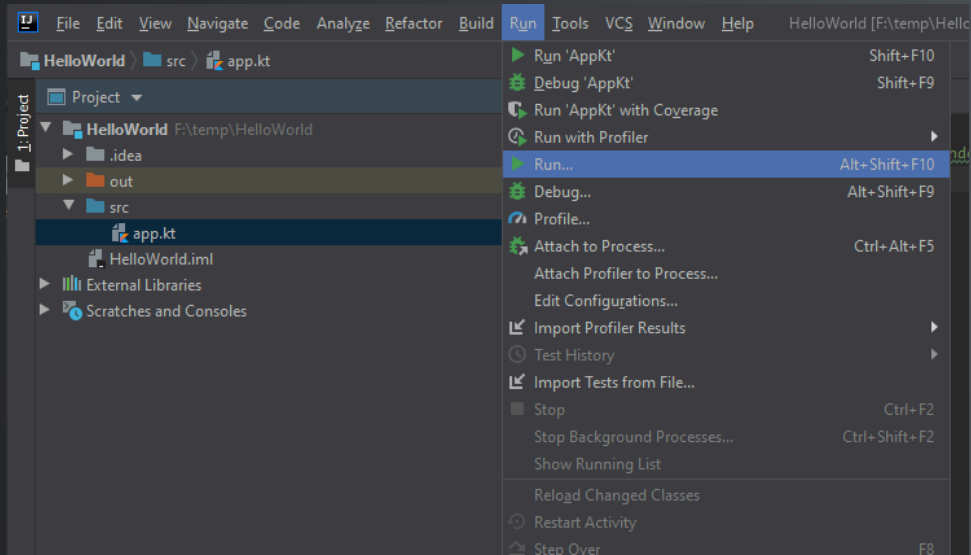
Kotlin et IntelliJ IDEA



Kotlin et IntelliJ IDEA

Exécuter le code

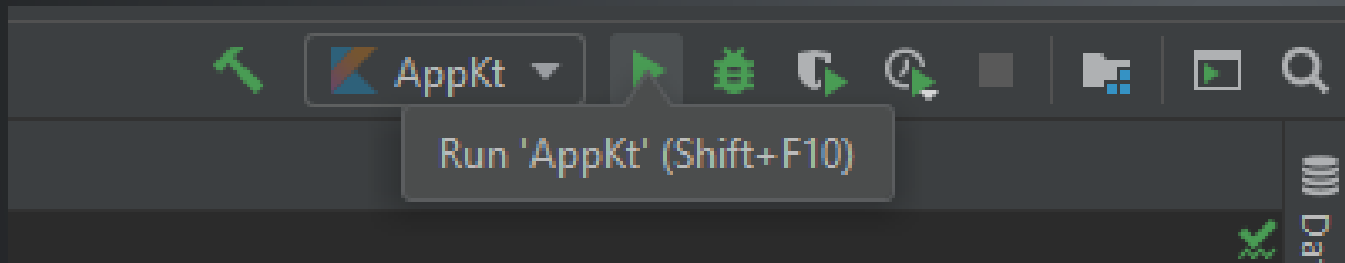
Ou bien encore :



Kotlin et IntelliJ IDEA

Exécuter le code

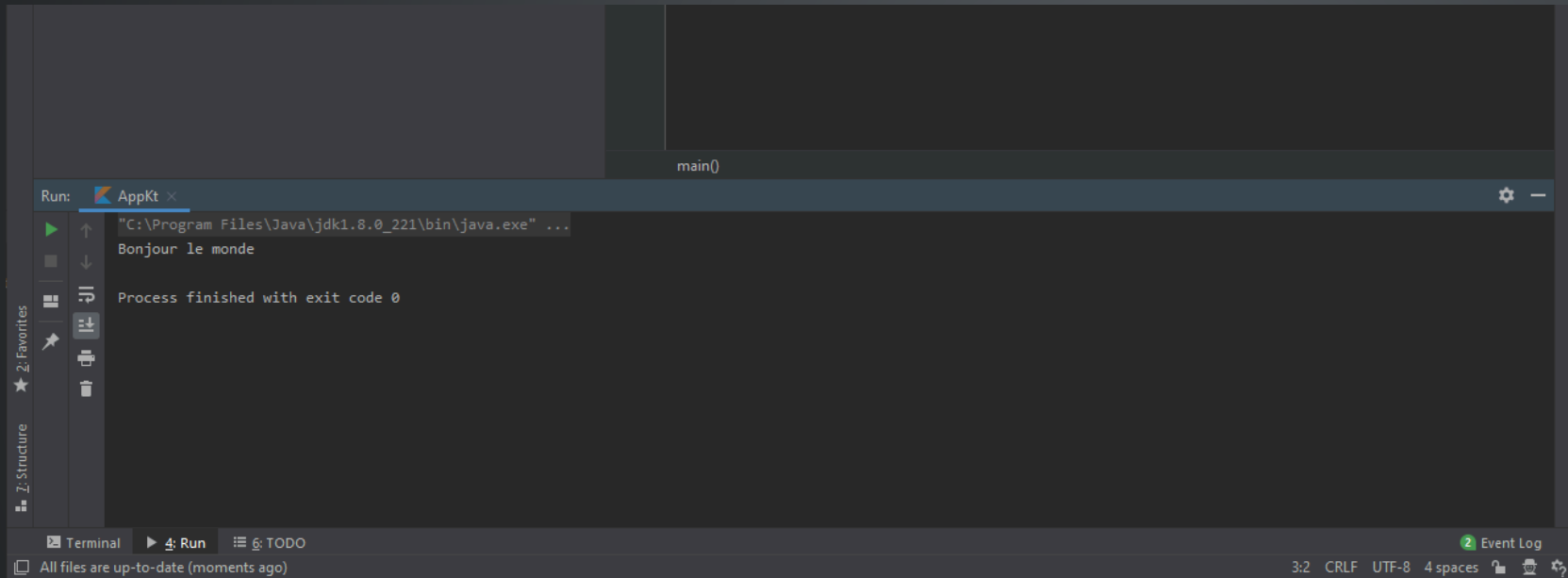
Ce qui permet d'avoir un nouveau raccourci :



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).

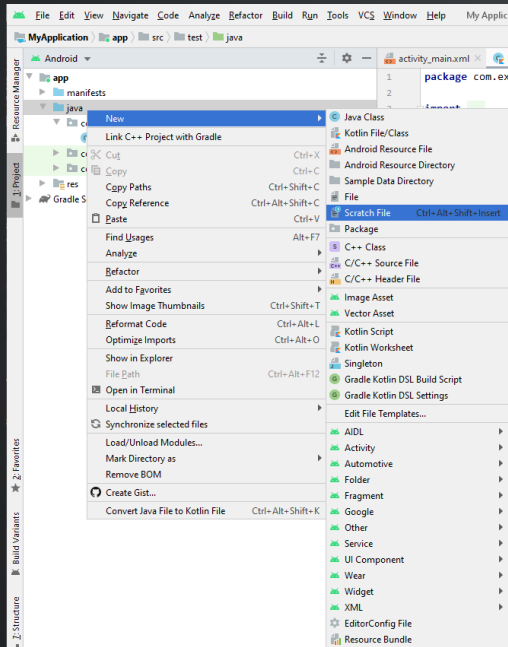
Le scratch

Kotlin et IntelliJ IDEA

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

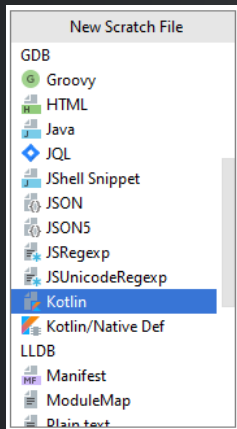
Au choix :

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



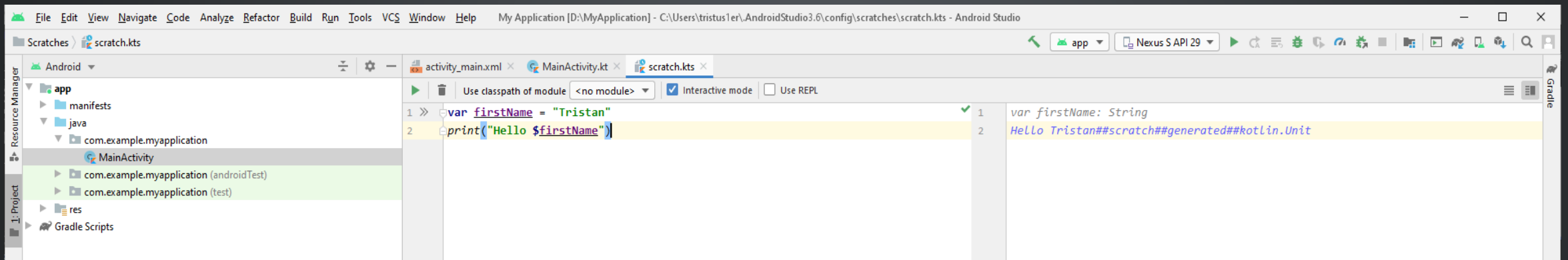
Kotlin et IntelliJ IDEA

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 de nommage des méthodes de test

On peut utiliser des noms avec des espaces entourés de ` . (Note : cela ne fonctionnera pas en Android).

Les `_` sont aussi autorisés :

```
1 class MyTestCase {  
2     @Test fun `ensure everything works`() { ... }  
3  
4     @Test fun ensureEverythingWorks_onAndroid() { ... }  
5 }
```

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 `//`.

Nommage des propriétés

Les conventions utilisées avec Kotlin

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

Nommage des propriétés internes

Si une classe comporte deux propriétés qui conceptuellement parlant, représentent la même chose, mais l'une fait partis de l'API et l'autre est un détail de l'implémentation, dans ce cas, vous pouvez préfixer d'un `_` la propriété privée :

```
1 class C {  
2     private val _elementList = mutableListOf<Element>()  
3  
4     val elementList: List<Element>  
5         get() = _elementList  
6 }
```

[Copier](#)

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

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

Copier

Déclaration de variables en Kotlin

Le code :

```
1 val name: String = "Tristan"  
2 val age: Int = 41  
3 val isDeveloper: Boolean = true
```

Copier

Est équivalent à :

```
1 val name = "Tristan"  
2 val age = 41  
3 val isDeveloper = true
```

Copier

Déclaration de variables en Kotlin

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

```
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 }
```

Copier

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 :

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

[Copier](#)

Deviendra en Kotlin :

```
1 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.

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

[Copier](#)

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).

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

[Copier](#)

Nombres

Solution

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

[Copier](#)

Nombres

Aller plus loin sur les numériques

Nombres

Les valeurs :

- Décimaux : 123
- Long (marqués avec un L) : 123L
- Hexadécimaux : 0x0F
- Binaires : 0b10001001
- Doubles par défaut : 123.5, 123.5e10
- Floats (marqués avec un f ou F) : 123.5f

Nombres

Le caractère "souligné" peut être utilisé pour améliorer la visibilité (depuis la version 1.1).

```
1 val oneMillion = 1_000_000
2 val creditCardNumber = 1234_5678_9012_3456L
3 val socialSecurityNumber = 999_99_9999L
4 val hexBytes = 0xFF_EC_DE_5E
5 val bytes = 0b11010010_01101001_10010100_10010010
```

Copier

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 caractère peut être converti explicitement :

```
1 fun decimalDigitValue(c: Char): Int {  
2     if (c !in '0'..'9')  
3         throw IllegalArgumentException("Out of range")  
4     return c.toInt() - '0'.toInt() // Explicit conversions to numbers  
5 }
```

Copier

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.

```
1 class Array<T> private constructor() {  
2     val size: Int  
3     operator fun get(index: Int): T  
4     operator fun set(index: Int, value: T): Unit  
5  
6     operator fun iterator(): Iterator<T>  
7         // ...  
8 }
```

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

```
1 fun main() {  
2     val oddArray = Array(11) { i -> (i * 2) }  
3  
4     for(i in oddArray) {  
5         print("$i ")  
6     }  
7 }
```

[Copier](#)

Les tableaux

Exercice

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

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

[Copier](#)

Les tableaux

Solution

```
1 fun main() {  
2     val alphabetArray = Array(26) { i -> (i+'a'.toInt()).toChar() }  
3  
4     for (i in alphabetArray) {  
5         print("$i ")  
6     }  
7 }
```

[Copier](#)

Les tableaux

Aller plus loin sur les tableaux

Les tableaux

Kotlin propose des tableaux spécifiques contenant des types primitifs : `ByteArray`, `ShortArray`, `IntArray`, etc, sans la couche d'encapsulation. Ces classes n'ont aucune relation avec la classe `Array`, mais disposent des mêmes méthodes et propriétés.

```
1 val x: IntArray = intArrayOf(1, 2, 3)
2 x[0] = x[1] + x[2]
3
4 // Tableau d'int de taille 5 contenant les valeurs [0, 0, 0, 0, 0]
5 val arr = IntArray(5)
6
7 // Exemple : initialisation des valeurs contenues dans le tableau avec une constante
8 // Tableau d'int de taille 5 contenant les valeurs [42, 42, 42, 42, 42]
9 val arr = IntArray(5) { 42 }
10
11 // Exemple : initialisation des valeurs contenues dans le tableau en utilisant une lambda (fonction)
12 // Tableau d'int de taille 5 contenant les valeurs [0, 1, 2, 3, 4] (valeurs égales à leur position dans le tableau)
13 var arr = IntArray(5, { it * 1 })
```

[Copier](#)

Les chaînes de caractères (String)

Les chaînes de caractères sont représentées par la classe `String` qui sont immutables. 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)

Il est possible de retirer les espaces au début des lignes, avec la méthode `.trimMargin()` qui utilise le symbole `|` par défaut.

```
1 val text = """
2   |Tell me and I forget.
3   |Teach me and I remember.
4   |Involve me and I learn.
5   |(Benjamin Franklin)
6   """.trimMargin()
7 println(text)
```

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

```
1 fun main() {  
2     fun hello(name: String): String = "Hello, ${name.toUpperCase()}"  
3     print(hello("Tristan"))  
4 }
```

[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 :

```
1 /*  
2 Les commentaires sur plusieurs  
3 lignes, pour pouvoir bien s'exprimer.  
4 Vous pouvez écrire autant que vous voulez.  
5 */  
6 // 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.

```
1 // Usage traditionnel
2 var max = a
3 if (a < b) max = b
4
5 // Avec else
6 var max: Int
7 if (a > b) {
8     max = a
9 } else {
10     max = b
11 }
12
13 // Comme une expression
14 val max = if (a > b) a else b
```

[Copier](#)

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 :

```
1 val max = if (a > b) {  
2     print("Choose a")  
3     a  
4 } else {  
5     print("Choose b")  
6     b  
7 }
```

[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.

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

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

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

Copier

Structures conditionnelles If et When

Selon le cas (when) (aller plus loin)

Le when avoir plusieurs conditions pour une même branche, séparées par une virgule :

```
1 when (x) {  
2     0, 1 -> print("x == 0 or x == 1")  
3     else -> print("otherwise")  
4 }
```

Copier

Les conditions peuvent être des expressions :

```
1 when (x) {  
2     parseInt(s) -> print("s encodes x")  
3     else -> print("s does not encode x")  
4 }
```

Copier

Structures conditionnelles If et When

Ou encore des intervalles :

```
1 when (x) {  
2     in 1..10 -> print("x is in the range")  
3     in validNumbers -> print("x is valid")  
4     !in 10..20 -> print("x is outside the range")  
5     else -> print("none of the above")  
6 }
```

Copier

On peut aussi tester le type du paramètre :

```
1 fun hasPrefix(x: Any) = when(x) {  
2     is String -> x.startsWith("prefix")  
3     else -> false  
4 }
```

Copier

On peut l'utiliser sans paramètre, les conditions sont alors obligatoirement des booléens, et remplacent un if-else if :

```
1 when {  
2     x.isOdd() -> print("x is odd")  
3     x.isEven() -> print("x is even")  
4     else -> print("x is funny")  
5 }
```

Copier

Structures conditionnelles If et When

Depuis Kotlin 1.3 il est possible de capturer le paramètre du when dans une variable :

```
1 fun Request.getBody() =  
2     when (val response = executeRequest()) {  
3         is Success -> response.body  
4         is HttpError -> throw HttpException(response.status)  
5     }
```

Copier

Boucles et ranges en Kotlin

Tant que faire se peut (while)

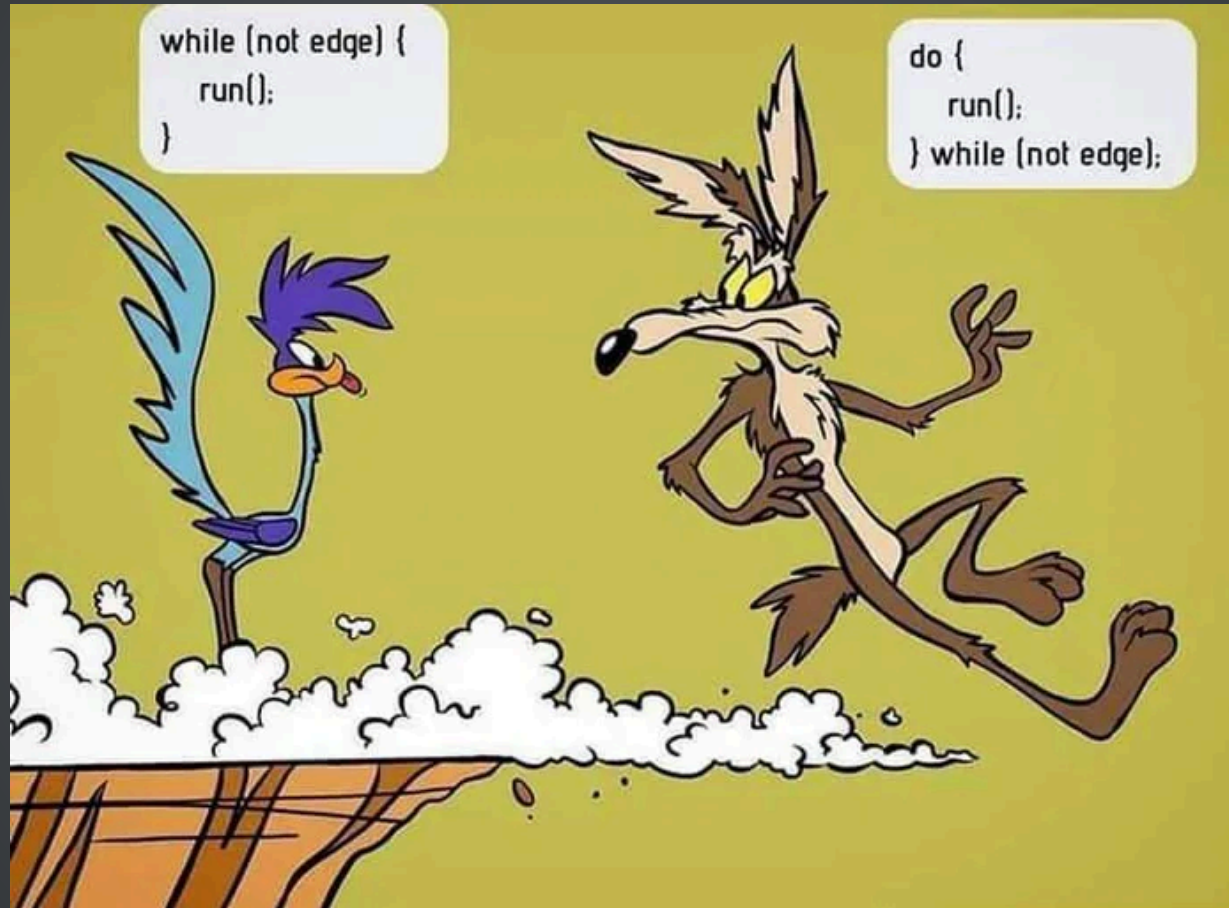
En Kotlin, la syntaxe du `while` est exactement la même qu'en Java :

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

Copier

While illustration

Boucles et ranges en Kotlin



Boucles et ranges en Kotlin

Boucle pour (for loop)

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

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

[Copier](#)

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

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

[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

```
1 var stringValue = "Une chaîne de caractères"
2 for (c in stringValue) {
3     print("$c ")
4 }
```

[Copier](#)

Solution bis :

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

[Copier](#)

Solution ter :

```
1 var stringValue = "Une chaîne de caractères"
2 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

```
1 fun main(args: Array<String>) {  
2     for (x in 0 until 10) {  
3         when {  
4             (x % 3 == 0 && x % 5 == 0) -> println("FizzBuzz")  
5             (x % 3 == 0) -> println("Fizz")  
6             (x % 5 == 0) -> println("Buzz")  
7             else -> println(x)  
8         }  
9     }  
10 }
```

[Copier](#)

Boucles et ranges en Kotlin

Solution bis

```
1 fun main(args: Array<String>) {  
2     for (x in 0 until 10) {  
3         println( when {  
4             (x % 3 == 0 && x % 5 == 0) -> "FizzBuzz"  
5             (x % 3 == 0) -> "Fizz"  
6             (x % 5 == 0) -> "Buzz"  
7             else -> x  
8         })  
9     }  
10 }
```

[Copier](#)

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 :

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

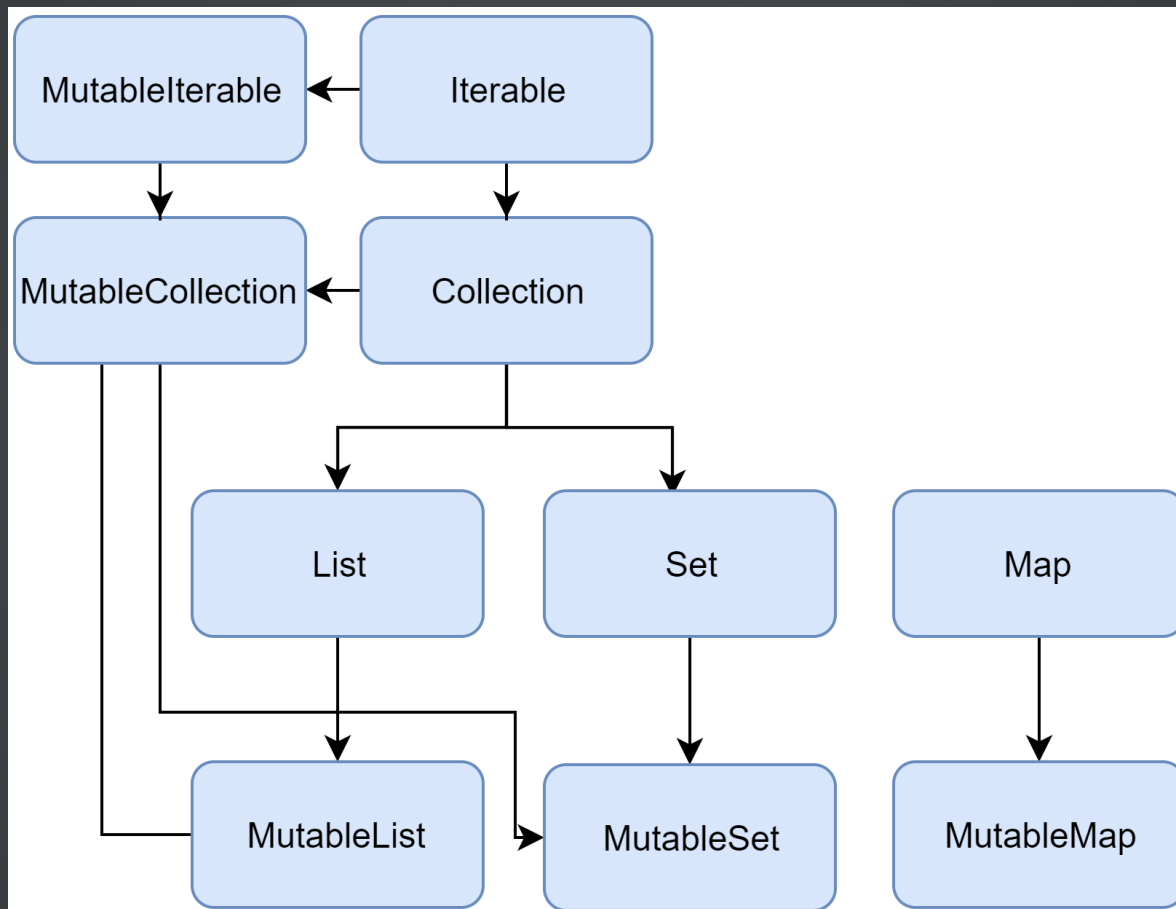
[Copier](#)

Exemple de tableau et de Map :

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

[Copier](#)

Collections en Kotlin



Packages et imports en Kotlin

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

```
1 package org.example
2
3 fun printMessage() { /*...*/ }
4 class Message { /*...*/ }
5
6 // ...
```

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 par défaut

Kotlin importe par défaut les packages suivants :

- `kotlin.*`
- `kotlin.annotation.*`
- `kotlin.collections.*`
- `kotlin.comparisons.*` (depuis 1.1)
- `kotlin.io.*`
- `kotlin.ranges.*`
- `kotlin.sequences.*`
- `kotlin.text.*`

Packages et imports en Kotlin

Imports par défaut (suite)

Selon la plate-forme cible, des packages supplémentaires sont importés :

- JVM
 - `java.lang.*`
 - `kotlin.jvm.*`
- JS
 - `kotlin.js.*`

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

Conflit de noms

Si deux classes importées ont le même nom (mais pas le même package), nous pouvons gérer le cas avec le mot clés :

```
1 import org.example.Message // Message is accessible
2 import org.test.Message as TestMessage // TestMessage stands for 'org.test.Message'
```

[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`

Packages et imports en Kotlin

Les bonnes pratiques

- Ne pas utiliser le mot `Utils` pour nommer un fichier source, qui apporte peu d'informations sur son contenu.
- Regrouper les classes qui ont un sens proche, dans un même fichier est recommandé, sous condition que le fichier ne soit pas trop gros (quelques centaines de lignes).
- De même, il est recommandé de regrouper les extensions correspondant au même client, dans un unique fichier.
- Utiliser des `val` ou des collections en lecture seule autant que possible.
- Utiliser de préférence `until` au lieu d'un intervalle ouvert :

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

Copier

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

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.

Les fonctions - Partie 1

- Fonctions en Kotlin.
- Fonctions "expression seule".
- Paramètres des fonctions en Kotlin.
- Fonctions Infix en Kotlin.
- Fonctions Anonyme en Kotlin.
- Tail recursion 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

Paramètres par défauts : surcharge

Les fonctions surcharge utilisent la même valeur par défaut. Il ne faut pas répéter la valeur par défaut :

```
1 open class A {  
2     open fun foo(i: Int = 10) { /*...*/ }  
3 }  
4  
5 class B: A() {  
6     override fun foo(i: Int) { /*...*/ } // No default value authorized  
7 }
```

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

```
1 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.

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

[Copier](#)

Paramètres des fonctions en Kotlin

Solution

```
1 fun strEq(s1: String, s2: String) = s1.equals(s2)
2 fun strEq(s1: String, s2: String, ignoreCase: Boolean) = if(ignoreCase) s1.toUpperCase().equals(s2.toUpperCase()) else s1.equals(s2)
3
4 fun main() {
5     println(strEq("Tristan", "TRISTAN"))
6     println(strEq("Tristan", "TRISTAN", true))
7 }
```

Copier

Solution bis :

```
1 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.

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

Copier

Paramètres des fonctions en Kotlin

Solution

```
1 fun tenFirstNumber(): Unit { for(i in 0 until 10) print("$i "); println() }
2 fun countdown(): Unit { for(i in 10 downTo 0) print("$i "); println() }
3 fun firstEvenNumbers(): Unit { for(i in 0 until 20 step 2) print("$i "); println() }
4 fun firstOddNumbers(): Unit { for(i in 1 .. 20 step 2) print("$i "); println() }
5
6 fun main() {
7     tenFirstNumber()
8     countdown()
9     firstEvenNumbers()
10    firstOddNumbers()
11 }
```

[Copier](#)

Paramètres des fonctions en Kotlin

Solution bis

```
1 fun tenFirstNumber() = (0..9).forEach { println(it) }
2 fun countdown() = (10 downTo 0).forEach { println(it) }
3 fun firstEvenNumbers() = (1..20 step 2).forEach { println(it) }
4 fun firstOddNumbers() = (0..9).forEach { println(it * 2) }
5
6 fun main() {
7     tenFirstNumber()
8     countdown()
9     firstEvenNumbers()
10    firstOddNumbers()
11 }
```

[Copier](#)

Paramètres des fonctions en Kotlin

Paramètres nommés

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

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

[Copier](#)

Paramètres des fonctions en Kotlin

Paramètres nommés utilisation

Nous pouvons utiliser les paramètres par défaut :

```
1 reformat(str)
```

[Copier](#)

Nous pouvons préciser tous les paramètres :

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

[Copier](#)

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

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

[Copier](#)

Paramètres des fonctions en Kotlin

Paramètres par défauts en premier

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

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

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 :

```
1 fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { /*...*/ }
2
3 foo(1) { println("hello") } // Utilise la valeur par défaut baz = 1
4 foo(qux = { println("hello") }) // Utilise les valeurs par défaut bar = 0 et baz = 1
5 foo { println("hello") } // Utilise les valeurs par défaut bar = 0 et baz = 1
```

[Copier](#)

Paramètres des fonctions en Kotlin

Nous ne sommes pas obligé d'utiliser tous les paramètres :

```
1 reformat(str, wordSeparator = '_')
```

[Copier](#)

Il est possible de mixer les paramètres de position et les paramètres nommés :

```
1 f(1, y = 2)
```

[Copier](#)

Mais l'ordre est important : il est impossible d'utiliser un paramètre nommé puis un paramètre de position :

```
1 f(x = 1, 2)
```

[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).

```
1 fun <T> asList(vararg ts: T): List<T> {  
2     val result = ArrayList<T>()  
3     for (t in ts) // ts is an Array  
4         result.add(t)  
5     return result  
6 }
```

[Copier](#)

Un seul paramètre peut être marqué variable.

Nous pouvons aussi utiliser le destructuring (opérateur `*`), si nous avons déjà une variable contenant une liste.

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

[Copier](#)

Fonctions Infix en Kotlin

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

Les conditions sont :

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

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

Copier

Fonctions Anonymes en Kotlin

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

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

[Copier](#)

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

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

[Copier](#)

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

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

[Copier](#)

Les fonctions récursives (Tail recursive functions)

Kotlin gère une façon de programmer appelée : "tail recursion".

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

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

Copier

Référence : "Tail recursive functions".

Les fonctions récursives (Tail recursive functions)

Ce code calcule le point invariant de cosinus, qui est une constante mathématique.

Le code appelle la fonction `Math.cos` plusieurs fois, en partant de 1.0 jusqu'à ce que le résultat ne change plus, en arrivant au résultat de : 0.7390851331706995 pour la précision `eps`.

Le code est équivalent à la forme plus classique :

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

Copier

Les fonctions récursives (Tail recursive functions)

Exercice

Modifions le code de la méthode `findFixPoint` pour visualiser le nombre d'appels récursifs effectués (par exemple en affichant un `.` à chaque itération).

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

[Copier](#)

Les fonctions récursives (Tail recursive functions)

Solution

Nous modifions le code de la condition else, pour afficher un "." à chaque appel.

```
1 tailrec fun findFixPoint(x: Double = 1.0): Double
2     = if (Math.abs(x - Math.cos(x)) < eps) x else {
3     print('.')
4     findFixPoint(Math.cos(x))
5 }
```

[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 :

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

Copier

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

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

Copier

Returns en Kotlin

Types de retours explicites

Les fonctions définies avec des accolades doivent explicitement préciser le type de retour (sauf si elles ne retournent rien, comme vu précédemment).

En effet, il n'est pas forcément très clair pour lecteur (et parfois même pour le compilateur) de déterminer le type de retour, en fonction de l'implémentation.

Returns en Kotlin

Fonctions locales

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

```
1 fun dfs(graph: Graph) {  
2     fun dfs(current: Vertex, visited: MutableSet<Vertex>) {  
3         if (!visited.add(current)) return  
4         for (v in current.neighbors)  
5             dfs(v, visited)  
6     }  
7  
8     dfs(graph.vertices[0], HashSet())  
9 }
```

Copier

Returns en Kotlin

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

```
1 fun dfs(graph: Graph) {  
2     val visited = HashSet<Vertex>()  
3     fun dfs(current: Vertex) {  
4         if (!visited.add(current)) return  
5         for (v in current.neighbors)  
6             dfs(v)  
7     }  
8  
9     dfs(graph.vertices[0])  
10 }
```

[Copier](#)

Returns en Kotlin

Exercice

Écrivez une fonction qui permet de déterminer si une valeur est paire en utilisant une fonction locale :

```
1 fun isMultiple(operand: Int): Boolean = TODO()
```

[Copier](#)

La fonction `isMultiple` retourne `true` si la valeur de `n` est multiple de `operand`.

La méthode à implémenter est la suivante :

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

[Copier](#)

Returns en Kotlin

Solution

```
1 fun isEven(n: Int): Boolean {  
2     fun isMultiple(operand: Int): Boolean = n % operand == 0  
3     return isMultiple(2)  
4 }  
5  
6 fun main() {  
7     println(isEven(2))  
8     println(isEven(3))  
9 }
```

[Copier](#)

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

Le constructeur ne comporte aucun code. Si nécessaire, il est possible de définir un **bloc d'initialisation** qui est préfixé par le mot clé `init`. Les blocs sont exécutés dans l'ordre d'apparition dans le code :

```
1 class InitOrderDemo(name: String) {  
2     val firstProperty = "First property: $name".also(::println)  
3  
4     init {  
5         println("First initializer block that prints ${name}")  
6     }  
7  
8     val secondProperty = "Second property: ${name.length}".also(::println)  
9  
10    init {  
11        println("Second initializer block that prints ${name.length}")  
12    }  
13 }
```

[Copier](#)

Les paramètres du constructeur peuvent être utilisés dans les blocs d'initialisation ou pour initialiser les propriétés déclarées dans le corps de la classe :

```
1 class Customer(name: String) {  
2     val customerKey = name.toUpperCase()  
3 }
```

[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 :

```
1 public class User {
2
3     // PROPERTIES
4     private String email;
5     private String password;
6     private int age;
7
8     // CONSTRUCTOR
9     public User(String email, String password, int age) {
10         this.email = email;
11         this.password = password;
12         this.age = age;
13     }
14
15     // GETTERS
16     public String getEmail() { return email; }
17     public String getPassword() { return password; }
```

Copier

En Kotlin son équivalent est :

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

Copier

Les constructeurs secondaires

Une classe

Les constructeurs secondaires sont préfixés par constructor :

```
1 class Person {
2     var children: MutableList<Person> = mutableListOf<Person>();
3     constructor(parent: Person) {
4         parent.children.add(this)
5     }
6 }
```

[Copier](#)

Si la classe comporte un constructeur primaire, chaque constructeur secondaire doit faire appel au constructeur primaire, directement ou indirectement via un autre constructeur secondaire. La délégation à un autre constructeur de la même classe est effectué en utilisant le mot clé `this` :

```
1 class Person(val name: String) {
2     var children: MutableList<Person> = mutableListOf<Person>();
3     constructor(name: String, parent: Person): this(name) {
4         parent.children.add(this)
5     }
6 }
```

[Copier](#)

Note : le code des blocs d'init font partis du constructeur primaire. La délégation au constructeur primaire est la première action effectuée dans un constructeur secondaire, donc le code de tous les blocs d'initialiseurs est exécuté avant le corps du constructeur secondaire, même si la classe n'a pas de constructeur primaire, la délégation est implicite :

```
1 class Constructors {
2     init {
3         println("Init block")
4     }
5 }
```

[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 :

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

[Copier](#)

Une classe

Propriété privée

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

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

[Copier](#)

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

```
1 class Person (var firstName: String = "", val lastName: String = "")
2
3 fun main() {
4     var father = Person("Anakin", "Skywalker")
5     println(father)
6     println( "${father.firstName} ${father.lastName}")
7
8     me.firstName = "Luke"
9     //     me.lastName = "Organa d'Alderaan"
10
11     println( "${me.firstName} ${me.lastName}")
12 }
```

[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

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

[Copier](#)

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` :

```
1 class Address {  
2     var name: String = "Holmes, Sherlock"  
3     var street: String = "221b Baker Street"  
4     var city: String = "London"  
5     var state: String? = null  
6     var zip: String = "NW1 6XE"  
7 }
```

[Copier](#)

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

```
1 fun copyAddress(address: Address): Address {  
2     val result = Address() // there's no 'new' keyword in Kotlin  
3     result.name = address.name // accessors are called  
4     result.street = address.street  
5     // ...  
6     return result  
7 }
```

[Copier](#)

Méthodes (Functions Members)

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

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

Copier

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

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

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](#)

Héritage en Kotlin

Si la classe enfant comporte un constructeur primaire, la classe parente doit être initialisée à cet endroit, en utilisant les paramètres du constructeur primaire.

Si la classe enfant n'a pas de constructeur primaire, alors chaque constructeur secondaire doit initialiser la classe parente en utilisant le mot clé `super`, ou déléguer à un constructeur secondaire qui le fera :

```
1 class MyView: View {  
2     constructor(ctx: Context): super(ctx)  
3  
4     constructor(ctx: Context, attrs: AttributeSet): super(ctx, attrs)  
5 }
```

[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 :

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

[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.

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

[Copier](#)

La surcharge

Kotlin permet de redéfinir la propriété directement dans le constructeur primaire avec le mot clé **override** :

```
1 interface Shape {  
2     val vertexCount: Int  
3 }  
4  
5 class Rectangle(override val vertexCount: Int = 4): Shape // Always has 4 vertices  
6  
7 class Polygon: Shape {  
8     override var vertexCount: Int = 0 // Can be set to any number later  
9 }
```

Copier

La surcharge

Ordre d'exécution

Pendant la construction de la nouvelle instance, la classe de base est initialisée en première, et ensuite l'initialisation de la classe enfant :

```
1 open class Base(val name: String) {  
2  
3     init { println("Initializing Base") }  
4  
5     open val size: Int =  
6         name.length.also { println("Initializing size in Base: $it") }  
7 }  
8  
9 class Derived(  
10     name: String,  
11     val lastName: String  
12 ): Base(name.capitalize().also { println("Argument for Base: $it") }) {  
13  
14     init { println("Initializing Derived") }  
15  
16     override val size: Int =  
17         (super.size + lastName.length).also { println("Initializing size in Derived: $it") }
```

[Copier](#)

Cela signifie que lorsque la classe primaire est créée, les propriétés déclarées ou redéfinies dans la classe dérivée, ne sont pas encore initialisées.

La surcharge

L'appel à la classe parente

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

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

[Copier](#)

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 :

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

Copier

L'objet compagnon

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

```
1 class MyClass1 {  
2     companion object {  
3         private const val TAG = "MyClass1"  
4         fun myMethod() { ... }  
5     }  
6 }
```

[Copier](#)

Interface en Kotlin

Les interfaces en Kotlin peuvent contenir des méthodes abstraites, mais aussi des méthodes implémentées. Ce qui les différencie 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` :

```
1 interface MyInterface {  
2     fun bar()  
3     fun foo() {  
4         // optional body  
5     }  
6 }
```

[Copier](#)

Implémenter une interface :

```
1 class Child: MyInterface {  
2     override fun bar() {  
3         // body  
4     }  
5 }
```

[Copier](#)

Interface en Kotlin

Héritage d'interfaces

Une interface peut hériter d'une autre interface, qui va compléter la première interface. Elle peut fournir une implémentation pour ses membres, et déclarer des nouvelles fonctions et propriétés. Les classes implémentant ce genre d'interface n'a besoin de définir que les implémentations manquantes :

```
1 interface Named {  
2     val name: String  
3 }  
4  
5 interface Person: Named {  
6     val firstName: String  
7     val lastName: String  
8  
9     override val name: String get() = "$firstName $lastName"  
10 }  
11  
12 data class Employee(  
13     // implementing 'name' is not required  
14     override val firstName: String,  
15     override val lastName: String,  
16     val position: Position  
17 ): Person
```

[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 :

```
1 open class User (var name: String, var firstName: String, var age: Int) {
2     open fun displayInformations() = println("$firstName $name is $age old")
3 }
4 class Admin(name: String, firstName: String, age: Int, var phoneNumber: String): User(name, firstName, age) {
5     override fun displayInformations() = println("$firstName $name is $age old, phone number: $phoneNumber")
6 }
7
8 fun main() {
9     var currentUser = User(firstName = "Tristan", name = "SALAUN", age = 41)
10    currentUser.displayInformations()
11    currentUser = Admin(firstName = "Tristan", name = "SALAUN", age = 41, phoneNumber = "0123456789")
12    currentUser.displayInformations()
13 }
```

[Copier](#)

Polymorphisme en Kotlin

Exercice

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

```
1 interface Animal {  
2     fun speak()  
3 }  
4  
5 class Dog: Animal  
6 class Bird: Animal  
7 class Duck: Animal  
8 class Snake: Animal
```

[Copier](#)

Polymorphisme en Kotlin

Solution

```
1 interface Animal {  
2     fun speak()  
3 }  
4  
5 class Dog: Animal { override fun speak() = println("Waf") }  
6 class Bird: Animal { override fun speak() = println("Cui cui") }  
7 class Duck: Animal { override fun speak() = println("Coin coin") }  
8 class Snake: Animal { override fun speak() = println("Ssssssss") }  
9  
10 fun main() {  
11     var myPet: Animal = Dog()  
12     myPet.speak()  
13     myPet = Bird()  
14     myPet.speak()  
15     myPet = Duck()  
16     myPet.speak()  
17     myPet = Snake()
```

[Copier](#)

Polymorphisme en Kotlin

Surcharge de fonctions

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

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

Copier

Polymorphisme en Kotlin

Surcharge de fonctions

Quelles seront les méthodes appelées lors de l'exécution ?

```
1 fun main() {  
2     val a: Number = 99  
3     val b = 1  
4     val c = 3.1  
5  
6     printNumber(a) //Which version of printNumber is getting used?  
7     printNumber(b) //Which version of printNumber is getting used?  
8     printNumber(c) //Which version of printNumber is getting used?  
9 }
```

Copier

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` :

```
1 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()`.

Les conditions

Data Classes en Kotlin

Afin d'assurer la consistance et le sens du code généré, la classe data doit se conformer aux conditions suivantes :

- Le constructeur primaire doit avoir au moins un paramètre.
- Tous les paramètres du constructeur primaire doivent être marqués `var` ou `val`.
- La classe data ne doit pas être abstraite (abstract), ouverte (open), scélée (sealed) ou interne (inner).
- (avant 1.1) Les classes data, peuvent seulement implémenter des interfaces.

De plus, les membres générés, suivent les règles suivantes :

- S'il y a une implémentation explicite des méthodes `equals()`, `hashCode()` ou `toString()` dans le corps de la classe, ou une implémentation marquée `final` dans la classe parente, alors ces fonctions ne sont pas générés : c'est le code existant qui est utilisé.
- Si un type parent comporte les fonctions `componentN()` qui sont open et retournent des types compatibles, la fonction correspondante est générée pour la classe data et surcharge celles du type parent. Si les fonctions du parent ne peuvent pas être surchargées, à cause d'une signature incompatible, ou si elles sont finales, alors une erreur est signalée.
- Étendre une classe qui comporte déjà une fonction `copy(...)` avec une signature correspondante est déprécié en Kotlin 1.2 et interdit en Kotlin 1.3.

Les propriétés déclarées dans le corps de Data Classes en Kotlin

Le compilateur utilise seulement les propriétés définies dans le constructeur primaire pour générer automatiquement les fonctions. Pour exclure une propriété de la génération de code, il suffit de la déclarer dans le corps de la classe :

```
1 data class Person(val name: String) {  
2     var age: Int = 0  
3 }
```

[Copier](#)

Seule la propriété `name` sera utilisée dans les fonctions `toString()`, `equals()`, `hashCode()` et `copy()` et il n'y aura qu'une seule fonction `component1()`. De ce fait, deux objets de type `Person` qui auraient des valeurs pour la propriété `age` différentes, seront considérés comme égaux.

```
1 data class Person(val name: String) {  
2     var age: Int = 0  
3 }  
4 fun main() {  
5     val person1 = Person("John")  
6     val person2 = Person("John")  
7     person1.age = 10  
8     person2.age = 20  
9     println("person1 == person2: ${person1 == person2}")  
10    println("person1 with age ${person1.age}: ${person1}")  
11    println("person2 with age ${person2.age}: ${person2}")  
12 }
```

[Copier](#)

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" :

```
1 val jane = User("Jane", 35)
2 val (name, age) = jane
3 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

```
1 data class Person (var firstName: String = "", var lastName: String = "")
2
3 fun main() {
4     var personTristan = Person("Tristan", "SALAUN")
5     var personMelody = personTristan.copy(firstName = "Mélody")
6     println(personTristan)
7     println(personMelody)
8 }
```

[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.

```
1 data class Person (var firstName: String = "", var lastName: String = "", var list: List<Int>)  
2  
3 fun main() {  
4     val listValue = listOf(1,2,3,4,5,6)  
5     var personTristan = Person("Tristan", "SALAUN", listValue)  
6     var personMelody = personTristan.copy(firstName = "Mélody")  
7     println(personTristan)  
8     println(personMelody)  
9 }
```

[Copier](#)

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 :

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

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 :

```
1 enum class Color(val rgb: Int) {  
2     RED(0xFF0000),  
3     GREEN(0x00FF00),  
4     BLUE(0x0000FF)  
5 }
```

[Copier](#)

Enum Classes en Kotlin

Les classes anonymes

Les constantes peuvent déclarer leur propre classe anonyme, avec leurs valeurs correspondantes, mais aussi surcharger les méthodes de base :

```
1 enum class ProtocolState {  
2     WAITING {  
3         override fun signal() = TALKING  
4     },  
5  
6     TALKING {  
7         override fun signal() = WAITING  
8     };  
9  
10    abstract fun signal(): ProtocolState  
11 }
```

[Copier](#)

Si une classe enum définit plusieurs membres, il faut séparer la définition des constantes de la définition des membres, par un point virgule.

Les valeurs d'une énumération, ne peut pas contenir d'imbrications autres que des inner classes (déprécié en Kotlin 1.2).

Enum Classes en Kotlin

Implémenter les interfaces dans les classes enum

Une classe enum peut implémenter une interface (mais ne dérive pas d'une classe), fournir une unique implémentation de l'interface pour toutes les valeurs ou une implémentation différent pour chaque, via sa classe anonyme. Cela est fait en ajoutant une interface à la déclaration de la classe enum, comme ci-dessous :

```
1 import java.util.function.BinaryOperator
2 import java.util.function.IntBinaryOperator
3
4 enum class IntArithmetics: BinaryOperator<Int>, IntBinaryOperator {
5     PLUS {
6         override fun apply(t: Int, u: Int): Int = t + u
7     },
8     TIMES {
9         override fun apply(t: Int, u: Int): Int = t * u
10    };
11
12    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
13 }
14
15 fun main() {
16     val a = 13
17     val b = 31
```

[Copier](#)

Travailler avec des constantes de la classe EnumClasses en Kotlin

Les classes enum, en Kotlin, disposent de fonctions qui permettent de lister les valeurs des constantes définies dans la classe, et d'obtenir un enum avec son nom. Les signatures de ces méthodes sont (en supposant que le nom de la classe est : EnumClass) :

```
1 EnumClass.valueOf(value: String): EnumClass
2 EnumClass.values(): Array<EnumClass>
```

[Copier](#)

La méthode `valueOf()` lève une `IllegalArgumentException` si le nom passé en paramètre ne correspond pas à une valeur définie dans la classe.

Depuis Kotlin 1.1, il est possible d'accéder à une constante d'une classe enum via une méthode générique, en utilisant les fonctions `enumValues<T>()` et `enumValueOf<T>()` :

```
1 enum class RGB { RED, GREEN, BLUE }
2
3 inline fun <reified T: Enum<T>> printAllValues() {
4     print(enumValues<T>().joinToString { it.name })
5 }
6
7 printAllValues<RGB>() // prints RED, GREEN, BLUE
```

[Copier](#)

Chaque constante de la classe enum possède des propriétés pour obtenir son nom et sa position dans la déclaration de la classe :

```
1 val name: String
2 val ordinal: Int
```

[Copier](#)

Enum Classes en Kotlin

Exercice

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

```
1 enum class Direction {  
2     NORTH, SOUTH, WEST, EAST  
3 }  
4 fun main() {  
5     val direction = Direction.NORTH  
6     when{  
7         true -> TODO()  
8     }  
9 }
```

[Copier](#)

Enum Classes en Kotlin

Solution

```
1 enum class Direction {  
2     NORTH, SOUTH, WEST, EAST  
3 }  
4 fun main() {  
5     val direction = Direction.NORTH;  
6     when(direction){  
7         Direction.NORTH -> println("On va au Nord.")  
8         Direction.SOUTH -> println("On va au Sud.")  
9         Direction.EAST -> println("On va à l'Est.")  
10        Direction.WEST -> println("On va à l'Ouest.")  
11    }  
12 }
```

[Copier](#)

Enum Classes en Kotlin

Solution bis

```
1 enum class Direction {  
2     NORTH, SOUTH, WEST, EAST  
3 }  
4  
5 fun main() {  
6     val direction = Direction.NORTH;  
7     println(  
8         when (direction) {  
9             Direction.NORTH -> "On va au Nord."  
10            Direction.SOUTH -> "On va au Sud."  
11            Direction.EAST -> "On va à l'Est."  
12            Direction.WEST -> "On va à l'Ouest."  
13        }  
14    )  
15 }
```

[Copier](#)

Nested Classes en Kotlin

Les classes peuvent être imbriquées dans d'autres classes :

```
1 class Outer {  
2     private val bar: Int = 1  
3     class Nested {  
4         fun foo() = 2  
5     }  
6 }  
7  
8 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 :

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

[Copier](#)

Nested Classes en Kotlin

Classes imbriquées anonymes

Les classes imbriquées anonymes sont créées en utilisant une expression object

```
1 window.addMouseListener(object: MouseAdapter() {  
2  
3     override fun mouseClicked(e: MouseEvent) { ... }  
4  
5     override fun mouseEntered(e: MouseEvent) { ... }  
6 })
```

[Copier](#)

Note : dans la JVM, si l'objet est une instance d'une interface fonctionnelle Java (c'est-à-dire, une interface avec une seule méthode abstraite), vous pouvez la créer en utilisant une expression lambda, préfixée du type de l'interface :

```
1 val listener = ActionListener { println("clicked") }
```

[Copier](#)

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 :

```
1 fun eval(expr: Expr): Double = when(expr) {  
2     is Const -> expr.number  
3     is Sum -> eval(expr.e1) + eval(expr.e2)  
4     NotANumber -> Double.NaN  
5     // the `else` clause is not required because we've covered all the cases  
6 }
```

[Copier](#)

Sealed Classes en Kotlin

Exemple sans classe scellée

Réolvons 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 Subtract(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

```
1 fun eval(expr: Expr): Double = when(expr) {  
2     is Const -> expr.number  
3     is Sum -> eval(expr.e1) + eval(expr.e2)  
4     is Subtract -> eval(expr.e1) - eval(expr.e2)  
5     is Multiply -> eval(expr.e1) * eval(expr.e2)  
6     is Divide -> eval(expr.e1) / eval(expr.e2)  
7     NotANumber -> Double.NaN  
8     // the `else` clause is not required because we've covered all the cases  
9 }
```

[Copier](#)

Les fonctions - Partie 2

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

Les fonctions - Partie 2

- Surcharge d'opérateurs en Kotlin.
- Expressions Lambda en Kotlin.
- Extensions de fonctions en Kotlin.
- Extensions de propriétés en Kotlin.
- Closures en Kotlin.

Surcharge d'opérateurs en Kotlin

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

Surcharge d'opérateurs en Kotlin

Les opérations unaires

Expression	Traduites en
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

Surcharge d'opérateurs en Kotlin

Exemple d'implémentation pour la classe Point :

```
1 data class Point(val x: Int, val y: Int)
2
3 operator fun Point.unaryMinus() = Point(-x, -y)
4
5 val point = Point(10, 20)
6
7 fun main() {
8     println(-point) // prints "Point(x=-10, y=-20)"
9 }
```

Copier

Surcharge d'opérateurs en Kotlin

Les opérations incrément et décrément

Expression	Traduites en
<code>a++</code>	<code>a.inc()</code>
<code>a--</code>	<code>a.dec()</code>

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

Surcharge d'opérateurs en Kotlin

Exercice

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

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

[Copier](#)

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

Surcharge d'opérateurs en Kotlin

Solution

```
1 data class Point(var x: Int, var y: Int) {  
2     operator fun inc(): Point {  
3         return Point(this.x + 1, this.y + 1)  
4     }  
5 }  
6 fun main() {  
7     var point1 = Point(1,1)  
8     println(point1)  
9     point1++  
10    println(point1)  
11    println(++point1)  
12    println(point1)  
13    println(point1++)  
14    println(point1)  
15  
16    // Compare with Int behaviour.  
17    var value = 2
```

[Copier](#)

Surcharge d'opérateurs en Kotlin

Les opérations binaires

On parle ici d'opérations qui nécessitent deux valeurs.

Surcharge d'opérateurs en Kotlin

Les opérations arithmétiques

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

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

Surcharge d'opérateurs en Kotlin

Exemple

Implémentation de l'opérateur + pour la classe Counter.

```
1 data class Counter(val dayIndex: Int) {  
2     operator fun plus(increment: Int): Counter {  
3         return Counter(dayIndex + increment)  
4     }  
5 }
```

Copier

Exercice : proposer un cas d'usage. Par exemple initialisons notre objet Counter avec la valeur 10 et incrémentons-la de 3.

Surcharge d'opérateurs en Kotlin

Solution

```
1 fun main() {  
2     var counter: Counter = Counter(10)  
3  
4     println(counter)  
5     println(counter + 3)  
6 }
```

[Copier](#)

Ou de manière équivalente :

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

[Copier](#)

Surcharge d'opérateurs en Kotlin

Exercice

Écrivez l'opérateur + pour la classe Point qui permet d'additionner 2 points (on additionne le x1 avec le x2 et le y1 avec le y2).

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

[Copier](#)

Surcharge d'opérateurs en Kotlin

Solution

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

[Copier](#)

Ou de manière équivalente :

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

[Copier](#)

Surcharge d'opérateurs en Kotlin

L'opérateur 'In'

Expression	Traduites en
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

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

Surcharge d'opérateurs en Kotlin

Opérateur d'accès indexé

Expression	Traduites en
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

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

Surcharge d'opérateurs en Kotlin

Opérateur d'invocation

Expression	Traduites en
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

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

Surcharge d'opérateurs en Kotlin

Opérateur d'affectation étendu

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

Surcharge d'opérateurs en Kotlin

Opérateurs d'égalité et d'inégalité

Expression	Traduites en
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

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

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

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

Surcharge d'opérateurs en Kotlin

Attention à la définition de l'égalité :

```
1 fun main() {  
2     val first = Integer(10)  
3     val second = Integer(10)  
4  
5     println(first == second)      // 1  
6     println(first.equals(second)) // 2  
7     println(first === second)    // 3  
8 }
```

Copier

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

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

Surcharge d'opérateurs en Kotlin

Opérateurs de comparaison

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

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

Lambda expression en Kotlin

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

```
1 fun <T, R> Collection<T>.fold(  
2     initial: R,  
3     combine: (acc: R, nextElement: T) -> R  
4 ): R {  
5     var accumulator: R = initial  
6     for (element: T in this) {  
7         accumulator = combine(accumulator, element)  
8     }  
9     return accumulator  
10 }
```

[Copier](#)

Dans le code ci-dessus, le paramètre `combine` à un type de fonction $(R, T) \rightarrow R$, 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 :

```
1 val items = listOf(1, 2, 3, 4, 5)
2
3 // Lambdas are code blocks enclosed in curly braces.
4 items.fold(0, {
5     // When a lambda has parameters, they go first, followed by '->'
6     acc: Int, i: Int ->
7     println("acc = $acc, i = $i, ")
8     val result = acc + i
9     println("result = $result")
10    // The last expression in a lambda is considered the return value:
11    result
12 })
```

[Copier](#)

```
1 // Parameter types in a lambda are optional if they can be inferred:
2 val joinedToString = items.fold("Elements:", { acc, i -> "$acc $i" })
3
4 // Function references can also be used for higher-order function calls:
5 val product = items.fold(1, Int::times)
```

[Copier](#)

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)`.

```
1 fun main() {  
2     val stringPlus: (String, String) -> String = String::plus  
3     val intPlus: Int.(Int) -> Int = Int::plus  
4  
5     println(stringPlus.invoke("<-", "->"))  
6     println(stringPlus("Hello, ", "world!"))  
7  
8     println(intPlus.invoke(1, 1))  
9     println(intPlus(1, 2))  
10    println(2.intPlus(3)) // extension-like call  
11 }
```

Copier

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 mot `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

Caractère souligné (underscore) pour les paramètres non utilisés

Si un paramètre de la lambda est inutilisé, alors on peut utiliser `_` à la place du paramètre :

```
1 map.forEach { _, value -> println("$value!") }
```

[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

```
1 val addition: (Int, Int) -> Int = TODO()
2 val subtraction: (Int, Int) -> Int = TODO()
3 val multiplication: (Int, Int) -> Int = TODO()
4 val division: (Int, Int) -> Int = TODO()
5
6 inline fun executeOperation(x: Int, y: Int, operation: (Int, Int) -> Int) = operation(x, y)
7
8 fun main() {
9     println(executeOperation(10, 5, addition))
10    println(executeOperation(10, 5, subtraction))
11    println(executeOperation(10, 5, multiplication))
12    println(executeOperation(10, 5, division))
13 }
```

[Copier](#)

Lambda expression en Kotlin

Solution

```
1 val addition = { x: Int, y: Int -> x + y }
2 val subtraction = { x: Int, y: Int -> x - y }
3 val multiplication = { x: Int, y: Int -> x * y }
4 val division = { x: Int, y: Int -> x / y }
5
6 inline fun executeOperation(x: Int, y: Int, operation: (Int, Int) -> Int) = operation(x, y)
7
8 fun main() {
9     println(executeOperation(10, 5, addition))
10    println(executeOperation(10, 5, subtraction))
11    println(executeOperation(10, 5, multiplication))
12    println(executeOperation(10, 5, division))
13 }
```

[Copier](#)

Simplification d'une méthode Java en Kotlin

Exemple de code en 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

Simplification d'une méthode Java en Kotlin

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

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

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 exercices :
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>`

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

[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>`.

```
1 val list = mutableListOf(1, 2, 3)  
2 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).

```
1 fun String.firstUpper(): Any = TODO()  
2  
3 // Exemple d'appel :  
4 fun main() {  
5     println("tristan".firstUpper()) // => Tristan  
6     println("TRISTAN".firstUpper()) // => Tristan  
7 }
```

[Copier](#)

Extensions

Solution

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

[Copier](#)

Autre solution :

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

[Copier](#)

Encore plus concis (sans utiliser le this) :

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

[Copier](#)

Extensions

Extensions de propriétés en Kotlin

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

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

[Copier](#)

Extensions

Exercice

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

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

[Copier](#)

Extensions

Solution

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

[Copier](#)

A noter, que le `this` dans `this.setCharAt(0, value)` est optionnel.

Extensions

Solution bis

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

[Copier](#)

A noter, que le `this` dans `this.setCharAt(0, value)` est optionnel.

Closures en Kotlin

Une expression lambda, ou une fonction anonyme (ou une fonction locale, ou encore une expression object) peuvent accéder à leur environnement (closure), c'est-à-dire aux variables définies en dehors de leur périmètre (scope). Exemple :

```
1 var ints = listOf(1, 2, -2, 3, 4, -15, 5, 6, -1)
2 var sum = 0
3 ints.filter { it > 0 }.forEach {
4     sum += it
5 }
6 println(sum)
```

[Copier](#)

Closures en Kotlin

Exercice

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

```
1 var ints = listOf(1, 2, -2, 3, 4, -15, 5, 6, -1)
2 var sum = 0
3 ints.filter { it > 0 }.forEach {
4     sum += it
5 }
6 println(sum)
```

[Copier](#)

Closures en Kotlin

Solution

```
1 fun main() {  
2     var ints = listOf(1, 2, -2, 3, 4, -15, 5, 6, -1)  
3     var sum = 0  
4     ints.filter { it > 0 }.forEach {  
5         sum += it  
6     }  
7     println(sum)  
8 }
```

[Copier](#)

Délégation

- Délégation de propriétés en Kotlin

Délégation

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

Concept de délégation en Kotlin

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

Délégation de fonctions en Kotlin

Exemple en Java :

```
1 interface Showable {  
2     void show();  
3 }  
4  
5 class View implements Showable {  
6  
7     @Override  
8     public void show() {  
9         System.out.println("View.show()");  
10    }  
11 }  
12  
13  
14 class CustomView implements View {  
15  
16     @Override  
17     public void show() {
```

Copier

Délégation de fonctions en Kotlin

Exemple en Kotlin

```
1 interface Nameable {  
2     var name: String  
3 }  
4  
5 class JackName: Nameable {  
6     override var name: String = "Jack"  
7 }  
8  
9 class LongDistanceRunner: Runnable {  
10     override fun run() {  
11         println("long")  
12     }  
13 }  
14  
15 class Person(name: Nameable, runner: Runnable): Nameable by name, Runnable by runner
```

[Copier](#)

```
1 fun main(args: Array<String>) {  
2     val person = Person(JackName(), LongDistanceRunner())  
3     println(person.name) //Jack  
4     person.run() //long  
5 }
```

[Copier](#)

Délégation de fonctions en Kotlin

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

Délégation de fonctions en Kotlin

Exercice

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

```
1 fun displayPerson(person: Person){  
2     println(person.name)  
3     person.run()  
4 }
```

[Copier](#)

Délégation de fonctions en Kotlin

Solution

```
1 interface Nameable {  
2     var name: String  
3 }  
4  
5 class JackName: Nameable {  
6     override var name: String = "Jack"  
7 }  
8 class TristanName: Nameable {  
9     override var name: String = "Tristan"  
10 }  
11  
12 class LongDistanceRunner: Runnable {  
13     override fun run() {  
14         println("long")  
15     }  
16 }  
17 class ShortDistanceRunner: Runnable {
```

[Copier](#)

Délégation de fonctions en Kotlin

Exercice

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

Délégation de fonctions en Kotlin

Solution

```
1 interface Nameable {  
2     var name: String  
3     var firstName: String  
4     var companyName: String  
5     var cityName: String  
6 }  
7  
8 class JackName: Nameable {  
9     override var name: String = "Jack"  
10    override var firstName: String = "Terence"  
11    override var companyName: String = "IBM"  
12    override var cityName: String = "London"  
13 }  
14 class TristanName: Nameable {  
15     override var name: String = "SALAUN"  
16     override var firstName: String = "Tristan"  
17     override var companyName: String = "STDev"
```

[Copier](#)

Délégation de propriétés en Kotlin

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

```
1 class Foo {  
2     var prop: String? = null  
3 }  
4  
5 val foo = Foo()  
6 foo.prop = "something"  
7 val another = foo.prop
```

Copier

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

```
1 var str: String  
2     get() = this.toString()  
3     set(value) {  
4         println(value)  
5         field = value  
6     }
```

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 :

```
1 import kotlin.reflect.KProperty
2
3 class Example {
4     val someName by NameDelegate()
5     val otherName by NameDelegate()
6 }
7
8 class NameDelegate {
9     operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
10         return property.name
11     }
12 }
13
14 fun main() {
15     val example = Example()
16     println(example.someName)
17     println(example.otherName)
```

[Copier](#)

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 :

```
1 import kotlin.reflect.KProperty
2
3 class Delegate {
4     operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
5         return "$thisRef, thank you for delegating '${property.name}' to me!"
6     }
7
8     operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
9         println("$value has been assigned to '${property.name}' in $thisRef.")
10    }
11 }
```

Copier

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 :

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

Copier

De manière plus classique/concise, nous utiliserons :

```
1 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 :

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

[Copier](#)

Délégation de propriétés en Kotlin

Exemple d'utilisation

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

[Copier](#)

Que va afficher le code ci-dessus ?

Délégation de propriétés en Kotlin

Exercice

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

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

[Copier](#)

Qu'affiche le code ci-dessus ?

Délégation de propriétés en Kotlin

Solution

```
1 import kotlin.properties.Delegates
2
3 fun main() {
4     var maxCount: Int by Delegates.observable(initialValue = 0) { property, oldValue, newValue ->
5         println("${property.name} is being changed from $oldValue to $newValue")
6     }
7
8     maxCount = 5
9     maxCount = 10
10    maxCount = 20
11 }
```

[Copier](#)

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.

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

Copier

Délégation de propriétés en Kotlin

Solution

```
1 import kotlin.properties.Delegates.vetoable
2
3 fun main() {
4     var age: Int by vetoable(initialValue = 0) { property, oldValue, newValue ->
5         newValue > 0
6     }
7
8     age = 10
9     println(age)
10    age = -1
11    println(age)
12    age = 30
13    println(age)
14    age = 0
15    println(age)
16 }
```

[Copier](#)

Délégation de propriétés en Kotlin

Exercice

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

Délégation de propriétés en Kotlin

Solution

```
1 import kotlin.properties.Delegates
2
3 fun main() {
4     var age: Int by Delegates.vetoable(initialValue = 0) { property, oldValue, newValue ->
5         if (newValue > 0) true else {println("${property.name} rejected value $newValue staying at value $oldValue"); false}
6     }
7
8     age = 10
9     println(age)
10    age = -1
11    println(age)
12    age = 30
13    println(age)
14    age = 0
15    println(age)
16 }
```

[Copier](#)

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) :

```
1 var person1: Person // Erreur sur cette ligne. à corriger.  
2  
3 fun main(args: Array<String>) {  
4     // initializing variable lately  
5     person1 = Person("Ted",28)  
6     println("${person1.name} is ${person1.age.toString()}")  
7 }  
8  
9 data class Person(var name:String, var age:Int)
```

[Copier](#)

Délégation de propriétés en Kotlin

Solution

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

[Copier](#)

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

```
1 import kotlin.properties.ReadWriteProperty
2 import kotlin.reflect.KProperty
3
4 class LogDelegate<T>: ReadWriteProperty<Any, T> {
5     private var value: T? = null
6     override fun getValue(thisRef: Any, property: KProperty<*>): T? {
7         println("LOG get $value")
8         return value
9     }
10    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
11        println("LOG set: $value")
12        this.value = value
13    }
14 }
15
16 class Demo {
17     var name by LogDelegate<String>()
```

[Copier](#)

Bonnes et mauvaises pratiques

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

Generics

- Generics en Kotlin.

Generics

- Generics en Kotlin.
- Generics et invariance en Kotlin.
- Covariance en Kotlin.
- Contravariance en Kotlin.

Generics en Kotlin

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

```
1 class Box<T>(t: T) {  
2     var value = t  
3 }
```

[Copier](#)

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

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

[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 :

```
1 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 :

```
1 fun <T> singletonList(item: T): List<T> {  
2     // ...  
3 }  
4  
5 fun <T> T.basicToString(): String { // extension function  
6     // ...  
7 }
```

[Copier](#)

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

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

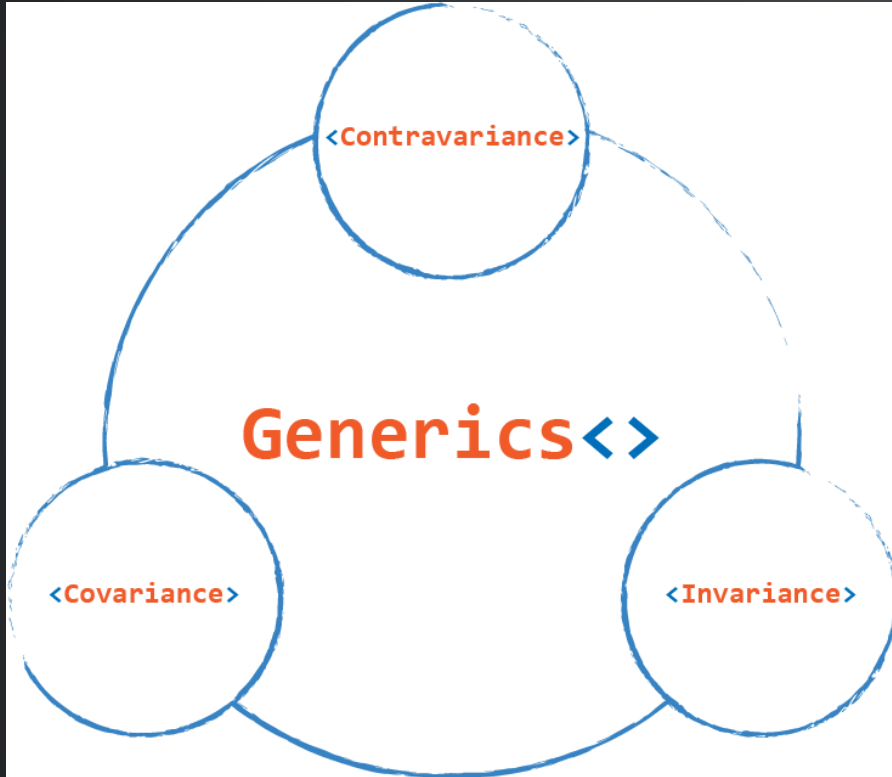
[Copier](#)

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

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

[Copier](#)

Generics et invariance en Kotlin



Generics et invariance en Kotlin

Invariance

Quand on utilise des classes simples, étendre de ces classes est simple. Mais quand on commence à utiliser des classes génériques, alors les règles se compliquent un peu.

L'invariance exprime le fait que bien que 2 classes aient une relation de hiérarchie, un type complexe (un générique) ne suit pas cette même hiérarchie. Pour que cela soit plus clair, prenons un exemple :

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

[Copier](#)

Considérons les types suivants :

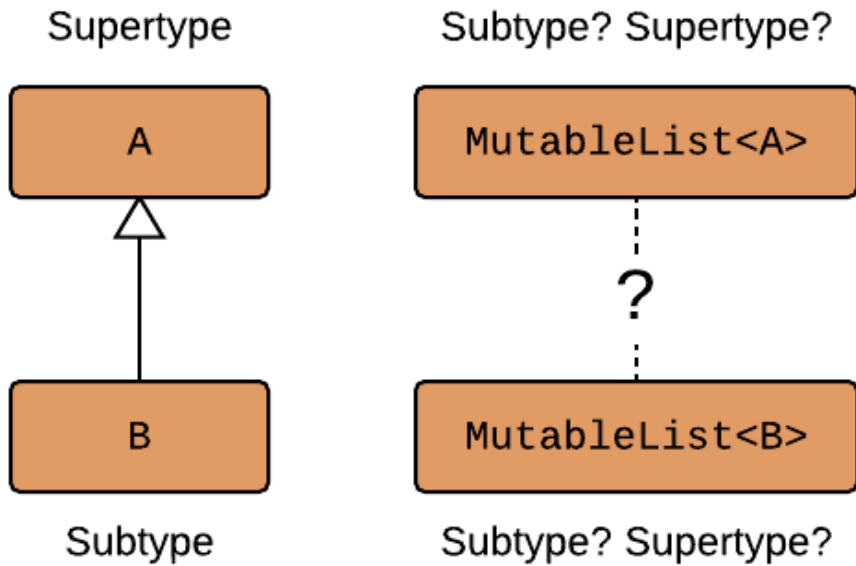
```
1 MutableList<A>
2 MutableList<B>
```

[Copier](#)

Generics et invariance en Kotlin

Relation ?

Quelle est la relation entre les deux `MutableList` ?



Generics et invariance en Kotlin

Testons

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

[Copier](#)

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

[Copier](#)

Generics et invariance en Kotlin

Réponse

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

Covariance en Kotlin

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

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

[Copier](#)

Considérons les types en lecture seule suivants :

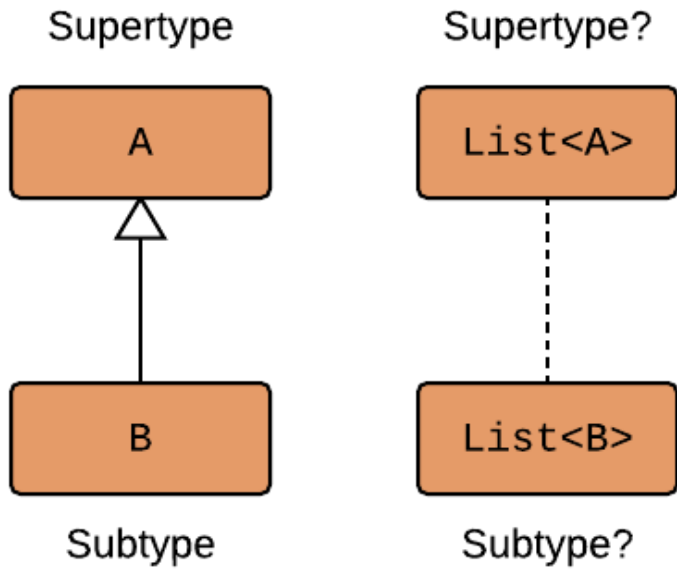
```
1 List<A>
2 List<B>
```

[Copier](#)

Covariance en Kotlin

Relation ?

Quelle est la relation entre les deux List ?



Covariance en Kotlin

Testons

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

[Copier](#)

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

[Copier](#)

Covariance en Kotlin

Réponse

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

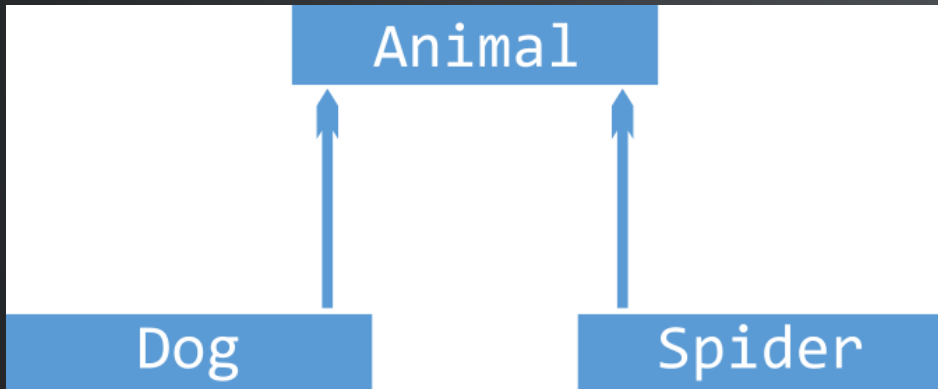
Covariance en Kotlin

Exercice

Prenons les classes suivantes :

```
1 abstract class Animal(val size: Int)
2 class Dog(val cuteness: Int): Animal(100)
3 class Spider(val terrorFactor: Int): Animal(1)
```

Copier



Covariance en Kotlin

Validons la hiérarchie

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

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

[Copier](#)

Tout est OK.

Covariance en Kotlin

Test avec les List

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

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

[Copier](#)

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



Contravariance en Kotlin

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

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

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

[Copier](#)

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

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

[Copier](#)

Contravariance en Kotlin

Comparaison d'animaux

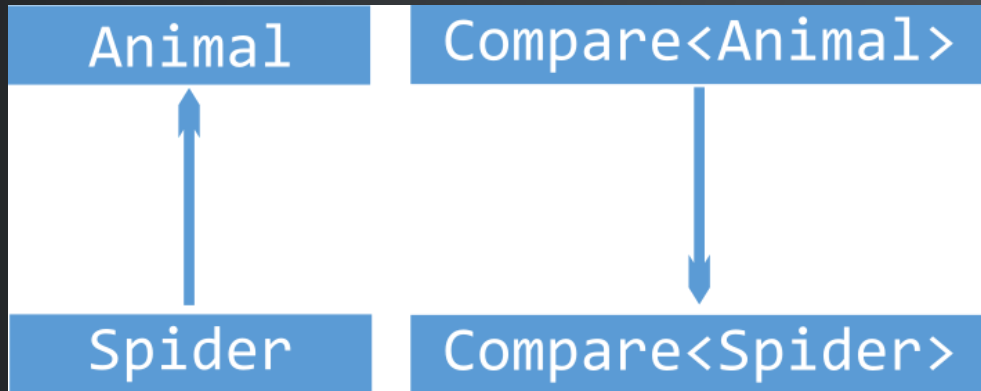
Si nous voulons comparer tous les animaux, le mécanisme doit fonctionner pour tous les chiens et les araignées :

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

[Copier](#)

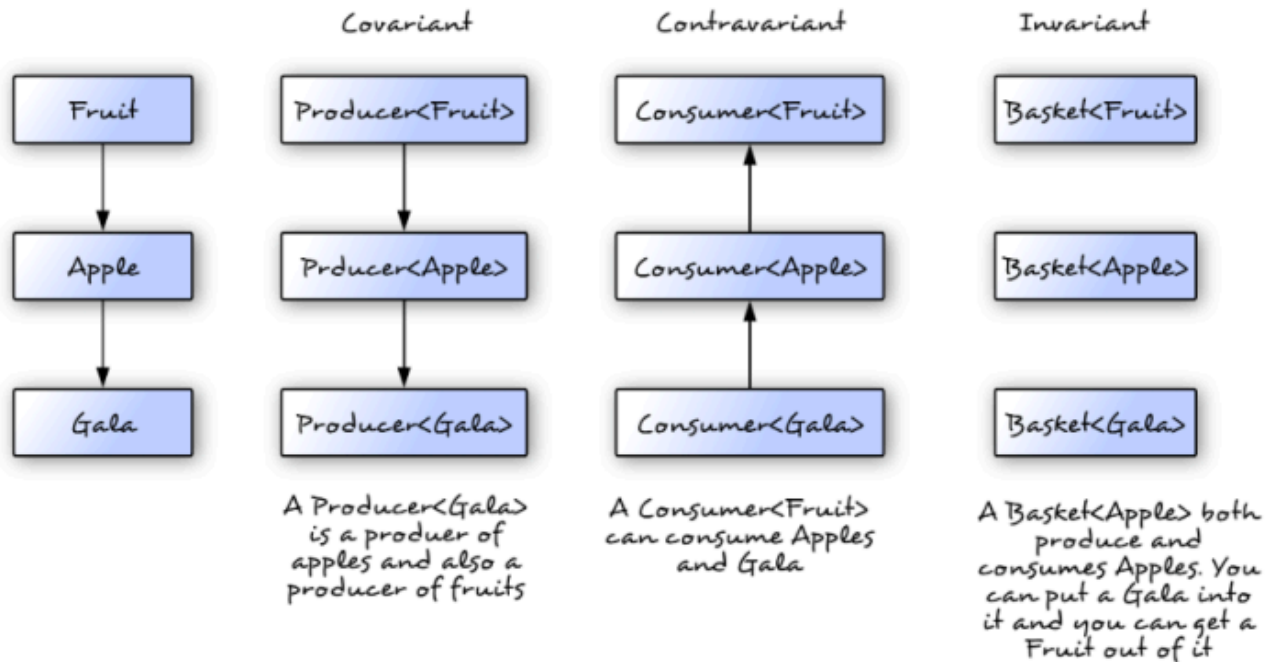
Contravariance en Kotlin

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



Autre exemple avec des fruits

Prenons un autre exemple avec des fruits



Autre exemple avec des fruits

Définition des fruits

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

[Copier](#)

Autre exemple avec des fruits

Expression des variances

```
1 class Variance{
2     val fruitProducer: () -> Fruit = ::Fruit
3     val appleProducer: () -> Apple = ::Apple
4     val galaProducer: () -> Gala = ::Gala
5     val fruitConsumer: (Fruit) -> Unit = ::eatFruit
6     val appleConsumer: (Apple) -> Unit = ::eatApple
7     val galaConsumer: (Gala) -> Unit = ::eatGala
8     val newFruitProducer1: () -> Fruit = appleProducer
9     val newFruitProducer2: () -> Fruit = galaProducer
10    val newAppleProducer: () -> Apple = galaProducer
11    val newGalaConsumer1: (Gala) -> Unit = appleConsumer
12    val newGalaConsumer2: (Gala) -> Unit = fruitConsumer
13    val newAppleConsumer: (Fruit) -> Unit = fruitConsumer
14 }
```

[Copier](#)

Autre exemple avec des fruits

Fonctions utilitaires

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

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.

Autres fonctionnalités

- Casting de types en Kotlin.
- Tuples.
- Déconstruction de valeurs.
- Gestion des exceptions.
- Déclaration de constantes.
- Initialisation tardive.
- Annotation en Kotlin.
- Bonnes et mauvaises pratiques.

Casting de types en Kotlin

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

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

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 :

```
1 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? :

```
1 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

```
1 class Person(var name: String)
2 class Other()
3
4 fun main() {
5     fun foo(x: Any) {
6         if (x is Person) {
7             println("${x.name}") // This wouldn't compile outside the if
8         }
9     }
10
11     var x: Any = Person("SALAUN")
12     var p = x as Person
13     var pNull = x as? Person
14
15     x = Other()
16     p = x as Person
17     pNull = x as? Person
```

[Copier](#)

Casting de types en Kotlin

Exercice

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

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Test {
5
6     private static int getDefaultSize(Object object){
7         if(object instanceof String) {
8             return ((String) object).length();
9         } else if(object instanceof List) {
10             return ((List) object).size();
11         }
12         return 0;
13     }
14
15     public static void main(String[] args) {
16
17         List list = new ArrayList();
```

[Copier](#)

Casting de types en Kotlin

Solution

```
1 private fun getDefaultSize(anyObject: Any): Int {  
2     if (anyObject is String) {  
3         return anyObject.length  
4     } else if (anyObject is List<*>) {  
5         return anyObject.size  
6     }  
7     return 0  
8 }
```

Copier

Ou encore plus concis :

```
1 private fun getDefaultSize(anyObject: Any) = when (anyObject) {  
2     is String -> anyObject.length  
3     is List<*> -> anyObject.size  
4     else -> 0  
5 }
```

Copier

Tuples

Les types standards

Kotlin propose dans la librairie standard des Tuples :

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

Tuples

Pair

[Tous les détails sur la page officielle.](#)

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

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

Tuples

Triple

[Tous les détails sur la page officielle.](#)

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

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

Tuples

Exercice

Écrivez une variable référençant une `Pair` de valeur, et les afficher séparément.
Procédez de la même manière avec `Triple`.
Les signatures des classes sont rappelées ci-dessous :

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

[Copier](#)

Tuples

Solution

```
1 fun main() {  
2  
3     val pairValue= Pair(1, "x")  
4     val tripleValue = Triple("Tristan","SALAUN", 40)  
5  
6     println(pairValue.first)  
7     println(pairValue.second)  
8     println(tripleValue.first)  
9     println(tripleValue.second)  
10    println(tripleValue.third)  
11 }
```

[Copier](#)

Deconstructing (Destructuring) Values

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

```
1 data class Person(var name:String, var age: Int)
2 val tristan = Person("Tristan", 44)
3 val (name, age) = tristan
```

[Copier](#)

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

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

[Copier](#)

Deconstructing (Destructuring) Values

Le code généré correspond à :

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

Copier

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

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

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

Copier

Deconstructing (Destructuring) Values

Exemple : retourner deux valeurs depuis une fonction

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

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

[Copier](#)

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

Deconstructing (Destructuring) Values

Exercice

Reprenons l'exemple avec les tuples et déstructurons les `Pair` dans des variables `a` et `b`.
Faisons de même pour `Triple` avec `c`, `d` et `e`.

Deconstructing (Destructuring) Values

Solution

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

[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

```
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 }
```

[Copier](#)

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](#)

L'objet compagnon

Pour les méthodes

Étudions l'exemple situé à l'adresse : <https://odelia-technologies.com/blog/object-companion-kotlin.html>

Une autre utilisation de l'objet companion est pour déclarer nos constantes, étant donné que le mot clé static n'existe pas en Kotlin :

```
1 companion object {  
2     private const val TAG = "ClassName"  
3 }
```

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` :

```
1 public class MyTest {  
2     lateinit var subject: TestSubject  
3  
4     @SetUp fun setup() {  
5         subject = TestSubject()  
6     }  
7  
8     @Test fun test() {  
9         subject.method() // dereference directly  
10    }  
11 }
```

[Copier](#)

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 se passe-t-il quand nous l'exécutons, pourquoi, et comment corriger le problème ?

```
1 class MyTest() {  
2     lateinit var subject: String  
3  
4     fun displaySubject() {  
5         println(subject)  
6     }  
7 }  
8  
9 fun main() {  
10     var myTest = MyTest()  
11     myTest.displaySubject()  
12 }
```

[Copier](#)

Initialisation tardive

Solution

Respectons notre contrat : initialisons la variable, comme promis :

```
1 fun main() {  
2     var myTest = MyTest()  
3     myTest.subject = "The best subject"  
4     myTest.displaySubject()  
5 }
```

[Copier](#)

Annotation en Kotlin

Définition

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

Annotation en Kotlin

Annoter un constructeur

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

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

[Copier](#)

Bonnes et mauvaises pratiques

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

Les annotations peuvent être mises en place pour générer du code standard lors de la précompilation.

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

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

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

```
1 open class Person(var firstName: String, val lastName: String)
2
3 class Friend(firstName: String, lastName: String, var age: Short): Person(firstName, lastName)
4 class Colleague(firstName: String, lastName: String, var role: Role): Person(firstName, lastName)
5 class Contact(firstName: String, lastName: String, var phoneNumber: String, var email: String): Person(firstName, lastName)
6
7 enum class Role {
8     MANAGER, CEO, WORKER
9 }
```

[Copier](#)

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

```
7 class Contact(firstName: String, lastName: String, var phoneNumber: PhoneNumber, var email: String): Person(firstName, lastName)
8
9 enum class Role {
10     MANAGER, CEO, WORKER
11 }
12
13 class PhoneNumber(type: PhoneType, value: String = "0000000000") {
14     var value: String by Delegates.vetoable(value) { property, oldValue, newValue ->
15         newValue.length == 10 && isNumber(newValue)
16     }
17 }
18
19 fun isNumber(s: String?): Boolean {
20     return if (s.isNullOrEmpty()) false else s.all { Character.isDigit(it) }
21 }
22
23 enum class PhoneType {
```

[Copier](#)

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

```
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     }
```

[Copier](#)

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

```
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) {
```

[Copier](#)

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

```
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
```

[Copier](#)

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

```
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) {
```

[Copier](#)

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

```
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 {
```

[Copier](#)

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, ...

```
1 import kotlin.random.Random
2
3 class RandomValues {
4     companion object {
5         val firstNameList = listOf("Jean", "Pierre", "Clément")
6         val lastNameList = listOf("DUPOND", "DUPONT", "MARTIN")
7
8         fun getFirstName(): String {
9             return firstNameList[Random.nextInt(0, firstNameList.size)]
10        }
11
12        fun getLastName(): String {
13            return lastNameList[Random.nextInt(0, lastNameList.size)]
14        }
15
16        fun getRole(): Role {
17            return Role.values()[Random.nextInt(0, Role.values().size)]
```

Copier

Exercice récapitulatif

Solution proposée

```
7     companion object {
8         fun getPerson(): Person {
9             val typeId = Random.nextInt(0, 3)
10            return when (typeId) {
11                0 -> Friend.getFriend()
12                1 -> Colleague.geColleague()
13                2 -> Contact.getContact()
14                else -> Friend.getFriend()
15            }
16        }
17    }
18 }
19
20 class Friend private constructor(firstName: String, lastName: String, age: Short, var address: Address? = null):
21     Person(firstName, lastName) {
22         val age: Short by Delegates.vetoable(age) { property, oldValue, newValue ->
23             newValue in 0..MAX_AGE
```

[Copier](#)

Exercice récapitulatif

9 - Interface

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

```
1 interface JsonExport {  
2     fun toJson(): String  
3 }  
4  
5 sealed class Person(var firstName: String, val lastName: String): JsonExport {  
6     ....  
7 }
```

[Copier](#)

Exercice récapitulatif

Solution proposée

```
4 interface JsonExport {  
5     fun toJson(): String  
6 }  
7  
8 sealed class Person(var firstName: String, val lastName: String): JsonExport {  
9     abstract override fun toString(): String  
10  
11     companion object {  
12         fun getPerson(): Person {  
13             val typeId = Random.nextInt(0, 3)  
14             return when (typeId) {  
15                 0 -> Friend.getFriend()  
16                 1 -> Colleague.geColleague()  
17                 2 -> Contact.getContact()  
18                 else -> Friend.getFriend()  
19             }  
20         }  
21     }  
22 }
```

[Copier](#)

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

```
1 val firstNameCumulatedSize = randomPersonList.fold(0) { acc, person ->
2     acc + person.firstName.length
3 }
4 println("$firstNameCumulatedSize / ${randomPersonList.size} = ${firstNameCumulatedSize.toFloat() / randomPersonList.size.toFloat()}")
5
6 val ageCumulated = randomPersonList.fold(0) { acc, person ->
7     if(person is Friend) {
8         acc + person.age
9     } else {
10         acc
11     }
12 }
13 println("$ageCumulated")
```

[Copier](#)

Exercice récapitulatif

11 - Opérateur

Nous allons maintenant définir l'opérateur `+`, pour notre classe `Person` qui :

- Prend un prénom aléatoire.
- Concatène les deux noms.
- Définis un âge à 0.

Pour l'exemple, nous allons rendre public de nouveau le constructeur de l'objet `Friend`.

Exercice récapitulatif

Solution proposée

```
1 operator fun Person.plus(other: Person) = Friend(RandomValues.getFirstName(), "${this.lastName}-${other.lastName}", 0 )
```

[Copier](#)

Interopérabilité

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

Interopérabilité

- Interopérabilité avec Java.
- De Kotlin au Java.
- Nulls de Java.
- Le Kotlin dans Java.
- Extensions de fonctions à partir du 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 :

```
1 import java.util.*
2
3 fun demo(source: List<Int>) {
4     val list = ArrayList<Int>()
5     // 'for'-loops work for Java collections:
6     for (item in source) {
7         list.add(item)
8     }
9     print(list)
10
11     // Operator conventions work as well:
12     for (i in 0..source.size - 1) {
13         list[i] = source[i] // get and set are called
14     }
15     print(list)
16 }
```

Copier

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

Copier

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 :

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

[Copier](#)

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

[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 :

```
1 fun main() {  
2     val customer = Customer()  
3  
4     customer.firstName = "Frodo"  
5     customer.lastName = "Baggins"  
6  
7     println("${customer.firstName} ${customer.lastName}")  
8 }
```

Copier

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 :

```
1 private String firstName;
2
3 public String getFirstName() {
4     return firstName;
5 }
6
7 public void setFirstName(String firstName) {
8     this.firstName = firstName;
9 }
```

Copier

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 remplaçant 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.

```
1 var firstName = ""  
2 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 :

```
1 // app.kt
2 package org.example
3 class Util
4
5 fun getTime(): Int { println("10h21"); return 10 }
```

[Copier](#)

```
1 // Java
2 import org.example.AppKt;
3 import org.example.Util;
4
5 public class Test {
6
7     Util utilVar = new Util();
8     int timeValue = AppKt.getTime();
9 }
```

[Copier](#)

De Kotlin au Java

Nom de la classe Java générée

Nous pouvons changer le nom de la classe Java généré en utilisant l'annotation `@JvmName` :

```
1 @file:JvmName("DemoUtils")
2
3 package org.example
4
5 class Util
6
7 fun getTime() { /*...*/ }
```

[Copier](#)

```
1 // Java
2 new org.example.Util();
3 org.example.DemoUtils.getTime();
```

[Copier](#)

De Kotlin au Java

Les champs d'instances

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

```
1 class User(id: String) {  
2     @JvmField val ID = id  
3 }
```

[Copier](#)

```
1 // Java  
2 class JavaClient {  
3     public String getID(User user) {  
4         return user.ID;  
5     }  
6 }
```

[Copier](#)

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

De Kotlin au Java

Champs statiques (statics)

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

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

De Kotlin au Java

statics / @JvmField

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

```
1 class Key(val value: Int) {  
2     companion object {  
3         @JvmField  
4         val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }  
5     }  
6 }
```

[Copier](#)

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

[Copier](#)

De Kotlin au Java

statics / lateinit

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

```
1 object Singleton {  
2     lateinit var provider: Provider  
3 }
```

[Copier](#)

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

[Copier](#)

De Kotlin au Java

statics / const

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

```
1 // file example.kt
2 object Obj {
3     const val CONST = 1
4 }
5
6 class C {
7     companion object {
8         const val VERSION = 9
9     }
10 }
11
12 const val MAX = 239
```

[Copier](#)

```
1 // Java
2 int const = Obj.CONST;
3 int max = ExampleKt.MAX;
4 int version = C.VERSION;
```

[Copier](#)

Surcharges

De Kotlin au Java

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

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

Exemple :

```
1 class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {  
2     @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*...*/ }  
3 }
```

[Copier](#)

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

```
1 // Constructors:  
2 Circle(int centerX, int centerY, double radius)  
3 Circle(int centerX, int centerY)  
4  
5 // Methods  
6 void draw(String label, int lineWidth, String color) { }  
7 void draw(String label, int lineWidth) { }  
8 void draw(String label) { }
```

[Copier](#)

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 :

```
1 // Java
2 public class Nullable {
3
4     public String get(){
5         return null;
6     }
7 }
```

[Copier](#)

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

[Copier](#)

Que se passe-t-il quand on lance le code ?

Que pouvons-nous faire pour éviter l'erreur ?

Nulls de Java

Solution

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

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 :

```
1 // Shape.kt
2 class Shape(var width: Int, var height: Int, val shape: String) {
3     var area: Int = 0
4     fun shapeMessage() {
5         println("Hi i am $shape, how are you doing")
6     }
7     fun draw() {
8         println("$shape is drawn")
9     }
10    fun calculateArea(): Int {
11        area = width * height
12        return area
13    }
14 }
```

[Copier](#)

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 :

```
1 public class FromKotlinClass {
2     public static void callShapeInstance() {
3         Shape shape = new Shape(5,5,"Square");
4         shape.shapeMessage();
5         shape.setHeight(10);
6         System.out.println(shape.getShape() + " width " + shape.getWidth());
7         System.out.println(shape.getShape() + " height " + shape.getHeight());
8         System.out.println(shape.getShape() + " area " + shape.calculateArea());
9         shape.draw();
10    }
11    public static void main(String[] args) {
12        callShapeInstance();
13    }
14 }
```

[Copier](#)

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` :

```
1 // Kotlin
2 object Singleton {
3     fun happy() {
4         println("I am Happy")
5     }
6 }
```

Copier

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

```
1 // Java
2 public static void main(String args[]) {
3     Singleton.INSTANCE.happy();
4 }
```

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` :

```
1 object Singleton {  
2  
3     fun happy() {  
4         println("I am Happy")  
5     }  
6  
7     @JvmStatic fun excited() {  
8         println("I am very Excited")  
9     }  
10 }
```

Copier

Ce qui donnera l'appel en Java :

```
1 public static void main(String args[]) {  
2     Singleton.INSTANCE.happy();  
3     Singleton.excited();  
4 }
```

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` :

```
1 fun logD(message: String) {  
2     Log.d("", message)  
3 }  
4 fun logE(message: String) {  
5     Log.e("", message)  
6 }
```

[Copier](#)

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

```
1 UtilsKt.logD("Debug");  
2 UtilsKt.logE("Error");
```

[Copier](#)

Extensions de fonctions à partir du Java

Exemple d'extension

Supposons que nous avons le code suivant :

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

[Copier](#)

Que l'on pourrait appeler en Kotlin :

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

[Copier](#)

En Java cela deviendra :

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

[Copier](#)

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

Java Réflexion avec Kotlin

Exercice

Reprenons notre classe Java Customer :

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

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 :

```
1 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 :

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

[Copier](#)

Standard Library

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

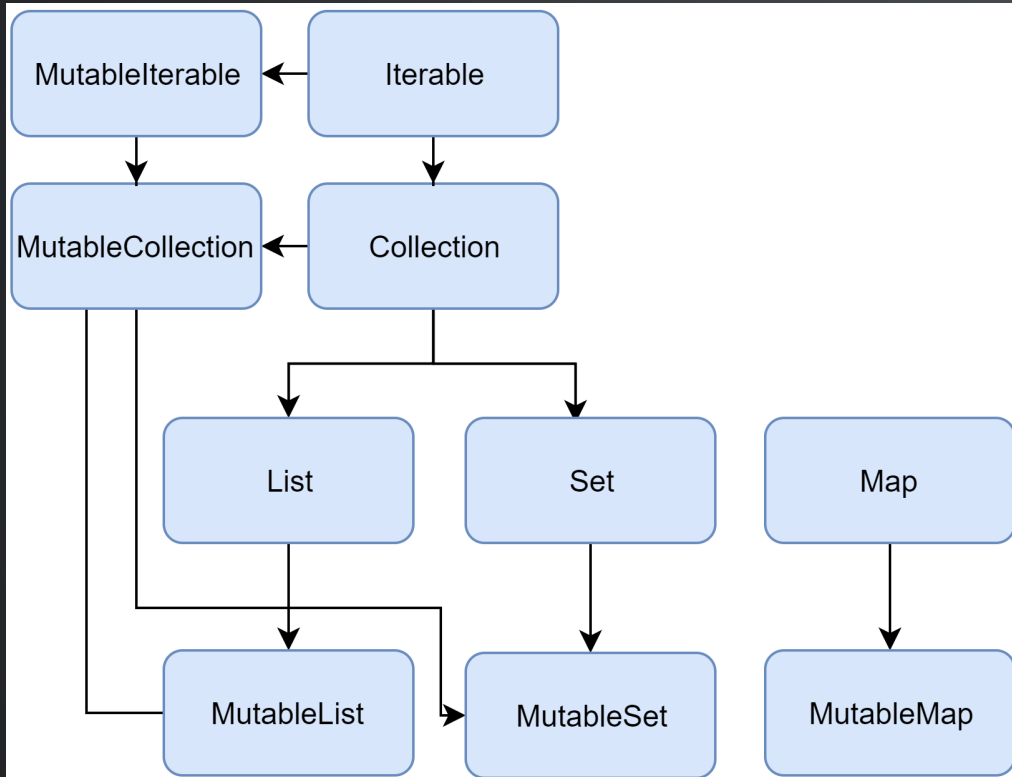
Kotlin Standard Library et collections dans Kotlin

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 définit les opérations pour itérer sur les éléments. C'est le type à utiliser pour gérer les différents types de collections. Dans les cas plus précis, préférer `List` ou `Set`.

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

Copier

Kotlin Standard Library et collections dans Kotlin

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

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

Copier

Kotlin Standard Library et collections dans Kotlin

Différences et similitudes entre `List<T>` et `Array<T>`

Exemples d'utilisation :

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

Copier

Les deux semblent similaires, ...

Kotlin Standard Library et collections dans Kotlin

Définition de `List<T>`

Une liste est une interface. C'est une collection générique et ordonnée d'éléments. Les méthodes dans cette interface permettent un accès en lecture seule aux éléments.

Les opérations de lecture et écriture sont définies dans l'interface `MutableList`.

Kotlin Standard Library et collections dans Kotlin

Définition de Array<T>

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

Les tableaux peuvent être concaténés pour donner un nouveau tableau :

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

[Copier](#)

Kotlin Standard Library et collections dans Kotlin

Les similitudes

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

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

[Copier](#)

Notons que les listes mutables peuvent changer de taille.

2. Les 2 permettent de stocker plusieurs éléments et font partis de la famille des collections :

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

[Copier](#)

Kotlin Standard Library et collections dans Kotlin

Les différences

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

```
1 val array = arrayOf(1, 2, 3)
2 array[2] = 4 // OK
3 for (element in array){
4     println("$element, ") // 1, 2, 4
5 }
6 val list = listOf("apple", "ball", "cow")
7 list[2] = "cat" //will not compile, lists are immutable
8 // println(list) = {apple, ball, cow}
9
10 val m = mutableListOf(1, 2, 3)
11 m[0] = m[1] // OK
```

[Copier](#)

Kotlin Standard Library et collections dans Kotlin

Les différences (suite)

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

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

[Copier](#)

Kotlin Standard Library et collections dans Kotlin

Les différences (suite)

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

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

Copier

Kotlin Standard Library et collections dans Kotlin

Les différences (suite)

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

Filtering, Mapping et Flatmapping en Kotlin

Entraînement en ligne

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

Kotlin lazy evaluation

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

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

[Copier](#)

Que remarquez-vous ?

Changez la valeur du `println` en :

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

[Copier](#)

Quel changement observez-vous, et pourquoi ?

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.

Programmation asynchrone

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

Le problème de la programmation asynchrone

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

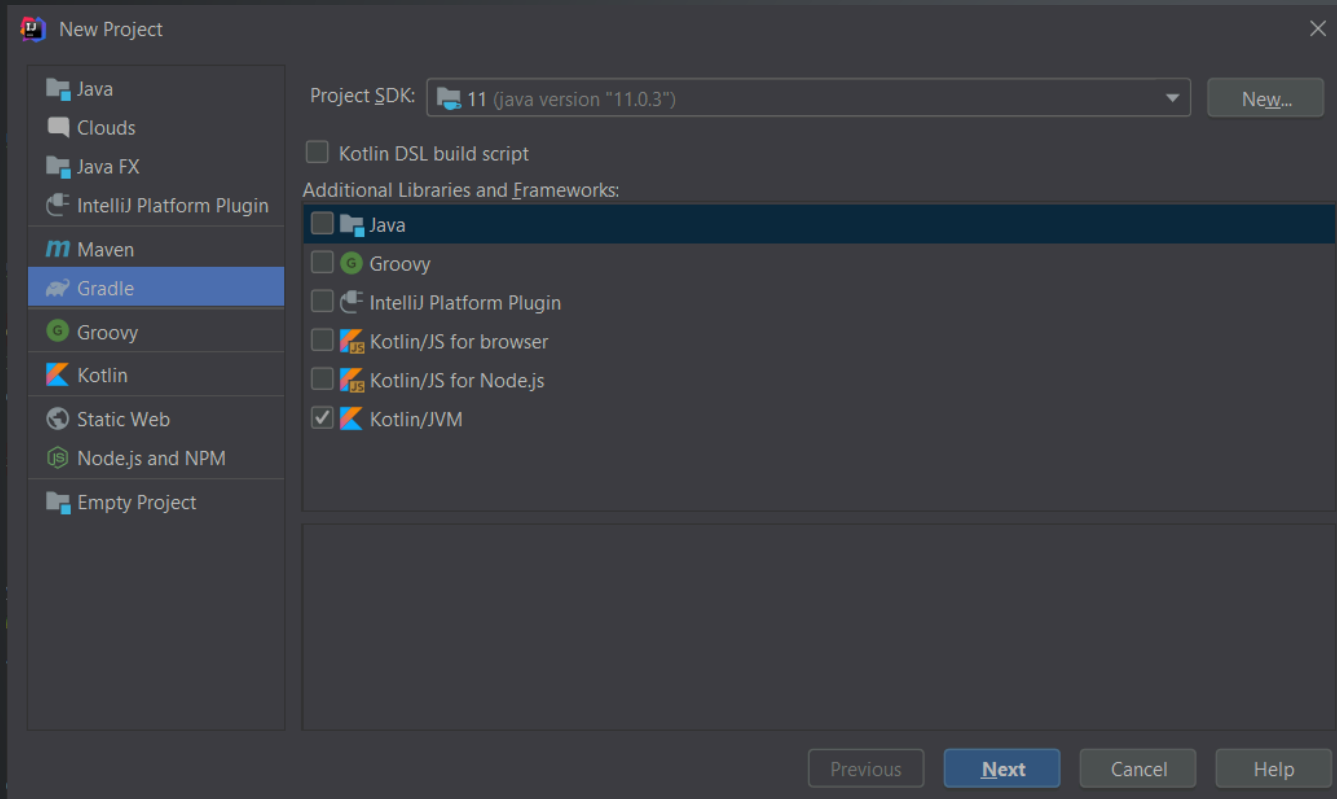
Plusieurs approches sont possibles pour résoudre ce problème :

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

Mise en place du projet

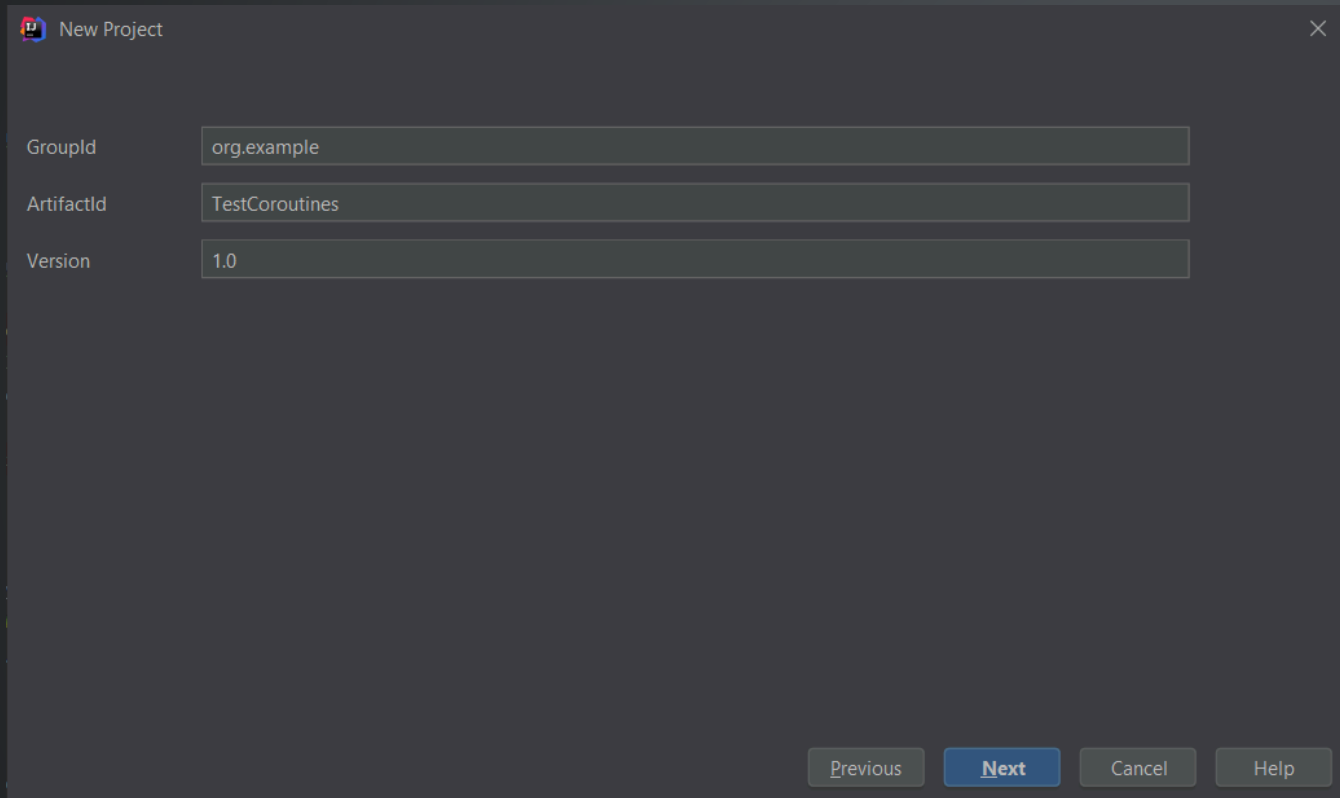
Coroutines en Kotlin

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



Coroutines en Kotlin

Nommage du module



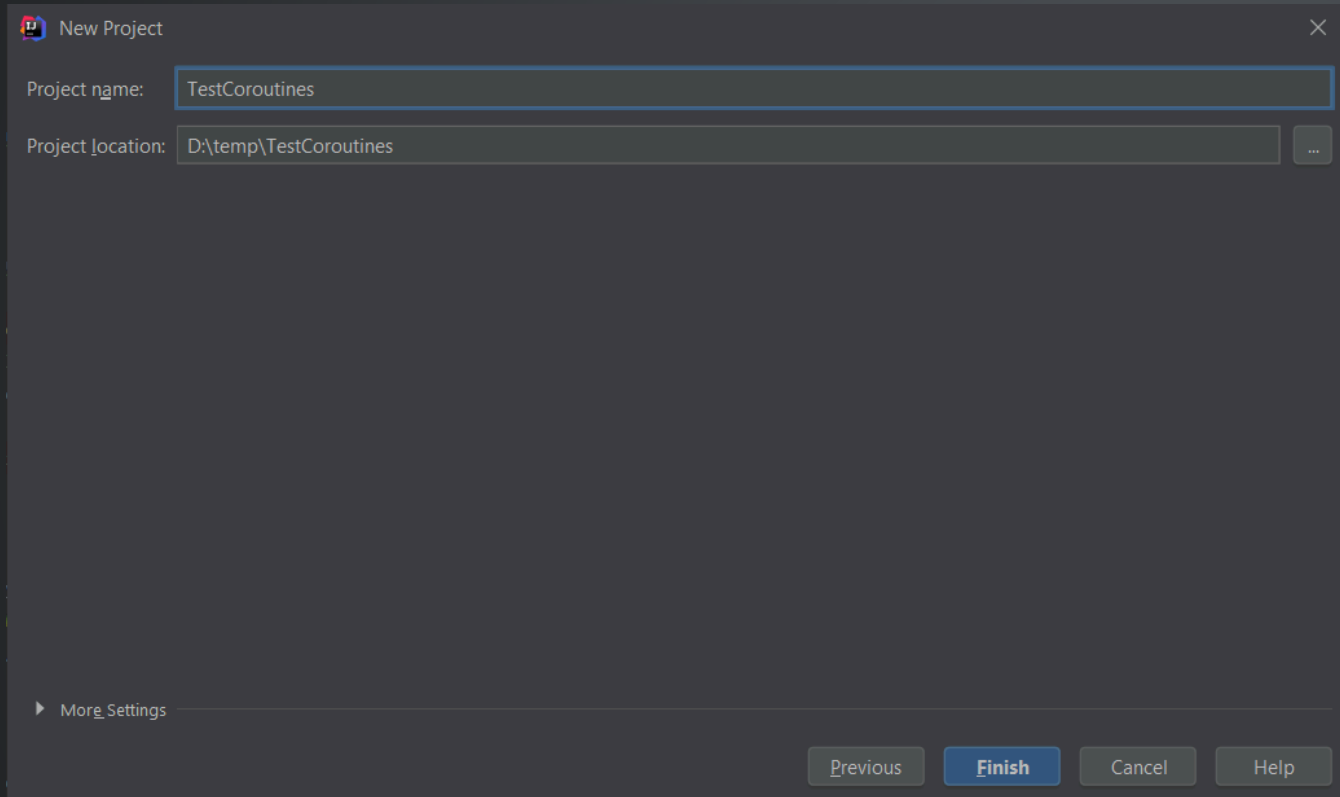
The screenshot shows the 'New Project' dialog box with the following fields and values:

Field	Value
GroupId	org.example
ArtifactId	TestCoroutines
Version	1.0

At the bottom of the dialog, there are four buttons: 'Previous', 'Next' (highlighted in blue), 'Cancel', and 'Help'.

Nommage du projet

Coroutines en Kotlin



New Project

Project name: TestCoroutines

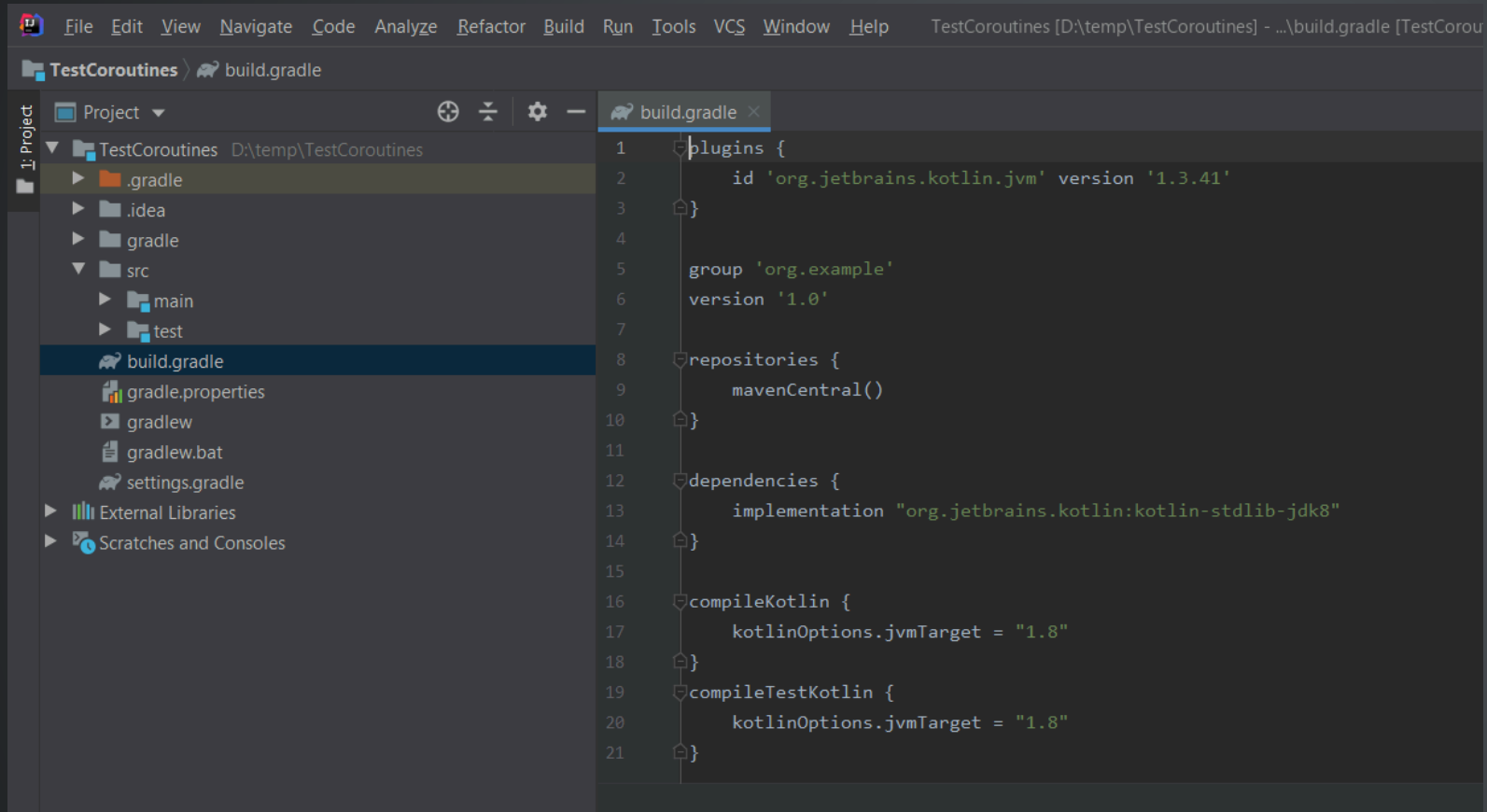
Project location: D:\temp\TestCoroutines

More Settings

Previous Finish Cancel Help

Coroutines en Kotlin

Ouvrez le fichier build.gradle



Coroutines en Kotlin

Ajoutez la dépendance

```
1 dependencies {  
2     ...  
3     implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.1")  
4 }
```

[Copier](#)

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 {}` :

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

[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 :

```
1 println("Start")
2
3 // Start a coroutine
4 GlobalScope.launch {
5     delay(1000)
6     println("Hello")
7 }
8
9 Thread.sleep(2000) // wait for 2 seconds
10 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

Coroutines en Kotlin

Blocage du thread principal

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

```
1 runBlocking {  
2     delay(2000)  
3 }
```

[Copier](#)

Ce qui nous donnera au final :

```
1 import kotlinx.coroutines.GlobalScope  
2 import kotlinx.coroutines.delay  
3 import kotlinx.coroutines.launch  
4 import kotlinx.coroutines.runBlocking  
5  
6 fun main() {  
7     println("Start")  
8  
9     // Start a coroutine  
10    GlobalScope.launch {  
11        delay(1000)  
12        println("Hello")  
13    }  
14  
15    // Step 1  
16    //Thread.sleep(2000) // wait for 2 seconds  
17 }
```

[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 :

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

Copier

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

Coroutines en Kotlin

Lançons beaucoup de thread

Écrivez le même code avec des coroutines :

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

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

```
1 val deferred = (1L..1_000_000L).map { n ->
2     GlobalScope.async {
3         n
4     }
5 }
6 val sum = deferred.sumBy { it.await() }
7 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 {}`

```
1 val deferred = (1L..1_000_000L).map { n ->
2     GlobalScope.async {
3         n
4     }
5 }
6 runBlocking {
7     val sum: Long = deferred.sumOf { it.await() }
8     println("Sum: $sum")
9 }
```

[Copier](#)

Nous devrions obtenir le résultat suivant :

```
1 Sum: 1784293664
```

[Copier](#)

Async et Await en Kotlin

Parallèle

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

```
1 val deferred = (1..1_000_000).map { n ->
2     GlobalScope.async {
3         delay(1000)
4         n
5     }
6 }
7 runBlocking {
8     val sum = deferred.sumOf { it.await() }
9     println("Sum: $sum")
10 }
```

[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 :

```
1 import kotlinx.coroutines.GlobalScope
2 import kotlinx.coroutines.async
3 import kotlinx.coroutines.runBlocking
4
5 fun main() {
6
7     suspend fun workload(n: Long): Long {
8         //delay(3000)
9         return n
10    }
11
12    val deferred = (1..1_000_000).map { n ->
13        GlobalScope.async {
14            workload(n)
15        }
16    }
17    runBlocking {
```

Copier

Async et Await en Kotlin

Code final version bis

Nous obtenons le code final, suivant :

```
1 import kotlinx.coroutines.GlobalScope
2 import kotlinx.coroutines.async
3 import kotlinx.coroutines.runBlocking
4
5 fun main() {
6
7     val deferred: List<Deferred<Long>> = (1..1_000_000L).map { n ->
8         GlobalScope.async { n }
9     }
10
11     runBlocking {
12         val sum = deferred.sumOf { it.await() }
13         println("Sum: $sum")
14     }
15 }
```

Copier

Yield en Kotlin

Construction de séquences

Nous allons construire une séquence, de manière paresseuse, par exemple la suite de fibonacci en utilisant la fonction `sequence` qui utilise `yield`:

```
1 fun main(args: Array<String>) {  
2     val fibonacciSeq = sequence {  
3         var a = 0  
4         var b = 1  
5  
6         yield(1)  
7  
8         while (true) {  
9             yield(a + b)  
10  
11             val tmp = a + b  
12             a = b  
13             b = tmp  
14         }  
15     }  
16  
17     // Print the first five Fibonacci numbers
```

[Copier](#)

Ce qui devrait nous afficher :

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

[Copier](#)

Yield en Kotlin

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

```
1 fun main(args: Array<String>) {  
2     val lazySeq = sequence {  
3         print("START ")  
4         for (i in 1..5) {  
5             yield(i)  
6             print("STEP ")  
7         }  
8         print("END")  
9     }  
10  
11     // Print the first three elements of the sequence  
12     lazySeq.take(3).forEach { print("$it ") }  
13 }
```

[Copier](#)

1. Combien de fois est affiché le message END ?
2. Que faudrait-il faire pour qu'il s'affiche ?
3. Que se passe-t-il si l'on demande par exemple 10 éléments de la séquence ?

Yield en Kotlin

Generation complète de la séquence

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

```
1 fun main(args: Array<String>) {  
2     val lazySeq = sequence {  
3         yield(0)  
4         yieldAll(1..10)  
5     }  
6  
7     lazySeq.forEach { print("$it ") }  
8 }
```

[Copier](#)

Yield en Kotlin

Génération filtrée de la séquence

Il est possible de décomposer notre génération de séquence :

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

[Copier](#)

Yield en Kotlin

Filtrer une séquence

Nous pouvons filter une séquence en utilisant la méthode `filter`. Par exemple prenons les 8 premières valeurs de la suite de fibonacci qui sont supérieures à 20 :

```
3      var a = 0
4      var b = 1
5
6      yield(1)
7
8      while (true) {
9          yield(a + b)
10
11         val tmp = a + b
12         a = b
13         b = tmp
14     }
15 }
16
17 // Print the first five Fibonacci numbers
18 println(fibonacciSeq.filter { it > 20 }.take(8).toList())
19 }
```

[Copier](#)

Yield en Kotlin

Exercice

- Reprenons l'exercice sur les personnes.
- Créons une suite (sequence) de personnes, en appelant la fabrique.
- Affichons les 10 premières personnes.
- Avec un prénom de 4 lettres.

Yield en Kotlin

Solution proposée

```
1 val personSequence = sequence {  
2     while (true)  
3         yield(Person.getPerson())  
4 }  
5  
6 fun main() {  
7     println(personSequence.filter { it.firstName.length == 4 }.take(10).toList())  
8 }
```

[Copier](#)

Reactive extension en Kotlin

Librairies

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

- [Présentation de RxKotlin](#)
- [Présentation du projet Reactor](#)

Bonnes et mauvaises pratiques

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

```
1  override fun onDestroy() {  
2      super.onDestroy()  
3      async.cancelAll()  
4  }
```

Copier

- Utilisez les coroutines, uniquement quand cela est nécessaire.

Scripts en Kotlin

Kotlin peut être utilisé pour écrire des scripts. Exemple :

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

Copier

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

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

Copier

Les fonctions inline

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

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

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

[Copier](#)

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

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

[Copier](#)

Les fonctions inline

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

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

[Copier](#)

Ce modifieur va impacter la fonction, en elle-même, mais aussi la lambda passée en paramètre : tout sera mis en ligne à l'endroit de l'appel.

Utiliser cette technique pourra augmenter la taille du code compilé, il faut donc faire attention à son usage (éviter d'inliner des grosses fonctions), mais l'on gagnera en performances, particulièrement lors d'appel dans des boucles.