RÉSUMÉ THÉORIQUE - FILIÈRE DÉVELOPPEMENT DIGITAL

Option - Applications mobiles

M210 - Programmer en Kotlin

35 heures

SOMMAIRE

MODALITÉS PÉDAGOGIQUES

1

LE GUIDE DE SOUTIEN

Il contient le résumé théorique et le manuel des travaux pratiques

2

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life

3

DES CONTENUS TÉLÉCHARGEABLES

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life

4

DU CONTENU INTERACTIF

Vous disposez de contenus interactifs sous forme d'exercices et de cours à utiliser sur WebForce Life

5

DES RESSOURCES EN LIGNES

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage

PARTIE 1

Découvrir les fondamentaux de Kotlin

Dans ce module, vous allez :

- Découvrir les fondamentaux de Kotlin

04 heures

CHAPITRE n° 1

INTRODUIRE LES DIFFÉRENCES MAJEURES AVEC JAVA

Ce que vous allez apprendre dans ce chapitre :

- Découvrir les fonctionnalités de Kotlin
- Comprendre l'interopérabilité de Kotlin avec Java

02 heures

CHAPITRE n° 1

INTRODUIRE LES DIFFÉRENCES MAJEURES AVEC JAVA

1. **Généralités sur les fonctionnalités KOTLIN**
2. Communication entre JAVA et Kotlin
3. Interopérabilité avec Java

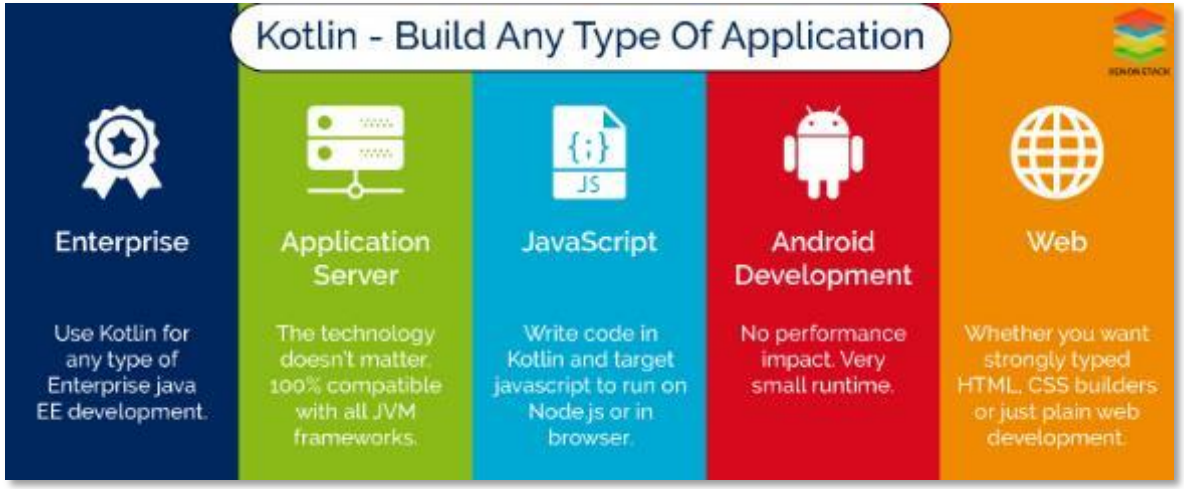
Qu'est ce que Kotlin ?

- Kotlin est un langage de programmation open source, orienté objet et fonctionnel, avec un typage statique, qui cible la machine virtuelle Java (JVM), Android, JavaScript et le code natif. Il est développé par JetBrains, par une équipe de programmeurs basée à Saint-Pétersbourg en Russie. Le nom a été inspiré du nom de l'île de Kotline, située près de Saint-Pétersbourg.
- Le projet a démarré en 2010 et était open source dès le début. La première version officielle 1.0 a été publiée en février 2016.

https://kotlinlang.org

Qu'est ce que Kotlin ?

- Kotlin est principalement développé par une équipe d'ingénieurs de JetBrains (l'équipe actuelle compte plus de 1500 personnes). Le principal concepteur linguistique est Andrey Breslav. En plus de l'équipe de base, il y a aussi plus de 553 contributeurs externes sur GitHub.
- La version actuelle de Kotlin est la version 1.7.20, publiée le 1 août 2022.
- Kotlin est protégé par la Fondation Kotlin et sous licence Apache 2.
- Kotlin supporte le développement des solutions backend, frontend et le développement mobile.



Kotlin is suitable for a range of application areas. Source: Gill 2017

PARTIE 1

Applications développées avec Kotlin

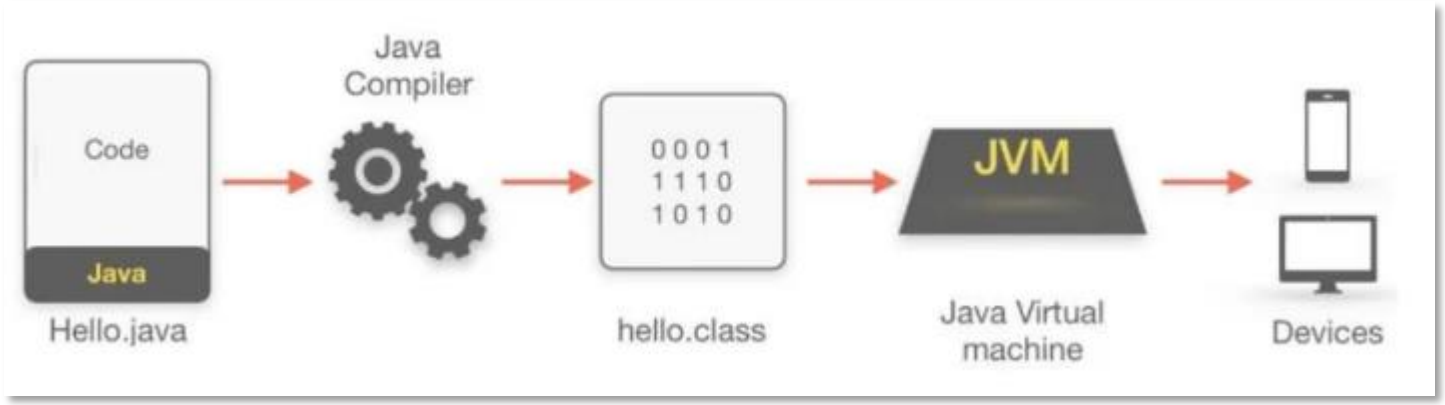
- Kotlin est le langage officiel pour le développement mobile, Actuellement plusieurs applications sont développé avec Kotlin.

Par exemple :

Kotlin est-il un langage orienté objet ou un langage fonctionnel ?

- Kotlin est à la fois un langage orienté objet et un langage fonctionnel. Vous pouvez programmer en orienté objet ou en procédural / fonctionnel. Il est aussi possible de construire une application avec un mixte de ces deux styles de programmation. Avec un support de première classe pour des fonctions telles que les fonctions d'ordre supérieur, les types de fonctions et les lambdas, Kotlin est un excellent choix si vous faites ou explorez la programmation fonctionnelle.

L'exemple suivant montre comment le code est compilé par le Java compiler et le JVM :

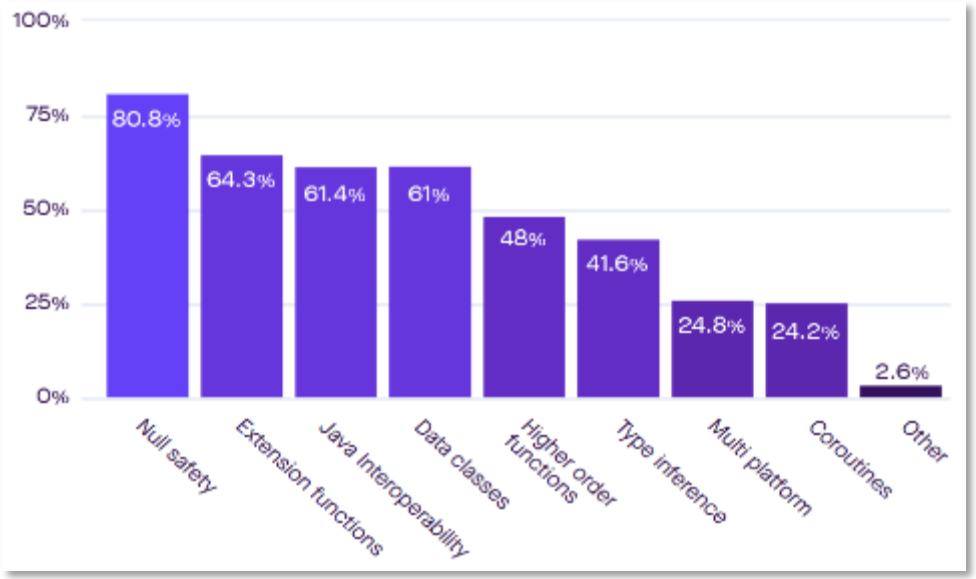


https://kotlinlang.org

PARTIE 1

Java vs Kotlin : Concision du code

- La comparaison d'une classe Java avec une classe Kotlin équivalente démontre la concision du code Kotlin. Pour effectuer la même opération que la classe Java, une classe Kotlin nécessite moins de code.
- Les extensions Android de Kotlin permettent d'importer une référence à une vue dans le fichier d'activité. Cela permet de travailler avec cette vue comme si elle faisait partie de l'activité.
- Une enquête de 2018 révèle les fonctionnalités préférées de Kotlin :



Source: Pusher 2018

Java vs Kotlin : Concision du code

- Kotlin nécessite relativement moins de ligne de code que Java, ce qui le rend plus lisible et compréhensible. Par exemple, le 'Switch case' en Java a été converti en quelques lignes de code en utilisant ‘when' en Kotlin.

Kotlin

```
when(days)
{
  lundi -> println("premier jour")
  Mardi -> println("deuxième jour")
  Mercredi -> println("deuxième jour")
  else -> println("n'existe pas ")
}
```

JAVA

```
switch(days)
{
  case lundi :
  System.out.println("premier jour");
  break;
  case mardi :
  System.out.println("deuxième jour");
  break;
  case mardi :
  System.out.println("troisième jour");
  break;
  default:
  System.out.println("n'existe pas ");
  break;
}
```

Java vs Kotlin : Coroutines

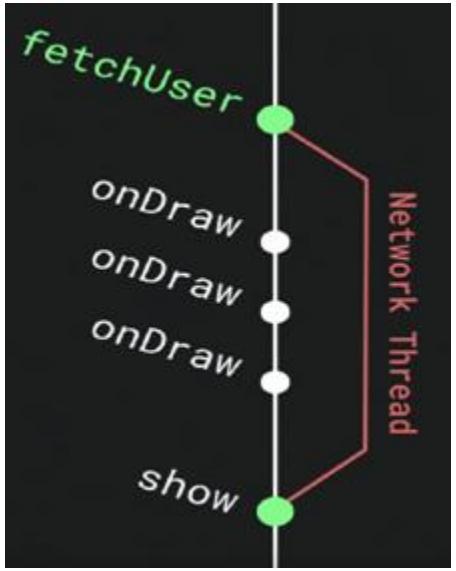
- Les travaux intensifs du CPU et les E/S du réseau sont des opérations de longue haleine. Le thread appelant est bloqué jusqu'à la fin de l'opération. Comme Android est monofil par défaut, l'interface utilisateur d'une application est complètement gelée dès que le fil principal est bloqué.
- La solution traditionnelle à ce problème en Java consiste à créer un thread d'arrière-plan pour les travaux longs ou intensifs. Cependant, la gestion de plusieurs threads entraîne une augmentation de la complexité ainsi que des erreurs dans le code.
- Kotlin permet également la création de threads supplémentaires. Cependant, il existe une meilleure façon de gérer les opérations intensives en Kotlin, connue sous le nom de coroutines. Les coroutines sont sans stack, ce qui signifie qu'elles nécessitent une utilisation de la mémoire plus faible que les threads.

Java vs Kotlin : Coroutines

- Les coroutines sont capables d'exécuter des tâches longues et intensives en suspendant l'exécution sans bloquer le thread, puis en reprenant l'exécution à un moment ultérieur. Elles permettent de créer un code asynchrone non bloquant qui semble être synchrone.
- Le code utilisant les coroutines est non seulement clair mais aussi concis. De plus, les coroutines permettent de créer des styles supplémentaires de programmation asynchrone non bloquante tels que async/await.

Voici à quoi ressemble une Coroutine :

```
CoroutineScope(Dispatchers.Main + Job()).launch
{
  val user = fetchUser() // A suspending function running in the I/O thread
  updateUser(user) // Updates UI in the main thread
}
private suspend fun fetchUser():User = with Context(Dispatchers.IO) {
  //fetches the data from server and returns user data
}
```



- Nous récupérons les données du serveur dans le thread d'arrière-plan, puis mettons à jour l'interface utilisateur dans le thread principal.
- Étant donné que dans cette partie concerne le concept de base de Kotlin, nous discuterons plus en détail des Coroutines dans la partie *Introduire les coroutines*.

PARTIE 1

Java vs Kotlin : Fonctions d'extension

- Kotlin permet aux développeurs d'étendre une classe avec de nouvelles fonctionnalités via des fonctions d'extension. Ces fonctions, bien que disponibles dans d'autres langages de programmation comme C#, ne sont pas disponibles en Java.
- La création d'une fonction d'extension est facile en Kotlin. Il suffit de préfixer le nom de la classe qui doit être étendue au nom de la fonction créée.

Java vs Kotlin : Fonctions d'extension

- Kotlin permet aux développeurs d'étendre une classe avec de nouvelles fonctionnalités via des fonctions d'extension. Ces fonctions, bien que disponibles dans d'autres langages de programmation comme C#, ne sont pas disponibles en Java.
- La création d'une fonction d'extension est facile en Kotlin.
- Il suffit de préfixer le nom de la classe qui doit être étendue au nom de la fonction créée.
- Pour rendre ce concept plus clair, prenons l'exemple de concaténation de chaînes pour ajouter trois chaînes, nous savons qu'il n'y a pas de fonction intégrée disponible dans Java/Kotlin, nous pouvons ajouter/combiner plus de trois chaînes, donc, pour implémenter cela, nous allons ajoutez la fonction dans Main.kt sans modifier la classe String et qui fera partie de la classe String :

```
fun main() {
  val str1 = "I"
  val str2 = "love"
  val str3 = "Kotlin"
  print(str1.add(str2, str3))
}
fun String.add(str2: String, str3:String): String{
  return this + str2 + str3
}
```

- **Le résultat est :** I love kotlin

Java vs Kotlin : Support natif de la délégation

- La délégation représente le processus par lequel un objet récepteur délègue des opérations à un second objet délégué. Kotlin prend en charge la composition par rapport au modèle de conception de l'héritage au moyen de la délégation de première classe, également connue sous le nom de délégation implicite.
- La délégation de classe est une alternative à l'héritage dans Kotlin. Elle permet d'utiliser des héritages multiples. En outre, les propriétés déléguées de Kotlin empêchent la duplication du code.

Java vs Kotlin : Champs non privés

- L'encapsulation est essentielle dans tout programme pour atteindre un niveau souhaitable de maintenabilité.
- En encapsulant la représentation d'un objet, on peut imposer la façon dont les appelants interagissent avec lui. Il est possible de modifier la représentation sans devoir modifier les appelants, à condition que l'API publique reste inchangée.
- Les champs non privés ou publics en Java sont utiles dans les scénarios où les appelants d'un objet doivent modifier sa représentation en conséquence. Cela signifie simplement que ces champs exposent la représentation d'un objet aux appelants. Kotlin n'a pas de champs non privés.

Java vs Kotlin : Sécurité de la valeur Nulle

- Java permet aux développeurs d'attribuer une valeur nulle à n'importe quelle variable. Cependant, s'ils essaient d'utiliser une référence d'objet qui a une valeur nulle, à l'exception NullPointerException.
- Contrairement à Java, tous les types sont non nuls par défaut dans Kotlin. Si les développeurs essaient d'assigner ou de retourner des valeurs nulles dans le code Kotlin, ils échoueront au moment de la compilation. Cependant, il existe un moyen de contourner ce problème. Afin d'assigner une valeur nulle à une variable en Kotlin, il est nécessaire de marquer explicitement cette variable comme nullable.
- Il n'existe pas d'exception NullPointerException en Kotlin. S'il existe une telle exception en Kotlin, il est fort probable qu'il ait été explicitement attribué une valeur nulle ou que cela soit dû à un code Java externe.

Java vs Kotlin : Bibliothèques de traitement des annotations avec Kotlin

- En plus de fournir un support pour les frameworks et bibliothèques Java existants, Kotlin a également une disposition pour les frameworks Java avancés reposant sur le traitement des annotations.
- Cependant, l'utilisation d'une bibliothèque Java qui utilise le traitement des annotations dans Kotlin nécessite de l'ajouter au projet Kotlin d'une manière un peu différente de ce qui est requis pour une bibliothèque Java qui n'utilise pas le traitement des annotations.
- Il est nécessaire de spécifier la dépendance à l'aide du plugin kotlin-kapt. Ensuite, l'outil de traitement des annotations Kotlin doit être utilisé à la place de l'annotationProcessor.
- Indépendamment de toutes les dissemblances entre les deux langages de programmation, ils sont totalement interopérables. Java et Kotlin compilent tous deux en bytecode. Cela signifie qu'il est possible d'appeler du code Java à partir de Kotlin et vice-versa.
- Cette flexibilité présente deux avantages. Premièrement, elle facilite la prise en main de Kotlin en introduisant progressivement le code Kotlin dans un projet Java. Deuxièmement, les deux langages peuvent être utilisés simultanément dans tout projet de développement d'applications Android.

Java vs Kotlin

- En consultant le tableau ci-dessous, vous pourrez facilement analyser quel langage de programmation conviendra le mieux à un projet de développement d'application Android.

Paramètres	JAVA	Kotlin
Facile à utiliser	Difficile	Simple
Performance	Presque pareil	Haut niveau
Popularité	Haut niveau	Haut niveau
Évolutivité	Niveau modéré	Haut niveau
Communauté	Haut niveau	Haut niveau
Cross platform	Plate-forme limitée	Plusieurs plate-forme
Documentation	Haut niveau	Haut niveau

CHAPITRE n° 1

INTRODUIRE LES DIFFÉRENCES MAJEURES AVEC JAVA

1. Généralités sur les fonctionnalités KOTLIN
2. **Communication entre JAVA et Kotlin**
3. Interopérabilité avec Java

Communication entre JAVA et Kotlin

Le code Kotlin peut être facilement appelé depuis Java. Par exemple, les instances d'une classe Kotlin peuvent être créées et exploitées de manière transparente dans des méthodes Java. Cependant, il existe certaines différences entre Java et Kotlin qui nécessitent une attention particulière lors de l'intégration du code Kotlin dans Java.

Propriétés :

Une propriété Kotlin est compilée avec les éléments Java suivants :

- une méthode getter, avec le nom calculé en ajoutant le préfixe get
- une méthode setter, avec le nom calculé en ajoutant le préfixe set (uniquement pour les propriétés var)
- un champ privé, avec le même nom que le nom de la propriété (uniquement pour les propriétés avec des champs de sauvegarde)

Communication entre JAVA et Kotlin

Par exemple, var firstName : String se compile dans les déclarations Java suivantes :

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

Si le nom de la propriété commence par is, une règle de mappage de nom différente est utilisée : le nom du getter sera le même que le nom de la propriété, et le nom du setter sera obtenu en remplaçant is par set. Par exemple, pour une propriété isOpen, le getter s'appellera isOpen() et le setter s'appellera setOpen(). Cette règle s'applique aux propriétés de tout type, pas seulement booléennes.

CHAPITRE n° 1

INTRODUIRE LES DIFFÉRENCES MAJEURES AVEC JAVA

Interopérabilité avec Java

- Lorsque Kotlin a été développé, il fonctionnait uniquement sur JVM , il fournit donc un ensemble complet de fonctionnalités qui facilitent l'appel de Kotlin depuis Java.
- Par exemple, les objets des classes Kotlin peuvent être facilement créés et leurs méthodes peuvent être invoquées dans les méthodes Java. Cependant, il existe certaines règles sur la façon dont le code Kotlin peut être utilisé en Java .

Propriétés Kotlin

- Une propriété dans Kotlin est définie en Java comme un champ privé portant le même nom que celui de la propriété, et une fonction getter et setter , avec get et set préfixés au nom de la propriété. Ce champ privé existe dans la classe java générée à partir du fichier kotlin.
- Par exemple, la propriété var age: Int , est compilée dans le code suivant en Java :

```
public int getAge() {
  return age;
}

public void setAge(int age) {
  this.age = age;
}
```

- Ces propriétés sont accessibles à l'aide de l'objet de la classe, de la même manière qu'en Java. Cependant, si le nom de la propriété commence par is , le mot-clé get est ignoré dans le nom de la fonction getter .

Fonctions au niveau du package

- Toutes les fonctions définies dans un fichier Kotlin, au sein d'un package, sont compilées en méthodes statiques en Java au sein d'une classe dont le nom de classe est une combinaison du nom du package et du nom du fichier.

Par exemple, s'il existe un package nommé kotlinPrograms et un fichier Kotlin nommé firstProgram.kt avec le contenu suivant.

```
package kotlinPrograms
class myClass {
fun add(val a:Int, val b:Int): Int {
  return a + b;
  }
}
```

- Cette fonction peut être invoquée en Java à l'aide de la syntaxe suivante :

```
new kotlinPrograms.firstProgram.myClass()
kotlinPrograms.firstProgramkt.add(3,5);
```

Fonctions au niveau du package

- Nous pouvons changer le nom de la classe Java générée en utilisant l'annotation **@JvmName**.

```
//Kotlin file
@file: Jvmname("Sample")
package kotlinPrograms
class myClass {
    fun add(val a: Int, val b: Int): Int {
        return a + b;
    }
}
```

- Cette fonction peut être invoquée en Java à l'aide de la syntaxe suivante :

```
//JAVA
new kotlinPrograms.firstProgram.myClass();
kotlinPrograms.Sample.add(3,5);
```

Fonctions au niveau du package

- Cependant, avoir plusieurs fichiers avec le même nom est logiquement une erreur. Pour pallier ce problème, Kotlin donne à son compilateur la possibilité de créer une classe de façade qui porte un nom particulier et qui contient toutes les déclarations de tous les fichiers portant le même nom. Pour permettre la création d'une telle classe de façade, l'annotation @JvmMultiFileClass dans tous les fichiers.

Exemple :

```
//Kotlin file
@file: Jvmname("Sample")
@file: JvmMultiFileClass

package sample.example
class print() { .......
}
```

Champs statiques

- Les propriétés dans Kotlin qui sont déclarées dans un objet nommé ou un objet compagnon sont utilisées comme champs statiques en Java. Pour accéder à ces champs en Java, ceux-ci doivent être annotés avec l'annotation @JvmField, le modificateur lateinit ou doivent être déclarés avec un modificateur const .

Exemple :

```
// filename Program.kt
// Property in a companion object
  class abc {
     companion object {
        @JvmField
        val x = 5;
     }
  }
// a constant property
const val y = 5;
// Java Usage
abc.x
ProgramKt.y
```

Méthodes statiques

- Les méthodes définies au niveau du package sont toujours générées en tant que méthodes statiques dans le fichier Java. De plus, les méthodes définies dans les objets nommés et les objets compagnons si elles sont annotées avec l'annotation @JvmStatic sont générées en tant que méthodes statiques. Cette annotation déclare la fonction suivante comme étant une fonction de classe.

Exemple pour l'objet compagnon :

```
// filename Program.kt
  class abc {
     companion object {
       @JvmStatic fun add(val a: Int, val b: Int): Int {
          return a + b;
       }
       fun sub(val a: Int, val b: Int): Int {
           return a - b;
       }
     }
  }
//JAVA usage
  abc.add(); // succes
  abc.sub(); // error : not a static method
  abc.companion.add(); // instance method remains
  C.companion.sub(); // the only way it works
```

- De même, cela fonctionne pour l'objet nommé.

Champs d'instance

- Kotlin fournit une fonctionnalité permettant d'utiliser une propriété en tant que champ d'instance en Java. Pour ce faire, annotez la propriété avec l'annotation @JvmField. Ces champs d'instance ont la même visibilité que la propriété Kotlin. Cependant, la propriété doit avoir un champ de support et ne doit pas être déclarée avec les modificateurs private , open , const et override .

```
class Test(data: Int) {
@JvmField val id = data
}
```

- Cette propriété est désormais accessible en Java en tant que :

```
Test obj = new Test(5);
System.out.println(obj.id);
```

Exceptions vérifiées

- Toutes les exceptions dans Kotlin sont décochées. Ainsi, les signatures Java des fonctions Kotlin ne déclarent ni ne gèrent les exceptions levées. Pour surmonter ce problème, la fonction Kotlin doit être annotée avec l'annotation @Throws spécifiant l'exception qui sera levée. Dans ce cas, la signature Java déclarera également la fonction à lancer.

Exemple :

```
// a sample kotlin function
// filename program.kt
package Sample

fun print() {
  throws IOException()
}

// Java code trying to call the above function
try {
  Sample.Program.print();
}

// This statement causes error because does not declare IO exception in
throws list
```

Exceptions vérifiées

- Donc, pour résoudre l'erreur, nous déclarons l'annotation @Throws en haut.

```
package Sample

@Throws(IOException::class)
fun print() {
  throws IOException()
}
```

Question 1

Kotlin est développé par ?

a) Google

b) JetBrains

c) Microsoft

d) Adobe

Question 2

Pouvons-nous utiliser Kotlin dans le langage de programmation Java ?

a) Oui

b) Non

Question 3

Pouvons-nous exécuter le code Kotlin sans Jvm ?

a) Oui

b) Non

Question 4

Kotlin a-t-il le mot-clé statique ?

a) Oui

b) Non

Question 5

Kotlin a été développé sous la licence ..... ?

a) Apache 1.0

b) Apache 2.0

c) Apache 1.1

d) Aucune de ces réponses

Question 6

Vous mettez à jour une classe Java vers Kotlin. Que devez-vous utiliser pour remplacer les champs statiques de la classe Java ?

a) objet anonyme

b) propriété statique

c) objet compagnon

d) backing field

Question 7

Quelles sont les trois méthodes de cette classe ? class Person

a) equals(), hashCode(), and toString()

b) equals(), toHash(), and super()

c) print(), println(), and toString()

d) clone(), equals(), and super()

Question 8

Laquelle de ces cibles kotlin ne supporte-t-elle pas?

a) .NET CLR

b) JavaScript

c) LLVM

d) JVM

Correction du QCM

1. b)
2. a)
3. a)
4. a)
5. b)
6. c)
7. a)
8. a)

CHAPITRE n° 2

PRÉPARER L'ENVIRONNEMENT DE DÉVELOPPEMENT

Ce que vous allez apprendre dans ce chapitre :

- Configurer Kotlin
- Découvrir la structure d'une application Kotlin

02 heures

CHAPITRE n° 2

PRÉPARER L'ENVIRONNEMENT DE DÉVELOPPEMENT

1. **Installation et première ligne de code**
2. Structure d'une application Kotlin

Configuration de Gradle

- Kotlin-Gradle-plugin est utilisé pour compiler le code Kotlin avec Gradle. Fondamentalement, sa version devrait correspondre à la version de Kotlin que vous souhaitez utiliser. Par exemple, si vous souhaitez utiliser Kotlin 1.0.3, vous devez utiliser Kotlin-Gradle-plugin version 1.0.3.
- C'est une bonne idée d'externaliser cette version dans Gradle.properties ou dans ExtraPropertiesExtension :

```
buildscript {
  ext.kotlin_version = '1.0.3'

  repositories {
    mavenCentral()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version«
  }

}
```

- Ensuite, vous devez appliquer ce plug-in à votre projet. La façon dont vous procédez diffère lorsque vous ciblez différentes plates-formes.

Sources Kotlin et Java

- Les sources Kotlin et les sources Java peuvent être stockées dans le même dossier ou placées dans des dossiers différents. La convention par défaut consiste à utiliser différents dossiers :

```
project
   - src
      - main (root)
         - kotlin
         - java
```

- La propriété sourceSets correspondante doit être mise à jour si vous n'utilisez pas la convention par défaut :

```
sourceSets.main {
 java.srcDirs("src/main/myJava", "src/main/myKotlin«
)
 }
```

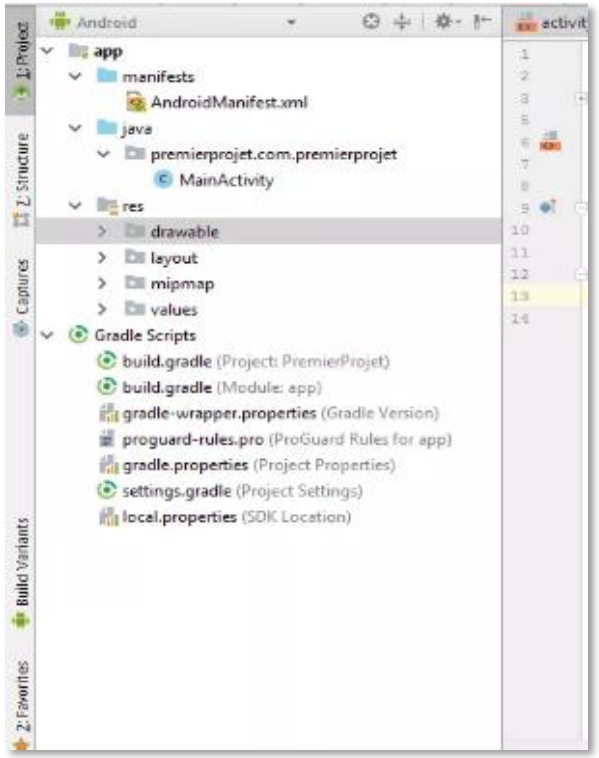
CHAPITRE n° 2

PRÉPARER L'ENVIRONNEMENT DE DÉVELOPPEMENT

1. Installation et première ligne de code
2. **Structure d'une application Kotlin**

Découvrir la structure d'une application Kotlin

- Chaque projet dans Android Studio contient un ou plusieurs modules avec des fichiers de code source ou des fichiers ressource.
- Vous pouvez trouvez des modules tels que :
    - Le module d'application d'Android
    - Les modules des librairies
    - Le module Google App Engine
- Par défaut Android Studio affiche vos fichiers de projet dans la vue Android qui se présente comme dans l'image suivante :
- Cet vue est organisée en module pour vous permettre d'accéder aux principaux fichiers source du module concerné.



PARTIE 1

Découvrir la structure d'une application Kotlin

Tous les fichiers de construction du projet sont visibles sous Gradle Script et chaque module d'application contient les fichiers suivants :

1. **Le Fichier manifeste :** le fichier manifeste (AndroidManifest.xml) contient toutes les informations de votre application telles que :
    - Les permissions
    - L'icône de l'application
    - Le nom de l'application
    - Les composants(Activité, service, BroadcastReceiver, ContenProvider) de votre application
    - Les méta données etc.

Voici un exemple de fichier manifeste :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="premierprojet.com.premierprojet">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Découvrir la structure d'une application Kotlin

2. **Le répertoire Java :** ce répertoire contient tous les fichiers de code source de votre application, y compris les codes de teste JUnit.
3. **Le répertoire res :** ce répertoire contient toutes les ressources de votre application telles que :
    - Les fichiers drawable
    - Une ou plusieurs chaines de caractère utilisées dans votre application
    - Un ou plusieurs fichiers layout
    - Les couleurs etc.

- Le répertoire res contient plusieurs autres sous-répertoires tels que :
    - **drawable :** ce répertoire contient les images, les fichiers drawable de votre application
    - **layout :** ce répertoire contient tous les fichiers layout qui définissent les différentes vues de votre application
    - **mipmap :** ce répertoire contient plusieurs icônes utilisées dans l'application telles que l'icône de l'application
    - **values :** ce répertoire contient plusieurs fichiers tels que le fichier :
        - **colors.xml :** ce fichier contient les couleurs utilisées dans votre application
    - **strings.xml :** ce fichier contient les chaines de caractère utilisées dans votre application
    - **style.xml :** ce fichier contient tous les style de votre application
- Android Studio utilise le gradle comme système de construction de votre projet.
- Le gradle utilise plusieurs fichiers pour définir l'ensemble des configurations de construction qui seront appliquées dans le module application de votre projet ou tous les modules de votre projet

PARTIE 2

S’INITIER À LA PROGRAMMATION KOTLIN

Dans ce module, vous allez :

- Manipuler les objets en Kotlin
- Utiliser les structures de données
- Gérer les erreurs en Kotlin

13 heures

CHAPITRE n° 1

CRÉER DES OBJETS ET CLASSE

Ce que vous allez apprendre dans ce chapitre :

- Créer des classes et héritage
- Comprendre le concept NullSafety en kotlin
- Utiliser des interfaces
- Créer des objets Kotlin

04 heures

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 1

CRÉER DES OBJETS ET CLASSE

1. **Classes et héritage**
2. NullSafety
3. Interfaces
4. Objets Kotlin

Les classes et objet dans Kotlin

- Kotlin est à la fois un langage de programmation orienté objet et fonctionnel.Donc Kotlin supporte les concepts de classe, objet qu'on peut retrouver dans un langages de programmation orienté tel que Java.
- La classe et les objets sont les concepts de base du langage de programmation orienté objet. Ceux-ci prennent en charge les concepts OOP d'héritage, d'abstraction, etc.

À suivre

Dans cette partie ,vous allez apprendre à créer une classe, ajouter des constructeurs à vos classes, créer des objets de ces classes et enfin découvrir les concepts de constructeur primaire et secondaire dans kotlin.

- Une classe est un modèle de définition des objets ayant des caractéristiques identiques(même propriétés et opérations). A partir d'une classe vous pouvez créer plusieurs d'objets et chaque objet sera une instance de cette classe.
- Dans Kotlin, vous pouvez déclarer une classe avec le mot clé class.
- Il existe deux types de déclarations en Kotlin. Tout d'abord il y a celles définies via le mot-clef class qui correspondent au concept de classes de Java.
- Ensuite il existe les déclarations définies via le mot-clef object qui correspondent à une classes qui sera obligatoirement un singleton.

Les classes et objet dans Kotlin

- Syntaxe de la déclaration de classe :

```
class className {
  // class header
  // property
  // member function
}
```

  - **Nom de la classe :** chaque classe a un nom spécifique
  - **En-tête de classe :** l'en-tête se compose des paramètres et des constructeurs d'une classe
  - **Corps de la classe** : entouré d'accolades, contient des fonctions membres et d'autres propriétés
- L'en-tête et le corps de la classe sont facultatifs ; s'il n'y a rien entre les accolades, le corps de la classe peut être omis.

```
class emptyClass
```

Les classes et objet dans Kotlin

- Le constructeur est une méthode spéciale de classe utilisé pour initialiser les propriétés d'un objet de la classe lors de la création de l'objet.
- Le constructeur est appelé durant la création de la classe. Vous devez aussi savoir que si vous ne déclarez pas un constructeur, le compilateur crée un constructeur par défaut pour votre classe qui servira à instancier la classe. Dans kotlin,il existe deux types de constructeur: le constructeur primaire et le constructeur secondaire.
- Le constructeur primaire fait partie de l'entête de la classe et est déclaré après le nom de la classe comme suit.

```
class Personne constructor (val nom: String, val prenom: String ){
}
```

Les classes et objet dans Kotlin

- Vous pouvez créer une objet de cette classe de la manière suivante :

```
fun main(){
  val personne = Personne("VIGAN","Joel")
}
class Personne constructor (val nom: String, val prenom: String ){
}
```

- Si le constructeur primaire ne possède aucune annotation ou des modificateurs d'accès(public, private,protected etc), le mot clé constructor peut être omit comme suit.

```
class Personne (val nom: String, val prenom: String ){
}
```

Le bloc d'initialisation

- Vous pouvez voir le constructeur primaire comme l'entête d'une fonction avec des paramètres qui peuvent être utilisés à l'intérieur de la fonction.De la même façon que les propriétés d'une fonction sont accessibles dans la fonction, les propriétés du constructeur primaire sont accessibles dans toute la classe.
- Le constructeur primaire ne doit contenir aucun code.Le code d'initialisation de la classe peut être défini dans le bloc d'initialisation qui est préfixé avec le mot clé init .

```
class Personne constructor(var nom: String, var prenom: String) {
  init {
     nom = nom.toUpperCase()
  }
}
```

- Dans cette exemple le bloc d'initialisation est utilisé pour transformer le nom en majuscule
- Vous pouvez déclarer plusieurs blocs d'initialisation dans une classe et ces blocs d'initialisation seront exécutés dans l'ordre dans lequel ils apparaissent.

```
class Personne (var nom: String, var prenom: String ){
  init {
     nom=nom.toUpperCase()
     println("Le nom de la personne est $nom")
  }
  init {
     prenom=prenom.toUpperCase()
     println("Le prenom de la personne est $prenom")
  }}
```

Le bloc d'initialisation

- L'exemple suivant montre comment créer une instance de la classe Personne.

```
fun main(){
  val personne = Personne("vigan","Joel")
}
class Personne (var nom: String, var prenom: String ){
  init {
     nom=nom.toUpperCase()
     println("Le nom de la personne est $nom")
  }
  init {
     prenom=prenom.toUpperCase()
     println("Le prenom de la personne est $prenom")
  }
}
```

- **Le résultat est :**
  - Le nom de la personne est VIGAN
  - Le prenom de la personne est JOEL

Le bloc d'initialisation

- Le bloc d'initialisation peut aussi être utilisé pour initialiser les propriétés membres de la classe comme suit.

```
fun main(){
  val personne = Personne("vigan","Joel")
}
class Personne (var nom: String, var prenom: String ){
  val dateNaissance: Int
  val email: String
  init {
     dateNaissance = 2019
     email="toto@gmail.com"
     println("La date de naissance est $dateNaissance")
     println("L'email est $email")
  }
```

- **Le résultat est :**
  - La date de naissance est 2019
  - L'email est toto@gmail.com

Le bloc d'initialisation

- ET enfin, vous pouvez aussi initialiser les propriétés de la classe directement dans le corps de la classe comme suit

```
fun main(){
  val personne = Personne("vigan","Joel",2019)
}
class Personne (var nom: String, var prenom: String,val dateNaissance:
Int ){
  val mDateNaissance: Int= dateNaissance //Initialisation
  init {
     println("La date de naissance est $mDateNaissance")
  }
}
}
```

- **Le résultat est :**
    - La date de naissance est 2019

L'héritage dans Kotlin

- L'héritage est l'un des concepts les plus importants de la programmation orienté objet. Il permet à une nouvelle classe d'hériter de toutes les fonctionnalités (propriétés et fonctions membres) d'une classe existant.
- Ainsi, la classe qui hérite des fonctionnalités est appelée classe dérivée ou classe fille ou sous classe et la classe dont les fonctionnalités sont héritées est appelée classe de base ou classe parent.

À suivre

- Dans cette partie vous allez découvrir comment créer une classe dérivée dans kotlin.

L'héritage dans Kotlin

- Toutes les classes dans kotlin ont une super classe commune appelée Any. Toutes les classes que vous créez dans kotlin héritent implicitement de la classe Any sans que cela ne soit explicitement déclaré.
- Prenons comme exemple la classe Jeu suivante :

```
class Jeu
```

- La classe Jeu hérite implicitement de la classe Any. Puisque la classe Any possède trois(3) méthodes equals(), hashCode() et toString().Ces méthodes sont donc toutes définies dans toutes les classes que vous allez créer.c
- Vous pouvez par exemple redéfinir ou implémenter ces méthodes dans la classe Jeu et ajouter des fonctionnalité qui seront propre à la classe Jeu.

L'héritage dans Kotlin

- Voici comment créer une classe Chien qui hérité d'une classe de base Animal

```
open class Animal{
}

class Chien:Animal(){
}
```

- Par défaut,toutes les classes que vous créez dans Kotlin sont **final** c'est à dire des classes qui ne peuvent pas être héritées.
- Pour permettre à une classe d'être héritable vous devez la marquer avec **le modificateur open** comme cela est fait avec la **classe Animal** précédente.
- Vous devez aussi remarquer que dans l'exemple précédent , la classe dérivée Jeu déclare explicitement son super type Animal en le plaçant dans son entre après les deux points. Une classe dérivée ne se content pas juste de marquer sont super type, elle doit aussi initialiser la classe de base(son super type).

L'héritage dans Kotlin

- Si la classe de base ne possède pas de constructeur primaire alors,la classe dérivée doit initialiser la classe de base en appelant le constructeur par défaut de la classe de base comme dans l'exemple précédent ou comme dans l'exemple suivant

```
open class Personne {
}

class Eleve:Personne(){
}
```

- Vous devez constater dans cet exemple que la classe dérivée Eleve initialise la classe de base Animal en appelant le constructeur par défaut de la classe Animal.
- Si La classe de base possède un constructeur primaire avec des paramètres et si la classe dérivée aussi possède un constructeur primaire, la classe dérivé doit initialiser la classe de base dans son entête avec les paramètres qui sont passés dans son constructeur primaire.

L'héritage dans Kotlin

- Donc le constructeur primaire de la classe dérivée doit avoir dans la liste de ses paramètres,les paramètres du constructeur primaire de la classe de base pour pouvoir initialiser la classe de base lors de la création de la classe dérivée. Voici un exemple qui illustre le cas d'une classe de base et une classe dérivé avec tout deux un constructeur primaire

```
open class Animal (val name: String){
}
class Chien(name: String):Animal(name){
}
```

- Vous devez constater que la **classe dérivée Chien** initialise le classe de base **Animal** en appelant le constructeur primaire de la classe de base(**Animal**) et en passant son paramètre **name** au constructeur primaire de la classe de base(**Animal**).

L'héritage dans Kotlin

- Si la classe dérivée ne possède pas de constructeur primaire et le classe de base possède un constructeur primaire,tous les constructeurs secondaires de la classe dérivée doivent se charger d'initialiser la classe de base avec le mot clé super ou déléguer l'initialisation à tout autre constructeur.
- Voici un exemple qui illustre le case d'une classe dérivée sans constructeur primaire et une classe de base avec un constructeur primaire

```
open class Animal (val name: String){
}
class Chien:Animal{
  constructor(name: String):super(name){
  }
  /*
  * Ce constructeur délègue l'initialisation à un autre
constructeur
  * de ce classe avec le mot clé this
  * */
  constructor(name: String,race: String ):this(name){
  }
}
```

L'héritage dans Kotlin

- Dans cette exemple la classe dérivée Chien initialise la classe de base Animal à partir de son constructeur secondaire avec le mot clé this.
- La classe dérivée peut aussi initialiser la classe de base en appelant un constructeur primaire ou un constructeur secondaire de la classe de base comme dans l'exemple suivant :

```
open class Personne(val nom: String){
  constructor(name: String,prenom:String):this(name){
  }
}
class Eleve:Personne{
  constructor(nom:String):super(nom){
  }
  constructor(nom: String,prenom:
String):super(nom,prenom){
  }
}
```

- Dans cet exemple, la classe dérivée **Eleve** initialise la classe de base **Personne** à partir d'un constructeur secondaire de la classe de base **Animal** avec le mot clé **super**

L'héritage dans Kotlin - Les fonctions membres

- Lorsqu'une classe dérive d'une classe de base, elle hérite des fonctions membres de la classe de base. Toute comme les classes, par défaut les fonctions membres que vous ajoutez dans une classe sont marqué final c'est à dire qu'elles ne peuvent pas être redéfinies dans une classe dérivée.
- Tout d'abord pour permettre à une fonction d'être redéfinie(implémentée) dans une classe dérivée, vous devez la marquer open. Ensuite vous devez marquer la fonction membre de la classe de base avec le modificateur override dans la classe dérivée sinon, le compilateur vous signalera une erreur.
- Voici un exemple qui illustre comment redéfinir une fonction membre d'une classe de base dans une classe dérivée.

```
fun main(){
  val chien=Chien("Bouledogue")
  chien.afficher()
}

open class Animal (val name: String){
  open fun afficher(){
     println("Mon nom d'animal est $name")
  }
}
```

- **Le résultat est :** Mon nom de chien est Bouledogue

L'héritage dans Kotlin - Les fonctions membres

- Une fonction membre marquée override dans une classe dérivée peut aussi être redéfini dans une autre classe dérivé.si vous souhaitez empêcher cette fonction d'être redéfinie dans une autre classe, vous devez la marquer final. Voir l'exemple suivant.

```
open class Animal (val name: String){
  open fun afficher(){
     println("Mon nom d'animal est $name")
  }
}
class Chien(name: String):Animal(name){
  final override fun afficher() {
     println("Mon nom de chien est $name")
  }
}
```

- Dans cet exemple, la fonction membre afficher est marquée final pour empêcher qu'elle soit redéfinie dans une autre classe dérivée.

L'héritage dans Kotlin - Les propriétés redéfinies

- La redéfinition d'une propriété dans une classe dérivée fonctionne comme la redéfinition de fonction membre dans une classe dérivée.C'est à dire la propriété à redéfinir doit être marquée open dans la classe de base et lorsqu'elle est déclarée à nouveau ,dans la classe dérivé, elle doit être marquée override .

Voir l'exemple suivant :

```
open class Animal (val name: String){
  open var couleur: String = "Noir"
  open fun afficher(){
     println("Mon nom d'animal est $name")
  }
}
class Chien(name: String):Animal(name){
  override var couleur: String = "Bringé"
  override fun afficher() {
     println("Mon nom de chien est $name")
  }
}
```

- Dans cet exemple, la propriété couleur est marquée **open** dans la classe de base et marquée override dans la classe dérivée.

L'héritage dans Kotlin - Redéfinition d'une propriété val en var

- Vous pouvez aussi redéfinir une propriété immuable(propriété marquée **val**) en une propriété mutable (propriété marquée var).
- Cela est possible parce qu'une **propriété val** déclare un getter.En définissant à nouveau dans une classe dérivé une propriété étant immuable en une propriété mutable(**propriété marqué var**), vous pourrai ajouter en plus un setter dans la classe dérivée.
- Donc puisque c'est un ajout de setter ,et non une suppression du getter alors ça fonctionne.

Voici un exemple qui illustre la redéfinition dune propriété val en var :

```
open class Animal (val name: String){
  open val couleur: String = "Noir"
  open fun afficher(){
     println("Mon nom d'animal est $name")
  }
}
class Chien(name: String):Animal(name){
  override var couleur: String = "Bringé"
  override fun afficher() {
     println("Mon nom de chien est $name")
  }
}
```

L'héritage dans Kotlin - Redéfinition d'une propriété val en var

- Dans cet exemple la propriété couleur est immuable dans la classe de base (propriété marquée val) alors qu'elle est mutable dans la classe dérivée. Si vous créez une instance de la classe de base(Animal) vous ne pouvez pas modifier la couleur défini comme suit.

```
fun main(){
  val animal=Animal("Chien")
  animal.couleur="Merle" //Erreur du compilateur
}
```

- Dans la classe Animal, la propriété couleur est immuable(propriété marquée val) Tandis que pour la classe dérivée de la classe animal (Chien) , vous pouvez modifier la valeur comme suit.

```
fun main() {
  val animal = Chien("Chien")
  animal . couleur ="Merle" //Erreur du compilateur
}
```

- Dans la classe Chien , la propriété couleur est mutable(propriété marquée var)

L'héritage dans Kotlin - Redéfinition des getters et setters

- Vous pouvez redéfinir une propriété d'une classe de base dans une classe dérivée en initialisant à nouveau la propriété ou en définissant des getters et setters personnalisés .

Voir l'exemple suivant :

```
open class Animal (val name: String){
  open var couleur: String = "Noir"
  open fun afficher(){
     println("Mon nom d'animal est $name")
  }
}
class Chien(name: String):Animal(name){
  override var couleur: String = "Bringé"
     get()=field.capitalize()
     set(value) {
        field = value.capitalize()
     }

  override fun afficher() {
     println("Mon nom de chien est $name")
  }
}
```

- Dans cet exemple, les accesseurs personnalisés de la propriété couleur sont définis dans la classe dérivée.

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 1

CRÉER DES OBJETS ET CLASSE

1. Classes et héritage
2. **NullSafety**
3. Interfaces
4. Objets Kotlin

Kotlin Null Safety

- L'un des bugs les plus récurent que l'on rencontre dans les applications c'est NullPointerException. Cela se produit par exemple lorsqu'un utilisateur clique sur un bouton dans une application et il y'a un appel d'une méthode sur une variable contenant une valeur null.
- Il est possible d'éliminer cette exception qui se produit durant l'exécution des applications. Cela est possible grâce au système de type de Kotlin qui vise à éliminer le risque de référence null dans le code.

À suivre

- Dans cette partie, nous allons voir comment éliminer les risques de référence null en **Kotlin** (**Kotlin Null Safety**).

Kotlin Null Safety - Types Null (Nullable types)

- Par défaut dans kotlin, le compilateur n'autorise pas aux variables de référencer une valeur null. Par exemple, le code suivant génère une erreur dans Kotlin

```
var nom: String = "Joel"
nom = null; //Erreur: Null can not be a value of non-null type String
```

- C'est grâce au système de nullabilité dans kotlin, que vous pouvez explicitement déclarer une variable pouvant contenir une valeur null ou non.
- Ce qui permet donc au compilateur de savoir à l'avance qu'une variable peut référencer une valeur null. Il pourra ainsi vous imposer de vérifier si cette variable est null avant d'appeler une méthode sur cette variable ou d'accéder à une propriété de cette variable.
- C'est ainsi que Kotlin vous permet d'éliminer l'exception NullPointerException. Nous allons voir cela plus en détail.

Kotlin Null Safety - Types Null (Nullable types)

- Pour notifier au compilateur dans kotlin que vous souhaitez déclarer une variable pouvant contenir une valeur null,
- Vous devez ajouter un point d'interrogation (?) après le type de la variable comme suit :
- En ajoutant un point d'interrogation à la fin du type de la variable **nom** dans l'exemple précédent, le compilateur sait que la variable **nom** peut contenir un valeur null et donc vous empêcher d'appeler une méthode ou d'accéder à une propriété de la variable **nom**, pour éviter l'exception **NullPointerException**.
- Le code suivant ne fonctionne pas dans Kotlin :

```
var nom: String? = "Joel"
nom=null
nom.length // Le compilateur génere une erreur
nom.trim() // Le compilateur génere une erreur
```

- Avant d'appeler une méthode ou d'accéder à une propriété d'une variable pouvant référencer une valeur null, vous devez vérifier que cette variable n'est pas null.

Kotlin Null Safety - Types Non Null (Non-Null Types)

- Par défaut, une variable dans Kotlin est de type non null. Donc vous n'avez besoin de ne rien faire pour déclarer une variable de type non null

```
var nom: String = "Joel"
val nb: Int = nom.length // Correct
nom.trim() // Correct
```

- A présent vous savez comment indiquer au compilateur que vous souhaitez avoir une variable pouvant contenir une valeur null ; et que le compilateur vous empêchera d'appeler une méthode ou d'accéder à une propriété de cette variable pouvant contenir une valeur null.
- Il vous reste à présent de savoir comment effectuer une vérification sur une variable pouvant contenir une valeur null avant de l'utiliser. Le compilateur vous aura ainsi épargné de l'exception **NullPointerException**.
- Il existe plusieurs manières de vérifier si une variable est null avant de pouvoir l'utiliser

Kotlin Null Safety - L'opérateur d'appel sécurisé ?.

- Il existe dans Kotlin, une manière simple et concise d'appeler une méthode sur une variable si et seulement si cette variable ne fait pas référence à une valeur null.
- Cela est possible grâce à l'opérateur « ? ». En fait cet opérateur combine deux choses à la fois :
    - Vérifier si la variable contient une valeur null
    - L'appel d'une méthode sur la variable si la valeur référencée par la variable n'est pas null. Tout cela se fait en une seul expression
- Voir le code suivant :

```
var nom: String? = "Joel"
nom?.toLowerCase()
```

Kotlin Null Safety - L'opérateur d'appel sécurisé ?. et la méthode let

- En utilisant l'opérateur « ?. » et la méthode **let**, vous pourrez exécuter un bloc de codes lorsque la variable n'est pas null. Dans un prochain tutoriel nous parlerons de la méthode let dans Koltin.

```
var nom: String? = "Joel"
nom?.let{
  it.toLowerCase()
  it.trim()
}
```

Kotlin Null Safety - L'opérateur Elvis ? : (Elvis operator)

- L'opérateur Elvis permet de définir une valeur par défaut lorsque la variable fait référence à un valeur null.

Voir l'exemple de code suivant :

```
var nom: String? = "Joel"
val nomPersonne = nom ?: "Noe"
```

- Vous pouvez combiner l'opérateur d'appel sécurisé « ?. » et l'opérateur Elvis « ?: » pour affecter une valeur par défaut lorsque la variable sur laquelle la méthode est appelée est null ou lorsque la variable sur laquelle on n'essaie d'accéder a une propriété null .
- Donc l'opérateur Elvis permet de fournir une valeur par défaut ou d'exécuter un bloc de codes lorsque la valeur contenu dans la variable vérifiée par l'opérateur d'appel sécurisé est null.

Kotlin Null Safety - L'opérateur !!

- L'opérateur d'assertion non null !!, contrairement à l'opérateur d'appel sécurisé « ?. » permet de convertir une variable de type null en une variable de type non null puis lance l'exception **NullPointerException** si :
    - cette variable contient une valeur null
    - on essaie d'accéder à une propriété de cette variable

Voir l'exemple de code suivant :

```
var nom: String? = null
nom!!.toLowerCase() //Erreur
```

- Dans cette partie nous avons appris comment éliminer les risques de référence null en vérifiant de manière simple et concise les variables pouvant contenir une valeur null avant de les utiliser.

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 1

CRÉER DES OBJETS ET CLASSE

1. Classes et héritage
2. NullSafety
3. **Interfaces**
4. Objets Kotlin

Les interfaces dans kotlin

- Comme les classes abstraites, les interfaces ne peuvent pas être instanciées mais peuvent contenir des déclarations de méthodes abstraites et de méthodes implémentées.
- Ce qui rend une interface différente d'une classe abstraite est que, une interface ne peut pas conserver d'état, c'est à dire que vous ne pouvez pas déclarer un propriété et l'affecter une valeur dans une interface. Les propriétés sont donc abstraites ou fournissent une implémentation des accesseurs.
- Un interface est définie avec le mot clé interface suivit du nom de l'interface

```
interface MyInterface{
  fun methode1()
  fun methode2(){
     println("Méthode 1")
  }
}
```

- Dans cet exemple, l'interface MyInterface possède une méthode abstraite et une méthode implémentée

Les interfaces dans kotlin

- Une classe ou un objet peut implémenter une interface. Les méthodes implémentées dans la classe dérivée doivent être marquées override.

```
interface MyInterface{
  fun methode1()
  fun methode2(){
     println("Méthode 1")
  }
}
class MyClasse:MyInterface{
  override fun methode1() {
  }
}
```

Les interfaces dans kotlin - Les propriétés dans une interface.

- Vous pouvez déclarer des propriétés dans une interface. Une propriété déclarée dans une interface peut être abstraite ou fournir l'implémentation de ses accesseurs.
- Il faut noter que vous n'avez pas besoin de marquer la propriété avec **le mot clé abstraite** pour la rendre abstraite. Dès que vous déclarer la propriété dans l'interface, elle est par défaut abstraite.
- Puisque qu'une propriété abstraite n'est pas initialisée dans une interface, la propriété ne possède pas de champs auto généré(**backing field**) dont l'identificateur est field.
- Les accesseurs n'ont donc pas accès au champ auto généré. **Le backing field (field**) est l'identificateur utilisé comme référence à une propriété pour accéder ou modifier cette propriété dans les accesseurs de la propriété.
- Voici un exemple qui montre comment déclarer une propriété abstraite dans une interface.

```
interface MyInterface{
  val propriete1: String
  fun methode1(){
     println(propriete1)
  }
}
class MyClasse:MyInterface{
  override val propriete1: String="Propriété"
}
```

Les interfaces dans kotlin - Les propriétés dans une interface.

- Cet exemple illustre cette fois-ci comment déclarer une propriété et définir ses accesseurs dans une interface.

```
fun main(){
  val myClasse=MyClasse()
  myClasse.methode1()

}
interface MyInterface{
  val propriete1: String
     get() = "Propriété"
  fun methode1(){
     println(propriete1)
  }
}
class MyClasse:MyInterface{

}
```

Le résultat est : Propriété

- Dans cet exemple la classe dérivée ne redéfinie pas la propriété déclarée dans l'interface puisque l'accesseur fourni déjà la valeur de la variable

Les interfaces dans kotlin - L'héritage d'interface

- Une interface peut dériver d'une autre interface, et peut implémenter les membres de l'interface de base. Chaque interface peut aussi déclarer ses nouvelles propriétés et fonctions membres.

Prenons l'exemple suivant :

```
interface Perimetre{
  var perimetre: Int
}
interface Calcul:Perimetre{
  val largeur: Int
  val longueur: Int
  override var perimetre: Int get() = (longueur +
largeur) * 2
     set(value) {}
}
```

- Dans cette exemple, l'interface Calcul qui dépend de l'interface Perimetre par la propriété perimetre redéfinie la propriété perimetre. Puisque la propriété perimetre est mutable, vous devez fournir les accesseurs(getter et setter) de cette propriété dans l'interface dérivée. Comme les propriétés ne peuvent pas être initialisées dans une interface, elle ne possède donc pas de champ auto généré(backing field), le setter est quand même défini mais n'affect aucune valeur(un setter vide). Le getter quand à lui renvoie une valeur qui ne dépend pas du champ auto généré (backing field) .

Les interfaces dans kotlin - L'héritage d'interface

- Vous pouvez ne pas définir de setter, il suffit juste de déclarer la propriété périmètre comme immuable (une propriété marquée val) comme suit interface

```
Perimetre{
  val perimetre: Int
}
interface Calcul:Perimetre{
  val largeur: Int
  val longueur: Int
  override val perimetre: Int get() = (longueur + largeur) * 2 }
```

- Toute classe qui hérite de ces interfaces devra fournir une implémentation des propriétés et méthodes qui n'ont pas encore été défini dans les interfaces.

Prenons l'exemple suivant :

```
fun main(){
  val rectangle= Rectangle(200,200)
  val perimetre = rectangle.perimetre
  println("Le périmètre du rectangle est $perimetre")
}
interface Perimetre{
  val perimetre: Int
}
interface Calcul:Perimetre{
  val largeur: Int
  val longueur: Int
  override val perimetre: Int get() = (longueur + largeur) * 2
}
class Rectangle(override val largeur: Int, override val longueur: Int):Calcul{
}
```

PARTIE 2

CHAPITRE n° 1

CRÉER DES OBJETS ET CLASSE

1. Classes et héritage
2. NullSafety
3. Interfaces
4. **Objets Kotlin**

Les objets Kotlin

C'est une unité de base de la programmation orientée objet et représente les entités réelles, qui ont des états et des comportements. Les objets sont utilisés pour accéder aux propriétés et aux fonctions membres d'une classe. Dans Kotlin, nous pouvons créer plusieurs objets d'une classe.

Un objet est composé de :

- **Etat :** Il est représenté par les attributs d'un objet. Il reflète également les propriétés d'un objet.
- **Comportement :** Il est représenté par les méthodes d'un objet. Il reflète également la réponse d'un objet à d'autres objets.
- **Identité :** Elle donne un nom unique à un objet et permet à un objet d'interagir avec d'autres objets.

https://kotlinlang.org

Les objets Kotlin

Créer un objet

Nous pouvons créer un objet en utilisant la référence de la classe.

```
var obj = className()
```

Accéder à la propriété de la classe :

Nous pouvons accéder aux propriétés d'une classe à l'aide d'un objet. Commencez par créer un objet à l'aide de la référence de classe, puis accédez à la propriété.

```
obj.nameOfProperty
```

Accéder à la fonction membre de la classe :

Nous pouvons accéder à la fonction membre de la classe en utilisant l'objet.

```
obj.funtionName(parameters)
```

Les objets Kotlin

Programme Kotlin de création de plusieurs objets et d'accès à la propriété et à la fonction membre de la classe :

```
class employee {
   // Constructor Declaration of Class
   var name: String = ""
   var age: Int = 0
   var gender: Char = 'M'
   var salary: Double = 0.toDouble()
   fun insertValues(n: String, a: Int, g: Char, s: Double) {
      name = n
      age = a
      gender = g
      salary = s
      println("Name of the employee: $name")
      println("Age of the employee: $age")
      println("Gender: $gender")
      println("Salary of the employee: $salary")
   }
   fun insertName(n: String) {
      this.name = n
   }
}

fun main(args: Array<String>) {
var obj = employee()
var obj2 = employee()
obj.insertValues("Praveen", 50, 'M', 500000.00)
obj2.insertName("Aliena")
println("Name of the new employee: ${obj2.name}")
}
```

CHAPITRE n° 2

MAÎTRISER LES CLASSES ENUM ET SEALED

Ce que vous allez apprendre dans ce chapitre :

- Créer des classes enum
- Comprendre la différence entre la classe Enum et la classe Sealed
- Créer des classes sealed

02 heures

CHAPITRE n° 1

MAÎTRISER LES CLASSES ENUM ET SEALED

1. **Sealed class**
2. Data class

La classe scellée

- Kotlin fournit un nouveau type de classe important qui n'est pas présent dans Java. Celles-ci sont connues sous le nom de **classes scellées**.
- Comme le mot scellé le suggère, les classes scellées se conforment à des hiérarchies de classes **restreintes** ou **délimitées** .
- Une classe scellée définit un ensemble de sous-classes en son sein. Il est utilisé lorsqu'il est connu à l'avance qu'un type sera conforme à l'un des types de sous-classe.
- Les classes scellées garantissent la sécurité des types en limitant les types à faire correspondre au moment de la compilation plutôt qu'au moment de l'exécution.
- Une classe scellée est comme une extension aux énumérations car l'ensemble des valeurs d'une énumération est également restreint.
- Mais il n'existe qu'une seule instance de chaque constant d'une énumération alors qu'une sous classe d'une classe scellée peut avoir plusieurs instances pouvant contenir un état.

La classe scellée - Déclarer une classe scellée ou sealed

- Pour déclarer une classe comme étant scellée :

```
sealed class Personne
```

- Les classes scellées sont essentiellement utiles lorsqu'elles sont utilisées dans une évaluation avec une expression when.
- Par exemple, supposons que vous souhaitez créer une classe Forme qui a deux (2) sous classes Carre, Rectangle dont voici l'implémentation.

```
open class Forme(val longueur: Int,val largeur: Int){
}
class Rectangle (longueur: Int,largeur: Int) :Forme(longueur,largeur){
}
class Carre (longueur: Int,largeur: Int) :Forme(longueur,largeur){
}
```

La classe scellée

- Ensuite vous souhaitez par exemple créer une instance de la classe Rectangle et utiliser une expression when pour évaluer le type de forme dont il s'agit durant l'exécution du programme comme suit.

```
fun main() {
  val rectangle = Rectangle(200, 200)
  println(check(rectangle))
}
fun check(forme: Forme): String {
  val formeResult = when (forme) {
     is Rectangle -> "C'est un rectangle"
     is Carre -> "C'est un carré"
     else -> "Ce n'est pas une forme"
  }
  return formeResult
}
```

- **Le résultat est :** C'est un rectangle

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 2

Maitriser les classes data et sealed

Data class

Les projets de grande envergure comportent plusieurs classes qui sont uniquement destinées à contenir des données. Bien que ces classes n'aient que très peu ou pas de fonctionnalités, un développeur doit écrire beaucoup de code standard en Java.

En général, le développeur doit définir un constructeur, plusieurs champs pour stocker les données, des fonctions getter et setter pour chacun des champs, ainsi que des fonctions equals(), hashCode() et toString().

Kotlin propose une méthode très simple pour créer de telles classes. Le développeur Kotlin n'a qu'à inclure le mot-clé data dans la définition de la classe, le compilateur se chargera lui-même de l'ensemble de la tâche.

Data class

Prenons un exemple de syntaxe pour une classe de données écrite en Java et Kotlin pour obtenir une comparaison claire :

La classe Java :

```
class Book {
  private String name;
  private String author;
  private int bookPhoto;

  public Book(String name, String author, int bookPhoto) {
    this.name = name;
    this.author = author;
    this.bookPhoto = bookPhoto;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getAuthor() {
    return author;
  }
  public void setAuthor(String author) {
    this.author = author;
  }
  public int getBookPhoto() {
    return bookPhoto;
  }
  public void setBookPhoto(int bookPhoto) {
    this.bookPhoto = bookPhoto;
  }
}
```

Data class

La classe Kotlin :

```
data class Book(val name:String, val author:String,val bookPhoto: Int)
```

L'exemple de code ci-dessus met clairement en évidence la brachylogie de Kotlin.

En dehors de cela, l'utilisation des fonctions Delegates, KotlinLib et One-line, aucune utilisation de "findViewbyids", un code plus court pour "setOnClickListener" sont quelques exemples supplémentaires qui aident à réduire le boilerplate code, faisant de Kotlin un langage puissant .

CHAPITRE n° 3

Utiliser les classes nested et enum

Ce que vous allez apprendre dans ce chapitre :

- Utiliser les classes nested
- Utiliser la classe enum

04 heures

CHAPITRE n° 3

Utiliser les classes nested et enum

1. **Nested class**
2. Enum

Nested class

- Les classes peuvent être imbriquées dans d'autres classes :

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

- Vous pouvez également utiliser des interfaces avec imbrication. Toutes les combinaisons de classes et d'interfaces sont possibles : Vous pouvez imbriquer des interfaces dans des classes, des classes dans des interfaces et des interfaces dans des interfaces.

```
interface OuterInterface {
 class InnerClass
 interface InnerInterface
}
class OuterClass {
 class InnerClass
 interface InnerInterface
}
```

Inner classes

- Une classe imbriquée marquée comme interne peut accéder aux membres de sa classe externe. Les classes internes portent une référence à un objet d'une classe externe :

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

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 3

Utiliser les classes nested et enum

1. Nested class
2. **Enum**

La classe Enum

- Un type énuméré (appelé souvent énumération ou juste enum, parfois type énumératif ou liste énumérative) est un type de données qui consiste en un ensemble de valeurs constantes. Ces différentes valeurs représentent différents cas ; on les nomme énumérateurs. Lorsqu'une variable est de type énuméré, elle peut avoir comme valeur n'importe quel cas de ce type énuméré.
- Une énumération est un type de données utilisé pour définir un ensemble de constant. Dans Kotlin chaque constant de l'énumération est un objet .
- Ces constantes sont séparées par une virgule.
- Une énumération se déclare en ajoutant le mot clé enum devant le mot class. En effet dans kotlin une énumération est une classe.

```
enum class STATUT{
  LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMANCHE
}
```

La classe Enum - Initialiser les constantes d'une énumération

- Puisque une énumération est une classe et une constante est une instance d'une **classe enum** alors les constantes peuvent être initialisées en passant des paramètres au constructeur de la **classe enum**.
- Il faut ensuite définir chaque constante avec les valeurs correspondant aux paramètres du constructeur de la **classe enum**.

```
enum class STATUT(val jour: Int){
LUNDI(1),MARDI(2),MERCREDI(3),JEUDI(4),VENDREDI(5),SAMEDI(6),DIMANCHE(7)
}
```

- comme suite pour accéder à la valeur jour de chaque constante :
    - **STATUT.LUNDI.jour**

La classe Enum - Les classes anonymes et les constantes d'énumération

- Puisque chaque constante d'une énumération est une instance d'une classe enum alors elles peuvent définir leurs classes anonymes contenant leur méthodes et les méthodes à redéfinies de la classe enum de base.
- Si la classe enum définie des membres, il faut séparer les constantes et les membres de la classe enum par un point virgule.
- Pour appeler les méthodes redéfinies dans chaque constante d'énumération :
    - SEMAINE.DIMANCHE.numeroSemaine()

```
enum class SEMAINE{
  LUNDI{
     override fun numeroSemaine() {}
  },MARDI {
     override fun numeroSemaine() {}
  },MERCREDI {
     override fun numeroSemaine() {}
  },JEUDI {
     override fun numeroSemaine() {}
  },VENDREDI {
     override fun numeroSemaine() {}
  },SAMEDI {
     override fun numeroSemaine() {}
  },DIMANCHE {
     override fun numeroSemaine() {}
  };
  abstract fun numeroSemaine()
}
```

La classe Enum - Opérations sur les constantes d'énumérations

- Les classes enum ont des méthodes synthétiques permettant de lister les constantes d'énumération définies ou d'obtenir une constante enum par son nom.
- Il s'agit des méthodes valueOf(value: String) et values() .
- La méthode **valueOf(value: String)** lève l'exception **IllegalArgumentException** si le nom spécifié ne correspond à aucun des constantes d'énumération définies dans la **classe enum**.
- Pour retourner le nombre de constante de l'énumération, on utilise la méthode **values()** comme suite :

```
println(SEMAINE.values().size)
```

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 4

MAITRISER LES EXTENSIONS

Ce que vous allez apprendre dans ce chapitre :

- Utiliser les collections dans Kotlin
- Créer des extensions

03 heures

CHAPITRE n° 4

MAITRISER LES EXTENSIONS

1. **Extensions**
2. Collections et ranges

Les extensions

- Kotlin offre la possibilité d'étendre une classe avec de nouvelles fonctionnalités sans avoir à hériter de la classe ou à utiliser des modèles de conception tels que Decorator. Cela se fait via des déclarations spéciales appelées extensions .
- Par exemple, vous pouvez écrire de nouvelles fonctions pour une classe à partir d'une bibliothèque tierce que vous ne pouvez pas modifier. De telles fonctions peuvent être appelées de la manière habituelle comme s'il s'agissait de méthodes de la classe d'origine. Ce mécanisme est appelé fonctions d'extension .
- Il existe également des propriétés d'extension qui vous permettent de définir de nouvelles propriétés pour les classes existantes.

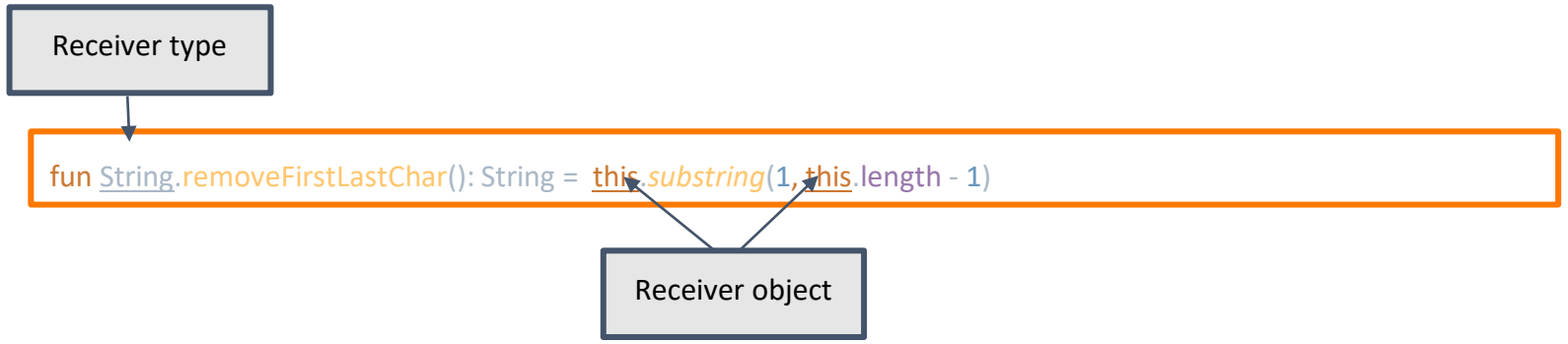
Exemple : Supprimer le premier et le dernier caractère de la chaîne :

```
fun String.removeFirstLastChar(): String =  this.substring(1, this.length - 1)

fun main(args: Array<String>) {
  val myString= "Hello Everyone"
  val result = myString.removeFirstLastChar()
  println("First character is: $result")
}
```

Les extensions

- **Le resultats est :** First character is: ello Everyon
- Ici, une fonction d'extension removeFirstLastChar() est ajoutée à la classe String. Le nom de la classe est le type de récepteur (classe String dans notre exemple). Le mot clé this à l'intérieur de la fonction d'extension fait référence à l'objet récepteur.



Extension de fonction

- Pour déclarer une extension de fonction , vous devez préfixer son nom avec le type receveur ou avec le nom de la classe que vous souhaitez ajouter une fonctionnalité.
- L'exemple suivant vous montre comment ajouter une fonction d'extension **supprimer()** dans la **classe Client**

```
fun main(){
  fun Client.supprimer (){
     println("Suppression d'un client")
  }
  val client = Client("VIGAN","Noé")
  client.ajout()
  client.supprimer()
}
class Client(val nom:String,val prenom: String){
  fun ajout(){
     println("Ajout client")
  }
}
```

- **Le résultat est :**
  - Ajout client
  - Suppression d'un client
- Dans cet exemple, la **fonction d'extension supprimer()** est ajouté à la classe Client en utilisant la notation pointée **Client.supprimer**.
- Ensuite, une instance de la **classe Client** est créé et les fonctions **ajout()** et **supprimer()** sont appelées sur l'objet client.

Ajouter des fonctionnalités aux classes de librairie

- Il existe plusieurs fonctions d'extension dans les librairies standard de **Kotlin**.
- **Kotlin** vous permet aussi d'ajouter des fonctionnalités aux classes de librairie standard sans les dériver grâce aux fonctions d'extension.
- Voici un exemple qui vous montre comment ajouter un fonction d'extension aux librairies standard de **Kotlin**

```
fun main(){
  fun MutableList<String>.shift(){
     if(!this.isEmpty()){
        this.removeAt(0)
     }
     else throw NoSuchElementException("Collection vide")
  }
  val listName: MutableList<String> = mutableListOf("Noé","Jean","Carole","Rose","Luc")
  println(listName)
  listName.shift()
  println(listName)
}
```

Ajouter des fonctionnalités aux classes de librairie en utilisant les fonctions d'extension

- **Le résultat est :**
  - [Noé, Jean, Carole, Rose, Luc]
  - [Jean, Carole, Rose, Luc]
- Dans cet exemple, nous ajoutons la fonction d'extension **shift()** à la classe **MutableList.** La fonction **shift()** supprime le premier élément de la collection.
- Vous devez constater aussi que cette fonction s'applique uniquement sur les chaines de caractère.

À noter

- **Le mot clé this** fait référence à l'objet récepteur (Celui sur lequel on applique la fonction d'extension **shift().)**

Nullable receiver

À noter

- les extensions peuvent être définies avec un type de récepteur nullable. De telles extensions peuvent être appelées sur une variable objet même si sa valeur est null, et peuvent vérifier this == null à l'intérieur du corps. C'est ce qui vous permet d'appeler toString() dans Kotlin sans vérifier null : la vérification se fait à l'intérieur de la fonction d'extension.

```
fun Any?.toString(): String {
  if (this == null) return "null"
  // après la vérification nulle, 'ceci' est converti automatiquement en un type non nul, donc le toString() ci-dessous
  // se résout en fonction membre de la classe Any
  return toString()
}
```

Extension properties et compagnon

- Comme pour les fonctions, Kotlin prend en charge les propriétés d'extension :

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

À noter

- Etant donné que les extensions n'insèrent pas réellement de membres dans les classes, il n'existe aucun moyen efficace pour une propriété d'extension d'avoir un champ de sauvegarde. **C'est pourquoi les initialiseurs ne sont pas autorisés pour les propriétés d'extension.** Leur comportement ne peut être défini qu'en fournissant explicitement des getters/setters.

Exemple :

```
val House.number = 1 // erreur : les initialiseurs ne sont pas autorisés pour les propriétés d'extension
```

- Si une classe à un objet compagnon défini, vous pouvez également définir des fonctions d'extension et des propriétés pour l'objet compagnon.
- Tout comme les membres normaux de l'objet compagnon, ils peuvent être appelés en utilisant uniquement le nom de classe comme qualificatif :

```
fun main() {
  val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet est l'implémentation par défaut
  val numbersBackwards = setOf(4, 3, 2, 1)
  println(numbers.first() == numbersBackwards.first())
  println(numbers.first() == numbersBackwards.last())
}
```

PARTIE 2

La visibilité des extensions

- Les extensions utilisent la même visibilité des autres entités que les fonctions régulières déclarées dans la même portée.

Par exemple :

- Une extension déclarée au plus haut niveau d'un fichier à accès aux autres déclarations private premier niveau dans le même fichier ;
- Si une extension est déclarée en dehors de son type de récepteur, une telle extension ne peut pas accéder aux membres private du récepteur .

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE n° 4

Maîtriser les extensions

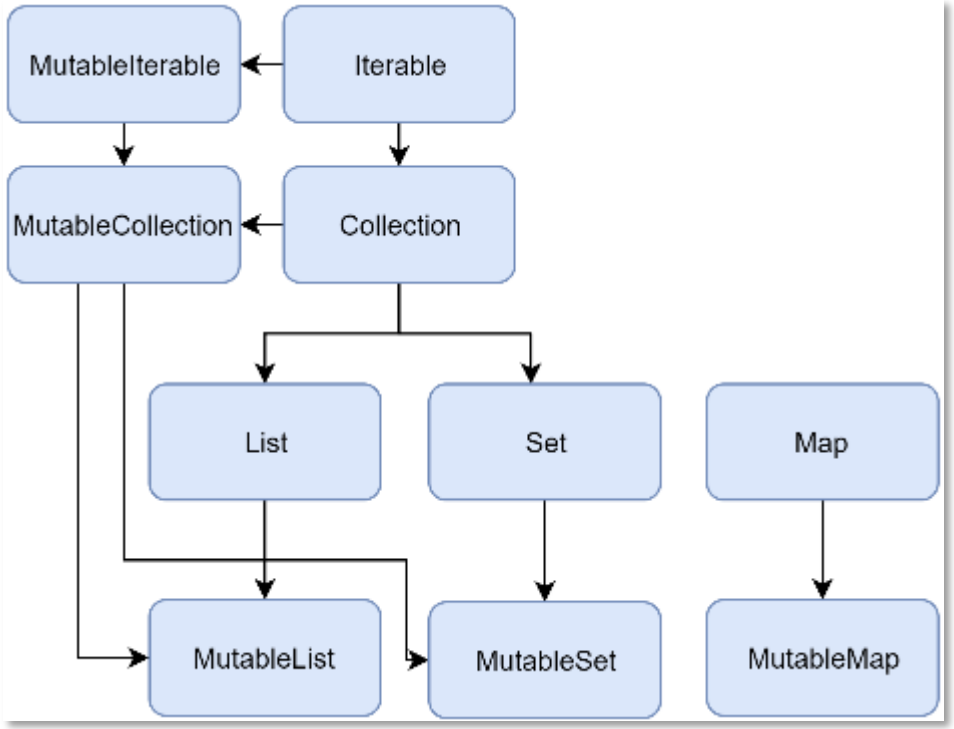
1. Extensions
2. **Collections et ranges**

Aperçu des collections Kotlin

- La bibliothèque standard de Kotlin fournit un ensemble complet d'outils pour la gestion des collections - des groupes d'un nombre variable d'éléments (éventuellement zéro) qui partagent une importance pour le problème en cours de résolution et sont exploités couramment.
- Les collections sont un concept commun à la plupart des langages de programmation. Si vous êtes familier avec les collections Java ou Python, par exemple, vous pouvez sauter cette introduction et passer aux sections détaillées.
- Une collection contient généralement un certain nombre d'objets (ce nombre peut également être zéro) du même type. Les objets d'une collection sont appelés éléments ou items . Par exemple, tous les étudiants d'un département forment une collection qui permet de calculer leur âge moyen. Les types de collecte suivants sont pertinents pour Kotlin :
    - **List** est une collection ordonnée avec accès aux éléments par des indices - des nombres entiers qui reflètent leur position. Les éléments peuvent apparaître plusieurs fois dans une liste. Un exemple de liste est une phrase : c'est un groupe de mots, leur ordre est important, et ils peuvent se répéter.
    - **Set** est une collection d'éléments uniques. Il reflète l'abstraction mathématique de l'ensemble : un groupe d'objets sans répétitions. Généralement, l'ordre des éléments d'ensemble n'a aucune signification. Par exemple, un alphabet est un ensemble de lettres.
    - **Map** (ou dictionnaire) est un ensemble de paires clé-valeur. Les clés sont uniques et chacune d'elles correspond exactement à une valeur. Les valeurs peuvent être des doublons. Les cartes sont utiles pour stocker des connexions logiques entre des objets, par exemple, l'ID d'un employé et son poste.

Collection types

- Vous trouverez ci-dessous un diagramme des interfaces de collections Kotlin :



https://kotlinlang.org

- Passons en revue les interfaces et leurs implémentations.

Collection types

- La bibliothèque standard Kotlin fournit des implémentations pour les types de collections de base:ensembles,listes et cartes.Une paire d'interfaces représente chaque type de collection :
  - Une interface en lecture seule qui fournit des opérations pour accéder aux éléments de collection.
  - Une interface mutable qui étend l'interface en lecture seule correspondante avec des opérations d'écriture : ajout, suppression et mise à jour de ses éléments.

```
fun main() {
//sampleStart
  val numbers = mutableListOf("one", "two", "three", "four")
  numbers.add("five")   // C'est acceptable
  //nombres = mutableListOf("six", "sept") // erreur de compilation
//sampleEnd
}
```

À noter

- La modification d'une collection mutable ne nécessite pas qu'elle soit une var: les opérations d'écriture modifient le même objet de collection mutable, donc la référence ne change pas. Cependant, si vous essayez de réaffecter une collection val , vous obtiendrez une erreur de compilation.

Collection

- Collection<T> est la racine de la hiérarchie des collections. Cette interface représente le comportement courant d'une collection en lecture seule : récupération de la taille, vérification de l'appartenance à un élément, etc. Collection hérite de l'interface Iterable<T> qui définit les opérations d'itération des éléments. Vous pouvez utiliser Collection comme paramètre d'une fonction qui s'applique à différents types de collection. Pour des cas plus spécifiques, utilisez les héritiers de la Collection : List et Set .

```
fun printAll(strings: Collection<String>) {
  for(s in strings) print("$s ")
  println()
}
fun main() {
  val stringList = listOf("one", "two", "one")
  printAll(stringList)

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

Collection

- **MutableCollection** est une Collection avec des opérations d'écriture, telles que add et remove .

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

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

List

- **List<T>** stocke les éléments dans un ordre spécifié et leur fournit un accès indexé. Les indices commencent à zéro - l'indice du premier élément - et vont à lastIndex qui est le (list.size - 1) .

```
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element \"two\" ${numbers.indexOf("two")}")
```

- Les éléments de liste (y compris les valeurs NULL) peuvent dupliquer : une liste peut contenir un nombre quelconque d'objets égaux ou d'occurrences d'un seul objet. Deux listes sont considérées comme égales si elles ont les mêmes tailles et des éléments structurellement égaux aux mêmes positions.

```
data class Person(var name: String, var age: Int)
fun main() {
  val bob = Person("Bob", 31)
  val people = listOf(Person("Adam", 20), bob, bob)
  val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
  println(people == people2)
  bob.age = 32
  println(people == people2)
}
```

PARTIE 2

List

- **MutableList<T>** est une List avec des opérations d'écriture spécifiques à une liste, par exemple, pour ajouter ou supprimer un élément à une position spécifique.

```
fun main() {
  val numbers = mutableListOf(1, 2, 3, 4)
  numbers.add(5)
  numbers.removeAt(1)
  numbers[0] = 0
  numbers.shuffle()
  println(numbers)
}
```

- Comme vous le voyez,les listes sont,à certains égards,très similaires aux tableaux.Toutefois,il existe une différence importante:la taille d'un tableau est définie lors de l'initialisation et n'est jamais modifiée;à l'inverse,une liste n'a pas de taille prédéfinie;la taille d'une liste peut être modifiée à la suite d'opérations d'écriture:ajout,mise à jour ou suppression d'éléments.
- Dans Kotlin, l'implémentation par défaut de List est ArrayList que vous pouvez considérer comme un tableau redimensionnable.

Set

- **Set<T>** stocke des éléments uniques ; leur ordre est généralement indéfini. null éléments null sont également uniques : un Set ne peut contenir qu'un seul null . Deux ensembles sont égaux s'ils ont la même taille, et pour chaque élément d'un ensemble, il y a un élément égal dans l'autre ensemble.

```
fun main() {
  val numbers = setOf(1, 2, 3, 4)
  println("Number of elements: ${numbers.size}")
  if (numbers.contains(1)) println("1 is in the set")

  val numbersBackwards = setOf(4, 3, 2, 1)
  println("The sets are equal: ${numbers == numbersBackwards}")
}
```

Set

- **MutableSet** est un Set avec des opérations d'écriture de MutableCollection . L'implémentation par défaut de Set – LinkedHashSet – préserve l'ordre d'insertion des éléments. Par conséquent, les fonctions qui reposent sur l'ordre, telles que first() ou last() , renvoient des résultats prévisibles sur de tels ensembles.

```
fun main() {
  val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet est l'implémentation par défaut
  val numbersBackwards = setOf(4, 3, 2, 1)

  println(numbers.first() == numbersBackwards.first())
  println(numbers.first() == numbersBackwards.last())
}
```

- Une implémentation alternative - HashSet - ne dit rien sur l'ordre des éléments, donc appeler de telles fonctions dessus renvoie des résultats imprévisibles.
- Cependant, HashSet nécessite moins de mémoire pour stocker le même nombre d'éléments.

Map

- **Map<K, V>** n'est pas un héritier de l'interface Collection ; Cependant, il s'agit également d'un type de collection Kotlin. Une Map stocke des paires clé-valeur (ou entrées) ; les clés sont uniques, mais différentes clés peuvent être associées à des valeurs égales. L'interface Map fournit des fonctions spécifiques, telles que l'accès à la valeur par clé, la recherche de clés et de valeurs, etc.

```
fun main() {
  val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

  println("All keys: ${numbersMap.keys}")
  println("All values: ${numbersMap.values}")
  if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
  if (1 in numbersMap.values) println("The value 1 is in the map")
  if (numbersMap.containsValue(1)) println("The value 1 is in the map") // identique au précédent
}
```

Map

- Deux Map contenant des paires égales sont égales quel que soit l'ordre des paires.

```
fun main() {
  val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
  val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)
  println("The maps are equal: ${numbersMap == anotherMap}")
}
```

- MutableMap est une Map avec des opérations d'écriture de carte, par exemple, vous pouvez ajouter une nouvelle paire clé-valeur ou mettre à jour la valeur associée à la clé donnée.

```
fun main() {
  val numbersMap = mutableMapOf("one" to 1, "two" to 2)
  numbersMap.put("three", 3)
  numbersMap["one"] = 11
  println(numbersMap)
}
```

- L'implémentation par défaut de Map - LinkedHashMap - préserve l'ordre d'insertion des éléments lors de l'itération de la carte. À son tour, une implémentation alternative - HashMap - ne dit rien sur l'ordre des éléments.

PARTIE 3

MAÎTRISER LES FONCTIONS ET LAMBDAS

Dans ce module, vous allez :

- Manipuler les expressions lambdas
- Faire des appels asynchrones
- Utiliser les expressions et opérateurs
- Comprendre le Lazy loading
- Utiliser les fonctions anonymes
- Utiliser les high-order Functions et inline

08 heures

CHAPITRE 1

DÉCLARER DES FONCTIONS

Ce que vous allez apprendre dans ce chapitre :

- Manipuler les expressions et opérateurs
- Faire des appels asynchrones
- Comprendre le lazy loading

02 heures

CHAPITRE 1

DÉCLARER DES FONCTIONS

1. **Expressions et opérateurs**
2. Appels asynchrones
3. Lazy loading

Expressions et opérateurs

- Les opérateurs sont des caractères spéciaux qui effectuent des opérations sur les opérandes (variables, valeurs).
- Dans Kotlin tout est objet, même les types de donnée de base tels que Int, Boolean et Char, dans le sens où vous pouvez appeler des fonctions et des propriétés membres sur n'importe quelle variable.
- Puisque ces types de donnée de base sont des objets, les opérations que vous effectuez sur ces types sont représentées en interne par des appels de fonction.
- Par exemple l'opération de soustraction a-b de deux nombres a et b est représentée en interne par l'appel de la fonction a.minus(b). Dans Kotlin, on appelle le système de transformation d'une expression un appel de fonction operator overloading.
- Il existe dans Kotlin, plusieurs opérateurs organisés dans plusieurs catégories.

Les opérateurs mathématiques ou numériques

- Les opérations d'addition, de soustraction, de multiplication etc., sont appelées opérations binaires.

Opération	Description	Expression	Fonction appelée
+	Addition	a+b	a.plus(b)
-	Soustraction	a-b	a.minus(b)
/	Division	a / b	a.div(b)
*	Multiplication	a * b	a.times(b)
%	Modulo	a % b	a.rem(b)

Les opérateurs logiques sur le type Booléen

- Kotlin fournit plusieurs opérateurs logiques pour effectuer des opérations sur le type booléens. Les opérateurs logiques sont :

Opérateur	Nom
||	OU logique
&&	ET logique
!	Non Logique

Les opérateurs de comparaison

- Voici ci-dessous la liste des opérateurs de comparaison :

Opération	Description	Expression	Fonction appelée
>	supérieur que	a > b	a.compareTo(b) > 0
<	Inférieur que	a < b	a.compareTo(b) < 0
>=	supérieur ou égale à	a >= b	a.compareTo(b) >= 0
<=	Inférieur ou égale à	a <= b	a.compareTo(b) <= 0
==	Est égale à	a == b	a?.equals(b) ?: (b === null)
!=	Pas égale à	a != b	!(a?.equals(b) ?: (b === null))

CHAPITRE 1

DÉCLARER DES FONCTIONS

1. Expressions et opérateurs
2. **Appels asynchrones**
3. Lazy loading

Techniques de programmation asynchrone

- Depuis des décennies, en tant que développeurs, nous sommes confrontés à un problème à résoudre : comment empêcher nos applications de se bloquer.
- Que nous développions des applications de bureau, mobiles ou même côté serveur, nous voulons éviter que l'utilisateur attende ou, pire encore, provoquer des blocages de l'application ou des crashs.
- Il existe de nombreuses approches pour résoudre ce problème, notamment :
    - Threading
    - Callbacks
    - Futures, promises, and others
    - Reactive Extensions
    - Coroutines

Threading

- Les threads sont l'approche la plus connue pour éviter le blocage des applications.

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // makes a request and consequently blocks the main thread
    return token
}
```

Threading

- Supposons dans le code ci-dessus que preparePost est un processus de longue durée et bloquerait par conséquent l'interface utilisateur. Ce que nous pouvons faire, c'est le lancer dans un thread séparé. Cela nous permettrait alors d'éviter le blocage de l'interface utilisateur. C'est une technique très courante, mais qui présente une série d'inconvénients :
    - Les threads ne sont pas infinis. Le nombre de threads pouvant être lancés est limité par le système d'exploitation sous-jacent. Dans les applications côté serveur, cela pourrait provoquer un goulot d'étranglement majeur.
    - Les threads ne sont pas toujours disponibles. Certaines plates-formes, telles que JavaScript, ne prennent même pas en charge les threads.
    - Les threads ne sont pas faciles. Déboguer les threads, éviter les conditions de concurrence sont des problèmes courants que nous rencontrons dans la programmation multi-thread.

Callbacks

- Avec les callbacks, l'idée est de passer une fonction en paramètre à une autre fonction, et de faire invoquer celle-ci une fois le processus terminé.

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}
fun preparePostAsync(callback: (Token) -> Unit) {
    // make request and return immediately
    // arrange callback to be invoked later
}
```

Callbacks

- Cela ressemble en principe à une solution beaucoup plus élégante, mais présente encore une fois plusieurs problèmes :
    - Difficulté des nested callbacks. Habituellement, une fonction utilisée comme callbacks finit souvent par avoir besoin de son propre callbacks. Cela conduit à une série de nested callbacks qui conduisent à un code incompréhensible. Le motif est souvent appelé l'arbre de Noël titré (les accolades représentent les branches de l'arbre).
    - La gestion des erreurs est compliquée. Le modèle d'imbrication rend la gestion et la propagation des erreurs un peu plus compliquées.

Futures, promises, et autres

- L'idée derrière les futures ou les promesses (il existe également d'autres termes auxquels on peut faire référence en fonction du langage/de la plate-forme), est que lorsque nous effectuons un appel, on nous promet qu'à un moment donné, il reviendra avec un objet appelé une promesse, qui peut ensuite être opéré.

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }
}
fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    return promise
}
```

Futures, promises, et autres

- Cette approche nécessite une série de changements dans notre façon de programmer, notamment :
    - Modèle de programmation différent. Semblable aux callbacks, le modèle de programmation s'éloigne d'une approche impérative descendante vers un modèle compositionnel avec des appels enchaînés. Les structures de programme traditionnelles telles que les boucles, la gestion des exceptions, etc. ne sont généralement plus valides dans ce modèle.
    - Différentes API. Il est généralement nécessaire d'apprendre une toute nouvelle API telle que thenCompose ou thenAccept, qui peut également varier d'une plate-forme à l'autre.
    - Type de retour spécifique. Le type de retour s'éloigne des données réelles dont nous avons besoin et renvoie à la place un nouveau type Promise qui doit être introspecté.
    - La gestion des erreurs peut être compliquée. La propagation et l'enchaînement des erreurs ne sont pas toujours simples.

Extensions réactives

- Les extensions réactives (Rx) ont été introduites dans C# par Erik Meijer. Bien qu'il ait été définitivement utilisé sur la plate-forme .NET, il n'a vraiment pas été adopté par le grand public jusqu'à ce que Netflix le porte sur Java, en le nommant RxJava. Depuis lors, de nombreux ports ont été fournis pour une variété de plates-formes, y compris JavaScript (RxJS).
- L'idée derrière Rx est d'évoluer vers ce qu'on appelle des flux observables dans lesquels nous considérons maintenant les données comme des flux (des quantités infinies de données) et ces flux peuvent être observés. Concrètement, Rx est simplement l'Observer Pattern avec une série d'extensions qui nous permettent d'opérer sur les données.
- En approche, c'est assez similaire à Futures, mais on peut penser qu'un Future renvoie un élément discret, alors que Rx renvoie un flux. Cependant, similaire au précédent, il introduit également une toute nouvelle façon de penser à notre modèle de programmation, célèbre pour sa formulation : **"tout est un flux, et c'est observable"**
- Cela implique une manière différente d'aborder les problèmes et un changement assez important par rapport à ce à quoi nous sommes habitués lors de l'écriture de code synchrone. L'un des avantages par rapport à Futures est qu'étant donné qu'il est porté sur de nombreuses plates-formes, nous pouvons généralement trouver une expérience d'API cohérente, peu importe ce que nous utilisons, que ce soit C #, Java, JavaScript ou tout autre langage où Rx est disponible.
- De plus, Rx introduit une approche un peu plus agréable de la gestion des erreurs.

Coroutines

- L'approche de Kotlin pour travailler avec du code asynchrone utilise des coroutines, qui est l'idée de calculs suspendables, c'est-à-dire l'idée qu'une fonction peut suspendre son exécution à un moment donné et reprendre plus tard.
- Cependant, l'un des avantages des coroutines est que, pour le développeur, l'écriture de code non bloquant est essentiellement la même que l'écriture de code bloquant. Le modèle de programmation en lui-même ne change pas vraiment.
- Voir la partie 4 **Maîtriser les aspects avancés de Kotlin , le chapitre 2 introduire les coroutines pour plus de détails.**

CHAPITRE 1

DÉCLARER DES FONCTIONS

1. Expressions et opérateurs
2. Appels asynchrones
3. **Lazy loading**

Lazy loading

- Kotlin est un langage de programmation robuste dans le sens où il vous évite l'une des erreurs des classes la plus commune qui est la NullPointerException.
- A cet effet, une variable déclarée dans Kotlin est non nulle par défaut. Et lorsque vous déclarez une nouvelle variable dans Kotlin, vous devez toujours l'initialiser.
- Vous pouvez initialiser la variable soit en définissant une valeur par défaut de son type, soit en définissant sa valeur à nul. Par contre pour initialiser la variable à une valeur nul, vous devez explicitement dire à Kotlin que la variable peut prendre une valeur nul.
- Qu'en est-il alors de si l'on souhaite créer un objet et ne pas initialiser certaines de ses variables membres lors de leurs déclarations comme on sait le faire en Java?
- Ou qu'en est-il de si vous souhaitez initialiser une variable plus tard via une injection de dépendance?
- A cet effet, Kotlin nous permet de retarder l'initialisation d'une variable de plusieurs manières.

Le mot clé lateinit

- Pour retarder l'initialisation d'une variable non nul, vous pouvez utiliser le mot clé lateinit lors de la déclaration de la variable comme suit.

```
fun main(){
 val personn =Personn("VIGAN","Noé ")
}
data class Personn(val nom: String,val prenom: String){
  lateinit var leg: Leg//Initialisation retardée
  init {
     leg=Leg(2)
  }
  fun affiche()="Le nom de pieds de $nom $prenom est ${leg.nbleg}"
}
data class  Leg(val nbleg: Int){}
```

- Une variable non nul déclarée avec le mot clé lateinit lors de sa déclaration peut être initialisée plus tard partout dans votre code.
- Mais vous devez obligatoirement initialiser la variable avant de l'utiliser pour éviter de générer l'exception : **UninitializedPropertyAccessException: lateinit property has not been initialized.**

Le mot clé lateinit

- Par exemple vous aller générer l'exception précédente si vous utilisez une variable sans l'initialiser comme suit.

```
data class Personn(val nom: String,val prenom: String){
  lateinit var leg: Leg
  init {
     print("Le nom de pieds de $nom $prenom est ${leg.nbleg}")//génrère l'exception
     leg=Leg(2)
  }
  fun affiche()="Le nom de pieds de $nom $prenom est ${leg.nbleg}"
}
data class  Leg(val nbleg: Int){
}
```

À noter

- Le mot clé lateinit peut être uniquement utilisé avec une variable var. Si vous déclarez une variable val avec lateinit comme suit, le compilateur vous signalera donc une erreur.

Le mot clé lateinit

```
data class Personn(val nom: String,val prenom: String){
  lateinit val leg: Leg //Le compilateur signale une erreur
  init {
     leg=Leg(2)
  }
  fun affiche()="Le nom de pieds de $nom $prenom est ${leg.nbleg}"
}
```

- De plus, vous ne pouvez pas utiliser lateinit avec les types primitives tels que Long,Int ,Double etc.

Voir l'exemple :

```
lateinit var n1: Int = 13  //Erreur
lateinit var nom: String="" //Erreur
```

- lateinit ne s'utilise pas dans un constructeur de classe. Si vous souhaitez utiliser lateinit dans la déclaration d'une variable dans une classe, vous devez déclarer la variable dans le corps de la classe comme suit.

Le mot clé lateinit

- Pour tester si une variable lateinit est déjà initialisée, vous pouvez utiliser la fonction **isInitialized** comme suit :

```
data class Personn(val nom: String,val prenom: String){
 lateinit var leg: Leg
 init {
   if(::leg.isInitialized){
       print("Le nom de pieds de $nom $prenom est ${leg.nbleg}")
       //S'exécute uniquement si la variable leg est déja initialisée
   }
    leg=Leg(2)
 }
 fun affiche()="Le nom de pieds de $nom $prenom est ${leg.nbleg}"
}
data class  Leg(val nbleg: Int){
}
```

Le mot clé by lazy

- Vous pouvez aussi utiliser le mot clé lazy pour retarder l'initialisation d'une variable. Contrairement à une variable déclarée avec le mot clé lateinit, une variable déclarée avec le mot clé lazy est initialisée lorsqu'elle veut être utilisée pour la première fois.

Voir l'exemple suivant :

```
val nom :String by lazy {
    "VIGAN"
}
```

- Vous devez noter que lazy s'utilise avec une variable déclarée val. Puisque qu'une variable déclarée avec le mot val est immuable, alors lorsque vous initialisez une variable déclarée val avec le mot clé lazy et que vous l'initialisez avant sa première utilisation, sa valeur ne peut plus changer.
- Si vous déclarez une variable var avec le mot clé lazy, le compilateur vous signalera qu'il y a une erreur comme dans l'exemple suivant.

```
var nom :String by lazy {//le compilateur signale une erreur
    "VIGAN"
}
```

CHAPITRE 2

MANIPULER LES EXPRESSIONS LAMBDAS ET FONCTIONS ANONYMES

Ce que vous allez apprendre dans ce chapitre :

- Manipuler les expressions lambdas
- Utiliser les fonctions anonymes

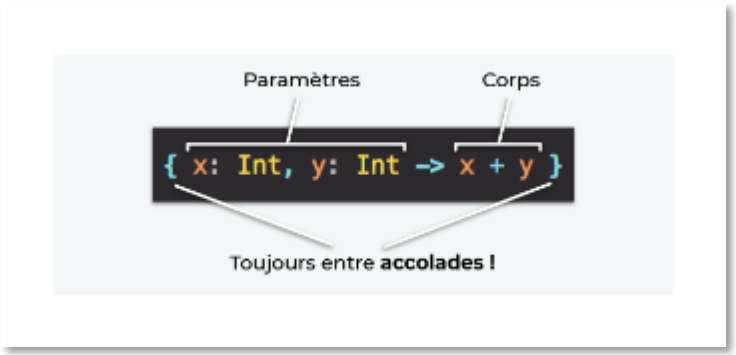
03 heures

CHAPITRE 2

MANIPULER LES EXPRESSIONS LAMBDAS ET FONCTIONS ANONYMES

Les expressions lambdas

- Dans cette partie, nous allons apprendre l'expression lambdas et la fonction anonyme dans Kotlin. Bien que syntaxiquement similaires, les lambdas Kotlin et Java ont des fonctionnalités très différentes.
- L'expression Lambdas et la fonction anonyme sont toutes deux des littéraux de fonction , ce qui signifie que ces fonctions ne sont pas déclarées mais transmises immédiatement en tant qu'expression.
- Comme nous le savons, la syntaxe des lambdas Kotlin est similaire à celle des Lambdas Java. Une fonction sans nom est appelée fonction anonyme.
- Pour l'expression lambda, nous pouvons dire qu'il s'agit d'une fonction anonyme.



Syntaxe d'une lambda en Kotlin : les paramètres et le corps sont placés entre accolades

https://openclassrooms.com

Les expressions lambdas

- Prenons un exemple d'une fonction prend deux int comme paramètres, les ajoute et renvoie un int, ce que nous pouvons réaliser de deux manières :
  - **sans les expressions lambdas :**

```
fun addTwoNumber (a: Int , b: Int) : Int {
  return a+b
}
val sumOfTwoNumber = addTwoNumber (2,3)
```

  - **avec les expressions lambdas :**

```
val sumOfTwoNumber : (Int, Int) -> Int = { a: Int, b: Int -> a + b }
```

- Les expressions lambda et les fonctions anonymes sont des « littéraux de fonction », c'est-à-dire des fonctions qui ne sont pas déclarées, mais transmises immédiatement en tant qu'expression. Ils nous fournissent un moyen compact d'écrire du code.

Inférence de type en lambdas

- L'inférence de type de Kotlin aide le compilateur à évaluer le type d'une expression lambda. Vous trouverez ci-dessous l'expression lambda à l'aide de laquelle nous pouvons calculer la somme de deux entiers.

```
val sum = {a: Int , b: Int -> a + b}
```

- Ici, le compilateur Kotlin l'auto-évalue comme une fonction qui prend deux paramètres de type Int et renvoie la valeur Int.

```
(Int,Int) -> Int
```

- Si nous voulions renvoyer la valeur String, nous pouvons le faire à l'aide de la fonction intégrée toString().

Exemple :

```
val sum1 = { a: Int, b: Int ->
  val num = a + b
  num.toString()     //convert Integer to String
}
fun main(args: Array<String>) {
  val result1 = sum1(2,3)
  println("The sum of two numbers is: $result1")
}
```

PARTIE 3

Inférence de type en lambdas

- **Le résultats est :**
  - The sum of two numbers is: 5
  - Dans le programme ci-dessus, le compilateur Kotlin l'auto-évalue comme une fonction qui prend deux valeurs entières et renvoie String.

```
(Int,Int) -> String
```

Déclaration de type en lambdas

- Nous devons déclarer explicitement le type de notre expression lambda.
- Si lambda ne renvoie aucune valeur, nous pouvons utiliser : **Modèle** d'unité : (Entrée) -> Sortie

Exemples Lambdas avec type de retour :

```
val lambda1: (Int) -> Int = (a -> a * a)
val lambda2: (String,String) -> String = { a , b -> a + b }
val lambda3: (Int)-> Unit = {print(Int)}
```

- **Lambdas peut être utilisé comme extension de classe :**

```
val lambda4: String.(Int) -> String = {this + it}
```

- Ici, il représente le nom implicite du paramètre unique et nous en discuterons plus tard.

Déclaration de type en lambdas

- Programme Kotlin lorsque les lambdas sont utilisés comme extension de classe

```
val lambda4 : String.(Int) -> String = { this + it }
fun main(args: Array<String>) {
  val result = "Geeks".lambda4(50)
  print(result)
}
```

- **Le résultats est :** Geeks50
- Dans l'exemple ci-dessus, nous utilisons l'expression lambda comme extension de classe.
- Nous avons passé les paramètres selon le format donné ci-dessus. ce mot-clé est utilisé pour la string et ce mot-clé est utilisé pour le paramètre Int passé dans le lambda. Ensuite, le code_body concatène à la fois les valeurs et retourne au résultat de la variable.

it : nom implicite d'un seul paramètre

- Dans la plupart des cas, lambdas contient le seul paramètre. Ici, **it** est utilisé pour représenter le paramètre unique que nous passons à l'expression lambda.

Exemple utilisant la forme abrégée de la fonction lambda :

```
val numbers = arrayOf(1,-2,3,-4,5)
fun main(args: Array<String>) {
  println(numbers.filter {  it > 0 })
}
```

- **Le résultats est :** [1, 3, 5]

Exemple utilisant la forme longue de la fonction lambda :

```
val numbers = arrayOf(1,-2,3,-4,5)
fun main(args: Array<String>) {
  println(numbers.filter {item -> item > 0 })
}
```

- **Le résultats est :** [1, 3, 5]

Renvoyer une valeur à partir de l'expression lambda

- Après l'exécution de lambda, la valeur finale est renvoyée par l'expression lambda. N'importe laquelle de ces valeurs entières, de string ou boolean peut être renvoyée par la fonction lambda.

Exemple pour renvoyer la valeur String par la fonction lambda :

```
val find =fun(num: Int): String{
  if(num % 2==0 && num < 0) { return "Number is even and negative" }
  else if (num %2 ==0 && num >0){  return "Number is even and positive" }
  else if(num %2 !=0 && num < 0){ return "Number is odd and negative" }
  else { return "Number is odd and positive" }
}
fun main(args: Array<String>) {
  val result = find(112)
  println(result)
}
```

- **Le résultats est :** Number is even and positive

Fonction anonyme

- Les fonctions anonymes sont, comme leur nom l'indique, des fonctions qui ne vont pas posséder de nom. En effet, lorsqu'on crée une fonction, nous ne sommes pas obligés de lui donner un nom à proprement parler.
- Généralement, on utilisera les fonctions anonymes lorsqu'on n'a pas besoin d'appeler notre fonction par son nom c'est-à-dire lorsque le code de notre fonction n'est appelé qu'à un endroit dans notre script et n'est pas réutilisé.
- En d'autres termes, les fonctions anonymes vont très souvent simplement nous permettre de gagner un peu de temps dans l'écriture de notre code et à le rendre plus clair en ne le polluant pas avec des noms inutiles.
- Une fonction anonyme est très similaire à une fonction régulière à l'exception du nom de la fonction qui est omis de la déclaration. Le corps de la fonction anonyme peut être une expression ou un bloc.

Fonction anonyme

Exemple 1 : Corps de la fonction en tant qu'expression

```
fun(a: Int, b: Int) : Int = a * b
```

Exemple 2 : Corps de la fonction sous forme de bloc

```
fun(a: Int, b: Int): Int {
  val mul = a * b
  return mul
}
```

- **Type de retour et paramètres :**
    1. Le type de retour et les paramètres sont également spécifiés de la même manière que pour la fonction régulière, mais nous pouvons mettre les paramètres s'ils peuvent être déduits du contexte.
    2. Le type de retour de la fonction peut être déduit automatiquement de la fonction s'il s'agit d'une expression et doit être spécifié explicitement pour la fonction anonyme s'il s'agit d'un bloc de corps.

Différence entre les expressions lambda et les fonctions anonymes

- La seule différence est le comportement des retours non locaux. Une instruction return sans étiquette revient toujours de la fonction déclarée avec le mot-clé fun. Cela signifie qu'un retour à l'intérieur d'une expression lambda reviendra de la fonction englobante, alors qu'un retour à l'intérieur d'une fonction anonyme reviendra de la fonction anonyme elle-même.

Exemple pour appeler la fonction anonyme :

```
// anonymous function with body as an expression
val anonymous1 = fun(x: Int, y: Int): Int = x + y
// anonymous function with body as a block
val anonymous2 = fun(a: Int, b: Int): Int {
  val mul = a * b
  return mul}
fun main(args: Array<String>) {
  //invoking functions
  val sum = anonymous1(3,5)
  val mul = anonymous2(3,5)
  println("The sum of two numbers is: $sum")
  println("The multiply of two numbers is: $mul") }
```

- **Le résultats est :**
  - The sum of two numbers is: 8
  - The multiply of two numbers is: 15

CHAPITRE 2

MANIPULER LES EXPRESSIONS LAMBDAS ET FONCTIONS ANONYMES

1. Expressions lambdas
2. **Gestion des exceptions**

Une exception est un événement indésirable ou inattendu, qui se produit pendant l'exécution d'un programme, c'est-à-dire au moment de l'exécution, qui perturbe le flux normal des instructions du programme. La gestion des exceptions est une technique qui nous permet de gérer les erreurs et d'éviter les plantages d'exécution qui peuvent arrêter notre programme.

Il existe deux types d'exceptions :

- Exception vérifiée – Exceptions généralement définies sur les méthodes et vérifiées au moment de la compilation, par exemple IOException, FileNotFoundException, etc.
- Exception non vérifiée – Exceptions généralement dues à des erreurs logiques et vérifiées au moment de l'exécution, par exemple NullPointerException, ArrayIndexOutOfBoundException, etc.

Dans Kotlin, nous n'avons que des exceptions non contrôlées et ne pouvant être détectées qu'au moment de l'exécution. Toutes les classes d'exception sont des descendantes de la classe **Throwable** .

Nous utilisons généralement l'expression **throw** pour lancer un objet d'exception :

```
throw Exception("Throw me")
```

Certaines des exceptions courantes sont :

- **NullPointerException :** elle est levée lorsque nous essayons d'invoquer une propriété ou une méthode sur un objet null.
- **Exception arithmétique :** elle est levée lorsque des opérations arithmétiques non valides sont effectuées sur des nombres. Par exemple « diviser par zéro ».
- **SecurityException :** elle est levée pour indiquer une violation de la sécurité.
- **ArrayIndexOutOfBoundException :** elle est levée lorsque nous essayons d'accéder à la valeur d'index invalide d'un array.

Programme Kotlin de lancement d'exception arithmétique :

```
fun main(args : Array<String>){
   var num = 10 / 0      // throws exception
 println(num)
}
```

Le résultats est : Exception in thread "main" java.lang.ArithmeticException: / by zero

Dans le programme ci-dessus, nous initialisons la numvariable avec la valeur 10/0 , mais nous savons qu'en arithmétique, la division par zéro n'est pas autorisée.

Pendant que nous essayons d'exécuter le programme, il lève une exception.

Bloc try-catch Kotlin

Dans Kotlin, nous utilisons le bloc try-catch pour la gestion des exceptions dans le programme. Le bloc try contient le code responsable de la levée d'une exception et le bloc catch est utilisé pour gérer l'exception. Ce bloc doit être écrit dans la méthode principale ou dans d'autres méthodes. Le bloc Try doit être suivi soit du bloc catch, soit du bloc finally, soit des deux.

```
try {
// code that can throw exception
 } catch(e: ExceptionName) {
// catch the exception and handle it
 }
```

Programme Kotlin de gestion des exceptions arithmétiques à l'aide du bloc try-catch :

```
import kotlin.ArithmeticException
  fun main(args : Array<String>){
 try{
   var num = 10 / 0
    }    catch(e: ArithmeticException){
   // caught and handles it
     println("Divide by zero not allowed")
  } }
```

Bloc try-catch Kotlin

Explication

Dans le programme ci-dessus, nous avons utilisé le bloc try-catch. La numvariable qui peut lever une exception est entre les accolades du bloc try car la division par zéro n'est pas définie en arithmétique. L'exception est interceptée par le bloc catch et exécute l' println() instruction.

CHAPITRE 3

UTILISER LES FONCTIONS D'ORDRE SUPÉRIEUR ET FONCTIONS INLINE

Ce que vous allez apprendre dans ce chapitre :

- Utiliser les fonctions d’ordre supérieur
- Utiliser les fonctions inline
- Réflexion

03 heures

CHAPITRE 3

UTILISER LES FONCTIONS D'ORDRES SUPÉRIEURS ET FONCTIONS INLINE

1. **Fonctions d’ordres supérieurs**
2. Fonctions inline
3. Réflexion

Les fonctions d’ordres supérieurs

- Les fonctions Kotlin sont de première classe , ce qui signifie qu'elles peuvent être stockées dans des variables et des structures de données, transmises comme arguments à et renvoyées par d'autres fonctions d'ordre supérieur .
- Vous pouvez utiliser les fonctions de toutes les manières possibles pour d'autres valeurs non fonctionnelles.
- Pour faciliter cela, Kotlin, en tant que langage de programmation à typage statique, utilise une famille de types de fonctions pour représenter les fonctions et fournit un ensemble de constructions de langage spécialisées, telles que les expressions lambda .
- Une fonction d'ordre supérieur est une fonction qui prend des fonctions comme paramètres, ou qui renvoie une fonction.

Les fonctions d'ordres supérieurs

- Un bon exemple est l'idiome de programmation fonctionnel fold pour les collections, qui prend une valeur d'accumulation initiale et une fonction de combinaison et construit sa valeur de retour en combinant consécutivement la valeur d'accumulateur actuelle avec chaque élément de collection, remplaçant l'accumulateur :

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

- Dans le code ci-dessus, le paramètre combine a un type de fonction (R, T) -> R , il accepte donc une fonction qui prend deux arguments de types R et T et renvoie une valeur de type R . Il est invoqué à l'intérieur de la boucle for et la valeur de retour est ensuite affectée à accumulator .

Les fonctions d’ordres supérieurs

- Pour appeler fold , nous devons lui transmettre une instance du type de fonction en tant qu'argument, et les expressions lambda (décrites plus en détail ci-dessous) sont largement utilisées à cette fin sur les sites d'appel de fonction d'ordre supérieur :

```
fun main() {
  val items = listOf(1, 2, 3, 4, 5)
  // Les lambdas sont des blocs de code entourés d'accolades.
  items.fold(0, {
     // Lorsqu'un lambda a des paramètres, ils passent en premier, suivis de '->'
        acc: Int, i: Int ->
     print("acc = $acc, i = $i, ")
     val result = acc + i
     println("result = $result")
     // La dernière expression dans un lambda est considérée comme la valeur de retour :
     result})
  // Les types de paramètres dans un lambda sont facultatifs s'ils peuvent être déduits :
  val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })
  // Les références de fonction peuvent également être utilisées pour les appels de fonction d'ordre supérieur :
  val product = items.fold(1, Int::times)
  println("joinedToString = $joinedToString")
  println("product = $product")
}
```

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 3

UTILISER LES FONCTIONS D'ORDRES SUPÉRIEURS ET FONCTIONS INLINE

Les fonctions inline

- L'utilisation de fonctions d'ordre supérieur impose certaines pénalités à l'exécution : chaque fonction est un objet, et elle capture une fermeture, c'est-à-dire les variables auxquelles on accède dans le corps de la fonction. Les allocations de mémoire (à la fois pour les objets fonction et les classes) et les appels virtuels introduisent une surcharge d'exécution.
- Mais il semble que dans de nombreux cas, ce type de surcharge puisse être éliminé en insérant les expressions lambda. Les fonctions présentées ci-dessous sont de bons exemples de cette situation. C'est-à-dire que la fonction lock() pourrait être facilement intégrée aux sites d'appel. Considérons le cas suivant :

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

- Au lieu de créer un objet fonction pour le paramètre et de générer un appel,le compilateur pourrait émettre le code suivant :

```
l.lock()
try {
  foo()
}
finally {
  l.unlock()
}
```

Les fonctions inline

- Pour que le compilateur fasse cela, nous devons marquer la fonction lock() avec le modificateur inline :

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

- Le modificateur inline affecte à la fois la fonction elle-même et les lambdas qui lui sont transmises : tous ceux-ci seront intégrés dans le site d'appel.
- L'inlining peut faire grossir le code généré, cependant,si nous le faisons de manière raisonnable (c'est à dire en évitant d'inliner de grandes fonctions),les performances s'en trouveront améliorées,en particulier au niveau des sites d'appel "mégamorphiques" à l'intérieur des boucles.

Les fonctions noinline

- Si vous souhaitez que seuls certains des lambdas transmis à une fonction en ligne soient intégrés, vous pouvez marquer certains de vos paramètres de fonction avec le modificateur noinline :

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { ... }
```

- Les lambdas inlinables ne peuvent être appelés qu'à l'intérieur des fonctions inline ou passés en tant qu'arguments inlinables, mais les noinline peuvent être manipulés comme bon nous semble : stockés dans des champs, transmis, etc.
- Notez que si une fonction en ligne n'a pas de paramètres de fonction inlinables ni de paramètres de type réifié , le compilateur émettra un avertissement, car il est très peu probable que l'incorporation de telles fonctions soit bénéfique (vous pouvez supprimer l'avertissement si vous êtes sûr que l'incorporation est nécessaire en utilisant le annotation @Suppress("NOTHING_TO_INLINE")).

Non-local returns

- Dans Kotlin, nous ne pouvons utiliser qu'un return normal et non qualifié pour quitter une fonction nommée ou une fonction anonyme. Cela signifie que pour quitter un lambda, nous devons utiliser un label , et un return nu est interdit à l'intérieur d'un lambda, car un lambda ne peut pas renvoyer la fonction englobante :

```
fun ordinaryFunction(block: () -> Unit) {
  println("hi!")
}
fun foo() {
  ordinaryFunction {
    return // ERREUR : impossible de faire revenir `foo` ici
  }
}
fun main() {
  foo()
}
```

- Mais si la fonction à laquelle le lambda est passé est inlined, le retour peut l'être aussi, donc c'est autorisé :
    - inline fun inlined(block: () -> Unit) { println("salut!") }

Non-local returns

```
fun foo() {
  inlined {
    return // OK : le lambda est inline
  }
}
fun main() {
  foo()
}
```

- De tels retours (situés dans un lambda, mais sortant de la fonction englobante) sont appelés retours non locaux . Nous sommes habitués à ce genre de construction dans les boucles, que les fonctions en ligne contiennent souvent :

```
fun hasZeros(ints: List<Int>): Boolean {
  ints.forEach {
    if (it == 0) return true // renvoie de hasZeros
  }
  return false
}
```

Non-local returns

- Notez que certaines fonctions en ligne peuvent appeler les lambdas qui leur sont transmises en tant que paramètres non pas directement à partir du corps de la fonction, mais à partir d'un autre contexte d'exécution, tel qu'un objet local ou une fonction imbriquée. Dans de tels cas, le flux de contrôle non local n'est pas non plus autorisé dans les lambdas.
- Pour indiquer cela, le paramètre lambda doit être marqué avec le modificateur crossinline :

```
inline fun f(crossinline body: () -> Unit) {
  val f = object: Runnable {
     override fun run() = body()
  }
  // ...
}
```

Paramètres de type réifié

- Parfois, nous avons besoin d'accéder à un type qui nous est passé en paramètre :

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
  var p = parent
  while (p != null && !clazz.isInstance(p)) {
     p = p.parent
  }
  @Suppress("UNCHECKED_CAST")
  return p as T?
}
```

- Ici, nous remontons un arbre et utilisons la réflexion pour vérifier si un nœud a un certain type. Tout va bien, mais le site d'appel n'est pas très joli :

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

- Ce que nous voulons en fait,c'est simplement passer un type à cette fonction,c'est-à-dire l'appeler comme ceci :

```
treeNode.findParentOfType<MyTreeNode>()
```

Paramètres de type réifié

- Pour activer cela, les fonctions en ligne prennent en charge les paramètres de type réifié , nous pouvons donc écrire quelque chose comme ceci :

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
  var p = parent
  while (p != null && p !is T) {
     p = p.parent
  }
  return p as T?
}
```

- Nous avons qualifié le paramètre type avec le modificateur reified. Il est maintenant accessible à l'intérieur de la fonction, presque comme s'il s'agissait d'une classe normale. Puisque la fonction est inline, aucune réflexion n'est nécessaire. Les opérateurs normaux comme !is et as fonctionnent maintenant.
- En outre, nous pouvons l'appeler comme mentionné ci-dessus : myTree.findParentOfType<MyTreeNodeType>() .

Restrictions pour les fonctions inline de l'API publique

- Lorsqu'une fonction en ligne est public ou protected et ne fait pas partie d'une déclaration private ou internal, elle est considérée comme l'API publique d' un module .
- Il peut être appelé dans d'autres modules et est également intégré à ces sites d'appel.
- Cela impose certains risques d'incompatibilité binaire causés par des changements dans le module qui déclare une fonction inline dans le cas où le module appelant n'est pas recompilé après le changement.
- Pour éliminer le risque qu'une telle incompatibilité soit introduite par une modification de l' API non publique d'un module, les fonctions en ligne de l'API publique ne sont pas autorisées à utiliser des déclarations API non publiques, c'est-à-dire private déclarations privées et internal et leurs parties, dans leurs corps .
- Une déclaration internal peut être annotée avec @PublishedApi, ce qui permet son utilisation dans des fonctions en ligne d'API publiques. Lorsqu'une fonction internal ligne est marquée comme @PublishedApi, son corps est également vérifié, comme s'il était public.

CHAPITRE 3

UTILISER LES FONCTIONS D'ORDRES SUPÉRIEURS ET FONCTIONS INLINE

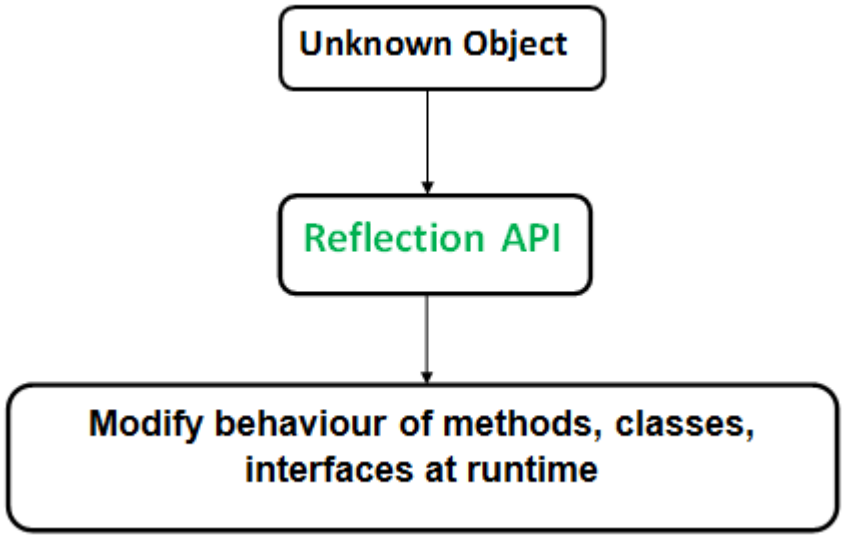
1. Fonctions d’ordres supérieurs
2. Fonctions inline
3. **Réflexion**

Réflexion

La réflexion est un ensemble de fonctionnalités de langage et de bibliothèque qui fournit la fonctionnalité d'introspection d'un programme donné au moment de l'exécution. La réflexion Kotlin est utilisée pour utiliser la classe et ses membres comme les propriétés, les fonctions, les constructeurs, etc… au moment de l'exécution.

Outre l'API de réflexion Java, Kotlin fournit également son propre ensemble d'API de réflexion, dans un style simple et fonctionnel. Les constructions Java Reflection standard sont également disponibles dans Kotlin et fonctionnent parfaitement avec son code.

Les reflets Kotlin sont disponibles en : kotlin.reflect package



https://kotlinlang.org

Caractéristiques de la réflexion Kotlin

1) Il donne accès aux propriétés et aux types nullables.
2) la réflexion Kotlin a quelques fonctionnalités supplémentaires par rapport à la réflexion Java.
3) la réflexion Kotlin aide à accéder au code JVM, écrit par un langage.

Références de classe :

Pour obtenir une référence de classe à l'exécution connue statiquement, utiliser l'opérateur de référence de classe. De plus, la référence à une classe peut également être obtenue à partir des instances de la classe. Ces références sont appelées références de classe délimitées . À l'aide d'instances, vous obtenez la référence au type exact auquel l'objet est conforme, en cas d'héritage.

Exemple pour démontrer les références de classe :

```
// A sample empty class
 class ReflectionDemo { }
 fun main() {     // Reference obtained using class name
    val abc = ReflectionDemo::class
  println("This is a class reference $abc")
   // Reference obtained using object
  val obj = ReflectionDemo()
 println("This is a bounded class reference ${obj::class}")
}
```

PARTIE 4

MAITRISER LES ASPECTS AVANCÉS DE KOTLIN

Dans ce module, vous allez :

- Utiliser les types checks et Casts
- Introduire les coroutines

08 heures

CHAPITRE 1

UTILISER LES TYPES CHECKS ET CASTS

Ce que vous allez apprendre dans ce chapitre :

- Utiliser les types checks et Casts
- Manipuler les unchecked casts

02 heures

CHAPITRE 1

UTILISER LES TYPES CHECKS ET CASTS

1. **Type safe builders**
2. Type Aliases

01 - Utiliser les types checks et Casts

Type safe builders

En utilisant des fonctions bien nommées en tant que constructeurs en combinaison avec des littéraux de fonction avec un récepteur, il est possible de créer des constructeurs de type statiquement typé dans Kotlin.

Les constructeurs de type sécurisé permettent de créer des langages spécifiques au domaine (DSL)basés sur Kotlin et adaptés à la construction de structures de données hiérarchiques complexes de manière semi-déclarative. Voici quelques exemples de cas d'utilisation des constructeurs :

- Générer un balisage avec du code Kotlin, tel que HTML ou XML ;
- Disposition par programmation des composants de l'interface utilisateur : Anko
- Configuration des routes pour un serveur Web : Ktor .

PARTIE 4

is et !is Opérateurs

- Nous pouvons vérifier si un objet est conforme à un type donné à l'exécution en utilisant l' opérateur is ou sa forme niée !is :

```
if (obj is String) {
  print(obj.length)
}
if (obj !is String) { // same as !(obj is String)
  print("Not a String")
}
else {
  print(obj.length)
}
```

Smart Casts

- Dans de nombreux cas, il n'est pas nécessaire d'utiliser des opérateurs de transtypage explicites dans Kotlin, car le compilateur suit les is - checks et les transtypages explicites pour les valeurs immuables et insère automatiquement les transtypages (sûrs) en cas de besoin :

```
fun demo(x: Any) {
  if (x is String) {
     print(x.length) // x est automatiquement converti en String
  }}
```

- Le compilateur est suffisamment intelligent pour savoir qu'un cast est sûr si une vérification négative conduit à un retour :

```
if (x !is String) return
print(x.length) // x is automatically cast to String
```

- ou à droite de && et || :

```
// x is automatically cast to string on the right-hand side of `||`
  if (x !is String || x.length == 0) return
// x is automatically cast to string on the right-hand side of `&&`
  if (x is String && x.length > 0) {
     print(x.length) // x is automatically cast to String
  }
```

Opérateur de casting « dangereux »

- Généralement, l'opérateur de transtypage lève une exception si le transtypage n'est pas possible. Ainsi, nous l'appelons **dangereux** . Le transtypage non sécurisé dans Kotlin est effectué par l'opérateur infixe comme (voir la priorité des opérateurs):

```
val x: String = y as String
```

- Notez que null ne peut pas être converti en String car ce type n'est pas nullable , c'est-à-dire que si y est null, le code ci-dessus lève une exception. Pour rendre ce code correct pour les valeurs nulles, utilisez le type nullable sur le côté droit de la distribution :

```
val x: String? = y as String?
```

- Veuillez noter que l'opérateur de distribution "unsafe" **n'est pas équivalent** à la unsafeCast<T>() disponible dans Kotlin/JS. unsafeCast n'effectuera aucune vérification de type, alors que l' opérateur de cast lève une ClassCastException lorsque le cast échoue.
- Pour éviter qu'une exception ne soit levée, on peut utiliser un opérateur de transtypage sécurisé comme ? qui renvoie null en cas d'échec :

```
val x: String? = y as? String
```

- Notez que malgré le fait que le côté droit de as ? est une String type non nul, le résultat du transtypage est nullable.

CHAPITRE 1

UTILISER LES TYPES CHECKS ET CASTS

1. Type safe builders
2. **Type Aliases**

01 - Utiliser les types checks et Casts

Type Aliases

Avec les alias de type, nous pouvons donner un alias à un autre type. C'est idéal pour donner un nom aux types de fonctions comme (String) -> Boolean ou type générique comme Pair<Person, Person> .

Les alias de type prennent en charge les génériques. Un alias peut remplacer un type par des génériques et un alias peut être un générique.

Les alias de type sont une fonctionnalité du compilateur. Rien n'est ajouté dans le code généré pour la JVM. Tous les alias seront remplacés par le type réel.

Type de fonction

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Type générique

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

PARTIE 4

Manipuler les unchecked casts

- Comme nous l'avons dit dans les slides précédents ,l'effacement des types rend impossible la vérification des arguments de type réels d'une instance de type générique au moment de l'exécution,et les types génériques dans le code peuvent être connectés les uns aux autres de manière insuffisante pour que le compilateur puisse garantir la sécurité des types.
- Malgré cela,nous avons parfois une logique de programme de haut niveau qui implique plutôt la sécurité des types.

Par exemple :

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
  TODO("Read a mapping of strings to arbitrary elements.")
}
// Nous avons enregistré une carte avec `Int`s dans ce fichier
val intsFile = File("ints.dictionary")
// Avertissement : transtypage non coché : `Map<String, *>` to `Map<String, Int>`
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

- Le compilateur génère un avertissement pour le cast dans la dernière ligne. La distribution ne peut pas être entièrement vérifiée au moment de l'exécution et ne fournit aucune garantie que les valeurs de la carte sont Int .
- Pour éviter les transtypages non vérifiés, vous pouvez reconcevoir la structure du programme : dans l'exemple ci-dessus, il pourrait y avoir des interfaces DictionaryReader<T> et DictionaryWriter<T> avec des implémentations de type sécurisé pour différents types. Vous pouvez introduire des abstractions raisonnables pour déplacer les transtypages non contrôlés du code appelant vers les détails de l'implémentation. L'utilisation appropriée de la variance générique peut également aider.

CHAPITRE 2

INTRODUIRE LES COROUTINES

Ce que vous allez apprendre dans ce chapitre :

- Découvrir les concepts de base des coroutines
- Création des coroutines

06 heures

CHAPITRE 2

INTRODUIRE LES COROUTINES

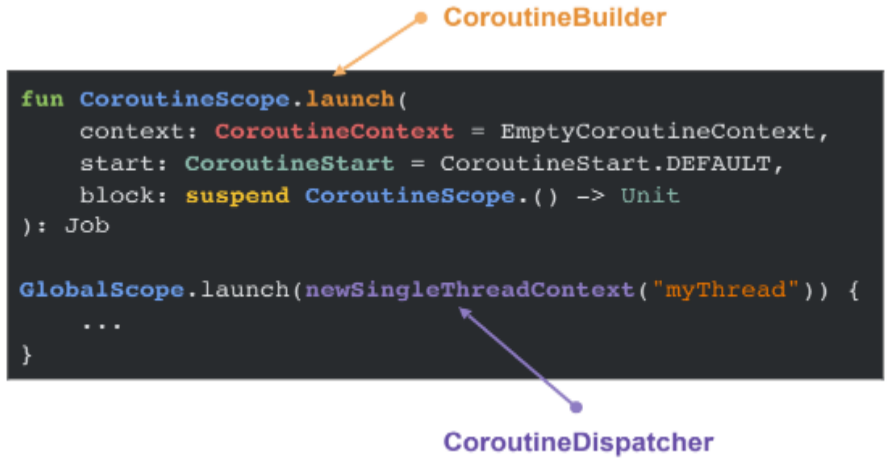
1. **Coroutines Scope**
2. Fonction launch
3. Fonction await
4. Jobs

CoroutineScope

- Un coroutine builder est une extension de CoroutineScope et hérite ainsi du coroutineContext pour pouvoir automatiquement propager à la fois les éléments du contexte et l'annulation.
- La meilleure manière de récupérer l'instance d'un scope, c'est avec les factories CoroutineScope et MainScope. On peut rajouter des éléments de contexte dans un scope grâce à l'opérateur plus.
- Chaque CoroutineBuilder (comme launch et async) et chaque scoping function (comme coroutineScope, withContext, etc) fournissent leurs propres scopes avec leur propre instance de la classe Job dans leur propre bloc de code. Par convention, ils vont tous attendre que les coroutines à l'intérieur de leur bloc ce soient terminés avant de finir leur propre exécution.
- Autrement dit, lorsque l'on crée une coroutine, cette dernière est définie dans un scope donné. Si vous lancez à l'intérieur de cette même coroutine un nouveau scope grâce par exemple à la méthode coroutineScope alors, la coroutine parente va devoir attendre cette dernière avant de pouvoir continuer à s'exécuter. Cela peut-être intéressant dans certains cas, on y reviendra dans la suite de cet article.

CoroutineContext & dispatchers

- Une coroutine tourne dans un contexte défini, représenté par le type CoroutineContext. Chaque contexte, inclut un objet CoroutineDispatcher, qui détermine le thred ou les threads utilisaient lors de l'exécution de la coroutine.
  - **Dispatchers.Default –** utilise un pool commun de threads partagés. Utile pour les tâches qui consomment des ressources CPU.
  - **Dispatchers.IO –** utilise un pool partagé de threads créé à la demande. Utile pour les opérations bloquantes d'I/O.
  - **newSingleThreadContext –** on crée et dispatch le code dans un thread prévu à cet effet.
  - **newFixedThreadPoolContext –** on fait la même chose qu'au-dessus mais sur un ThreadPool avec une taille définie.
  - On peut également préciser son propre **Executor**, il suffit de le convertir grâce à la méthode asCoroutineDispatcher.



https://kotlinlang.org/

PARTIE 4

CoroutineDispatcher en action

- Sur Android :
  - On ne peut pas faire d'appels réseaux sur le main thread, sinon une exception est declenchée.
  - De la même manière, nous ne pouvons pas modifier l'UI en dehors du thread principal.
- C'est pourquoi JetBrains, nous fournit la librairie coroutines-android, qui introduit un nouveau type de Dispatchers.main, qui va permettre de d'exécuter la suite du code d'une coroutine sur le thread principal. On peut le voir comme un runOnUIThread().
- Grâce à la classe CoroutineDispatcher, on va pouvoir avoir un outil efficace afin de contrôler facilement les appels réseaux de notre application. Exemple :

```
GlobalScope.launch(Dispatchers.Default) {
  val bitmap = DownloaderHelper.getBitmapFromURL(myURL)
  bitmap?.let {
     launch(Dispatchers.Main) {
        imageView.setImageBitmap(it)
     }
  }
}
```

- On lance le téléchargement d'une image depuis un thread en arrière-plan et dès que le téléchargement est terminé, on « dispatch » le résultat dans le main thread afin de pouvoir changer l'UI et d'afficher l'image.

- La librairie coroutines a été développée par JetBrains. Selon la documentation, les coroutines sont :
    - Des threads allégés.
    - Une pure abstraction de langage, pour l'utilisateur.
    - Une manière d'exécuter du code non bloquant et asynchrone.
    - Une coroutine ne dépend pas d'un thread en particulier.
    - Elle peut être suspendue dans un thread et reprise dans un autre.
    - Chaque coroutine peut communiquer avec elle même.
- Les coroutines sont simplement comme des « wrappers » de threads qui permettent de gérer de manière assez fine l'endroit et le moment où l'on va exécuter notre code.
- Les coroutines sont fournies par la librairie kotlinx.coroutines.

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

INTRODUIRE LES COROUTINES

1. Coroutines Scope
2. **Fonction launch**
3. Fonction await
4. Jobs

02 - Introduire les coroutines
Fonction launch

L'une des premières façons de démarrer une coroutine est d'utiliser la fonction launch. On pourra utiliser la fonction delay() pour simuler un traitement lourd ou de longue durée dans une coroutine. Cette fonction prend en paramètre un nombre en millisecondes. Il est important de préciser que le delay ne bloque que la coroutine, et non le Thread principal.

Une coroutine builder est une fonction qui va prendre en argument une suspending lambda, créer une coroutine et dans certains cas donner accès à un résultat.

Les coroutine builder launch() et future() par exemple sont définies dans une bibliothèque. La bibliothèque standard fournit des coroutine builder primitifs utilisées pour définir tous les autres coroutine builder.

Notre première coroutine

```
fun main() {
  GlobalScope.launch {
    delay(1000L)
    println("World!")
  }
  print("Hello,")
  Thread.sleep(2000L)
}
```

- Voici, un hello world classique obtenu avec une coroutine, regardons par ligne ce qui est fait :
  - Une nouvelle coroutine est lancée en arrière-plan grâce au mot-clé launch.
  - On fait une pause non bloquante pour 1 seconde (le temps par défaut est en ms) grâce à la méthode delay.
  - Après 1 seconde d'attente, on affiche le mot « World » .
  - Pendant ce temps, le main thread continue de tourner.
  - Il affiche le mot « Hello »
  - Enfin, on fait attendre le thread principal pendant 2 secondes pour maintenir la JVM en vie et ainsi laisser le temps à la coroutine de se terminer.

Les suspending functions

- Un concept assez important introduit avec les coroutines, ce sont les suspending functions. Sur l'exemple du dessus, notre coroutine contenait peu de code mais bien sur lorsque ce dernier devient assez conséquent, il est préférable de le scinder dans une fonction à part.
- Cette fonction doit alors être marquée avec le modifier suspend. Cela lui permet non seulement d'être utilisée dans une coroutine mais aussi de profiter de ces fonctionnalités, comme la méthode delay qu'on a vu au-dessus.
- Chose importante à savoir, une suspending function ne peut être appelée que depuis une autre suspending function ou alors à l'intérieur d'une coroutine !

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

INTRODUIRE LES COROUTINES

1. Coroutines Scope
2. Fonction launch
3. **Fonction await**
4. Jobs

Attend l'achèvement de cette valeur sans bloquer un thread et reprend lorsque le calcul différé est terminé, en renvoyant la valeur résultante ou en levant l'exception correspondante si le différé a été annulé.

Cette fonction de suspension est annulable. Si le Job de la coroutine en cours est annulé ou terminé alors que cette fonction de suspension est en attente, cette fonction reprend immédiatement avec CancellationException. Il existe une garantie d'annulation rapide. Si le job a été annulé alors que cette fonction était suspendue, il ne reprendra pas correctement.

Cette fonction peut être utilisée dans l'invocation de select avec la clause onAwait. Utilisez isCompleted pour vérifier l'achèvement de cette valeur différée sans attendre.

Coroutines builder

- Les builders sont utilisés pour préciser comment une coroutine doit se lancer :
    - Avec la commande launch
        - Cette dernière va alors retourner un objet Job.
        - Mais ne renverra pas de résultat.
    - Grâce à la méthode async
        - Qui va renvoyer un objet Deferred<T>, l'équivalent par exemple de PromiseKit pour Swift.
        - On peut alors utiliser la fonction await(), pour mettre en pause le thread courant et récupérer le résultat quand on en a besoin.
    - Avec runBlocking
        - Dont l'objectif est de lancer une coroutine et bloque le thread courant jusqu'à la fin de son exécution.

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

INTRODUIRE LES COROUTINES

1. Coroutines Scope
2. Fonction launch
3. Fonction await
4. **Jobs**

02 - Introduire les coroutines

Jobs

Un job est une chose annulable avec un cycle de vie qui aboutit à son achèvement. Les coroutines sont créées avec le constructeur de coroutines de lancement. Il exécute un bloc de code spécifié et se termine à l'achèvement de ce bloc.

Voici quelques propriétés du Job:

- Il est créé avec le constructeur de coroutine "launch". Il exécute un bloc de code spécifique et se termine à la fin de ce bloc;
- Une fois créée, la tâche est automatiquement lancée;
- Permet de manipuler le cycle de vie de la coroutine;
- Ils ont une hiérarchie, nous pouvons avoir des jobs parents et fils;
- Un job est annulé à l'aide de la fonction cancel();
- Si un job est annulé, tous ses parents et fils le seront également;
- L'exécution d'un job ne produit pas de valeur de résultat. Nous devrions utiliser une interface Deferred pour un job qui produit un résultat.

CoroutineStart

- Il est possible grâce à la classe CoroutineStart de préciser, **quand** une coroutine doit se lancer :
    - L'argument par défaut est **DEFAULT** (surprenant, non ?), cela signifie tout simplement que la coroutine démarre immédiatement.
    - **LAZY :** la coroutine va se lancer uniquement lorsqu'elle sera nécessaire.
    - **ATOMIC :** similaire à l'argument par défaut, mais la coroutine ne peut pas être annulée tant qu'elle n'a pas été exécutée.

Voici un exemple d'utilisation :

```
GlobalScope.async(start = CoroutineStart.LAZY) {
  ...
}
```

Annuler une coroutine

- Il est bien sûr possible d'annuler une coroutine précédemment lancée. Par exemple, si l'utilisateur ouvre une page qui nécessite l'appel d'une API mais qu'il ferme la page aussitôt, le résultat de l'appel n'aura donc plus de sens et peut-être annulé. Comme, on l'a vu au-dessus, un coroutine builder retourne un Job ou un Deferred, ces objets peuvent être utilisés afin d'annuler une coroutine. Prenons un exemple :

```
val job = launch {
  repeat(1000) { i ->
    println("job: I'm sleeping $i ...")
    delay(500L)
  }
}
delay(1300L)
println("main: I'm tired of waiting!")
job.cancel()
job.join()
println("main: Now I can quit.")
```

Annuler une coroutine

- Analysons ce petit bout de code :
    - On lance une coroutine grâce au coroutine builder launch, on récupère se faisaint une instance de la classe Job.
    - Cette dernière va itérer toutes les secondes et afficher un message avant de faire une pause de 500 ms.
    - Pour le main thread, on fait une pause de 1,3 s, puis on va appeler la méthode cancel et join de la classe Job.
        - cancel(): on arrêtera la coroutine à la prochaine itération.
        - join(): on attend la fin de la complétion de la coroutine.

À garder en tête

- À savoir, qu'il existe la méthode cancelAndJoin, qui a exactement le même effet que faire les appels aux deux méthodes cancel et join à la suite.

Rendre sa coroutine cancellable

- Il faut savoir que l'annulation d'une coroutine est coopérative ! Qu'est-ce que ça veut dire ? Cela signifie que le code doit être prévu pour être cancellable. Prenons un cas concret, si vous lancez une coroutine et que cette dernière effectue une opération sans vérifier si elle peut être annulée alors elle ne le sera pas. C'est peut-être encore flou, prenons l'exemple de la documentation :

```
val startTime = System.currentTimeMillis()
val job = launch(Dispatchers.Default) {
  var nextPrintTime = startTime
  var i = 0
  while (i < 5) { // computation loop
    if (System.currentTimeMillis() >= nextPrintTime) {
      println("job: I'm sleeping ${i++} ...")
      nextPrintTime += 500L
    }}
}
delay(1300L)
println("main: I'm tired of waiting!")
job.cancelAndJoin()
println("main: Now I can quit.")
```

Rendre sa coroutine cancellable

- **Le résultats est :**

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm sleeping 3 ...
job: I'm sleeping 4 ...
main: Now I can quit.
```

- Le problème est simple, à aucun moment, on ne vérifie l'état de la coroutine, comme on effectue un while, l'exécution se continuera tant que i ne sera pas supérieur à 5. Il faut donc rendre son code cancellable, pour faire cela, deux possibilités s'offrent à nous :
    - Une première solution serait d'utiliser une suspending function pour vérifier l'état de l'annulation. En utilisant la function yield par exemple.
    - La seconde approche, c'est tout simplement de vérifier l'état d'annulation de la coroutine, grâce à l'attribut: isActive, qui retourne true, si le job est encore actif (non complété et non annulé).
- Pour rendre annulable l'exemple précédent, il suffirait de remplacer le while (i < 5) par while (isActive). Enfin, la raison la plus évidente qui existerait pour annuler une coroutine, c'est qu'un certain temps a été atteint. Par exemple dans le cas d'une requête réseau, on pourrait vouloir arrêter la coroutine si un certain timeout a été atteint. Il existe déjà des méthodes pour faire cela : withTimeout et withTimeoutOrNull

PARTIE 5

UTILISER LES OUTILS ANDROID ET KOTLIN

Dans ce module, vous allez :

- Documenter le code Kotlin
- Manipuler les plugins Kotlin

02 heures

CHAPITRE 1

DOCUMENTER LE CODE KOTLIN

Ce que vous allez apprendre dans ce chapitre :

- Documenter le code Kotlin avec KDoc
- Documenter le code Kotlin avec Dokka

2 heures

CHAPITRE 1

DOCUMENTER LE CODE KOTLIN

1. **DOKKA**
2. KDOC

Dokka

- L'outil de génération de documentation de Kotlin s'appelle Dokka. **Consultez la partie TP** pour les instructions d'installation et d'utilisation.
- Dokka a des plugins pour Gradle, Maven et Ant, vous pouvez donc intégrer la génération de documentation dans votre processus de construction.
- Dokka est un moteur de documentation pour Kotlin, remplissant la même fonction que javadoc pour Java. Tout comme Kotlin lui-même, Dokka prend entièrement en charge les projets Java/Kotlin en langage mixte. Il comprend les commentaires Javadoc standard dans les fichiers Java et les commentaires KDoc dans les fichiers Kotlin, et peut générer de la documentation dans plusieurs formats, notamment Javadoc standard, HTML et Markdown.

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 1

DOCUMENTER LE CODE KOTLIN

1. DOKKA
2. **KDOC**

Kdoc

- Tout comme avec JavaDoc, les commentaires KDoc commencent par /** et se terminent par */ . Chaque ligne du commentaire peut commencer par un astérisque, qui n'est pas considéré comme faisant partie du contenu du commentaire.
- Par convention, le premier paragraphe du texte de la documentation (le bloc de texte jusqu'à la première ligne blanche) est la description sommaire de l'élément, et le texte suivant est la description détaillée.
- Chaque balise de bloc commence sur une nouvelle ligne et commence par le caractère @.

KDoc

Voici un exemple de classe documentée à l'aide de KDoc :

```kotlin
/**
* A group of *members*.
* This class has no useful logic; it's just a documentation example.
* @param T the type of a member in this group.
* @property name the name of this group.
* @constructor Creates an empty group.
*/
class Group<T>(val name: String) {
  /**
   * Adds a [member] to this group.
   * @return the new size of the group.
   */
  fun add(member: T): Int { ... }
}
```

KDoc - Block tags

- KDoc prend actuellement en charge les balises de bloc suivantes :
- **@param name :** Documente un paramètre de valeur d'une fonction ou un paramètre de type d'une classe, d'une propriété ou d'une fonction. Pour mieux séparer le nom du paramètre de la description, si vous le préférez, vous pouvez mettre le nom du paramètre entre parenthèses. Les deux syntaxes suivantes sont donc équivalentes :
- **@param name description. @param[name] description.**
- **@return :** Documente la valeur de retour d'une fonction.
- **@constructor :** Documente le constructeur primaire d'une classe.
- **@receiver :** Documente le récepteur d'une fonction d'extension.

KDoc - Block tags

- **@property name :** Documente la propriété d'une classe qui a le nom spécifié.Cette balise peut être utilisée pour documenter les propriétés déclarées dans le constructeur primaire, lorsque l'insertion d'un commentaire doc directement avant la définition de la propriété serait gênante.
- **classe @lance, classe @exception :** Documente une exception qui peut être levée par une méthode. Puisque Kotlin n'a pas d'exceptions vérifiées, il n'y a pas non plus d'attente que toutes les exceptions possibles soient documentées, mais vous pouvez quand même utiliser cette balise lorsqu'elle fournit des informations utiles aux utilisateurs de la classe.
- **@sample identifier :** Incorpore le corps de la fonction portant le nom qualifié spécifié dans la documentation de l'élément actuel, afin de montrer un exemple d'utilisation de l'élément.
- **@see identifier :** Ajoute un lien vers la classe ou la méthode spécifiée au bloc Voir aussi de la documentation.
- **@author :** Spécifie l'auteur de l'élément documenté.

KDoc - Block tags

- **@since :** Spécifie la version du logiciel dans laquelle l'élément à documenter a été introduit.
- **@suppress :** Exclut l'élément de la documentation génerée. Peut être utilisé pour les éléments qui ne font pas partie de l'API officielle d'un module mais qui doivent néanmoins être visibles de l'extérieur.

Attention

- KDoc ne prend pas en charge la balise @deprecated. À la place, veuillez utiliser l' annotation **@Deprecated**

KDoc - Liens vers les éléments

- Pour établir un lien avec un autre élément (classe, méthode, propriété ou paramètre),il suffit de mettre son nom entre crochets :
    - **Use the method [foo] for this purpose.**
- Si vous souhaitez spécifier une étiquette personnalisée pour le lien, utilisez la syntaxe de style référence Markdown :
    - **Use [this method][foo] for this purpose.**
- Vous pouvez également utiliser des noms qualifiés dans les liens.

À noter

- Contrairement à JavaDoc, les noms qualifiés utilisent toujours le caractère point pour séparer les composants, même avant un nom de méthode :
    - **Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.**

- Les noms dans les liens sont résolus en utilisant les mêmes règles que si le nom était utilisé à l'intérieur de l'élément documenté. En particulier, cela signifie que si vous avez importé un nom dans le fichier courant, vous n'avez pas besoin de le qualifier complètement lorsque vous l'utilisez dans un commentaire KDoc.

À noter

- KDoc ne dispose d'aucune syntaxe pour résoudre les membres surchargés dans les liens. Puisque l'outil de génération de documentation Kotlin place la documentation de toutes les surcharges d'une fonction sur la même page, l'identification d'une fonction surchargée spécifique n'est pas nécessaire pour que le lien fonctionne.

KDoc - documentation des modules et des package

- La documentation d'un module dans son ensemble, ainsi que des packages de ce module, est fournie dans un fichier Markdown séparé, et les chemins d'accès à ce fichier sont transmis à Dokka à l'aide du paramètre de ligne de commande -include ou des paramètres correspondants dans Ant, Maven et Plugins Gradle.
- À l'intérieur du fichier, la documentation du module dans son ensemble et des packages individuels est introduite par les en-têtes de premier niveau correspondant. Le texte de l'en-tête doit être **Module <module name>** pour le module et Package **<package qualified name>** pour un package.

Voici un exemple de contenu du fichier :

```
# Module kotlin-demo

The module shows the Dokka syntax usage.

# Package org.jetbrains.kotlin.demo

Contains assorted useful stuff.

## Level 2 heading

Text after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

Useful stuff in another package.
```

CHAPITRE 2

MANIPULER LES PLUGINS KOTLIN

Ce que vous allez apprendre dans ce chapitre :

- Utiliser Gradle
- Utiliser Kapt

02 heures

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

MANIPULER LES PLUGINS KOTLIN

1. **KAPT**
2. Gradle

Kapt

- Les processeurs d'annotation sont pris en charge dans Kotlin avec le plugin de compilation kapt . En un mot, vous pouvez utiliser des bibliothèques telles que Dagger ou Data Binding dans vos projets Kotlin.
- Pour utiliser Kapt il faut appliquer le kotlin-kapt Gradle :

```
plugins {
  id "org.jetbrains.kotlin.kapt" version "1.4.10"
}
```

- Alternativement, vous pouvez utiliser la syntaxe du apply plugin :

```
apply plugin: 'kotlin-kapt'
```

- Pour ajouter les dépendances respectives à l'aide de la configuration kapt dans votre bloc de dependencies :

```
dependencies {
  kapt 'groupId:artifactId:version'
}
```

Kapt

- Les tâches de traitement des annotations kapt sont mises en cache dans Gradle par défaut. Cependant, les processeurs d'annotation exécutent du code arbitraire qui ne peut pas nécessairement transformer les entrées de tâche en sorties, peut accéder et modifier les fichiers qui ne sont pas suivis par Gradle, etc.
- Si les processeurs d'annotation utilisés dans la construction ne peuvent pas être correctement mis en cache, il est possible de désactiver complètement la mise en cache pour kapt en ajoutant les lignes suivantes au script de construction, afin d'éviter les faux positifs de cache pour les tâches kapt :

```
kapt {
  useBuildCache = false
}
```

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

MANIPULER LES PLUGINS KOTLIN

1. KAPT
2. **Gradle**

Gradle

- Gradle est l'outil de gestion de dépendance et d'automatisation officiel d'Android. Il combine les atouts de Apache Maven, et de Apache Ant, ajoute ses propres fonctionnalités et utilise le DSL(Domain Specific Language) basé sur groovy (Un langage de la machine virtuel Java) pour déclarer les configurations de votre projet. Contrairement à Apache Maven et Apache Ant qui utilisent un fichier xml pour déclarer les configurations de projet .
- Voici quelques avantages qui font du gradle l'outil de construction le plus utilisé.
    - **Performance élevée :** Le gradle évite les taches inutiles en n'exécutant que les tâches dont les entrées ou les sorties ont changé. Vous pouvez également utiliser le build cache pour permettre la réutilisation des résultats des taches d'exécution précédente. Il existe bien d'autres optimisations implémentées par le gradle.
    - **Fondé sur la JVM :** Le gradle s'exécute sur la machine virtuel Java(JVM). Vous devez installer le JDK (Java Development Kit) pour l'utiliser.
    - **Extensibilité :** Vous pouvez facilement étendre le gradle et fournir vos propres types de tâche pour construire des modéles. Il ajoute de nouveaux concepts de construction tel que les product flavors et les build types

Le plugin Android du Gradle

- Le plugin Android du gradle est un plugin du gradle qui ajoute plusieurs fonctionnalités spécifiques pour la construction de vos applications Android. Vous pouvez aussi créer votre propre plugin Android pour le gradle qui exécutera durant la construction de votre application.
- Android Studio utilise le gradle wrapper pour complètement intégrer le plugin Android du gradle. Le plugin Android du gradle peux s'exécuter indépendamment de Android Studio.Ce qui signifie que vous pouvez construire votre application depuis Android Studio ou en ligne de commande.

Le fichier de construction du projet

- Ce fichier définit les configurations de construction qui s'appliquent à tous les modules de votre projet. Voici un exemple de ce fichier

```
buildscript {
  repositories {
     google()
     jcenter()
  }
  dependencies {
     classpath 'com.android.tools.build:gradle:3.1.3'
     // NOTE: Do not place your application dependencies here; they belong
     // in the individual module build.gradle files
  }
}
allprojects {
  repositories {
     google()
     jcenter()
  }
}
ext {
  compileSdkVersion = 28
}
task clean(type: Delete) {
  delete rootProject.buildDir
}
```

Le fichier de construction du projet

- **Buildscript :** Ce bloc vous permet de configurer les dépôts, les dépendances du gradle et des plugins du gradle
- **Repositories :** Ce bloc configure les dépôts utilisés par le gradle pour rechercher et télécharger les dépendances de votre projet.
- **Dependencies :** ce bloc configure les dépendances que le gradle a besoin pour construire votre projet.
- **Allprojects :** ce blog configure tous les dépôts et dépendances utilisé par tous les modules de votre projet

- Si votre projet contient plusieurs modules, vous pouvez définir des propriétés dans le bloc ext dans le fichier gradle.build (Project).
- Vous pouvez ensuite accéder à la propriété dans le fichier gradle.build de chaque module de votre projet comme dans l'exemple suivant avec la propriété **rootProject.ext.property_name**.

Le fichier de construction du projet

- Voici comment utiliser cette propriété dans le fichier build.gradle(app) de votre projet

```
apply plugin: 'com.android.application'

android {
  compileSdkVersion rootProject.ext.compileSdkVersion
  defaultConfig {
     applicationId "premierprojet.com.premierprojet"
     minSdkVersion 16
     targetSdkVersion 27
     versionCode 1
     versionName "1.0"
     testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }

}
```

Le fichier de construction de module

- Ce fichier permet de configurer les paramètres de construction d'un module spécifique.

Voici un exemple de fichier de construction du module :

```
apply plugin: 'com.android.application'
android {
  compileSdkVersion 27
  defaultConfig {
     applicationId "premierprojet.com.premierprojet"
     minSdkVersion 16
     targetSdkVersion 27
     versionCode 1
     versionName "1.0"
     testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }
  buildTypes {
     release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
     }}
}
dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar'])
  implementation 'com.android.support:appcompat-v7:27.1.1'
}
```