Programmer en KOTLIN
M210

Support du cours développement mobile

ETTAHERI Nizar
ettaheri.nizar@gmail.com

Chapitre 1

M210 Programmer en Kotlin

- **Utiliser les types checks et Casts**
- **Introduire les coroutines**

- **Type safe builders**

En utilisant des fonctions bien nommées en tant que générateurs en combinaison avec des littéraux de fonction avec récepteur, il est possible de créer des générateurs de type sécurisé et typés statiquement dans Kotlin.
Les constructeurs de type sécurisé permettent de créer des langages spécifiques au domaine (**DSL**) basés sur Kotlin adaptés à la construction de structures de données hiérarchiques complexes de manière semi-déclarative. Exemples de cas d'utilisation pour les constructeurs :

1. Génération de balisage avec du code Kotlin, tel que **HTML** ou **XML**
2. Configuration des routes pour un serveur Web : **Ktor**

Comment ça fonctionne

Supposons que vous deviez implémenter un constructeur de type sécurisé dans Kotlin.

Tout d'abord, définissez le modèle que vous souhaitez créer. Dans ce cas, vous devez modéliser des balises HTML. Cela se fait facilement avec un tas de classes.

Par exemple, HTML est une classe qui décrit la <html> balise définissant les enfants comme <head>et <body>.

Plus de détails dans le code :

```
html {
 // ...
    }
```

html est en fait un appel de fonction qui prend une expression lambda comme argument. Cette fonction est définie comme suit :

```
fun html(init: HTML.() -> Unit): HTML {
   val html = HTML()
   html.init()
   return html
    }
```

Cette fonction prend un paramètre nommé init, qui est lui-même une fonction. Le type de la fonction est HTML.() -> Unit, qui est un *type de fonction avec récepteur* . Cela signifie que vous devez passer une instance de type HTML(un *récepteur*) à la fonction, et vous pouvez appeler les membres de cette instance à l'intérieur de la fonction.

Le récepteur est accessible via le mot this.clé :

```
html {
    this.head { ... }
    this.body { ... }
     }
```

(headet bodysont des fonctions membres de HTML.)

Maintenant, thispeut être omis, comme d'habitude, et vous obtenez déjà quelque chose qui ressemble beaucoup à un constructeur :

```
html {
    head { ... }
    body { ... }
     }
```

Regardons le corps de la fonction html tel que défini ci-dessus. Il crée une nouvelle instance de HTML, puis il l'initialise en appelant la fonction qui est passée en argument (dans cet exemple cela revient à appeler head et body sur l' HTML instance), puis il retourne cette instance. C'est exactement ce qu'un constructeur doit faire.

Les fonctions head et body de la HTML classe sont définies de la même manière que html. La seule différence est qu'ils ajoutent les instances construites à la children collection de l'instance englobante HTML :

```
fun head(init: Head.() -> Unit): Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit): Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
      }
```

En fait, ces deux fonctions font exactement la même chose, vous pouvez donc avoir une version générique initTag:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
   tag.init()
   children.add(tag)
   return tag
    }
```

Donc, maintenant vos fonctions sont très simples :

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

Et vous pouvez les utiliser pour créer <head>et <body> baliser.

Considérez le code suivant :

```
import com.example.html.* // see declarations below

fun result() =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup to XML"}

            // an element with attributes and text content
            a(href = "https://kotlinlang.org") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // content generated by
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

Output

HTML encoding with Kotlin

this format can be used as an alternative markup to HTML

Kotlin

This is some **mixed** text. For more see the Kotlin project

some text

- 1*2 = 2
- 2*2 = 4
- 3*2 = 6
- 4*2 = 8
- 5*2 = 10

Code source (Intelij IDEA)

https://github.com/NizarETH/TypeSafeBuilderHtml

- **Type alias**

Les alias de type fournissent des noms alternatifs pour les types existants. Si le nom du type est trop long, vous pouvez introduire un autre nom plus court et utiliser le nouveau à la place.
Il est utile de raccourcir les types génériques longs.

Par exemple, il est souvent tentant de réduire les types de collection :

```
typealias NodeSet = Set<Network.Node>

    typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

Vous pouvez fournir différents alias pour les types de fonction :

```
typealias MyHandler = (Int, String, Any) -> Unit

    typealias Predicate<T> = (T) -> Boolean
```

Quiz 1 : T et K signifient quoi ?

Vous pouvez avoir de nouveaux noms pour les classes internes et imbriquées :

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
    typealias BInner = B.Inner
```

Les alias de type n'introduisent pas de nouveaux types. Ils sont équivalents aux types sous-jacents correspondants. Lorsque vous ajoutez typealias Predicate<T>et utilisez Predicate<Int>dans votre code, le compilateur Kotlin le développe toujours en (Int) -> Boolean. Ainsi vous pouvez passer une variable de votre type à chaque fois qu'un type de fonction générale est requis et inversement :

```
typealias Prédicat < T > = ( T ) -> Booléen

fun foo ( p : Prédicat < Int > ) = p ( 42 )

amusement principal () {
  val f : ( Int ) -> Booléen = { it > 0 }
  println ( foo ( f )) // affiche "true"

  val p : Prédicat < Int > = { it > 0 }
  println ( listOf ( 1 , - 2 ). filter ( p )) // affiche "[1]"
}
```

- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

- **Introduire les coroutines**

- **Coroutines Scope**
- **Fonction launch**
- **Fonction await**
- **Jobs**

Une coroutine

Une coroutine est une unité de traitement permettant d'exécuter du code **asynchrone**. Sur le principe, il s'agit d'un Thread "**allégé**". Son avantage étant qu'elle peut être suspendue et reprise plus tard. Une coroutine peut être suspendue dans un Thread et être reprise dans un autre. Elle ne dépend donc pas d'un Thread en particulier, ce qui apporte un avantage considérable lors de l'utilisation de plusieurs traitements asynchrones.

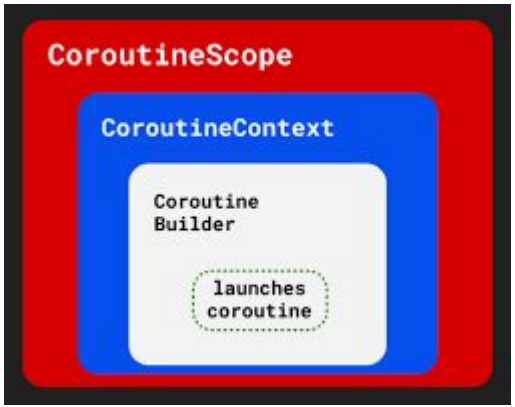
Du fait qu'une coroutine est plus légère qu'un Thread, il est possible d'en créer des centaines de milliers en parallèle sur un poste classique sans avoir de problème d' "**out of memory**", et donc de réaliser plusieurs traitements en même temps. Il est aussi possible de faire communiquer les coroutines entre elles, mais nous y reviendrons plus tard.

Les coroutines ont vocation à être utilisées notamment pour des traitements d'arrière-plan, tels que des **appels à des web services** pour charger des données, des **traitements lourds** qui ne nécessitent pas de bloquer le Thread principal, ou encore des traitements n'ayant pas le besoin de manipuler l'interface utilisateur (ou seulement lorsque le traitement est terminé). Dans un contexte Android, on pourra par exemple utiliser les coroutines lors d'une phase de login en affichant un composant de chargement, lors du chargement de données provenant d'un serveur, ou pour remplir une base locale en arrière-plan.

Coroutine Scope

Un objet **CoroutineScope** est un contexte qui applique l'annulation et d'autres règles à ses enfants et à leurs enfants de manière récursive.

Les fonctions permettant de créer des coroutines telles que **launch()** et **async()** étendent CoroutineScope.



Coroutine Scope

Il crée, exécute et garde la trace de toutes vos coroutines. Il fournit également des événements de cycle de vie comme le démarrage et la pause d'une coroutine.

Voici le résultat lorsque nous exécutons le code ci-dessus:

```
Program execution will now block
Task from GlobalScope
Task from runBlocking
Task from coroutineScope
Program execution will now continue
```

```
fun main() {
    println("Program execution will now block")
    runBlocking { this: CoroutineScope
        launch { this: CoroutineScope
            delay( timeMillis: 1000L)
            println("Task from runBlocking")
        }

        GlobalScope.launch { this: CoroutineScope
            delay( timeMillis: 500L)
            println("Task from GlobalScope")
        }

        coroutineScope { this: CoroutineScope
            launch { this: CoroutineScope
                delay( timeMillis: 1500L)
                println("Task from coroutineScope")
            }
        } ^runBlocking
    }
    println("Program execution will now continue")
}
```

1. **runBlocking** { ... } : crée une Coroutine de manière bloquante. Elle bloquera le thread principal ou le thread dans lequel elle est utilisée. Dans l'exemple ci-dessus, l'impression "**Program execution will now continue**" sera exécutée après la fin du bloc runBlocking.
2. **GlobalScope.launch**{ ... } : crée une nouvelle Coroutine, le scope sera le cycle de vie de l'application.
3. **coroutineScope** { ... } : Crée un nouveau scope personnalisé

   et ne se termine pas tant que toutes les Coroutines enfants

   ne sont pas terminées ;

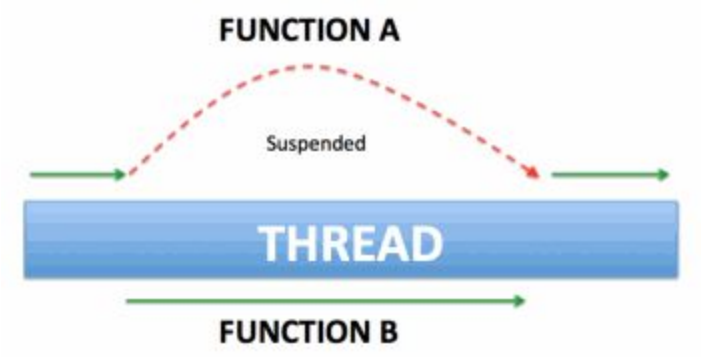
- Si le parent est annulé, tous les enfants sont annulés.
- Le parent attendra toujours l'achèvement de ses fils.

```
fun main() {
    println("Program execution will now block")
    runBlocking { this: CoroutineScope
        launch { this: CoroutineScope
            delay( timeMillis: 1000L)
            println("Task from runBlocking")
        }

        GlobalScope.launch { this: CoroutineScope
            delay( timeMillis: 500L)
            println("Task from GlobalScope")
        }

        coroutineScope { this: CoroutineScope
            launch { this: CoroutineScope
                delay( timeMillis: 1500L)
                println("Task from coroutineScope")
            }
        } ^runBlocking
    }
    println("Program execution will now continue")
}
```

Delay est une **fonction de suspension** spéciale. Elle suspend la Coroutine pendant un temps spécifique. La suspension d'une Coroutine ne bloque pas le thread sous-jacent, mais permet aux autres Coroutines de s'exécuter et d'utiliser le thread sous-jacent pour leur code. Nous verrons plus en détail les fonctions de suspension dans la section suivante.

Suspendre les fonctions:

Les fonctions de suspension sont comme l'épine dorsale des Coroutines. Il est donc très important de bien comprendre ce concept avant d'aller plus loin.



Une fonction de suspension est simplement une fonction qui peut être mise en pause et reprise ultérieurement. Elle peut exécuter une opération de longue durée et attendre qu'elle se termine sans se bloquer. La syntaxe d'une fonction suspensive est similaire à celle d'une fonction régulière, à l'exception de l'ajout du mot-clé **suspend**.

Notez que les fonctions de suspension sont **automatiquement synchronisées** avec les autres variables et fonctions du thread principal.

Voici le résultat lorsque nous exécutons le code ci-dessus:

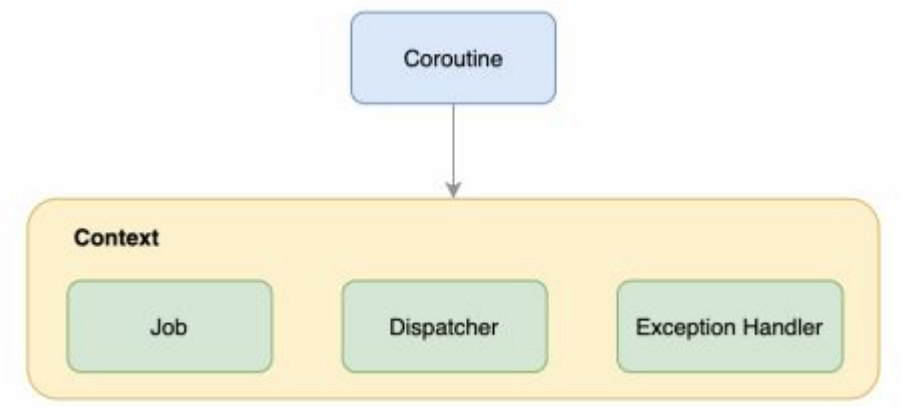
```
Hello, World!
Suspend functions are cool
There have been 2 calls so far
```

```
var functionCalls = 0

fun main() {
    GlobalScope.launch { completeMessage() }
    GlobalScope.launch { improveMessage() }
    print("Hello, ")
    Thread.sleep( millis: 2000L)
    println("There have been $functionCalls calls so far")
}

suspend fun completeMessage() {
    delay( timeMillis: 500L)
    println("World!")
    functionCalls++
}

suspend fun improveMessage() {
    delay( timeMillis: 1000L)
    println("Suspend functions are cool")
    functionCalls++
}
```

Quiz 2 : Combien de courtines on peut créer dans un projet ?

Le contexte

Les coroutines s'exécutent toujours dans un certain contexte qui est représenté par une valeur du type CoroutineContext.



Le contexte de la Coroutine est un ensemble de divers éléments. Les principaux éléments sont le Job de la Coroutine, son dispatcher et aussi son gestionnaire d'exception.

Le Job

Selon la documentation officielle, *"un job est une chose annulable avec un cycle de vie qui aboutit à son achèvement.*
Le Job est créé par 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.

```
runBlocking { this: CoroutineScope
    val job1 : Job  = launch { this: CoroutineScope
        delay( timeMillis: 3000L)
        println("Job1 launched")
    }

    job1.invokeOnCompletion { println("Job1 completed") }

    delay( timeMillis: 500L)
    println("Job1 will be cancelled")
    job1.cancel()
}
```

Voici le résultat lorsque nous exécutons le code ci-dessus:

```
Job1 will be cancelled
Job1 completed
```

Quiz 3 : Pourquoi Job cancelled avant completed ?

Comment récupérer la valeur d'un Job à la fin de son exécution?

Comme nous l'avons mentionné juste avant, nous allons utiliser **Deferred** qui est un Job avec un résultat. Il attendra et bloquera le thread actuel jusqu'à ce que nous récupérions le résultat.

En fait, il est créé avec le constructeur **asynchrone** Coroutine et le résultat peut être récupéré par la méthode **await()**, qui lève une exception si le Deferred a échoué.

Voici un exemple d'utilisation:

```
fun main() {
    runBlocking { this: CoroutineScope
        val firstDeferred : Deferred<Int> = async { getFirstValue() }
        val secondDeferred : Deferred<Int> = async { getSecondValue() }

        println("Doing some processing here")
        delay( timeMillis: 500L)
        println("Waiting for values")

        val firstValue : Int = firstDeferred.await()
        val secondValue : Int = secondDeferred.await()

        println("The total is ${firstValue + secondValue}")
    }
}

suspend fun getFirstValue(): Int {
    delay( timeMillis: 1000L)
    val value : Int = Random.nextInt( until: 100)
    println("Returning first value $value")
    return value
}

suspend fun getSecondValue(): Int {
    delay( timeMillis: 2000L)
```

Voici le résultat lorsque

nous exécutons le code ci-dessus:

```
Doing some processing here
Waiting for values
Returning first value 13
Returning second value 472
The total is 485
```

Await : attendre la fin de la tâche sans bloquer un thread.

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 arrête d'attendre l'étape d'achèvement et reprend immédiatement avec CancellationException .

Toutes les coroutines doivent être exécutées dans un Dispatcher, même si elles sont exécutées sur le thread principal.

Les coroutines peuvent se suspendre, et le Dispatcher est la chose qui sait comment les reprendre.

Pour spécifier l'endroit où les coroutines doivent s'exécuter, Kotlin fournit trois Dispatcher que vous pouvez utiliser :

- **Dispatchers.Main** : thread principal sur Android, interagit avec l'interface utilisateur et effectue un travail léger.

➢ Appel des fonctions de suspension;
➢ Appelez les fonctions de l'interface utilisateur;
➢ Mise à jour de LiveData.

- **Dispatchers.IO** : Optimisé pour les entrées/sorties de disque et de réseau en dehors du thread principal.

➢Demandes de bases de données;
➢ Lecture/écriture de fichiers;
➢ Mise en réseau.

- **Dispatchers.Default** : Optimisé pour le travail intensif du CPU en dehors du thread principal.

➢ Trier une liste;
➢ Analyse de JSON;
➢ Utils

Comment changer le contexte d'une Coroutine?

Ceci peut être facilement réalisé en utilisant la fonction suspend **withContext(Dispatcher)**. Cela permet de changer facilement le contexte de démarrer une Coroutine et de passer d'un Dispatcher à l'autre.

Voici un exemple de la façon dont nous pouvons l'utiliser pour passer du Dispatcher **par défaut** au Dispatcher **IO**:

```
runBlocking { this: CoroutineScope
    launch(Dispatchers.Default) { this: CoroutineScope
        println("First context: $coroutineContext")
        withContext(Dispatchers.IO) { this: CoroutineScope
            println("Second context: $coroutineContext")
        }
        println("Third context: $coroutineContext")
    }
}
```

La gestion correcte des exceptions a un impact énorme sur la façon dont les utilisateurs perçoivent notre application. Si votre application ne cesse de planter, l'utilisateur sera déçu et ne l'utilisera peut-être plus jamais.

À cette fin, nous avons deux options :

- Utilisation de try/catch;
- Utilisation de **CoroutineExceptionHandler**

Le CoroutineExceptionHandler est un élément optionnel d'un **CoroutineContext** permettant de gérer les exceptions non attrapées.

Voici comment nous pouvons définir un **CoroutineExceptionHandler**, chaque fois qu'une exception est attrapée, vous avez des informations sur le **CoroutineContext** où l'exception s'est produite et l'exception elle-même.

Voici comment nous pouvons définir un **CoroutineExceptionHandler**, chaque fois qu'une exception est attrapée, vous avez des informations sur le **CoroutineContext** où l'exception s'est produite et l'exception elle-même.

```
runBlocking { this: CoroutineScope
    val myHandler : CoroutineExceptionHandler = CoroutineExceptionHandler {coroutineContext, throwable ->
        println("Exception handled: ${throwable.localizedMessage}")
    }

    val job : Job = GlobalScope.launch(myHandler) { this: CoroutineScope
        println("Throwing exception from job")
        throw IndexOutOfBoundsException("exception in coroutine")
    }
    job.join()
}
```

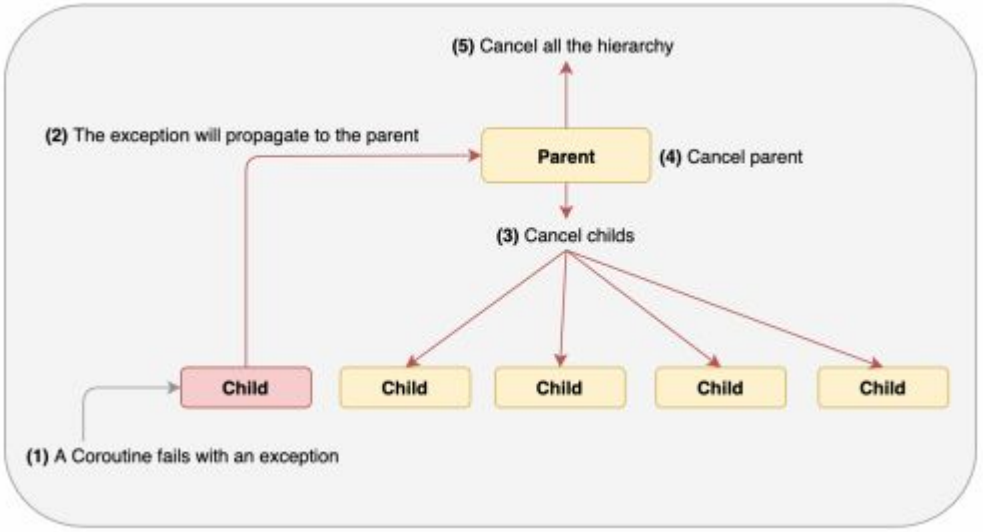
Voici le résultat lorsque nous exécutons le code ci-dessus:

job.join() // wait until child coroutine completes

```
Throwing exception from job
Exception handled: exception in coroutine
```

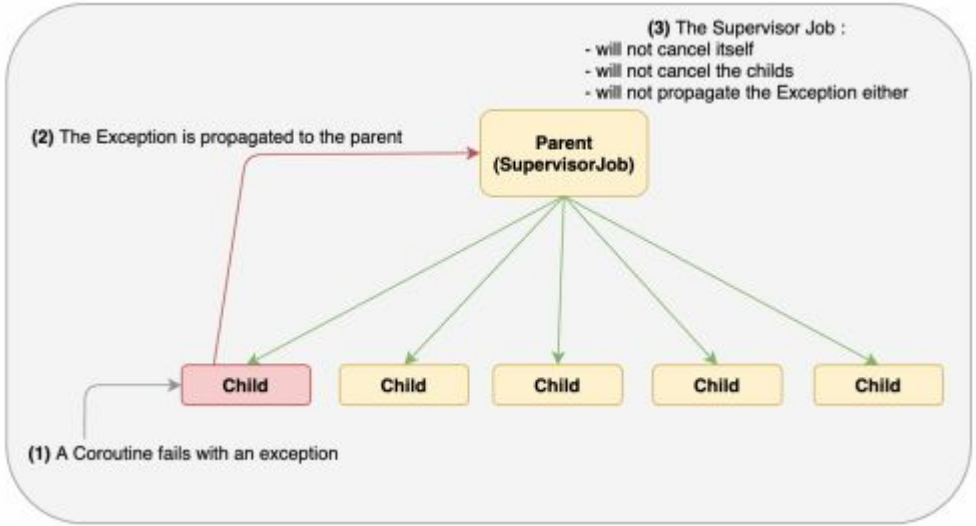
Lorsque la Coroutine échoue avec une exception, elle propage cette exception à son parent. Ensuite, le parent va :

- Annulez le reste de ses fils;
- Annuler son exécution;
- Propager l'exception vers son parent jusqu'à la racine de la hiérarchie et toutes les Coroutines déjà lancées dans le **CoroutineScope** seront également annulées.



Cette approche n'est pas toujours une bonne idée à utiliser. Par exemple, nous avons la fonction init() qui initialise les interactions de l'utilisateur avec les composants de l'interface utilisateur en utilisant une CoroutineScope. Imaginez que si une coroutine enfant échoue, le scope sera automatiquement annulé et l'interface utilisateur ne répondra plus car le scope entier sera annulé.

Pour y remédier, nous pouvons utiliser **SupervisorJob** qui est une implémentation différente d'un **job**.



Nous pouvons créer un **CoroutineScope**

en utilisant l'une des méthodes suivantes :

- val uiScope = CoroutineScope(SupervisorJob())
- supervisorScope { ... }

Les flux dans les coroutines

Un autre aspect intéressant des Coroutines est la possibilité d'utiliser des flux. Il s'agit d'un flux de valeurs qui sont calculées de manière asynchrone à partir d'une Coroutine.

- Les flux émettent des valeurs avec la fonction **emit()**;
- Flows reçoit les valeurs à l'aide de la fonction **collect()**;
- La fonction de construction des flux est la fonction **flow{..}**;
- Un flux est annulé lorsque la coroutine est annulée;
- Les flux sont des flux froids, en d'autres termes, le code à l'intérieur d'un constructeur de flux ne s'exécute pas tant que le flux n'est pas collecté.

Dans cet exemple, nous allons essayer d'émettre les valeurs d'une liste de nombres. Chaque fois que nous émettons une valeur, nous suspendons la coroutine pour un délai spécifique en utilisant la fonction ***delay***:

Voici le résultat lorsque nous exécutons le code ci-dessus:

```
Received prime number 7
Received prime number 11
Received prime number 13
Received prime number 17
Received prime number 19
Received prime number 23
Received prime number 29
Finished receiving numbers
```

```
fun main() {
    runBlocking { this: CoroutineScope
        println("Receiving prime numbers")
        sendPrimes().collect { it: Int
            println("Received prime number $it")
        }
        println("Finished receiving numbers")
    }
}

fun sendPrimes(): Flow<Int> = flow { this: FlowCollector<Int>
    val primesList : List<Int> = listOf(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    primesList.forEach { it: Int
        delay( timeMillis: it * 100L)
        emit(it)
    }
}
```

- Une liste peut également être convertie en un flux à l'aide de la fonction **asFlow()** :
- Nous pouvons créer un flux directement

  à partir de tout type d'objets en utilisant

  la fonction **flowOf()** :

```
fun main() {
    runBlocking { this: CoroutineScope
        sendNumbers().collect { it: String
            println("Received $it")
        }
    }
}

fun sendNumbers()
        = flowOf( ...elements: "One", "Two", "Three")
```

```
fun main() {
    runBlocking { this: CoroutineScope
        sendNumbers().collect { it: Int
            println("Received $it")
        }
    }
}

fun sendNumbers()
        = listOf(1, 2, 3).asFlow()
```

Opérateurs de débit

Flow dispose d'un tas d'opérateurs sympas pour

nous faciliter le codage:

Map: mettre en correspondance un flux avec

un autre flux

Dans cet exemple, nous allons faire correspondre
un flux de Int à un autre flux de String.
Chaque élément de la liste sera transformé
en String suivant :
"*mapping (valeur du numéro de la liste)*".

Opérateurs de débit

- Filter: Filtre les valeurs de débit avec une condition spécifique.

```
fun main() {
    runBlocking { this: CoroutineScope
//        mapOperator()
        filterOperator()
    }
}

suspend fun filterOperator() {
    (1..10).asFlow()
        .filter { it: Int
            it % 2 == 0
        }
        .collect { it: Int
            println(it)
        }
}
```

Opérateurs de débit

- transform: Opérateur de transformation général qui peut émettre **n'importe quelle** valeur en tout point.

```
fun main() {
    runBlocking { this: CoroutineScope
//        mapOperator()
//        filterOperator()
        transformOperator()
    }
}

suspend fun transformOperator() {
    (1..10).asFlow()
        .transform { this: FlowCollector<Int>
            emit( value: "Emitting string value $it")
            emit(it)
        }
        .collect { it: Any
            println(it)
        }
}
```

Opérateurs de débit

- **take** : N'utilise qu'un certain nombre de valeurs, ne tient pas compte du reste des valeurs émises
- **toList** : convertit un flux en une liste;
- **toSet** : convertit un flux en un ensemble qui n'a que des valeurs uniques (pas de valeurs dupliquées);
- **flowOn** : permet de basculer le contexte du flux.

Code source

https://gist.github.com/NizarETH/63fb18743a07b8b8f0cbf9b3cd88c6a1

- **Kapt**
- **Gradle**

- **Kapt**

Quand utiliser KAPT ?

La première question qui nous vient généralement à l'esprit lorsque nous parlons de traitement des annotations dans Android, en particulier venant d'un arrière-plan Java, est de savoir si vous devriez utiliser KAPT.

La réponse est que KAPT est l'outil officiel pour effectuer le traitement des annotations dans Kotlin, donc la réponse est toujours. C'est aussi longtemps que vous avez besoin du traitement des annotations, sinon, pourquoi l'ajouter ?

Inclure KAPT dans un projet Android

Pour commencer à utiliser KAPT dans un projet Android, vous devez l'ajouter dans les dépendances de

l'application comme vous le faites avec tout autre outil externe. Utilisez la ligne suivante pour l'inclure :

```
appliquer le plugin : 'kotlin-kapt'
```

C'est aussi simple que cela, mais il y a une erreur courante lors de l'ajout de cette dépendance au projet.

L'erreur indique "Le plugin Kotlin doit être activé avant 'kotlin-kapt'", la solution à cela est simple, mais lorsque vous avez des tonnes de lignes de code, cela peut être déroutant. Cette erreur signifie que lors de l'ajout de la ligne ci-dessus à votre fichier Gradle, vous devez d'abord ajouter le plugin Kotlin comme ceci :

```
appliquer le plugin : 'kotlin-android'
appliquer le plugin : 'kotlin-kapt'
```

A quoi sert KAPT ?

La réponse à cette question est simple, il est utilisé pour le traitement des annotations dans Kotlin. Mais si vous vous posez la question, cela signifie probablement que vous n'êtes pas familier avec le traitement des annotations.

Voyons d'abord ce que sont les annotations. Les annotations sont les mots que vous voyez en haut des fichiers et des méthodes précédés du symbole '@'. L'exemple le plus courant que vous avez peut-être déjà vu est l'annotation **@Override**. Mais il y en a beaucoup d'autres, dont certaines peuvent être directement interprétées par Kotlin, celles-ci sont appelées annotations intégrées. D'autres ont besoin d'un "processeur d'annotations", également connu sous le nom de KAPT, à utiliser dans votre code.

La définition des annotations dans la documentation Kotlin indique ce qui suit :

Les annotations sont des moyens d'attacher des métadonnées au code.

https://kotlinlang.org/docs/reference/annotations.html

Ok, mais… Que fait un processeur d'annotation ?

Les processeurs d'annotation remontent à l'époque où Android était basé sur Java. Il existe de nombreux tutoriels sur les annotations en Java car à un moment donné dans le passé, ils sont devenus des outils très puissants (ce qu'ils sont) et populaires.

Que ce soit en Java ou en Kotlin, les processeurs d'annotations sont des outils qui parcourent votre code pendant la compilation et génèrent eux-mêmes du code pour votre application. Cela explique pourquoi ils sont si puissants, si vous savez utiliser les annotations, ils peuvent vous faire gagner du temps en générant eux-mêmes du code.

Une grande chose à propos de KAPT est que si vous avez un projet avec des fichiers Java et Kotlin, il peut prendre en charge les deux.

L'inverse n'est pas vrai car APT (outil de traitement d'annotations Java) ne peut pas interpréter les annotations Kotlin.

kapt est en mode maintenance. Il sera à jour avec les dernières versions de Kotlin et Java, mais il n'y aura pas l'intention d'implémenter de nouvelles fonctionnalités.

Veuillez utiliser l' API de traitement des symboles Kotlin (**KSP**) pour le traitement des annotations. Consultez la liste des bibliothèques prises en charge par **KSP** .

Kotlin Symbol Processing (**KSP**) est une API que vous pouvez utiliser pour développer des plugins de compilateur légers. **KSP** fournit une API de plug-in de compilateur simplifiée qui exploite la puissance de Kotlin tout en maintenant la courbe d'apprentissage au minimum. Par rapport à kapt , les processeurs d'annotation qui utilisent KSP peuvent fonctionner jusqu'à **2 fois plus rapidement**.

- **Gradle**

Gradle est un **moteur de production** fonctionnant sur la plateforme Java. Il permet de construire des projets en **Java**, **Scala**, **Groovy** voire **C++**.

Gradle allie les atouts de **Apache Maven** et **Apache Ant** : il allie l'utilisation de conventions à la manière de **Maven** (convention plutôt que configuration) avec la flexibilité de Ant pour décrire les tâches de construction, avec une cohérence forte dans l'interface de programmation des tâches.

- L'outil **Gradle** a été développé pour la compilation d'exécutables multi-projets, qui tendent à être gourmands en espace. Son fonctionnement est basé sur une série de tâches de compilation qui sont exécutées de manière sérielle ou en parallèle.
- Un service web permet une visualisation des étapes de la compilation.
- Un système de *plugin* permet d'étendre les fonctionnalités du logiciel afin de supporter des fonctionnalités supplémentaires et d'autres langages de programmation.

Gradle est un logiciel libre distribué sous la licence Licence Apache 2.0. Sa version initiale date de 2007.

Gradle permet d'écrire des tâches de construction dans un fichier de construction en utilisant le langage Groovy. Il est possible d'importer des tâches standards qui permettent de construire des programmes utilisant un ou plusieurs langages (Java, Groovy…) ou qui permettent d'exécuter des activités d'ingénierie logicielle telles qu'exécuter les tests unitaires, assurer la qualité du code (**SonarQube**, **Checkstyle**)…

Quiz 3 : Alternative de Gradle ?

Gradle reprend certaines des idées fortes de Maven :

- convention plutôt que configuration
- cycle de vie
- gestion des dépendances à la manière d'Apache Ivy ou Maven
- référentiel (ou entrepôts)

Gradle présente les avantages suivants :

- possibilité de scripter la construction en Groovy dans le fichier de construction ;
- possibilité de changer le comportement par défaut de certaines tâches ;
- une notation compacte pour décrire les dépendances ;
- un moteur de production pensé pour produire des projets multi-langages.

Gradle permet de construire sans effort des projets utilisant d'autres langages que Java. Migrer de Maven vers Gradle se fait très facilement pour un projet respectant les conventions Maven.

Voici le fichier **build.gradle** :

```
apply plugin: 'java'
```

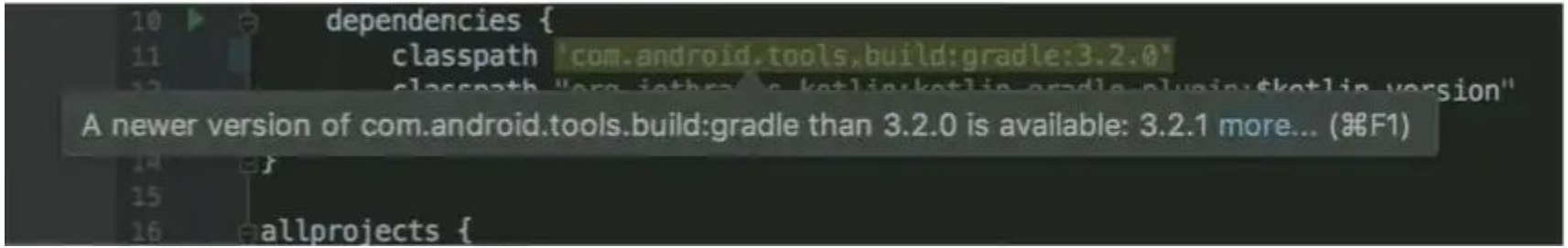
Executer **gradle build** donne la sortie suivante :

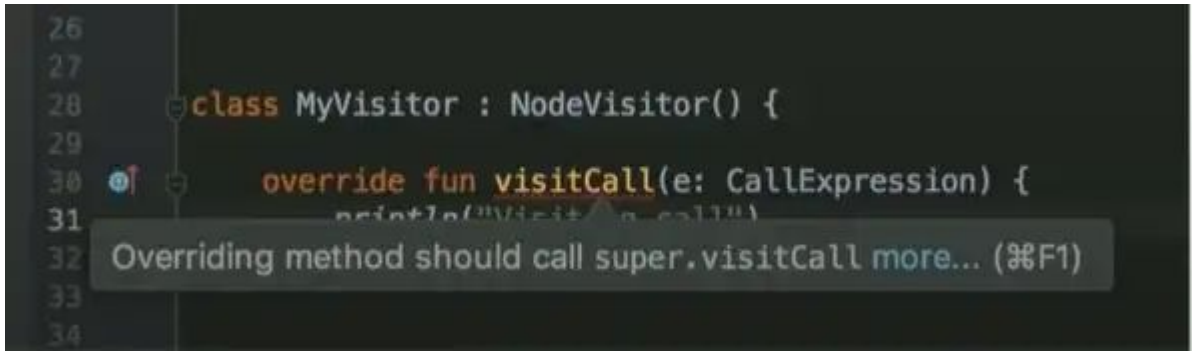
```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

  BUILD SUCCESSFUL
```

- **Lint**

Beaucoup d'entre nous savent probablement ce qu'est réellement Android **Lint**, même si nous ne connaissons pas le nom. Parce que c'est la chose qui affiche les messages d'avertissement dans l'IDE. Il peut s'agir d'avertissements ou d'erreurs qui feront échouer la construction.

Les images suivantes montrent un exemple de suggestions Lint.





Android Studio fournit un outil d'analyse de code appelé lint qui peut vous aider à identifier et à corriger les problèmes liés à la qualité structurelle de votre code. Par exemple, Lint peut nous aider à résoudre ces problèmes

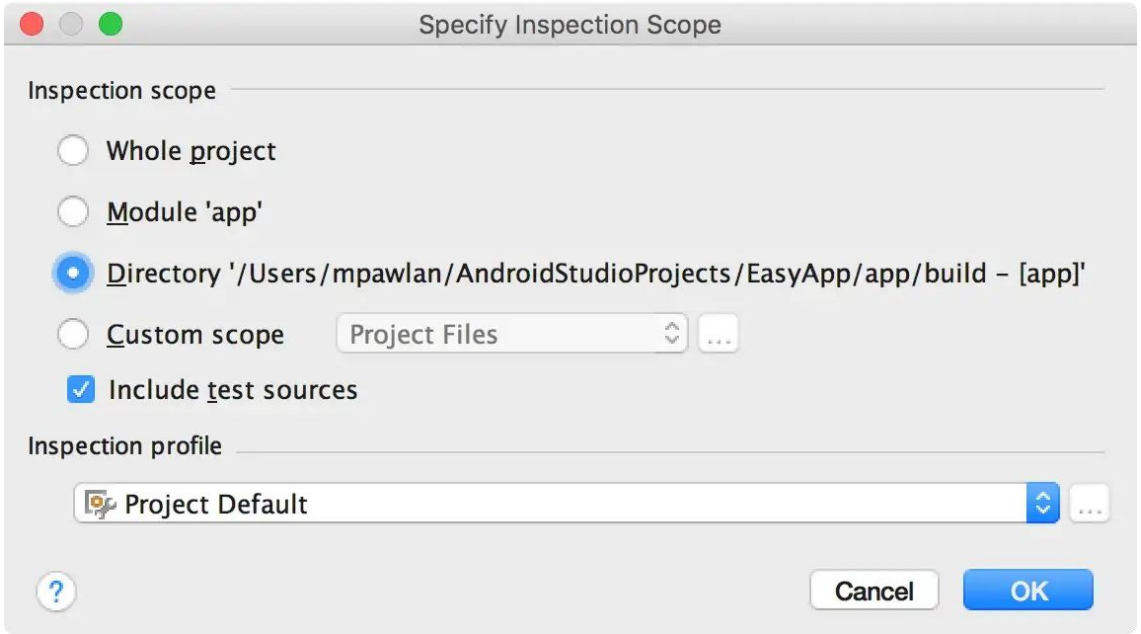
- Fichiers de ressources XML contenant des espaces de noms inutilisés
- Utilisation d'éléments obsolètes
- Les appels d'API qui ne sont pas pris en charge par les versions d'API cibles peuvent entraîner l'échec de l'exécution correcte du code
- Beaucoup plus..

- Les problèmes détectés dans notre code par Lint nous seront signalés avec quelques suggestions et un niveau d'avertissement. Nous pouvons utiliser la suggestion pour corriger notre code.
- Lint peut être personnalisé pour voir un type particulier d'erreur dans notre projet. Lorsque vous utilisez Android Studio, le processus d'inspection s'exécute chaque fois que nous construisons notre projet. Cependant, nous pouvons exécuter manuellement des inspections ou exécuter des Lint à partir de la ligne de commande.

Comment utiliser les Lint

Par Android Studio :

Pour inspecter manuellement un fichier dans Android Studio, cliquez sur **Code > Inspecter le code.** La fenêtre ci-dessous s'ouvrira :



Exécutez lint depuis la ligne de commande :

Appelez la tâche **Lint** pour votre projet en saisissant l'une des commandes suivantes à partir du répertoire racine de votre projet :
Sous Windows :

```
gradlew lint
```

Sous Linux ou Mac :

```
./gradlew lint
```

Lorsque l'outil Lint a terminé ses vérifications, il fournit des chemins vers les versions XML et HTML du rapport Lint.

Utilisation de la ligne de base :

Si nous travaillons sur un grand projet et que nous souhaitons trouver les futures erreurs qui pourraient survenir lors de l'ajout de codes supplémentaires à notre projet, nous pouvons ajouter une ligne de base au projet.

Ainsi, lint ignorera les problèmes de code précédents et nous avertira uniquement des nouvelles lignes de code ajoutées après la ligne de base.

Pour créer un instantané de référence, modifiez le `build.gradle` fichier de votre projet comme suit

```
android {
 lintOptions {
   baseline file("lint-baseline.xml")
 }
}
```

- **Product Flavors**

Utilisation de base des (Flavors) saveurs de produits Android/variantes de construction

Les saveurs de produits Android Studio sont une fonctionnalité puissante et notre plugin Android Gradle vous permet d'échanger des classes Java/Kotlin au moment de la compilation et ne nécessite pas de bibliothèques supplémentaires.

Voici quelques exemples typiques de dimensions de saveur :

- **Des flavors gratuites/payantes** pour obtenir deux APK différents qui seront publiés sur vos canaux de distribution.
- **Stable/expérimental** pour maintenir les expériences pendant un ensemble de sources différent et générer rapidement des versions bêta.

Les flavors Android sont également connues sous le nom de types de construction Android ou de variantes de construction Android. Elles constituent le moyen de développement d'applications Android natives pour implémenter différentes versions de la même application avec des modifications mineures. Voici la documentation officielle d'Android pour la configuration d'Android Product Flavors.

Avec les Product Flavors, ce coût est souvent économisé et investi davantage dans l'apport de fonctionnalités d'application plus avancées ou de nouveaux concepts à brancher.

Types de construction et variantes de construction

Les variantes de build sont le résultat de l'utilisation par Gradle d'un ensemble spécifique de règles pour combiner les paramètres, le code et les ressources configurés dans vos types de build et vos saveurs de produit.

Le type de construction applique différents paramètres de construction et d'empaquetage. Un exemple de types de construction sont "Debug" et "Release".

Création des products flavors

Il existe de nombreuses situations où nous utilisons des product flavors, l'une des situations les plus courantes où de nombreux développeurs optent pour cette fonctionnalité.

Une entreprise souhaite republier l'application existante avec des modifications mineures (logos, couleurs, fonctionnalités supplémentaires mineures) en tant qu'application de remplacement dans PlayStore. la fonctionnalité de base de l'application reste équivalente, mais il y aura quelques changements décoratifs dans l'application(Fête/Ramadan/Hallowen/événement sportif...).

C'est l'une des situations les plus simples où vous utiliserez **Product Flavors.**

Par défaut, Android Studio générera les types de construction **"debug"** et **"release" pour votre projet.**

Le débogage est le type de construction qui est utilisé une fois que nous exécutons l'appliance à partir de l'IDE directement sur un outil.

Une **version** est le type de build qui nécessite que vous signiez l'APK. les versions de version sont destinées à être téléchargées sur le Google Play Store. dans le type de version de version, nous obscurcissons le code à l'aide de ProGuard pour arrêter l'ingénierie inverse.

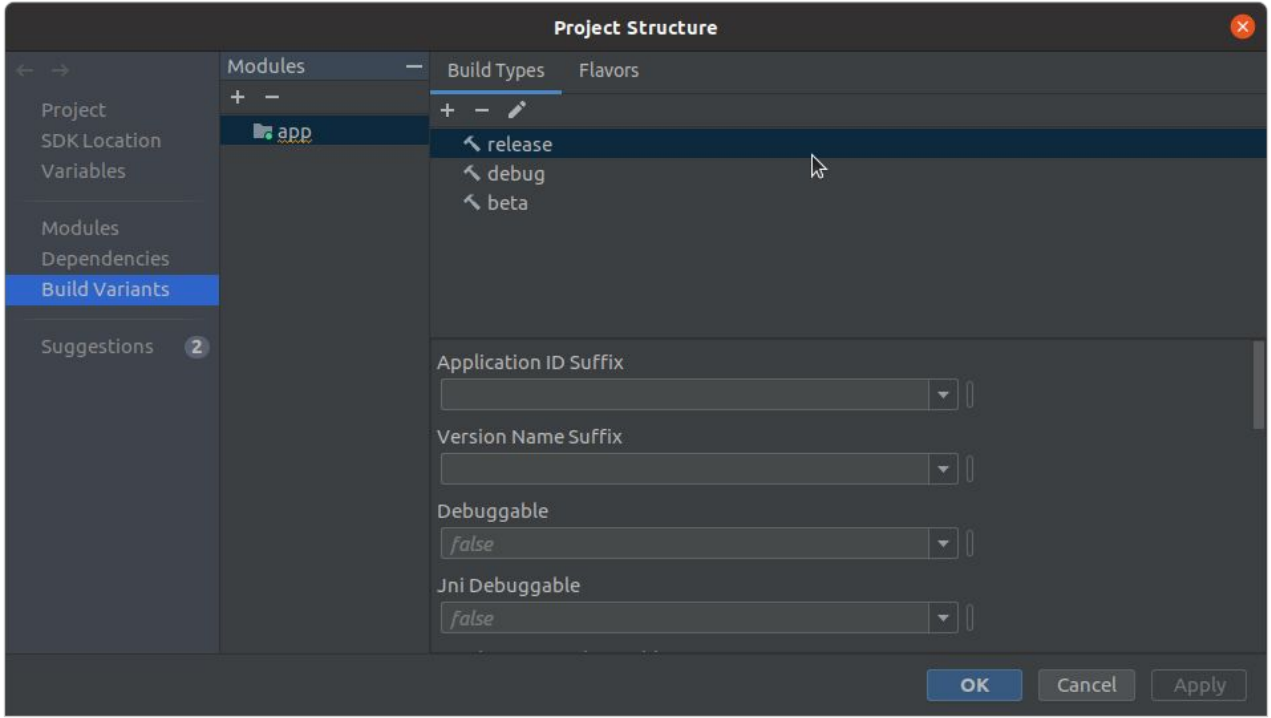
L'image suivante montre les types de construction par défaut.

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Ajoutons plus de types de construction et plus de propriétés au **buildConfig.**

```
buildTypes {

    debug{
        applicationIdSuffix ".debug"
        versionNameSuffix "-debug"
    }

    beta{
        applicationIdSuffix ".beta"
        versionNameSuffix "-beta"
    }

    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Le suffixe applicationId ajoute la chaîne à l'applicationId de l'application.
L'option Build Variant se trouve dans la partie gauche de l'écran dans Android Studio ou vous pouvez aller dans **Build >** sélectionner **Build Variant** ou vous pouvez utiliser la touche de raccourci pour appuyer sur **cmd + shift + A** et rechercher **« Build Variant »** :



Créer Product Flavor est assez facile dans Android Studio, nous devons simplement ajouter le bloc **productFlavors** à l'intérieur du bloc Android dans le fichier build.gradle au niveau de l'application.

```
flavorDimensions "ice_cream"
productFlavors {
    VanillaFlavor {
        applicationId "com.simform.vanillaflavor"
        dimension "ice_cream"
        versionCode 1
        versionName "1.0"
    }

    ChocolateFlavor {
        applicationId "com.simform.chocolateflavor"
        dimension "ice_cream"
        versionCode 1
        versionName "1.0"
    }

    StrawberryFlavor {
        applicationId "com.simform.strawberryflavor"
        dimension "ice_cream"
        versionCode 1
        versionName "1.0"
    }
}
```

Gestion des ressources et des fichiers Java/Kotlin pour différentes saveurs

À l'heure actuelle, nous savons tous comment créer différentes saveurs avec buildTypes, mais cela ne suffit pas dans le monde réel, nous devrions toujours être prêts à créer des ressources et des fichiers Java/Kotlin spécifiques à chaque variante.

Avant cela, vous devez simplement réaliser le bloc **sourceSets** dans Gradle, qui définit les répertoires de code qui seront disponibles dans les types de construction spécifiés. vous comprendrez cela en lisant la suite.

Fichiers de ressources

Voyons d'abord comment créer des fichiers de ressources pour chaque version.

Basculez vers la vue du projet dans le panneau de gauche, puis développez tous les répertoires jusqu'à src, puis cliquez avec le bouton droit sur src, puis accédez à **nouveau> Dossier> Dossier Res**

Ensuite, vous verrez une fenêtre contextuelle avec deux champs :

1. L'ensemble de sources cibles n'est rien d'autre que le répertoires pour la saveur sélectionnée.

2. L'emplacement du nouveau dossier est l'emplacement où le fichier sera créé.

De même, nous créerons un répertoire pour tous les types de build aromatisés si nécessaire.

Fichiers Java/Kotlin

La création d'un répertoire Java/Kotlin pour chaque saveur est similaire à la création de répertoires de ressources, la seule différence est que nous sélectionnons un dossier Java/Kotlin plutôt que le dossier res.

Observez qu'à la fin de la démo, une nouvelle ligne est ajoutée dans le fichier **build.gradle avec le bloc sourceSets** comme ci-dessous.

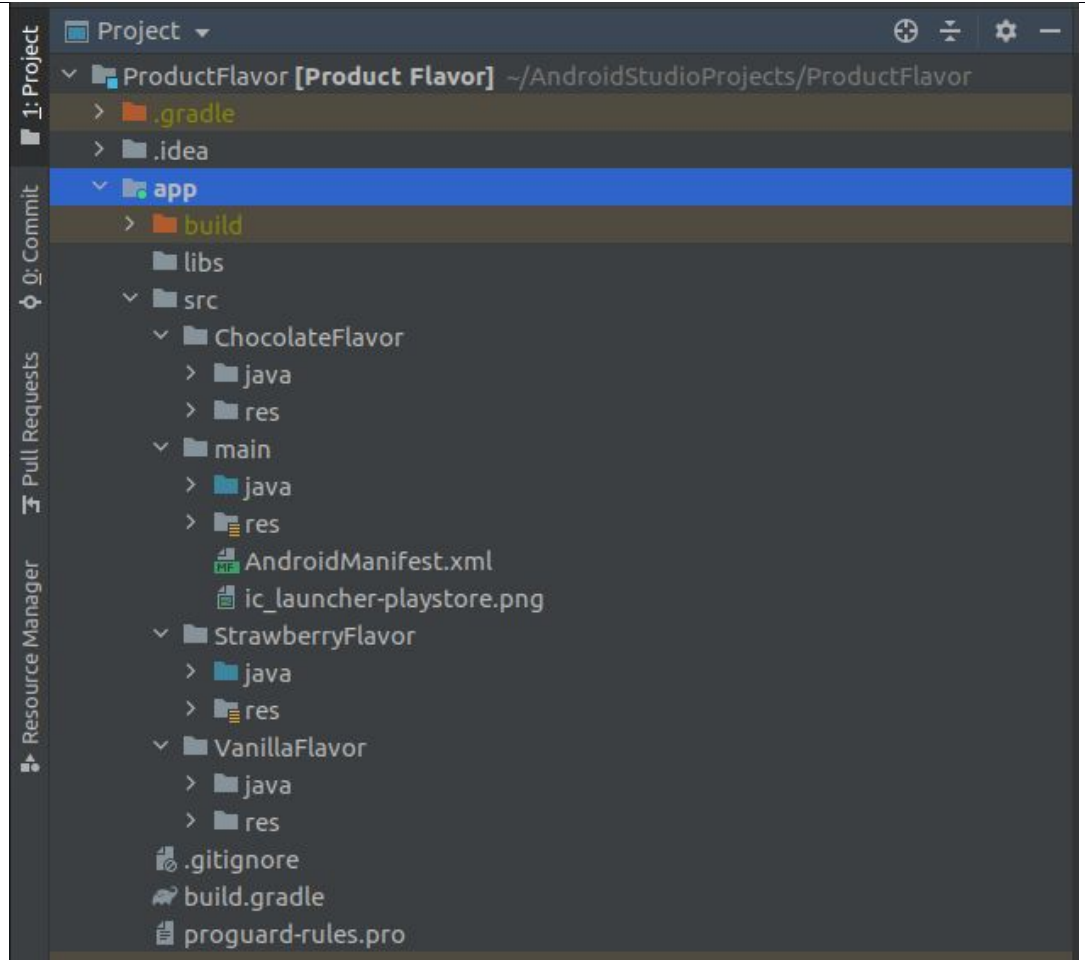
```
sourceSets {
    VanillaFlavor {
        res {
            srcDirs 'src/VanillaFlavor/res'
        }
        java {
            srcDirs 'src/VanillaFlavor/java'
        }
    }
    ChocolateFlavor {
        res {
            srcDirs 'src/ChocolateFlavor/res'
        }
        java {
            srcDirs 'src/ChocolateFlavor/java'
        }
    }
    StrawberryFlavor {
        res {
            srcDirs 'src/StrawberryFlavor/res'
        }
        java {
            srcDirs 'src/StrawberryFlavor/java'
        }
    }
}
```

L'ensemble de sources décrit les répertoires de code à inclure lors de la création d'APK pour un type de construction spécifique.

Ici, les répertoires du bloc principal seront inclus en tant que code standard.

VanillaFlavor et les autres types de construction Flavor incluront à la fois les répertoires de blocs principaux et également les répertoires du bloc Flavor.

Nous allons maintenant créer des sous-répertoires avec la même hiérarchie que dans le dossier principal pour présenter de nouvelles mises en page, dessinables et valeurs pour différentes saveurs.


Project
ProductFlavor [Product Flavor] ~/AndroidStudioProjects/ProductFlavor
.gradle
.idea
app
build
libs
src
ChocolateFlavor
java
res
main
java
res
AndroidManifest.xml
ic_launcher-playstore.png
StrawberryFlavor
java
res
VanillaFlavor
java
res
.gitignore
build.gradle
proguard-rules.pro
1: Project
0: Commit
Pull Requests
Resource Manager

Maintenant que vous avez trois saveurs différentes de la même application pour changer la variante de construction, Android Studio utilise sélectionnez **Build > sélectionnez Build Variant** dans la barre de menus (ou cliquez sur Build Variants dans la barre de fenêtres), puis sélectionnez une variante de construction dans le menu.

Maintenant, changez la variante de construction dans le menu déroulant et exécutez l'application, vous avez trois versions différentes de la même application (Captures d'écran jointes).

Maintenant, changez la variante de construction dans le menu déroulant et exécutez l'application,

vous avez trois versions différentes de la même application (Captures d'écran jointes).

Comment internationaliser son application Android

Android est l'un des systèmes d'exploitation mobiles populaires ayant des millions d'utilisateurs dans plus de 190 pays et grandissant jour après jour. Ainsi, lorsque vous souhaitez que votre application soit globalement performante, il est toujours judicieux de la traduire en plusieurs langues. C'est une tâche assez simple en Android.

Dans cet article, nous allons créer une application multilingue prenant en charge le français, l'anglais et l'allemand.

Comment ça marche?

Par défaut, android considère l'anglais comme langue principale et charge les ressources de chaîne à partir de res ⇒ values ⇒ strings.xml. Lorsque vous souhaitez ajouter un support pour une autre langue, vous devez créer un dossier de valeurs en ajoutant un trait d'union et le code de langue ISO. Par exemple, si vous souhaitez ajouter du support pour l'allemand, vous devez créer un dossier de valeurs nommé values-de et y conserver un fichier strings.xml avec toutes les chaînes traduites en allemand.

Elle fonctionne comme suit

1. Lorsque l'utilisateur change la langue de l'appareil via Paramètres ⇒ Langue et saisie, Android vérifie lui-même les ressources linguistiques appropriées dans l'application. (L'utilisateur sélectionne l'allemand par exemple)

2. Si l'application prend en charge la langue sélectionnée, Android recherche ses ressources de chaîne dans le dossier valeurs (code de langue ISO) du projet. (Pour l'allemand, il charge les valeurs de chaîne de valeurs-de / string.xml)

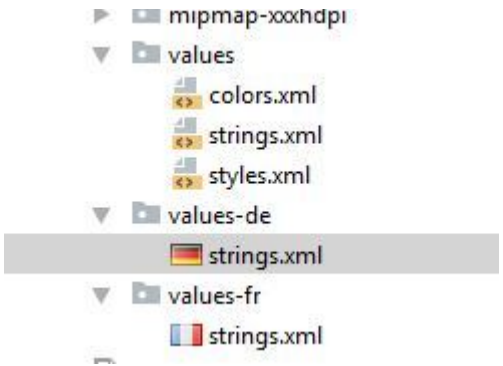
3. Si la langue supportée strings.xml manque une valeur de chaîne, android charge toujours les chaînes manquantes du fichier strings.xml par défaut i.e values / strings.xml

Il est donc obligatoire que le fichier stings.xml par défaut contienne toutes les valeurs de chaîne utilisées par l'application. Sinon, l'application va cracher.

Pour commencer, créez un nouveau projet android.

Maintenant, sous le dossier res, créez deux dossiers nommés values-de, values-fr et un fichier strings.xml dans chacun des dossiers.

Votre projet devrait ressembler à ceci une fois que vous avez créé les fichiers



Nous allons donc créer dans values-fr>strings.xml, une ressource avec la clé « hello» et la valeur « Bonjour tous le monde ! » ce qui donne :

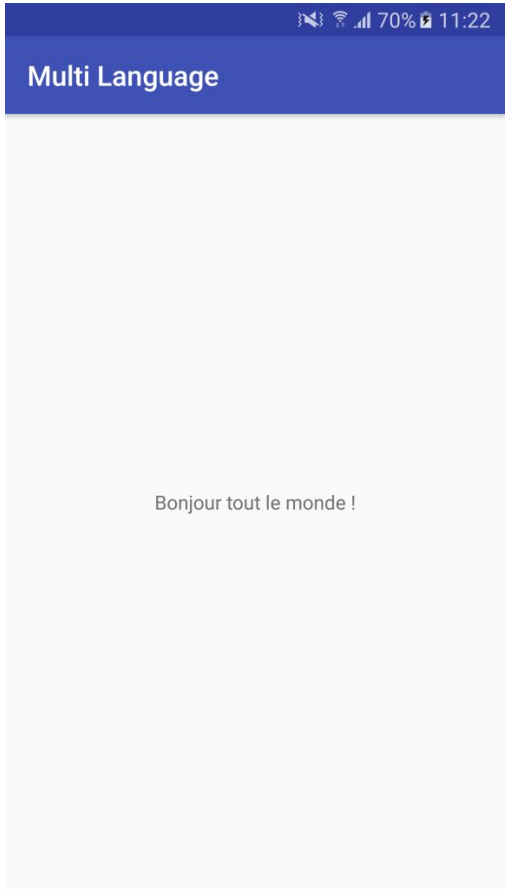
Répétez cette opération pour le dossier values-de mais cette fois ci en allemand.

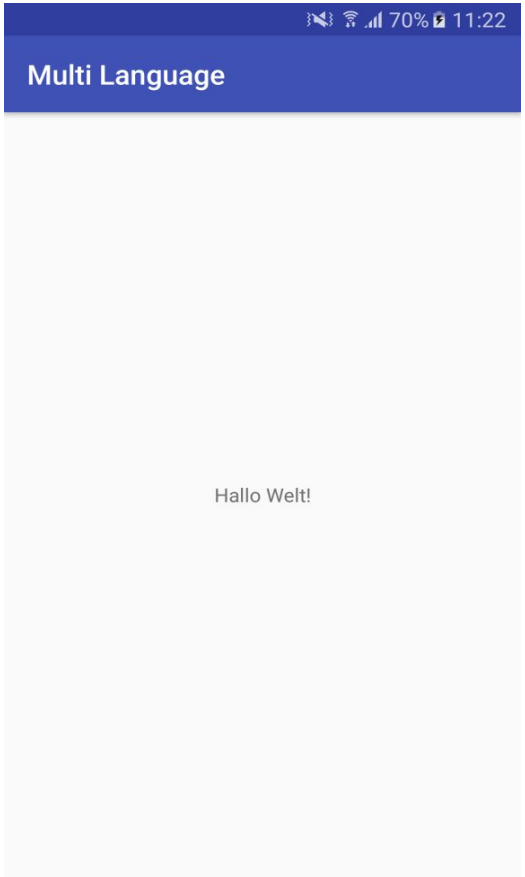
```
1  <resources>
2      <string name="app_name">Multi Language</string>
3      <string name="hello">Hallo Welt!</string>
4  </resources>
5
```

Le fichier activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.setiafanou.multilanguage.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

Exécutez votre application





Rappel :

- **GIT**

- **Les commandes de base de Git**

  I. Démarrer un dépôt Git

  Cloner un dépôt existant

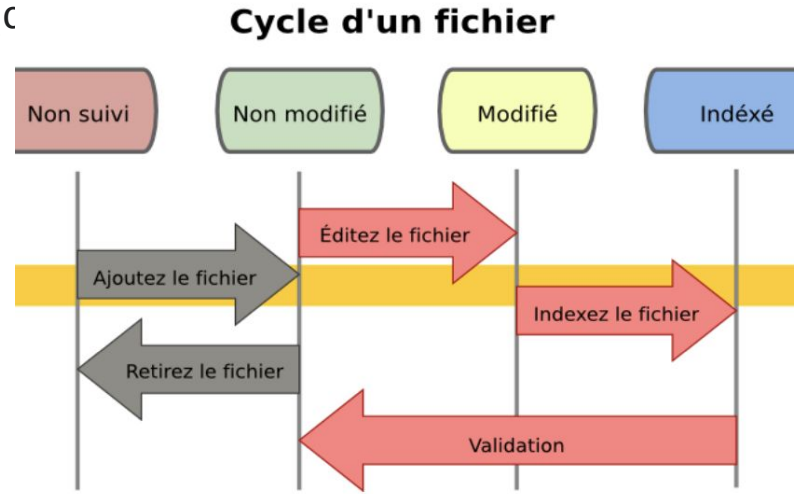
  $ git clone ssh://bob@serveur/git/projet.git

  $ git clone https://idul@projets.fsg.ulaval.ca/git/scm/cpp/projet.git

  La commande **«clone»** crée une copie d'un dépôt Git existant. Vous clonez un dépôt avec git clone [url]. On a à la fois le clone par l'url ssh qui demande une clé public et par https qui demande à chaque fois l'authentification.

II. Enregistrer des modifications dans un dépôt

Quatre états d'un projet Git: Non suivi: fichier n'étant (n'appartenant) pas ou plus géré par Git; Non modifié: fichier sauvegardé de manière sûre dans sa version courante dans la base de données du dépôt; Modifié: fichier ayant subi des modifications depuis la dernière fois qu'il a été soumis; Indexé: idem pour modifié, sauf qu'il sera pris instantané dans sa version courante de la prochaine soumission (c



II. Enregistrer des modifications dans un dépôt

a. Vérifier l'état des fichiers

$ git status

 # On branch master

 # Untracked files:

 # (use "git add ..." to include in what will be committed)

 # LISEZMOI nothing added to commit but untracked files present (use "git add" to track)

Untracked files : fichiers non suivis car il n'est pas indexé.

b. Indexer l'ajout ou les changements d'un fichier avant de soumettre (commit) les modifications

$ git add -A

II. Enregistrer des modifications dans un dépôt

c. Valider les modifications

$ git commit –m « Mon premier commit»

La commande «commit» est faite pour valider ceux qui a été indexés avec «git add». Pad d'indexe pas de validation. Après l'option –m est suivi d'un commentaire de l'utilisateur décrivant ce qui a été accompli et le fichier est ajouté au répertoire Git/dépôt (local) mais pas encore sur le dépôt distant.

d. Visualiser l'historique des validations **$ git log**

Par défaut, git log énumère en ordre chronologique inversé les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier

II. Enregistrer des modifications dans un dépôt

c. Valider les modifications

$ git commit –m « Mon premier commit»

La commande «commit» est faite pour valider ceux qui a été indexés avec «git add». Pad d'indexe pas de validation. Après l'option –m est suivi d'un commentaire de l'utilisateur décrivant ce qui a été accompli et le fichier est ajouté au répertoire Git/dépôt (local) mais pas encore sur le dépôt distant.

d. Visualiser l'historique des validations **$ git log**

Par défaut, git log énumère en ordre chronologique inversé les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier

II. Enregistrer des modifications dans un dépôt

e. Pousser son travail sur un dépôt distant

$ git push origin master La commande «push» sert à envoyer tout les «commits» effectués se trouvant dans le répertoire Git/dépôt (HEAD) de la copie du dépôt local vers le dépôt distant.

f. Récupérer et tirer depuis des dépôts distants **$ git pull** La commande «pull» permet de mettre à jour votre dépôt local des dernières validations (modifications des fichiers). La commande est faite avant l'indexation des modifications.

Faire une **branche** signifie diverger de la ligne principale de développement et continuer à travailler sans se préoccuper de cette ligne principale. La branche par défaut dans Git quand vous créez un dépôt s'appelle master et elle pointe vers le dernier des commits réaliser.

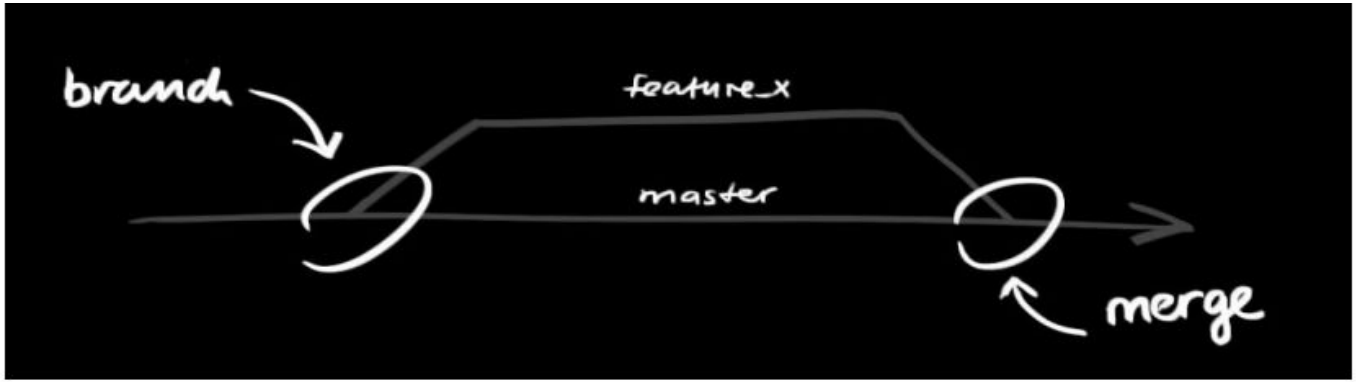
Pourquoi des branches ? Pouvoir se lancer dans des évolutions ambitieuses en ayant toujours la capacité de revenir à une version stable que l'on peut continuer à maintenir indépendamment. Pouvoir tester différentes implémentations d'une même fonctionnalité de manière indépendante.

1. Créer une nouvelle branche nommée « feature_x » **$ git branch feature_x**

2. Basculer vers une branche existante **$ git checkout feature_x**

Cela déplace (HEAD) le pointeur vers la branche feature_x.

Tous les commits à ce moment sont fait sur la branche courante.



3. Retourner sur la branche principale **$ git checkout master**

4. Supprimer la branche

$ git branch –d feature_x

5. Incorporation des modifications d'une branche dans la branche courante (HEAD) par fusionner (merge) $ git merge Fusion d'une autre branche avec la branche active (par exemple master).

Il est possible qu'il y ait des conflits à résoudre lors d'un merge.

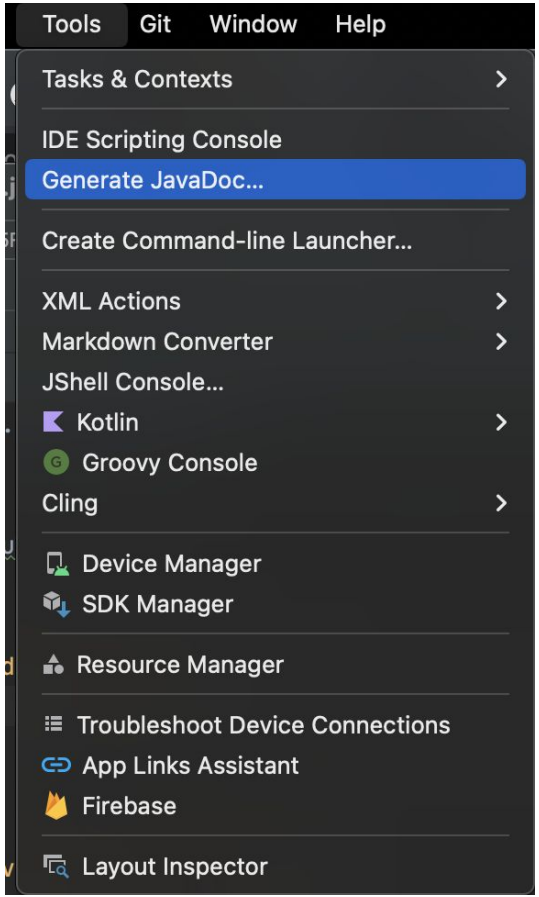
Lorsque vous modifiez différemment la même partie du même fichier dans les deux branches que vous souhaitez fusionner, Git ne sera pas capable de réaliser proprement la fusion : Aucun “commit” de fusion n’est créé et le processus est mis en pause. Vous devez alors régler ces conflits manuellement en éditant les fichiers indiqués par git: Faire git status qui donne les fichiers n’ayant pas pu être fusionnés (listés en tant que “unmerged”). Marquer les conflits comme résolus en faisant la commande git add ou git commit -a On peut, après résolution de tous les conflits, soumettre les modifications sous forme d’objet “commit” de fusion avec git commit –m « Mon premier commit» ou git push et terminer ainsi le processus de fusion.

Rappel :

- **JavaDoc**

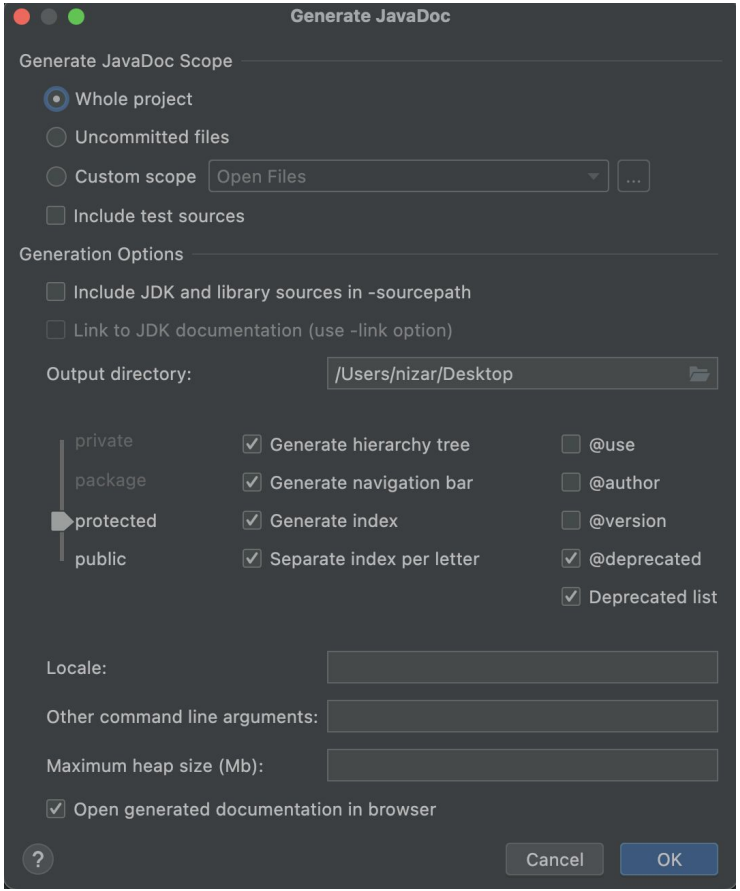
Aller à :

Tools-> Generate JavaDoc..



Sélectionner le chemin d'output

et cliquer sur Ok



C’est la fin du module

Merci !