

Kotlin

Par Omar Thioune

Présentation

- Kotlin est un langage de programmation créé par JetBrains en 2011 qui peut être exécuté en Java ou être compilé en JavaScript. Il a été fait afin d'avoir une version plus expressive que Java, donc du code Kotlin et Java peuvent exister dans la même application. En 2017, il a été pris en charge par Google afin d'être un support pour Android.

Comment en faire son usage

- Kotlin est le langage principal utilisé pour créer des applications mobiles Android.
- Kotlin est utilisé pour le développement backend
- Kotlin permet de créer des applications qui partagent du code entre différentes plateformes, comme Android, iOS, desktop et le web via kotlin multiplatform.
- Kotlin peut être compilé en JavaScript, ce qui permet de l'utiliser pour le développement front-end en complément des frameworks comme React.

- Kotlin est reconnu pour sa concision et sa clarté, ce qui permet de réduire la quantité de code nécessaire tout en intégrant des fonctionnalités puissantes comme les lambdas, les types nullable (qui aident à éviter les erreurs liées aux valeurs nulles), et les extensions de fonctions. Le langage est également conçu pour favoriser une programmation plus sécurisée, grâce à ses mécanismes de gestion des erreurs intégrés. Bien que Kotlin soit principalement utilisé dans le développement d'applications Android, il s'étend bien au-delà. Il est tout aussi efficace pour le développement de serveurs, l'écriture de scripts ou encore la création d'applications multiplateformes via Kotlin/Native.


Pourquoi Kotlin au lieu de java

- Kotlin présente plusieurs avantages par rapport à Java, notamment :
- **Concison et lisibilité** : Kotlin permet de réduire la quantité de code nécessaire pour accomplir une tâche, rendant le code plus facile à lire et à maintenir. Par exemple, la syntaxe pour la déclaration de variables est plus simple et directe.
- **Sécurité** : Kotlin a été conçu pour éviter les erreurs courantes rencontrées en Java, notamment les erreurs liées aux valeurs nulles. En Kotlin, les variables sont par défaut non nullables, sauf si spécifié autrement avec ?. Cela permet de réduire le nombre de bogues liés à la gestion des références nulles.
- **Interopérabilité avec Java** : Kotlin est complètement compatible avec Java. Il est donc possible de mélanger du code Kotlin et Java dans une même application. Cela facilite la migration des projets Java existants vers Kotlin sans avoir à réécrire tout le code.

Les variables

- En Kotlin, deux mots-clés principaux sont utilisés : **val** et **var**. Le mot-clé **val** crée des variables immuables, équivalentes à des constantes. Une fois leur valeur assignée, elle ne peut plus être modifiée, ce qui garantit leur immutabilité. À l'inverse, **var** est utilisé pour des variables mutables, dont la valeur peut être modifiée à tout moment.
- La syntaxe est simple : `val nom = "Omar"` ou `var âge = 19`. Kotlin utilise l'inférence de type, ce qui signifie que le type de la variable est déduit automatiquement. Cependant, il est possible de spécifier explicitement le type : `var taille: Double = 1.88`.
- Les variables peuvent également être nullables en utilisant le symbole `?` : `var adresse: String? = null`. Cela favorise la gestion de la nullité dans Kotlin.

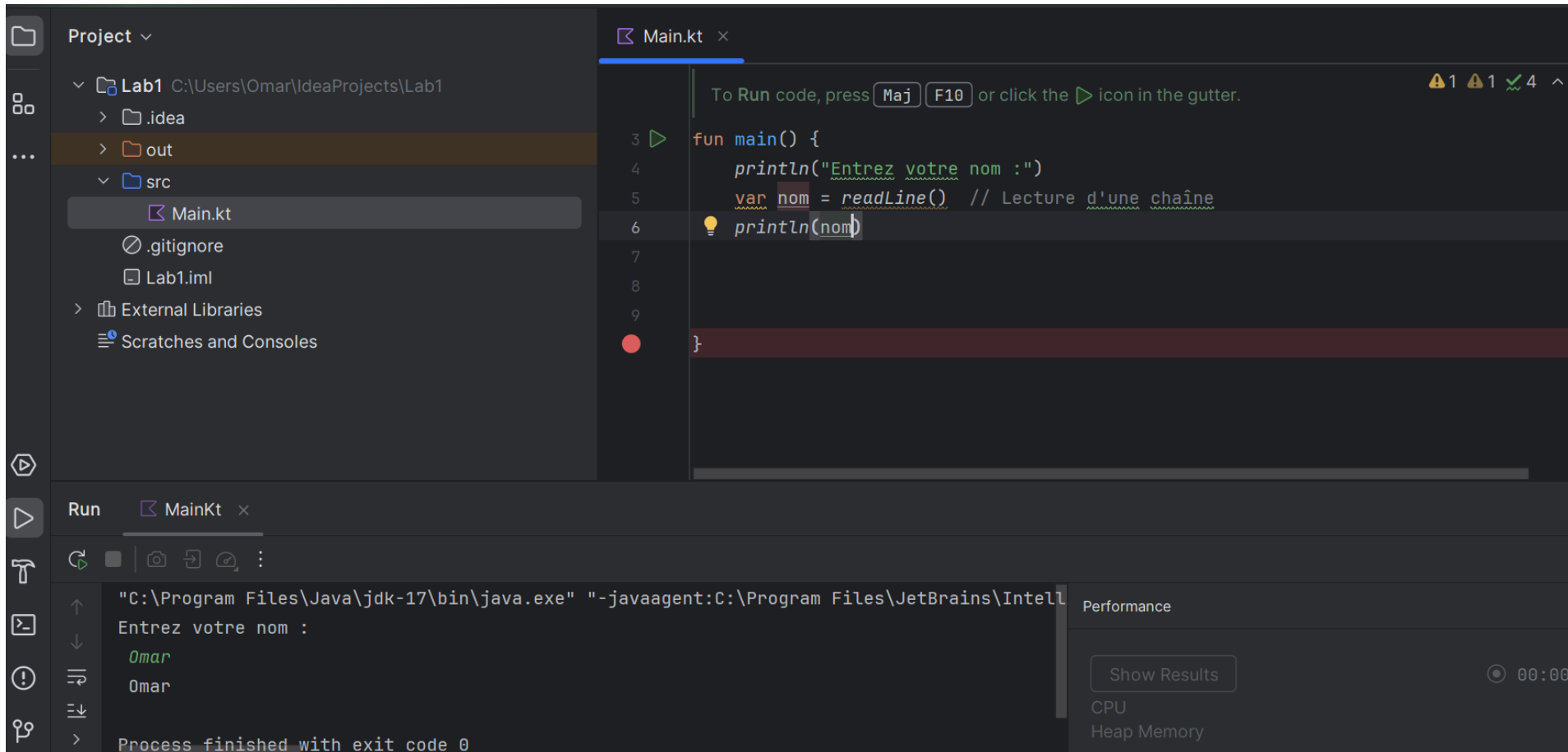
Exemple déclaration de variable

```
fun main() {  
    val nom = "Omar"  
    var age = 5  
    var nombre: Int = 10  
    var texte: String = "Bonsoir"  
    var reponse: Boolean = true  
     var taille: Double = 1.88  
  
}
```

Saisir les variables

- En Kotlin, la saisie de variables s'effectue principalement à l'aide de la fonction **readLine()**, qui permet de lire l'entrée utilisateur sous forme de chaîne de caractères. Cependant, comme cette fonction retourne toujours un type **String**, il est souvent nécessaire de convertir ces données dans le type désiré pour les manipulations ultérieures.
- Pour ce faire, Kotlin offre diverses méthodes de conversion qui permettent de transformer une chaîne de caractères en d'autres types de données plus spécifiques. Par exemple :
- **toInt()** : Convertit la chaîne en un entier. Attention, si la chaîne ne représente pas un nombre entier valide, cela peut provoquer une exception **NumberFormatException**.
- **toFloat()** : Convertit la chaîne en un nombre à virgule flottante (décimal).
- **toDouble()** : Similaire à **toFloat()**, mais avec une précision double.
- **toBoolean()** : Convertit la chaîne en une valeur booléenne (vrai ou faux). Les chaînes "true" ou "false" (non sensibles à la casse) sont reconnues, sinon la valeur par défaut est **false**.
- **toString()** : Bien que les entrées via **readLine()** soient déjà des chaînes, cette méthode est utile lorsqu'on veut explicitement convertir un autre type de données en chaîne de caractères.

Exemple saisie de variable



Concatenation et opérations arithmétiques

- Nous allons voir comment faire une concatenation des variables qui se font avec avec le \$ et les opérations arithmétique qui sont similaires à ceux de java

Exemple code

```
fun main() {  
    val numero1 = 15.05  
    val numero2 = 12.06  
    // addition  
    var resultat = numero1 + numero2  
    println("$numero1 + $numero2 = $resultat")  
    // soustraction  
    resultat = numero1 - numero2  
    println("$numero1 - $numero2 = $resultat")  
    // multiplacation  
    resultat = numero1 * numero2  
    println("$numero1 * $numero2 = $resultat")  
  
    resultat = numero1 / numero2  
    println("$numero1 / $numero2 = $resultat")  
}
```

Exemple resultat

```
15.05+ 12.06 = 27.11  
15.05 - 12.06 = 2.99  
15.05*12.06 = 181.503000000000001  
15.05 / 12.06 = 1.247927031509121  
  
Process finished with exit code 0
```

Les boucles

- Les boucles kotlin sont similaires à java pour la plupart du temps, mais il y a certaines exclusivité quant à la possibilité de parcourir une plage de nombre ou un tableau.
- Mettre deux petits points afin de parcourir une suite de nombre
- Utilise until afin d'arrêter avant le nombre qu'on spécifie
- downTo pour parcourir le tableau à reculons
- Step pour parcourir par le nombre spécifié

Exemple

```
Project: Lab1 C:\Users\Omar\IdeaProjects\Lab1
  > .idea
  > out
  > src
    Main.kt
    .gitignore
    Lab1.iml
  > External Libraries
  > Scratches and Consoles

To Run code, press Maj F10 or click the > icon in the gutter.

3 fun main() {
4   for (i in 1..5) print(i) // Affiche 1 à 5
5   println("")
6   for (i in 1..until(5)) print(i) // Affiche 1 à 4
7   println("")
8   for (i in 5 downTo 1) print(i) // Compte à rebours
9   println("")
10  for (i in 1..10 step 2) print(i) // Pas de 2
11
12
13 }
```

Run MainKt

"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I..."

12345
1234
54321
13579
Process finished with exit code 0

Performance

Show Results 00:00

Process Terminated

Les fonctions

- En Kotlin, les fonctions sont des blocs de code réutilisables qui permettent d'effectuer une tâche spécifique. Pour définir une fonction, il est nécessaire d'utiliser le mot-clé **fun**, suivi du nom de la fonction. Le nom de la fonction doit être choisi de manière descriptive afin de refléter clairement son rôle dans le programme.
- Ensuite, entre parenthèses, tu dois spécifier les paramètres que la fonction va accepter, chaque paramètre étant défini par un nom suivi de son type. Le type est indiqué après le nom du paramètre en utilisant le symbole deux-points (:). Il est important de noter que les types en Kotlin sont obligatoires pour les paramètres afin d'assurer une meilleure sécurité de type à la compilation.
- Après les parenthèses, il faut également indiquer le type de retour de la fonction. Le type de retour est la valeur que la fonction va produire après son exécution. Comme pour les paramètres, le type de retour est spécifié après le symbole deux-points (:), juste après les parenthèses fermées. Si la fonction ne retourne aucune valeur, on utilise le type **Unit**, qui est l'équivalent de `void` dans d'autres langages.

Example

```
Lab1 C:\Users\Omar\IdeaProjects\Lab1
├── .idea
├── out
├── src
│   └── Main.kt
├── .gitignore
└── Lab1.iml
External Libraries
Scratches and Consoles
```

To Run code, press **May** **F10** or click the **▶** icon in the gutter.

```
3 ▶ fun main() {
4
5
6     println(saluer(nom: "Omar"))
7
8
9 }
10
11 fun saluer(nom: String): String {
12     return "Bonjour, $nom!"
13 }
```

Run MainKt x

↑ "C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I Performance
↓ Bonjour, Omar!
—

Show Results

POO

- Maintenant on va observer comment faire de la POO en Kotlin qui a quelques différences avec java
- Classe et objets
- Héritage
- Private protected public
- Le polymorphisme

Les classes

- Alors la création de classe en kotlin est assez similiaire, cependant au niveau des getter et des setters Kotlin va se démarquer pour la simplicité de son code. Nous utiliser field afin de récupérer notre valeur et le set n'a pas autant de code à faire afin de le remplir

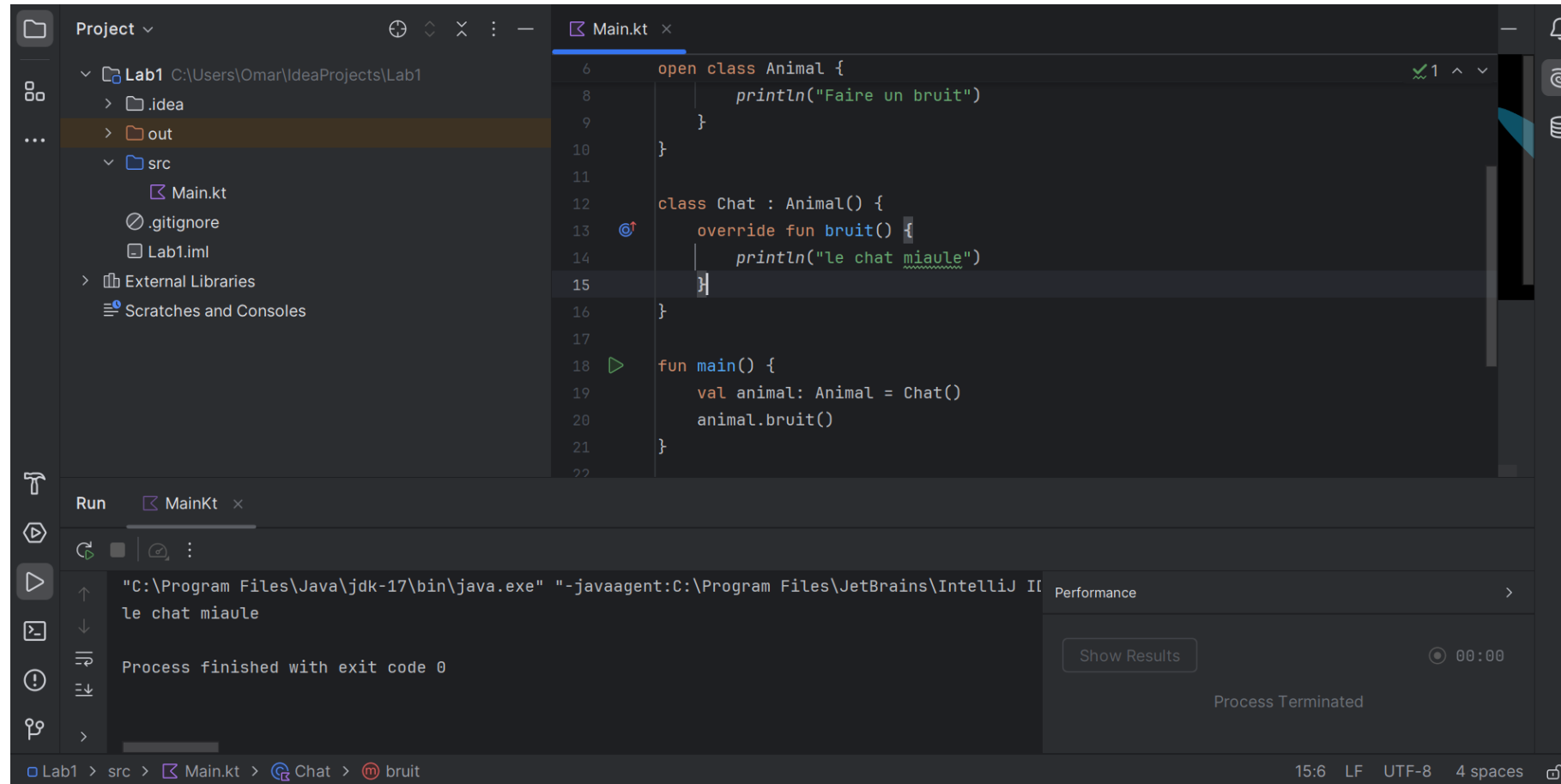
Exemple classes

```
class etudiant {  
    var nom: String = ""  
    get() = field  
    set(nom) {  
        field = nom  
    }  
  
    var age: Int = 0  
    get() = field  
    set(age) {  
        field = age  
    }  
}
```

Polymorphisme

- Le polymorphisme en Java repose sur les interfaces et les classes abstraites. Vous pouvez utiliser une référence de type parent pour désigner des objets de type enfant. Cela permet d'exécuter des méthodes redéfinies dans les classes enfants tandis qu'en Kotlin, toutes les sont final par défaut, ce qui signifie qu'elles ne peuvent pas être héritées à moins d'être marquées avec le mot-clé open

Exemple polymorphisme kotlin



Les héritages

- En Java, l'héritage est explicite avec le mot-clé `extends` pour les classes et `implements` pour les interfaces. En Kotlin, pour permettre l'héritage, les classes doivent être marquées avec `open` (par défaut, les classes sont `final`), et les interfaces sont déclarées avec `interface`.

Exemple héritage

```
open class Personnage(val nom: String, val pointsDeVie: Int) {
    open fun attaquer() {
        println("$nom attaque avec une arme.")
    }
}

class Guerrier(nom: String, pointsDeVie: Int) : Personnage(nom, pointsDeVie) {
    override fun attaquer() {
        println("$nom attaque avec une épée !")
    }
}

class Magicien(nom: String, pointsDeVie: Int) : Personnage(nom, pointsDeVie) {
    override fun attaquer() {
        println("$nom attaque avec des sorts magiques !")
    }
}

fun main() {
    val guerrier = Guerrier(nom: "Aragorn", pointsDeVie: 100)
    val magicien = Magicien(nom: "Gandalf", pointsDeVie: 80)

    guerrier.attaquer()
    magicien.attaquer()
}
```

Exemple resultat

```
Aragorn attaque avec une épée !  
Gandalf attaque avec des sorts magiques !
```


Exemple interface

```
interface Deplacable {  
    fun deplacer()  
}  
  
class Vehicule(val nom: String) : Deplacable {  
    override fun deplacer() {  
        println("$nom se déplace sur la route.")  
    }  
}  
  
class Hero(val nom: String) : Deplacable {  
    override fun deplacer() {  
        println("$nom court vers l'ennemi !")  
    }  
}  
  
fun main() {  
    val voiture = Vehicule(nom: "Voiture rapide")  
    val heros = Hero(nom: "Link")  
    voiture.deplacer()  
    heros.deplacer()  
}
```

resultat

```
Voiture rapide se déplace sur la route.  
Link court vers l'ennemi !
```

DATA CLASS

Data class est une fonctionnalité de kotlin qui permet de contenir des objets contenant des données. Cependant, data class contient déjà des fonctions utiles comme `toString()`, `equals()`, `hashCode()`, et `copy()` qui simplifie le code lorsqu'on essaye de manipuler des objets.

Lorsque que tu utilises uniquement `class`, ces quatre fonctions ne sont pas automatiquement créées, le `Copy` n'est pas disponible et devra être écrit par l'utilisateur et lorsqu'on voudra comparer les objets, ce sera leurs adresses mémoires qui seront comparées et non leurs propriétés.

Syntaxe d'un data class

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

fun main() {

}
```

ToString

- **toString()** : La fonction **toString()** est automatiquement générée pour fournir une représentation en chaîne de caractères de l'objet. Contrairement à la version par défaut de cette fonction dans les autres classes, qui affiche généralement une chaîne cryptique représentant l'adresse mémoire de l'objet, la version générée dans une **data class** renvoie une chaîne lisible et détaillée qui liste les propriétés de l'objet avec leurs valeurs respectives. Cela est très utile pour le débogage et l'affichage d'objets de manière compréhensible.

Equals

- **equals()** : La fonction **equals()** permet de comparer deux objets pour vérifier s'ils sont équivalents. Par défaut, l'égalité d'objets en Kotlin (comme dans la plupart des langages) compare les adresses mémoires, c'est-à-dire si deux objets sont littéralement le même objet en mémoire. Cependant, dans une **data class**, **equals()** compare les propriétés des deux objets pour déterminer s'ils sont égaux. Si toutes les propriétés ont les mêmes valeurs, les objets sont considérés comme égaux.

Copy

- **copy()** : La fonction **copy()** est l'une des fonctionnalités les plus puissantes des **data classes** en Kotlin. Elle permet de créer une nouvelle instance de l'objet avec les mêmes valeurs de propriétés que l'original, tout en te donnant la possibilité de modifier certaines propriétés si nécessaire. Cela facilite la création de nouvelles instances d'un objet dérivées d'un existant, sans avoir à répéter tout le processus de création manuelle.

Hash

- **hashCode()** : La fonction **hashCode()** génère un code de hachage pour l'objet, qui est un entier représentant l'état interne de l'objet. Ce code de hachage est généralement utilisé lorsque les objets sont stockés dans des structures de données comme des **HashMap** ou des **HashSet**, qui s'appuient sur le code de hachage pour organiser les objets en mémoire de manière efficace.
- En utilisant une **data class**, Kotlin génère un **hashCode()** basé sur les propriétés de la classe, ce qui garantit que deux objets ayant les mêmes propriétés auront le même code de hachage, et donc seront traités de manière égale dans les collections de type hash.

Dataclass

- Les fonctions utiles :
- ToString = envoie la chaine de caractères de l'objet
- Equals = renvoie un boolean selon si deux objets sont égaux
- Copy = copie l'objet à une autre
- Hash code = renvoie un code

Example

The screenshot displays the IntelliJ IDEA IDE interface. On the left, the Project tool window shows a project named 'Lab1' with a source file 'Main.kt'. The main editor window shows the following Kotlin code:

```
4  
5 fun main() {  
6  
7     val Humain1 = Humain( name: "Omar", age: 19)  
8  
9     println("Option toString = " + Humain1.toString());  
10  
11     val Humain2 = Humain1.copy()  
12  
13     println("Option Copy = " + Humain2.toString());  
14  
15     println("Option hashCode = " + Humain1.hashCode());  
16  
17     println("Option equals = " + Humain2.equals(Humain1));  
18  
19 }
```

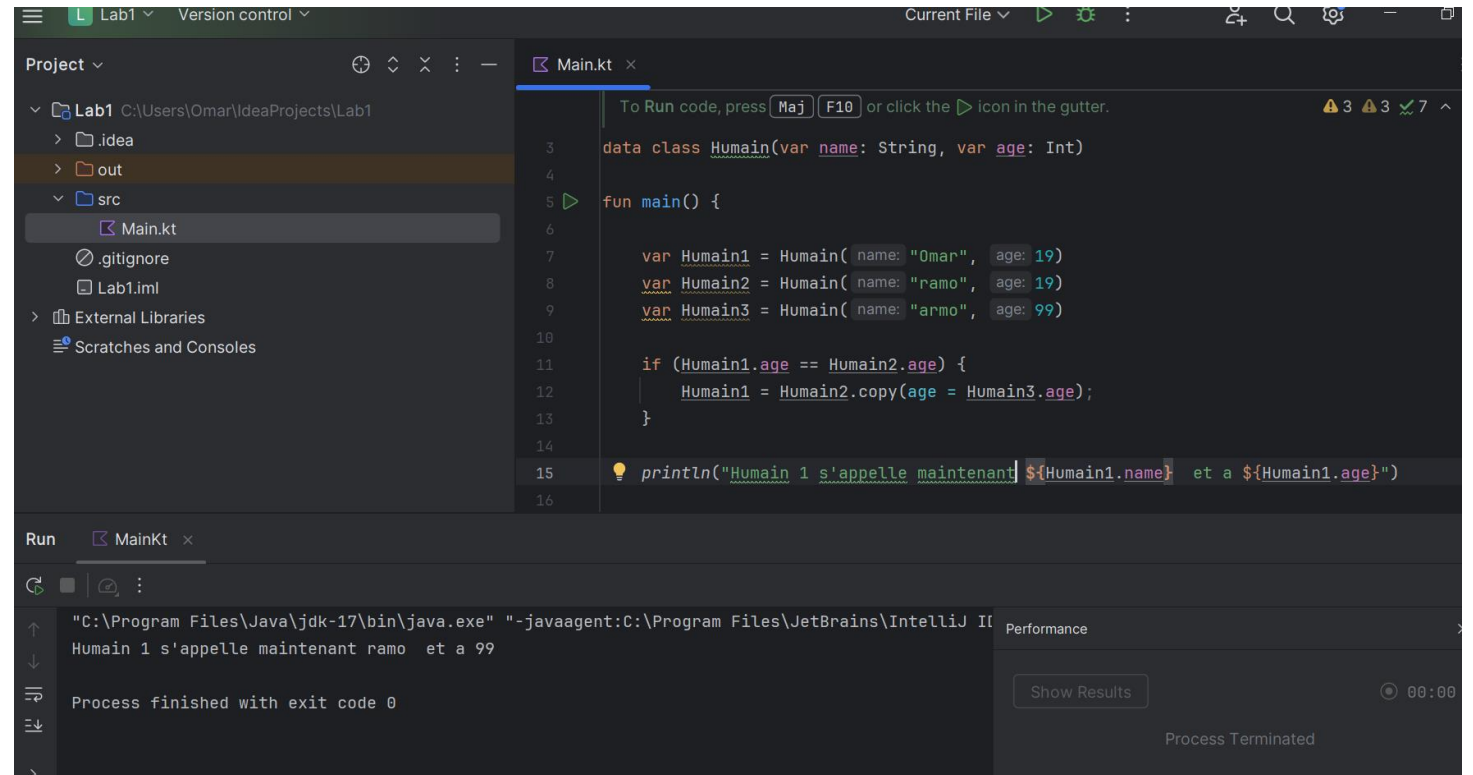
Below the code editor, the Run tool window shows the execution output for 'MainKt':

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I  
Option toString = Humain(name=Omar, age=19)  
Option Copy = Humain(name=Omar, age=19)  
Option hashCode = 76302148  
Option equals = true
```

On the right side of the Run tool window, there is a 'Performance' section with a 'Show Results' button and a 'Process Terminated' status.

Copy

- Avec la fonction copy tu peux évidemment copier un objet sur une autre mais tu peux faire une copy et modifier un des élément



The screenshot shows an IDE window with a project named 'Lab1'. The file explorer on the left shows the project structure: 'Lab1' (C:\Users\Omar\IdeaProjects\Lab1) containing '.idea', 'out', 'src', 'Main.kt', '.gitignore', and 'Lab1.iml'. The 'Main.kt' file is open in the editor. The code defines a data class 'Humain' with 'name' and 'age' properties, and a 'main' function. In the 'main' function, three 'Humain' objects are created: 'Humain1' (name: 'Omar', age: 19), 'Humain2' (name: 'ramo', age: 19), and 'Humain3' (name: 'armo', age: 99). A conditional statement checks if 'Humain1.age' equals 'Humain2.age'. If true, it calls 'Humain1.copy(age = Humain3.age)'. Finally, it prints the name and age of 'Humain1'. The 'Run' tab at the bottom shows the execution output: 'Humain 1 s'appelle maintenant ramo et a 99' and 'Process finished with exit code 0'. The 'Performance' tab shows 'Process Terminated'.

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

fun main() {

    var Humain1 = Humain( name: "Omar", age: 19)
    var Humain2 = Humain( name: "ramo", age: 19)
    var Humain3 = Humain( name: "armo", age: 99)

    if (Humain1.age == Humain2.age) {
        Humain1 = Humain2.copy(age = Humain3.age);
    }

    println("Humain 1 s'appelle maintenant: ${Humain1.name} et a ${Humain1.age}")
}
```

Run MainKt x

"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I

Humain 1 s'appelle maintenant ramo et a 99

Process finished with exit code 0

Performance

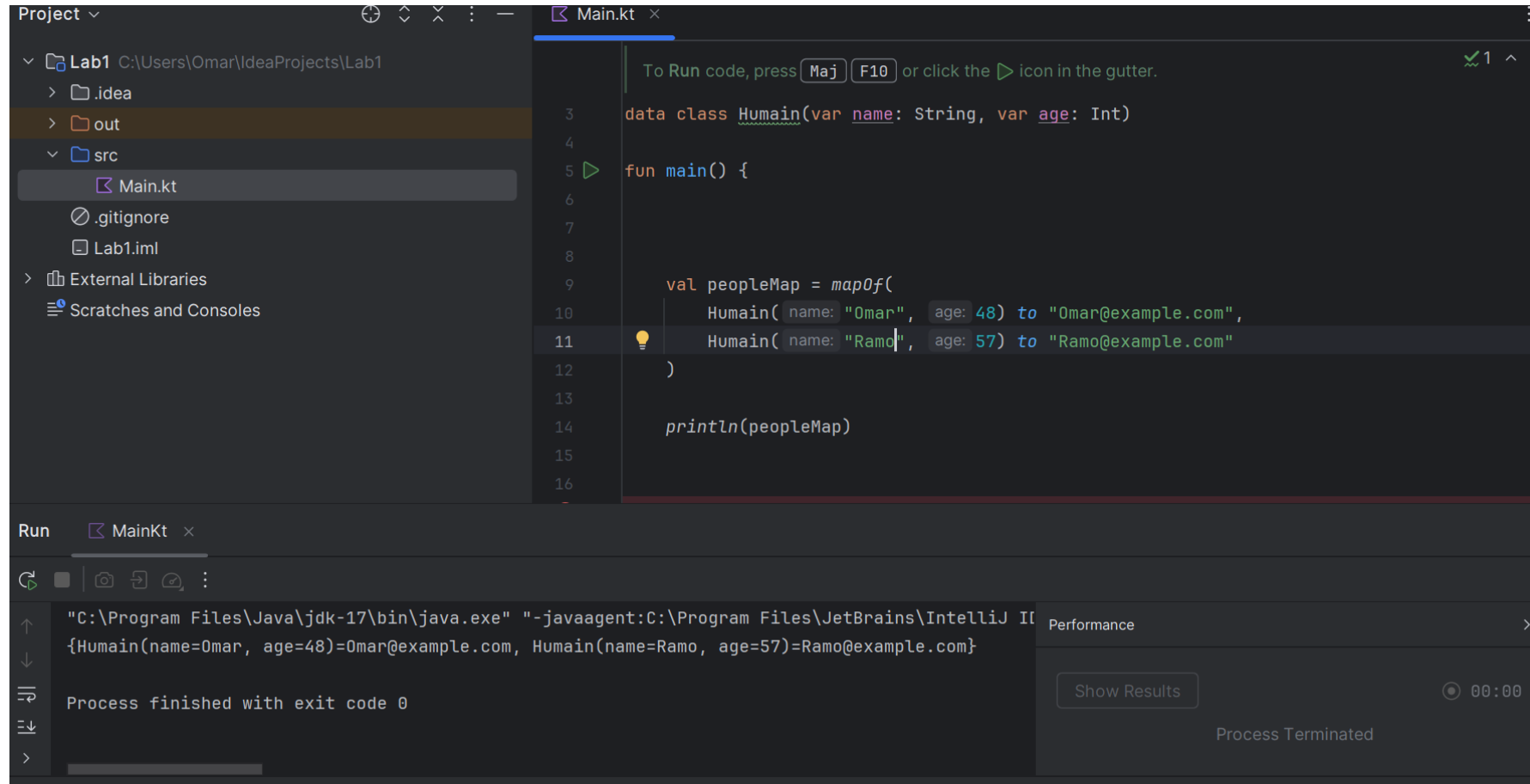
Show Results

Process Terminated

Le mappage

- Un Map est une collection qui stocke des paires clé-valeur. La clé dans un Map doit être unique, et c'est ici que l'utilisation de `data class` devient très utile. La clé sera comparée à l'aide des méthodes `equals()` et `hashCode()` pour garantir que chaque clé dans le Map est unique. Dans l'exemple que je fais demandé je vais assigner des objets à une clé-valeur. Tu peux aussi utiliser le mappage afin d'assigner un nom à un chiffre ou vis-versa

Utilisation de Map avec Kotlin



The screenshot displays the IntelliJ IDEA IDE interface. On the left, the Project Explorer shows a project named 'Lab1' with a source file 'Main.kt'. The main editor window shows the following Kotlin code:

```
3 data class Humain(var name: String, var age: Int)
4
5 fun main() {
6
7
8
9     val peopleMap = mapOf(
10         Humain(name: "Omar", age: 48) to "Omar@example.com",
11         Humain(name: "Ramo", age: 57) to "Ramo@example.com"
12     )
13
14     println(peopleMap)
15
16 }
```

Below the code editor, the Run tab is active, showing the command used to execute the program:

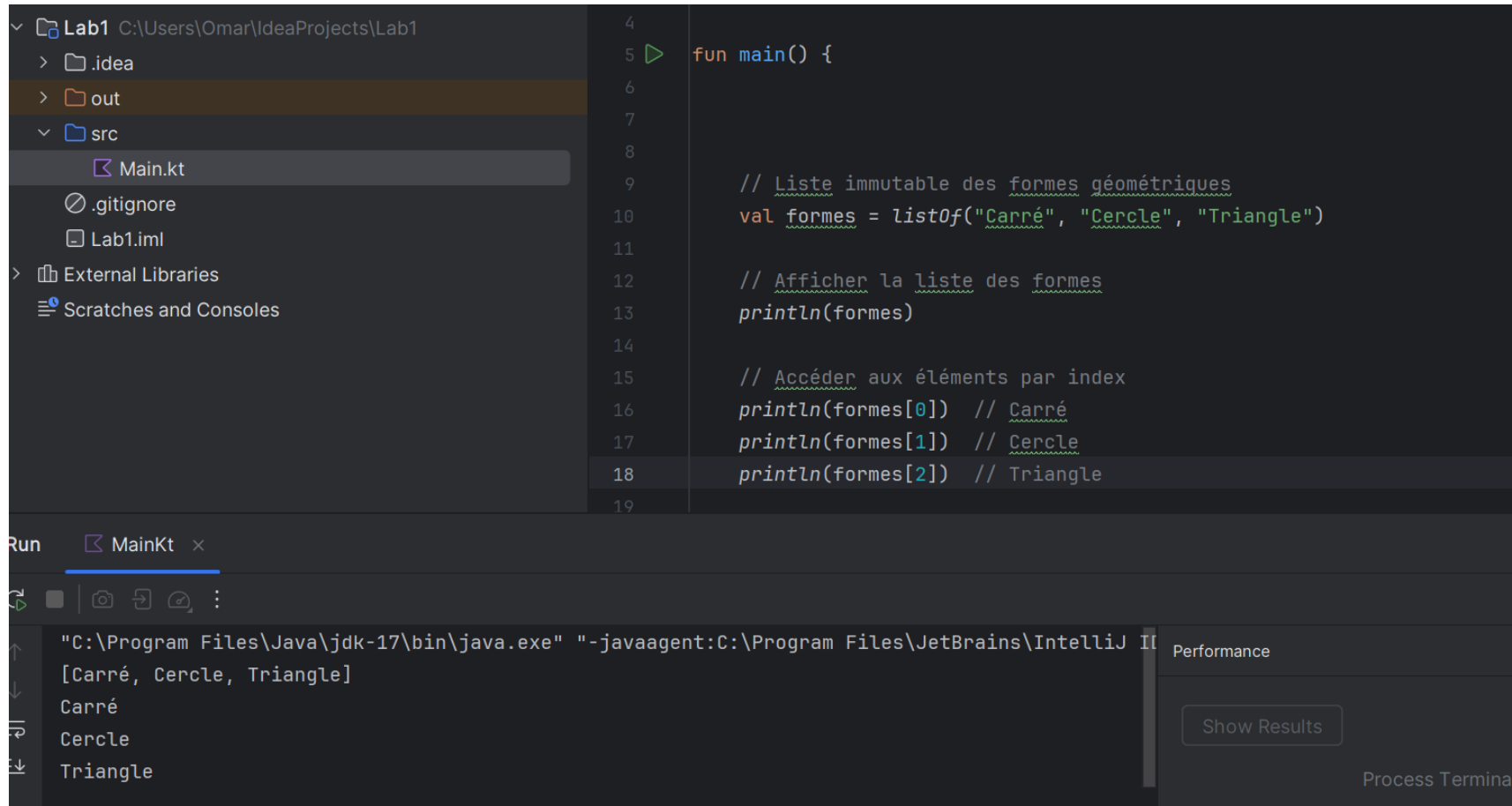
```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I..."
{Humain(name=Omar, age=48)=Omar@example.com, Humain(name=Ramo, age=57)=Ramo@example.com}
```

The output indicates that the process finished with exit code 0. On the right side of the Run tab, a Performance section shows 'Process Terminated' and a 'Show Results' button.

Les listes

- Une list en Kotlin est une collection ordonnée d'éléments. Par défaut, une list est immutable, ce qui signifie que vous ne pouvez pas modifier la liste (ajouter, supprimer ou modifier des éléments) après sa création. Si vous avez besoin d'une liste modifiable, vous devez utiliser une MutableList. On va utiliser ListOf(variable , variable) et lorsqu'on va utiliser mutableListOf on va pouvoir ajouter des variable avec .add , supprimer des variable avec .remove.

Exemple liste immutable avec ListOf



The screenshot displays an IDE interface with a project named 'Lab1'. The file explorer on the left shows the project structure, including 'src' and 'Main.kt'. The main editor window shows the following Kotlin code:

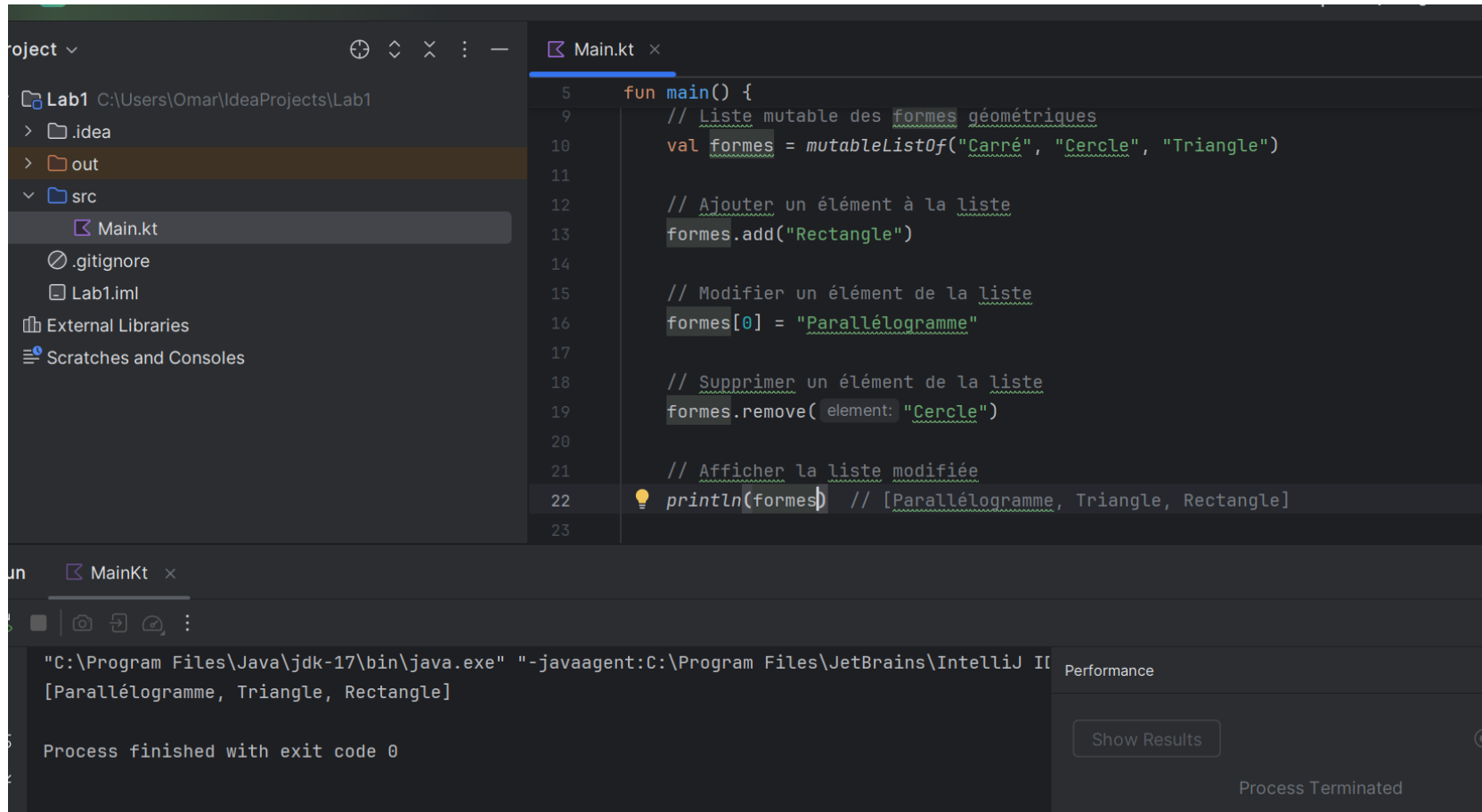
```
4  
5 fun main() {  
6  
7  
8  
9 // Liste immuable des formes géométriques  
10 val formes = listOf("Carré", "Cercle", "Triangle")  
11  
12 // Afficher la liste des formes  
13 println(formes)  
14  
15 // Accéder aux éléments par index  
16 println(formes[0]) // Carré  
17 println(formes[1]) // Cercle  
18 println(formes[2]) // Triangle  
19 }
```

Below the code editor, the 'Run' tab is active, showing the execution output for 'MainKt':

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I  
[Carré, Cercle, Triangle]  
Carré  
Cercle  
Triangle
```

On the right side of the Run tab, there is a 'Performance' section with a 'Show Results' button and a 'Process Termina' label.

Exemple list mutable



The screenshot displays the IntelliJ IDEA IDE interface. On the left, the Project tool window shows a project named 'Lab1' with a source file 'Main.kt'. The main editor window shows the following Kotlin code in 'Main.kt':

```
5 fun main() {  
9     // Liste mutable des formes géométriques  
10    val formes = mutableListOf("Carré", "Cercle", "Triangle")  
11  
12    // Ajouter un élément à la liste  
13    formes.add("Rectangle")  
14  
15    // Modifier un élément de la liste  
16    formes[0] = "Parallélogramme"  
17  
18    // Supprimer un élément de la liste  
19    formes.remove(element: "Cercle")  
20  
21    // Afficher la liste modifiée  
22    println(formes) // [Parallélogramme, Triangle, Rectangle]  
23 }
```

Below the code editor, the Run tool window shows the execution output:

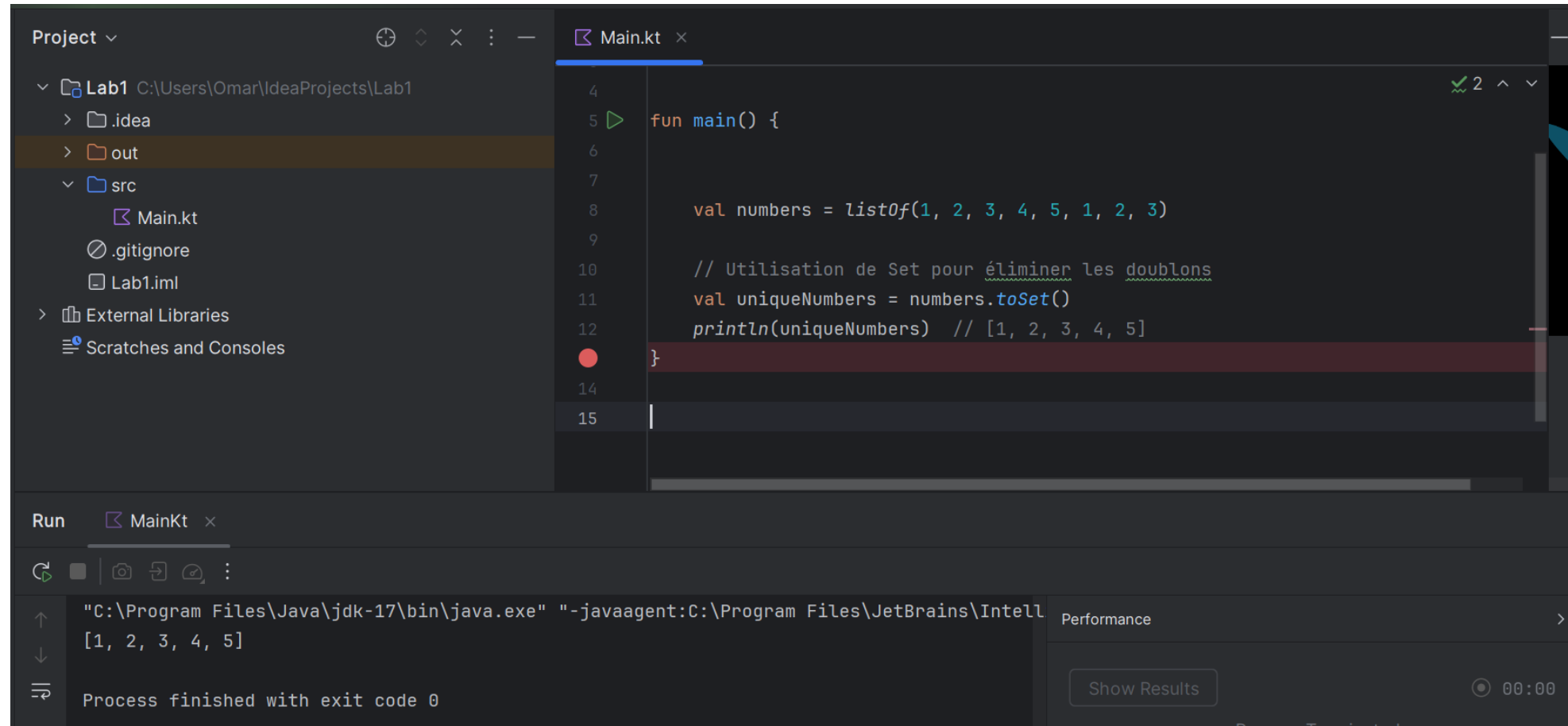
```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ I  
[Parallélogramme, Triangle, Rectangle]  
  
Process finished with exit code 0
```

On the right side of the Run tool window, there is a 'Performance' tab with a 'Show Results' button and a 'Process Terminated' status.

Set

- En Kotlin, un set est une collection qui ne permet pas de doublons, ce qui signifie que chaque élément dans un Set doit être unique. Les Set sont particulièrement utiles lorsque vous avez besoin de vérifier rapidement si un élément est déjà présent dans une collection, tout en garantissant l'unicité des éléments.
- Types de Set en Kotlin
- Il existe deux types principaux de Set en Kotlin :
- Set immutable : La version par défaut d'un Set. Vous ne pouvez pas modifier cette collection une fois qu'elle est créée.
- mutableSet : Une version modifiable de Set. Vous pouvez ajouter, supprimer ou modifier les éléments après la création de l'ensemble.
- Son usage est très similaire au list à part pour la partie des doublons

Exemple de set pour vérifier les doublons



The screenshot displays an IDE interface with a project named 'Lab1'. The 'Project' sidebar on the left shows the file structure, including 'src/Main.kt'. The main editor window shows the following Kotlin code in 'Main.kt':

```
4  
5 fun main() {  
6  
7  
8     val numbers = listOf(1, 2, 3, 4, 5, 1, 2, 3)  
9  
10    // Utilisation de Set pour éliminer les doublons  
11    val uniqueNumbers = numbers.toSet()  
12    println(uniqueNumbers) // [1, 2, 3, 4, 5]  
13 }  
14  
15
```

The code uses the `toSet()` method to convert a list of numbers into a set, which automatically removes any duplicate values. The output of the program is shown in the 'Run' console at the bottom: `[1, 2, 3, 4, 5]`. The console also indicates that the process finished with exit code 0.

Null safety

- Le **Null Safety** est l'une des fonctions les plus importantes de Kotlin, qui permet de prévenir les erreurs courantes liées à l'utilisation des références nulles, une cause fréquente de bugs dans les applications. En Java, une variable non initialisée ou qui contient la valeur null peut causer des exceptions de type `NullPointerException` (NPE) à l'exécution, ce qui est souvent difficile à corriger. Kotlin interdit par défaut l'utilisation de `null` pour les types non définis comme `String`, `Int`, etc. Cela permet de réduire les risques de NPE et d'améliorer la solidité du code.
- Dans Kotlin, pour qu'une variable accepte la valeur `null`, elle doit être explicitement déclarée comme **nullable** en utilisant le symbole `?`. Par exemple, une variable de type `String` qui peut être nulle doit être déclarée comme `String?`. Cela permet de signaler clairement où un `null` est accepté et d'obliger le développeur à gérer cette situation, soit avec des contrôles de type, soit à l'aide d'opérateurs spécifiques comme `?.`

La lecture de fichier

- En Kotlin, la lecture de fichiers est une tâche courante que l'on peut accomplir de manière simple et efficace grâce aux bibliothèques standard de Kotlin et Java. Kotlin offre des API intuitives et concises qui permettent de lire facilement des fichiers en quelques lignes de code. Que ce soit pour lire un fichier texte ligne par ligne, lire tout le contenu d'un fichier en une seule fois, ou traiter des fichiers de manière asynchrone, Kotlin propose des solutions adaptées.

Afficher le fichier au complet

```
import java.io.File

fun main() {
    // Chemin du fichier
    val fichier = File(pathname: "fichier.txt")

    // Lire tout le contenu du fichier en une seule fois
    val data = fichier.readText()

    // Afficher le contenu du fichier
    println(data)
}
```

Fichier ligne par ligne

```
import java.io.File

fun main() {
    val file = File(pathname: "fichier.txt")

    val lignes = file.readLines()

    for (ligne in lignes) {
        println(ligne)
    }
}
```

Kotlin et java

- Kotlin est entièrement interopérable avec Java, ce qui signifie que tu peux facilement utiliser du code Java dans un projet Kotlin, et inversement. C'est une fonctionnalité très puissante de Kotlin, qui permet de migrer progressivement des projets Java existants vers Kotlin, sans avoir à tout réécrire.

Les coroutines

- Les coroutines en Kotlin sont un outil puissant pour gérer des tâches asynchrones, c'est-à-dire des tâches qui peuvent s'exécuter en arrière-plan sans bloquer l'application. Elles sont très utiles quand tu veux effectuer des opérations comme des requêtes réseau, des lectures de fichiers ou des calculs complexes, tout en gardant ton programme réactif. Contrairement aux threads, les coroutines sont légères et permettent de mieux gérer les tâches concurrentes.

Utilité

- Pendant que ton programme attend la réponse, tu ne veux pas que l'application gèle devienne inutilisable pour l'utilisateur. Avec les coroutines, tu peux dire à Kotlin de suspendre l'exécution pendant qu'il attend la réponse, tout en continuant d'exécuter d'autres parties du programme. Une fois la tâche terminée, Kotlin reprend là où la coroutine s'était arrêtée.

Coroutine vs thread

Les coroutines prennent moins de mémoires que les threads. Créer un thread est coûteux en mémoire, tandis que des milliers de coroutines peuvent s'exécuter en même temps avec un faible coût. Les coroutines permettent d'attendre des résultats sans bloquer le thread principal, ce qui rend ton application plus fluide et réactive. Les coroutines gèrent automatiquement la concurrence et la reprise des tâches, ce qui rend le code asynchrone plus simple à lire et à écrire que les threads.

Exemple usage Coroutines

```
1  import kotlinx.coroutines.*
2
3  fun main() = runBlocking {
4      |
5      launch {
6          delay(timeMillis: 20000L)
7          println("arriere-plan")
8      }
9
10     println("programme actuel")
11 }
12
```

Bibliographie

- Page officiel de Kotlin
- chrome-extension://efaidnbmnnnibpcajpcgclefindmkaj/https://kotlinlang.org/docs/kotlin-reference.pdf

Le livre Head First Kotlin: A Brain-Friendly Guide de *de Dawn Griffiths, David Griffiths*

[Le livre Kotlin](#) Les fondamentaux du développement d'applications Android *de Anthony COSSON*

Le livre Learn Kotlin for Android Development The Next Generation Language for Modern Android Apps Programming — Peter Späth

Kotlin Atomic de Bruce Eckel