

Programmation des composants mobiles (Android)

Wieslaw Zielonka
zielonka@irif.fr

Cours et TD 4 h par semaine pendant 6 semaines.

note finale = 50% examen + 50% projet

pour la session de rattrapage les mêmes modalités.

La note du projet est conservée pour la session 2 (il n'y aura pas de soutenances de rattrapage du projet).

IDE : AndroidStudio

C'est le seul IDE autorisé à l'examen.

Installé sur les machines de l'UFR et vous devez installer sur vos machines et ne jamais passer à une nouvelle version.

Et ne pas utiliser d'autres IDE, comme par exemple IntelliJ IDEA.

Les applications Android : java, C++, Kotlin

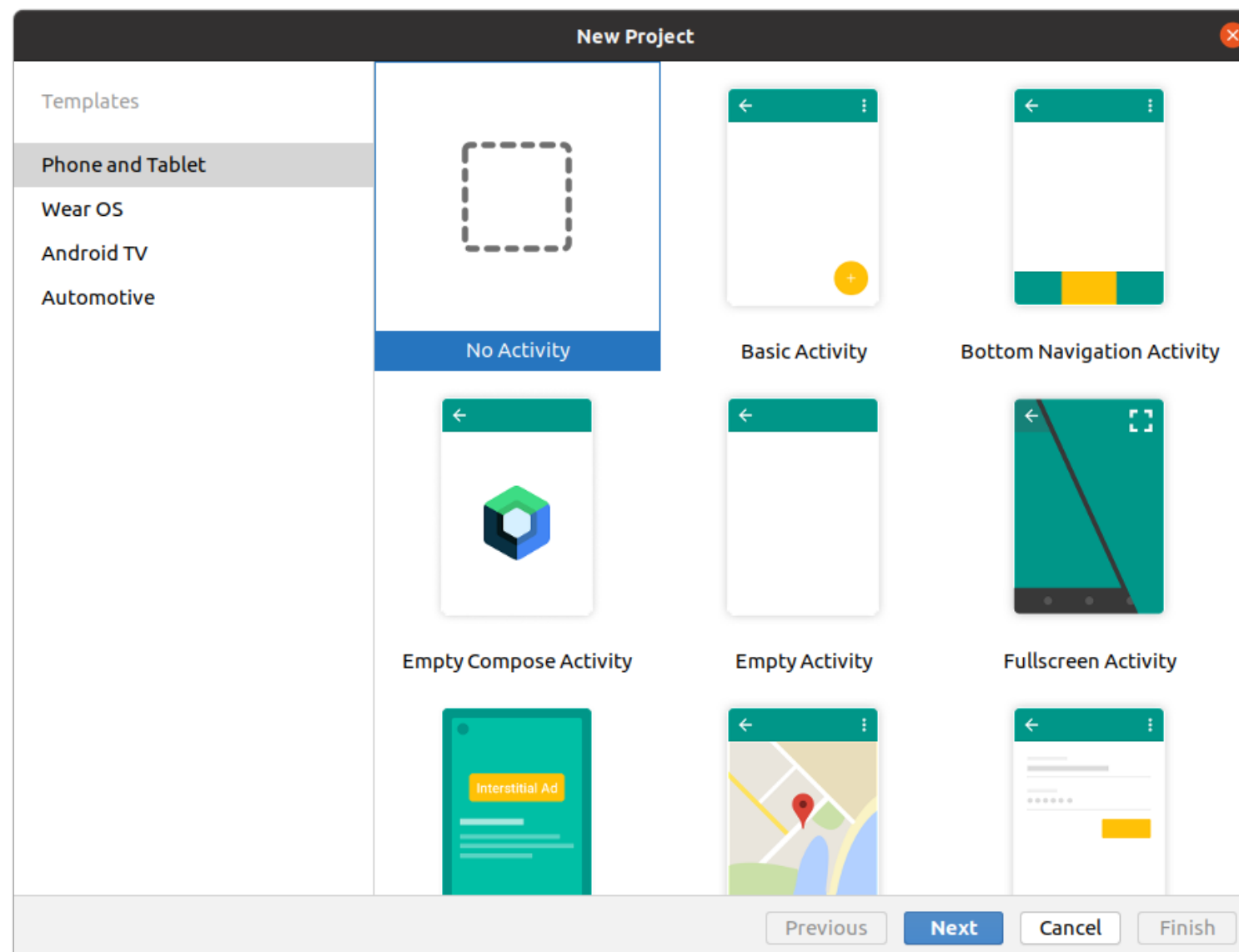
Dans ce cours j'utiliserai Kotlin.

Documentation et tutorials de kotlin :

<https://kotlinlang.org/docs/home.html>

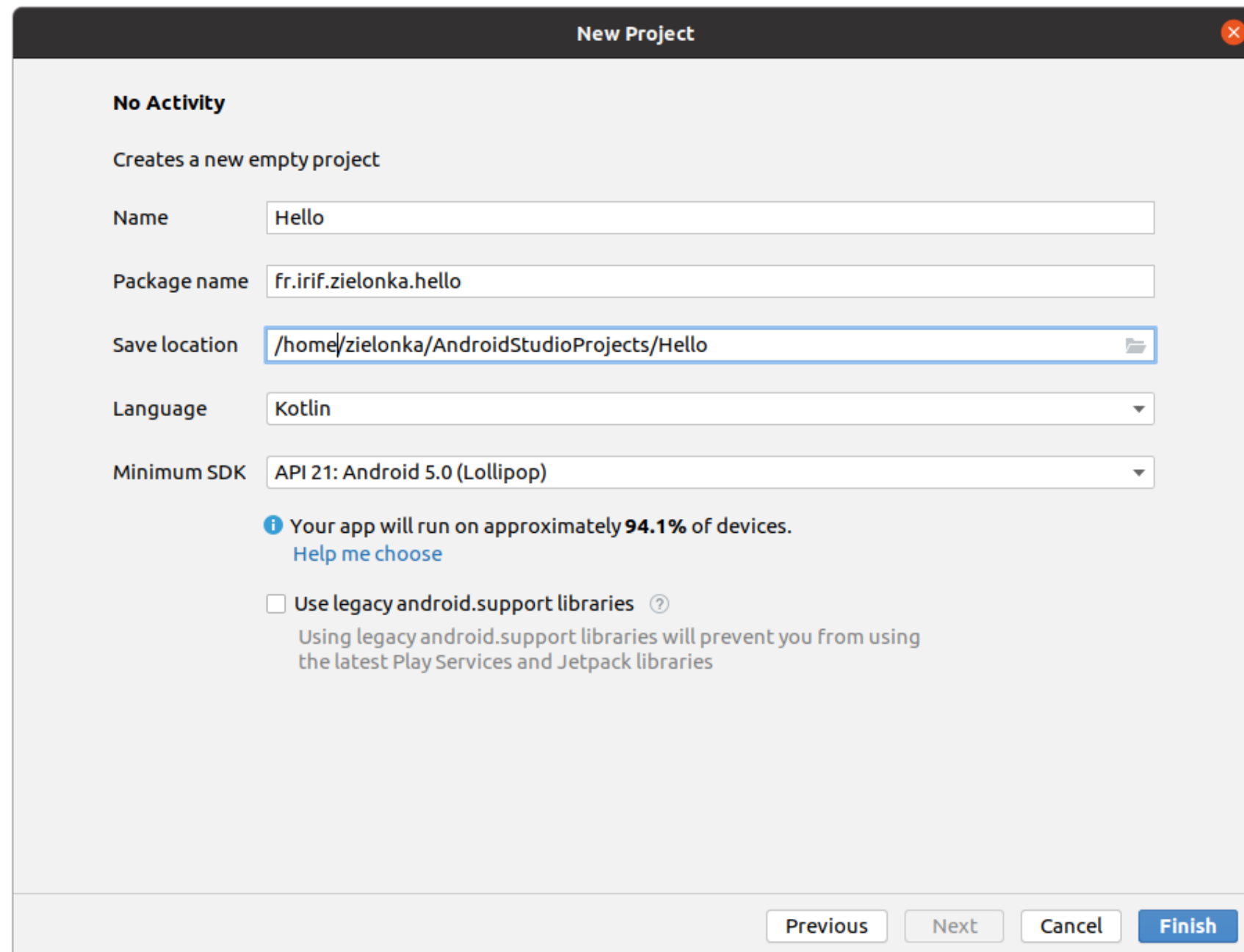
Première application kotlin

- Lancer AndroidStudio
- File -> New -> New Project
- sélectionner "No Activity" (pour une application sur le terminal)



Première application kotlin

- choisir le nom du projet (ici Hello)
- le nom de package (ici fr.irif.zielonka.hello)
- le répertoire du projet (sans doute par défaut)
- langage (Kotlin)



New Project

No Activity

Creates a new empty project

Name: Hello

Package name: fr.irif.zielonka.hello

Save location: /home/zielonka/AndroidStudioProjects/Hello

Language: Kotlin

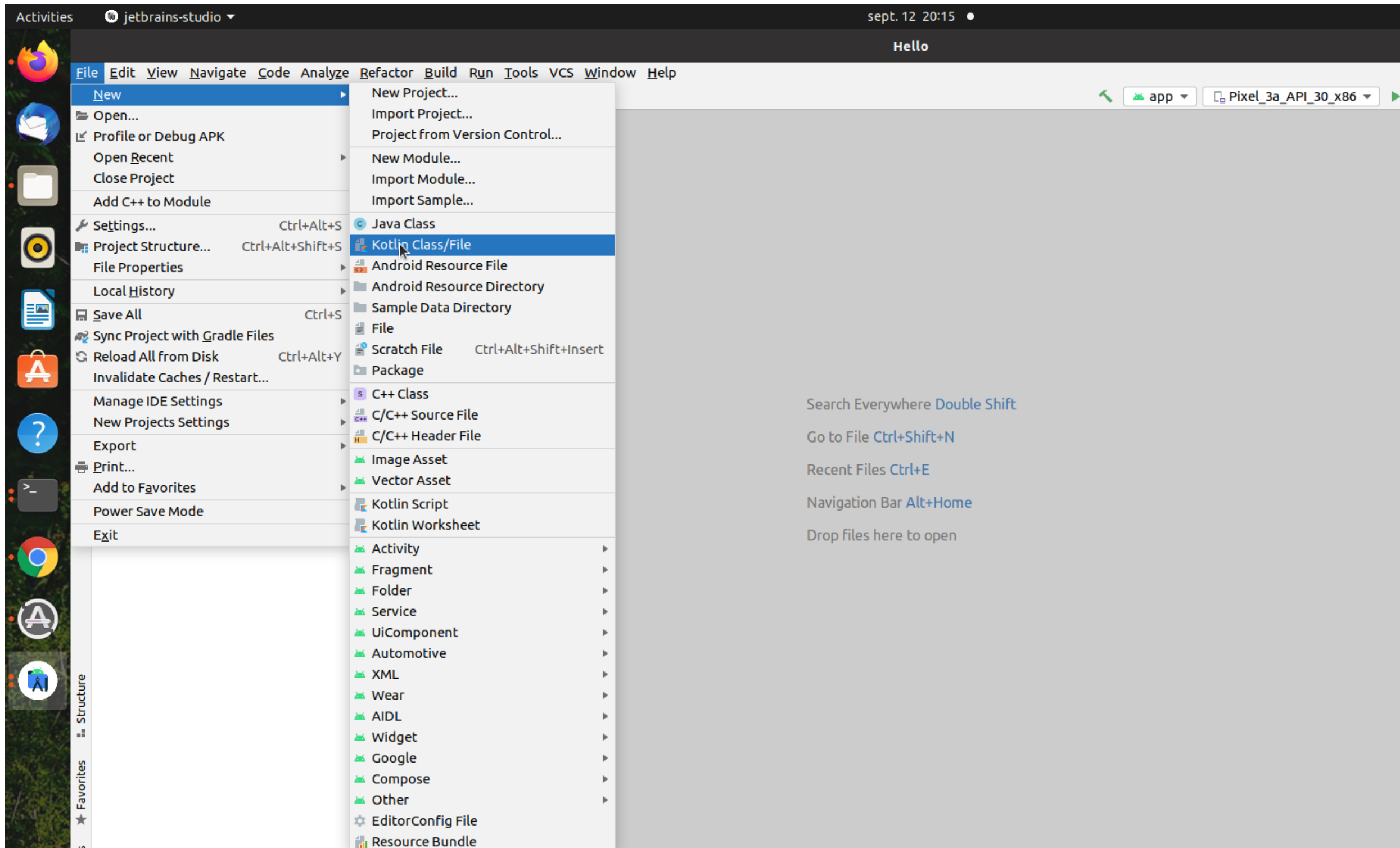
Minimum SDK: API 21: Android 5.0 (Lollipop)

i Your app will run on approximately **94.1%** of devices.
[Help me choose](#)

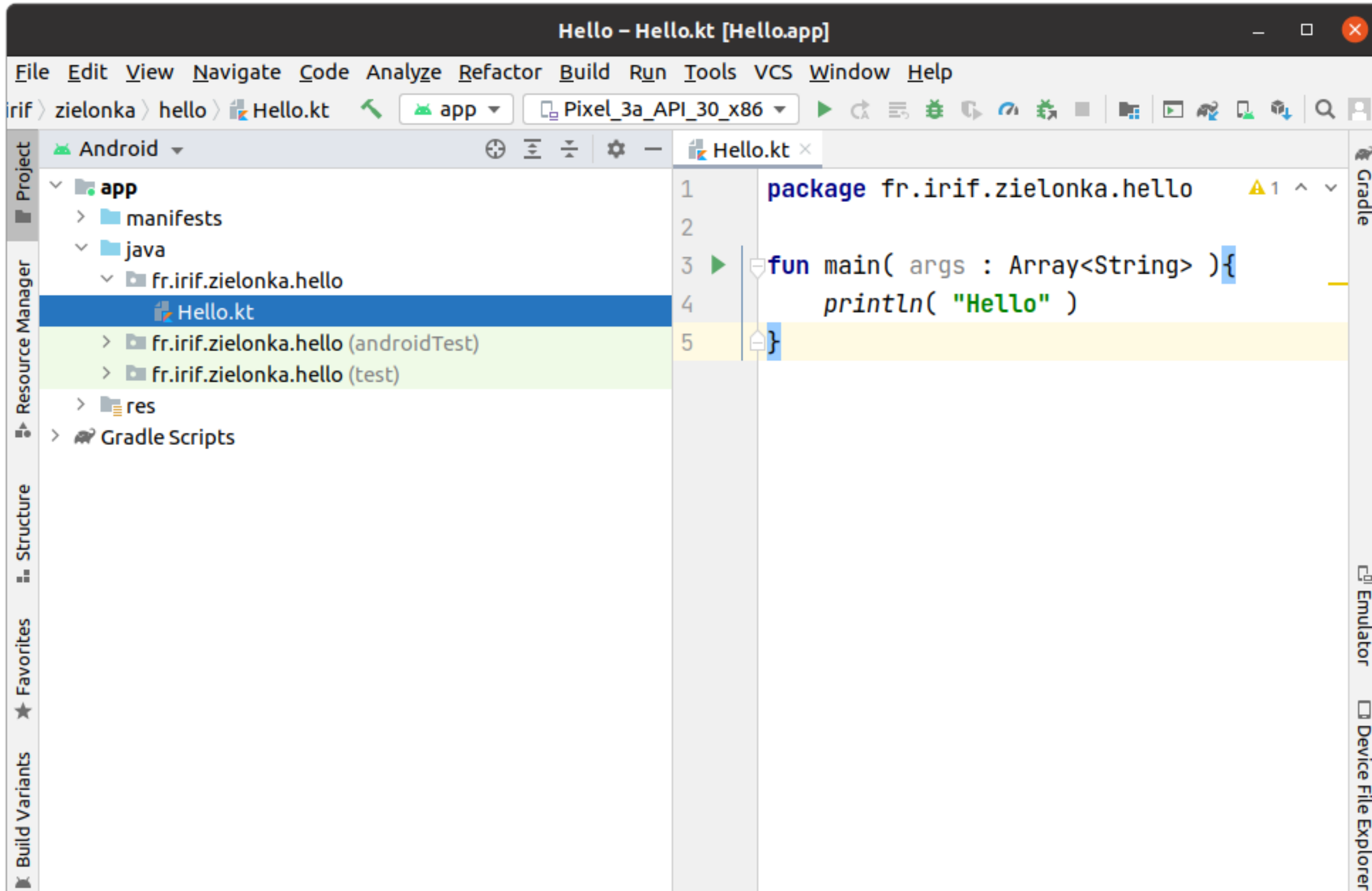
☐ Use legacy android.support libraries **?**
Using legacy android.support libraries will prevent you from using the latest Play Services and Jetpack libraries

Previous Next Cancel **Finish**

- File -> New -> Kotlin Class/File pour ajouter le fichier source
- choisir le nom du fichier (Hello.kt)



- Ecrire un programme dans le fichier Hello.kt



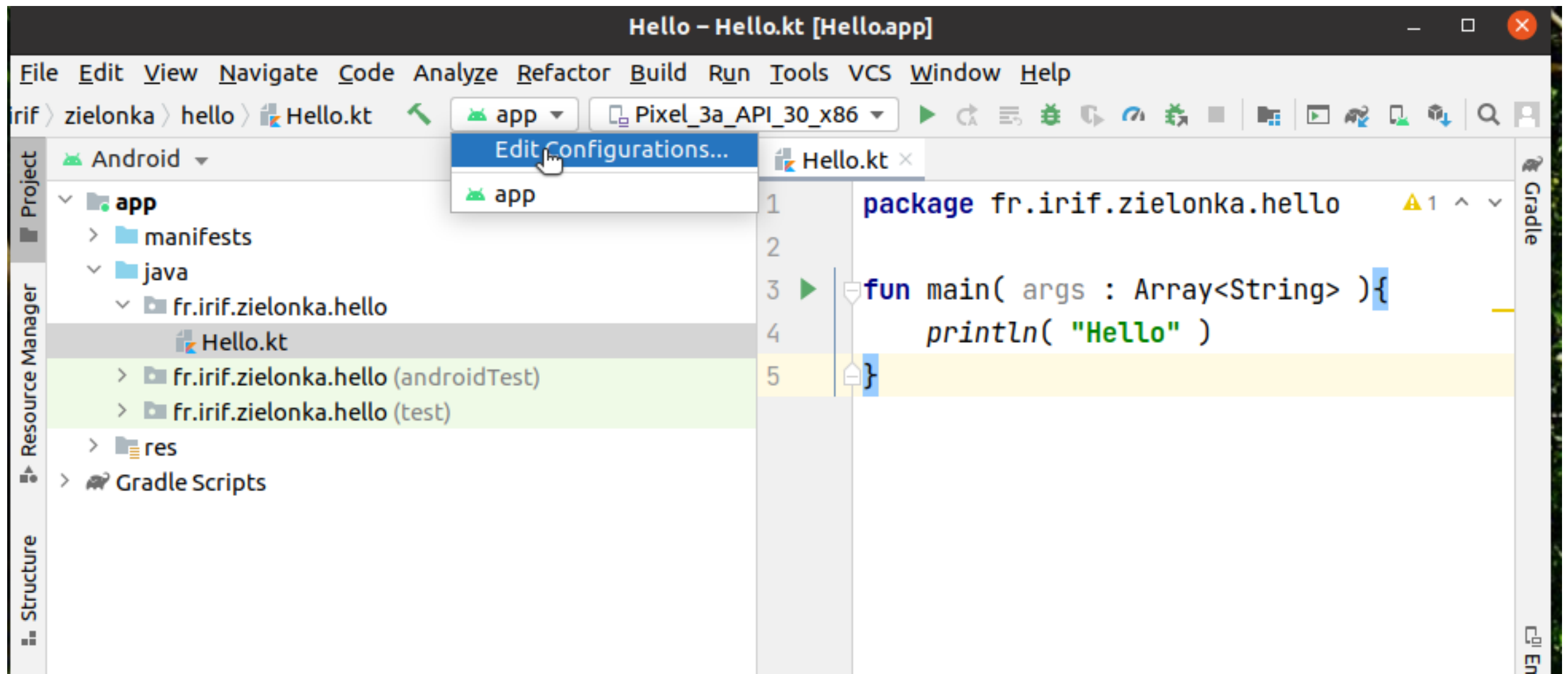
Le programme en mode terminal contient la fonction main :

```
fun main( argv : Array<String> ){
```

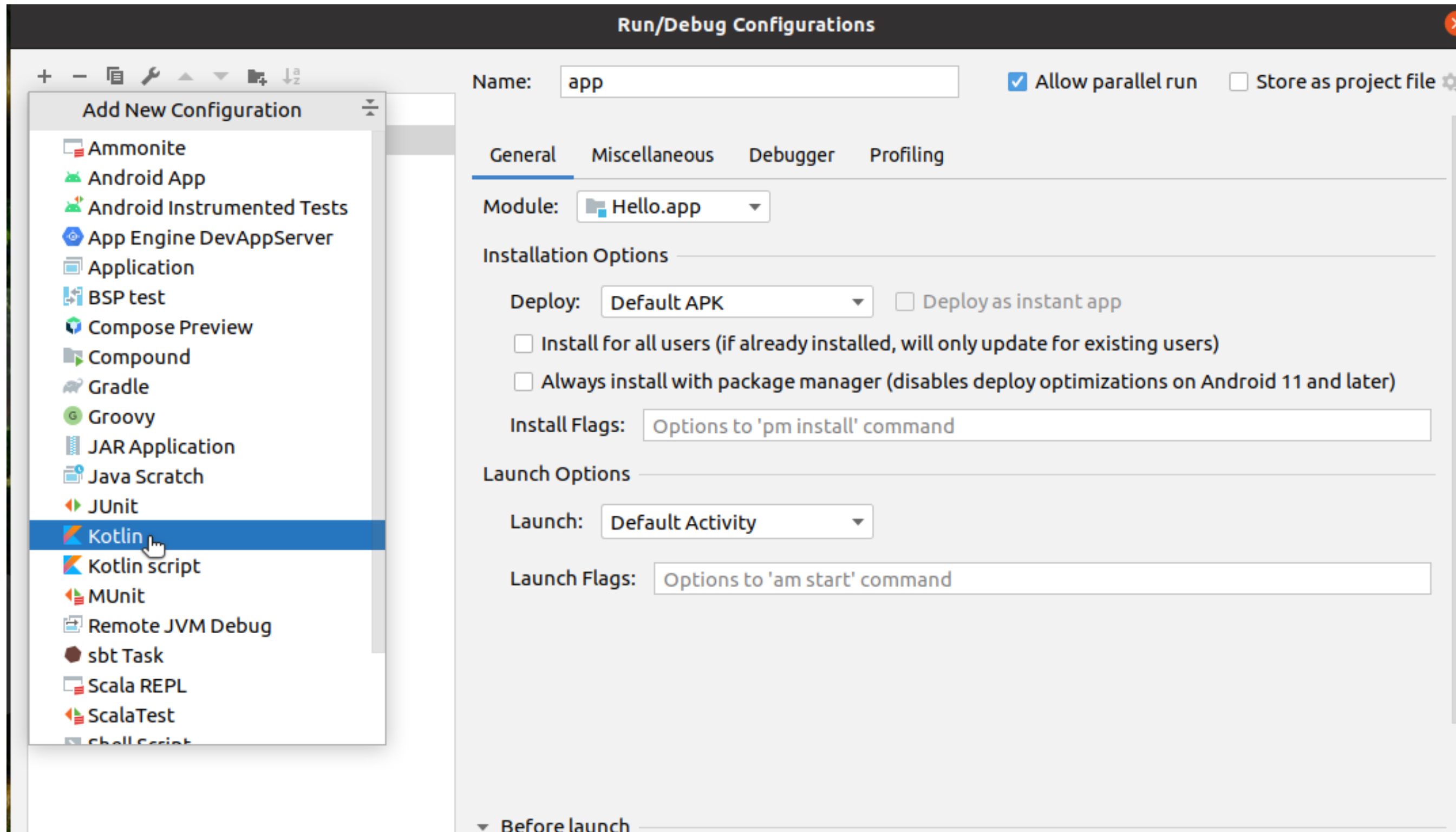
L'exécution du programme en mode terminal commence dans la fonction `main()`.

La configuration par défaut est une configuration pour Android. Il faut créer une nouvelle configuration pour une application sur le terminal. Sélectionner

app -> EditConfigurations

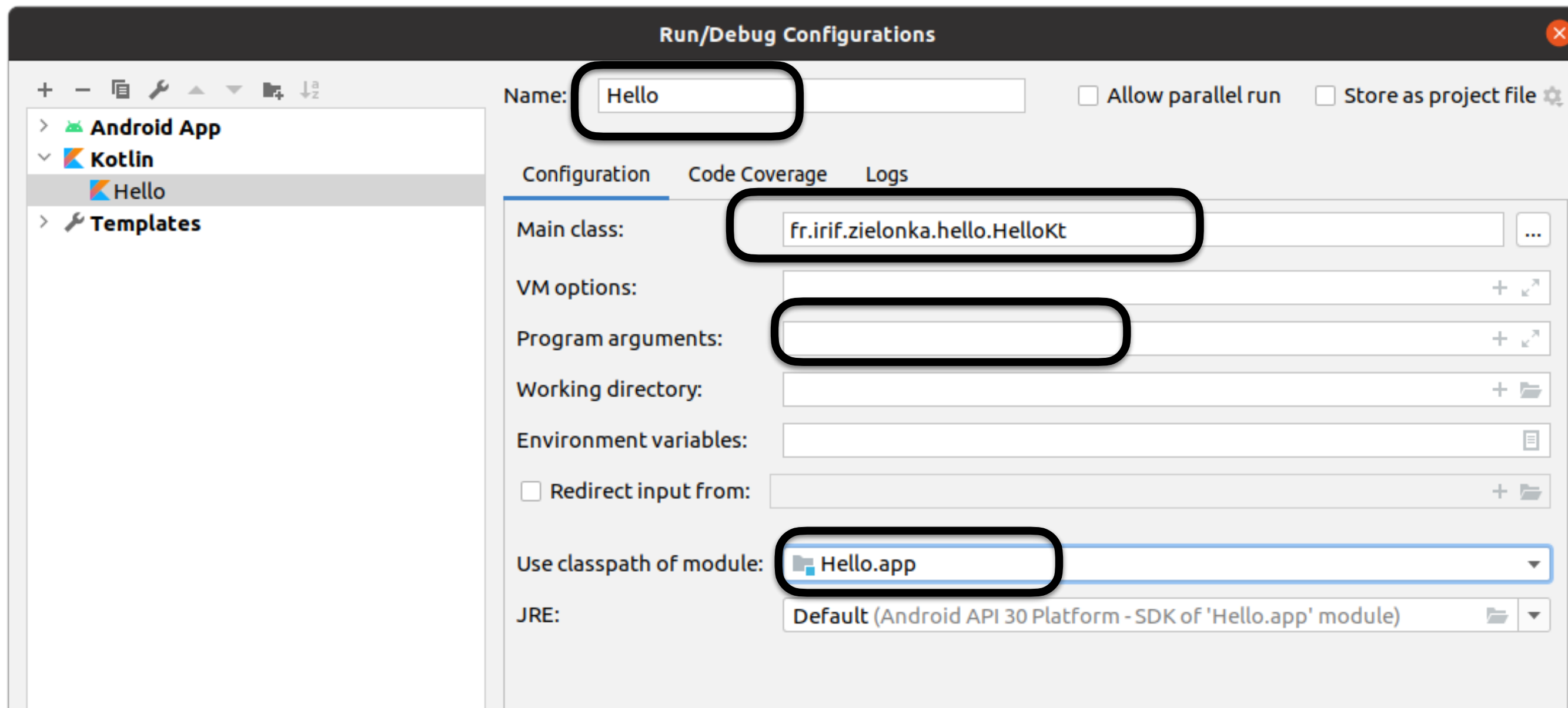


Cliquer sur + (à gauche du menu) et choisir le type de la nouvelle configuration : Kotlin



choisir

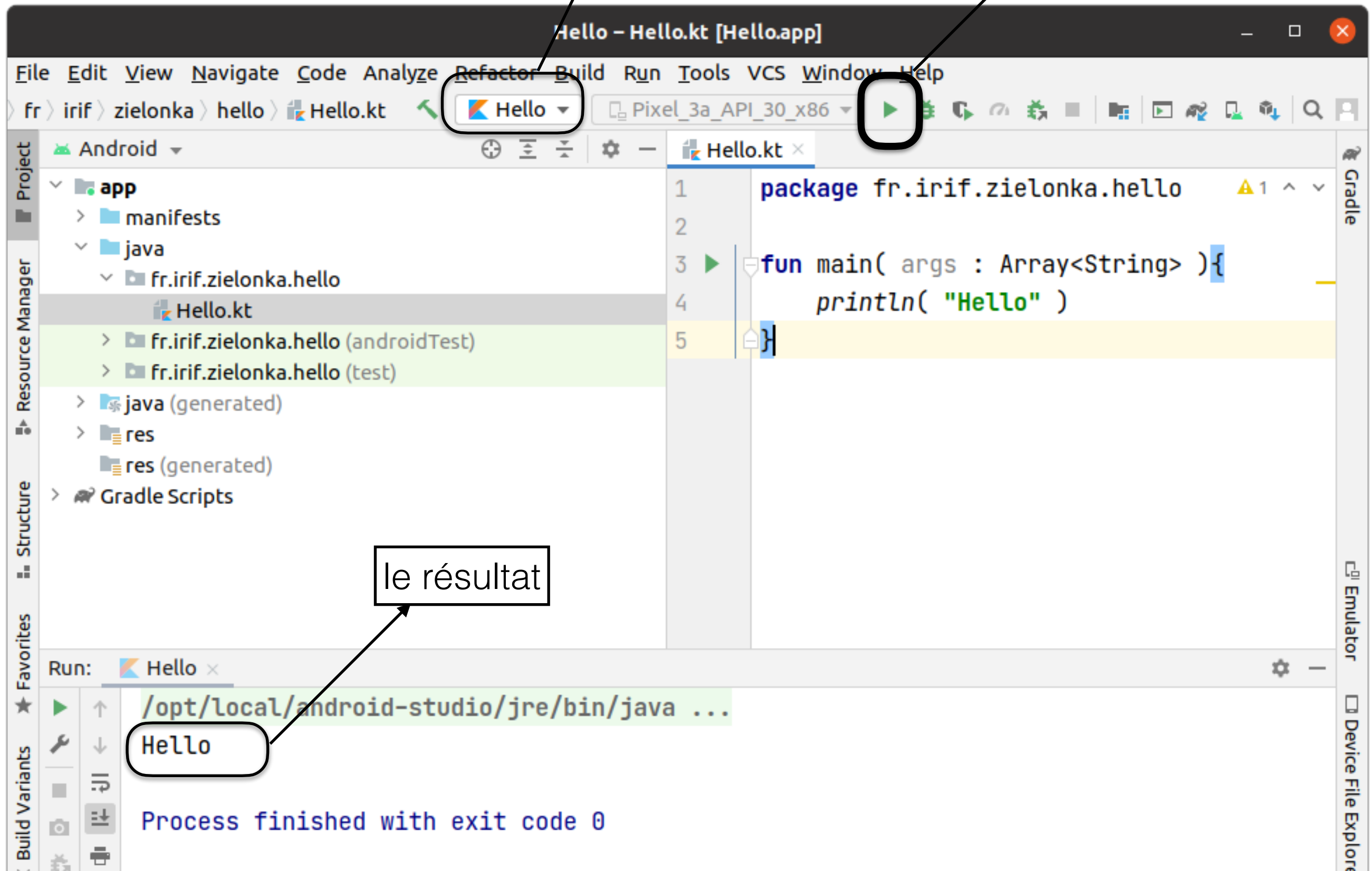
- le nom (ici Hello)
- le classpath du module (ici **Hello.app**, attention pas Hello, mais Hello.app)
- Main class : ici **fr.irif.zielonka.hello.HelloKt** . Le préfix est le package, le nom de la classe est le nom du fichier source (Hello) avec le suffixe Kt
- ajouter éventuellement les arguments du programme (arguments du main())



Compiler et exécuter le programme

dans la nouvelle configuration

compiler+exécuter



Kotlin

Les types numériques:

type	nombre de bits
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Déclarations de variables :

Les variables non-mutables :

```
val i: Int = 5
```

Une variable déclaré avec **val** est non-mutable,
sa valeur ne peut pas changer :

```
i=i+2 // erreur de compilation
```

Les variables mutables :

```
var j: Int
```

```
j = 8
```

```
j=j+5
```

Les variables mutables sont déclarées avec **var**

Pas de conversion automatique entre les types de base:

```
val k: Double = 6 // erreur de compilation
```

```
// k Double mais 6 de type Int
```

```
val k: Double = 6.0 //ici OK
```

Il existe de méthodes de conversion :

```
val x: Int = 4
```

```
val y: Double = x.toDouble()
```

Fonctions

Contrairement au java les fonctions peuvent être définies aussi bien à l'intérieure qu'à l'extérieurs de classes :

```
fun add( a: Int, b: Int): Int{  
    return a+b  
}
```

Le mot clé **fun** est obligatoire.

Si la valeur de la fonction est donnée par une expression on peut simplifier:

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

Si le type de retour peut être déduit du type de l'expression alors le type de retour peut être omis :

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

Il est possible de définir une fonction à l'intérieure d'une autre fonction.

Fonctions

```
fun g( a : Int, b: Int = 0, c: Int = 0, d: Int = 1): Double {  
    return (a+b+c)/d.toDouble()  
}
```

La fonction g possède 4 arguments de type Int. Les arguments b,c,d possèdent les valeurs par défaut.

```
val n1 = g(1,2,3,4)  
val n2 = g(1, d=6) // calcule (1+0+0)/6.0
```

les arguments nommés peuvent être utilisés dans n'importe quel ordre :

```
val n3 = g(1, d=8, b=5) //calcule (1+5+0)/8.0
```

La possibilité de spécifier les arguments par défaut permet dans la plupart de cas d'éviter la surcharge de fonctions.

Les types nullables

Pour chaque type T il existe le type T? qui contient toutes les valeurs de T plus la valeur null.

Cela permet d'éviter `NullPointerException` parce que le compilateur ne permet plus de compiler le code qui permettrait d'affecter la valeur null à une variable de type T.

```
var x: Int = 3
x = null           // erreur de compilation
```

```
var x: Int? = 3    //x peut prendre les valeurs Int ou null
x = null           //OK
```

Int est un sous-type de Int? (mais pas l'inverse)

```
val x: Int = 3
val y: Int? = x     //OK
```

```
val x: Int? = 4
val y: Int = x      //erreur de compilation parce que x peut être
                    //potentiellement null
```

```
val s: String? = f() //f() retourne un String ou null
val l = s.length     // ne compile pas, s peut être null et
                    // dans ce cas .length ne s'applique pas
```

Les types nullable opérateurs ?. !!. ?:

Opérateur ?.

```
val s: String? = F() // F peut retourner String ou null
val l = s.length     //ne compile pas, s peut être null
val n = s?.length    //compile et s?.length est soit la longueur de s
                      // si s est un String, soit null si s est null. Le type de n (déduit
                      // par le compilateur) est Int?
```

Possibilité d'enchaîner les opérateurs ?.

```
val ville1 = personnel[i]?.adresse?.city
// ville1 sera null si un d'éléments (personnel[i] ou adresse ou city) est null
```

Opérateur !!.

```
val ville2 = personnel[i]!!.adresse!!.city
//si personnel[i] ou adresse null alors NullPointerException, par contre city peut être null
```

Opérateur elvis ?:

```
var s = savedInstanceState?.getInt("key") ?: 0
```

Si la valeur de savedInstanceState?.getInt("key") est un Int alors s prendra cette valeur, si la valeur de cette expression est null alors s prendra la valeur 0

boucles

```
for( i in 0..10 ){ } //i=0,1,...,9,10
for( k in 0..1 ){ }
for( k in 0..1 step 2){ } //k == 0,2,4,...
for( k in 9 downTo 0 step 3 ){ }
for( k in 1 until 100 ){ } //1 <= k < 100
```

```
while() {}
```

```
do{ } while()
```

comme en java

if else

if else est une expression donc possède une valeur qu'on pourra utiliser dans d'autre expressions ou dans l'affectation :

```
val z = if( j > 0 )  
        1  
      else if( j < 0 )  
        -1  
      else 0
```

z prendra une de trois valeurs: -1 si j < 0, 0 si j == 0 et 1 si j > 0

plus besoin d'expression ternaire

break et continue (avec étiquette)

```
loop@  
for(i in 1..100)  
  for( j in 1..100 )  
    if(....) break @loop
```

break permet de sortir d'une boucle. Break avec une étiquette permet de sortir de boucles imbriquées.

classes

```
class Personne( val name: String, val firstName: String)
```

La classe Personne est dérivée de la classe **Any** (la classe à la racine de la hiérarchie de classes). Personne possède deux propriétés non-mutables : name et firstName. (Any remplace Object de java) .

val devant le noms de paramètres indique que les propriétés name et firstName ne peuvent pas être modifiées une fois l'objet Personne construit.

Cette définition de Personne est équivalente à

```
class Person( name: String, firstName: String ){  
    val name: String = name  
    val firstName = firstName  
}
```

Pour créer un objet Personne :

```
val pers = Personne( "Durand", "Thomas" )
```

Il n'y a pas de new. On appelle un constructeur comme une fonction.

Pour accéder à une propriété :

```
val n: String = pers.name
```

classes

Comme en java, la classe `Personne` hérite la fonction `toString`, et nous pouvons la redéfinir :

```
class Personne( val name: String, val firstName: String){  
    override fun toString(): String = "${name}, ${firstName}"  
}
```

Pour redéfinir une fonction il faut précéder la nouvelle fonction par `override` .
Dans une chaîne de caractères

`${expression}`

est remplacé par la valeur de l'expression.

Maintenant nous pouvons afficher l'objet de la classe `Personne` :

```
val p = Personne("Dupont", "Carole")  
println( p )    //équivalent à println( p.toString() )
```

La classe `Personne` ainsi définie est finale par défaut (impossible de définir une classe dérivée).

classes dérivées

pour pouvoir définir des classes dérivées de `Personne`
il faut que `Personne` soit **open**

↓
open `class` `Personne`(`val` `name`: `String`, `val` `firstName`: `String`){
 `override fun toString(): String` = `"${name}, ${firstName}"`
}

Employe dérivée de `Personne` :

```
class Employe( name: String, firstName: String,  
                val dateEmbauche: Calendar = Calendar.getInstance()):  
    Personne(name, firstName){  
        //redefinir toString()  
        override fun toString(): String {  
            return "${super.toString()} " +  
                    "${dateEmbauche.get(Calendar.YEAR)}/" +  
                    "${dateEmbauche.get(Calendar.MONTH)}/" +  
                    "${dateEmbauche.get(Calendar.DAY_OF_MONTH)}"  
        }  
    }
```

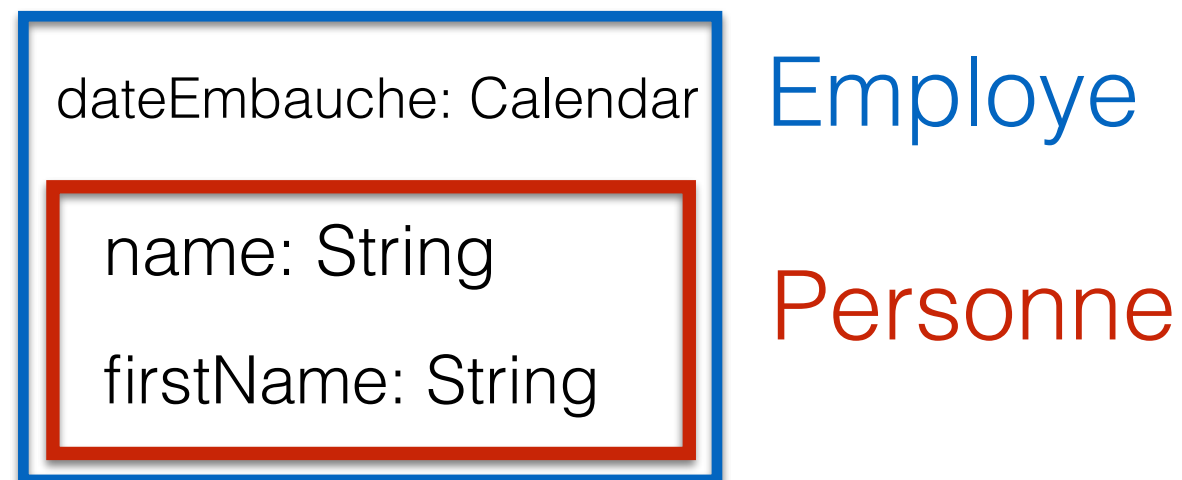
ou

open `class` `Employe`(...): `Personne`(...) si nous ne voulons pas que
`Employe` soit finale.

classes dérivée

```
class Employe( name: String, firstName: String,  
              val dateEmbauche: Calendar = Calendar.getInstance()):  
    Personne(name, firstName){  
  
    override fun toString(): String { ..... }  
}
```

Le constructeur Employé possède 3 paramètres : `name` et `firstName` servent à initialiser les attributs de même nom de la classe mère. **Il n'y a pas de `val` devant les noms de paramètres `name` et `firstName`.**



`dateEmbauche: Calendar` -- une nouvelle propriété de type `Calendar` avec la valeur par défaut `Calendar.getInstance()`. La classe `Calendar` c'est la classe de `java.util.Calendar`, il faut un `import`.

classes dérivée

Créer des employés:

```
import java.util.Calendar

val date = Calendar.getInstance()
date.set( 2020, 7, 22)
val employe3 = Employe3("Durand", "Maxime", date)

val a = Employe( "Lebon", "Pierre", date)
val b = Employe( "Bush", "John")
```

classes dérivée

```
class Employe( name: String, firstName: String,  
              val dateEmbauche: Calendar = Calendar.getInstance()):  
    Personne(name, firstName){  
  
    override fun toString(): String { ..... }  
}
```

Classe équivalente en java :

```
class Employe extends Personne{  
  
    Calendar dateEmbauche; //nouvelle propriété  
  
    Employe( String name, String firstName, Calendar dateEmbauche){  
        super(name, firstName);  
        this.dateEmbauche = dateEmbauche;  
    }  
    Employe( String name, String firstName ){  
        super(name, firstName);  
        this.dateEmbauche = Calendar.getInstance()  
    }  
    public String toString(){ ..... }  
}
```

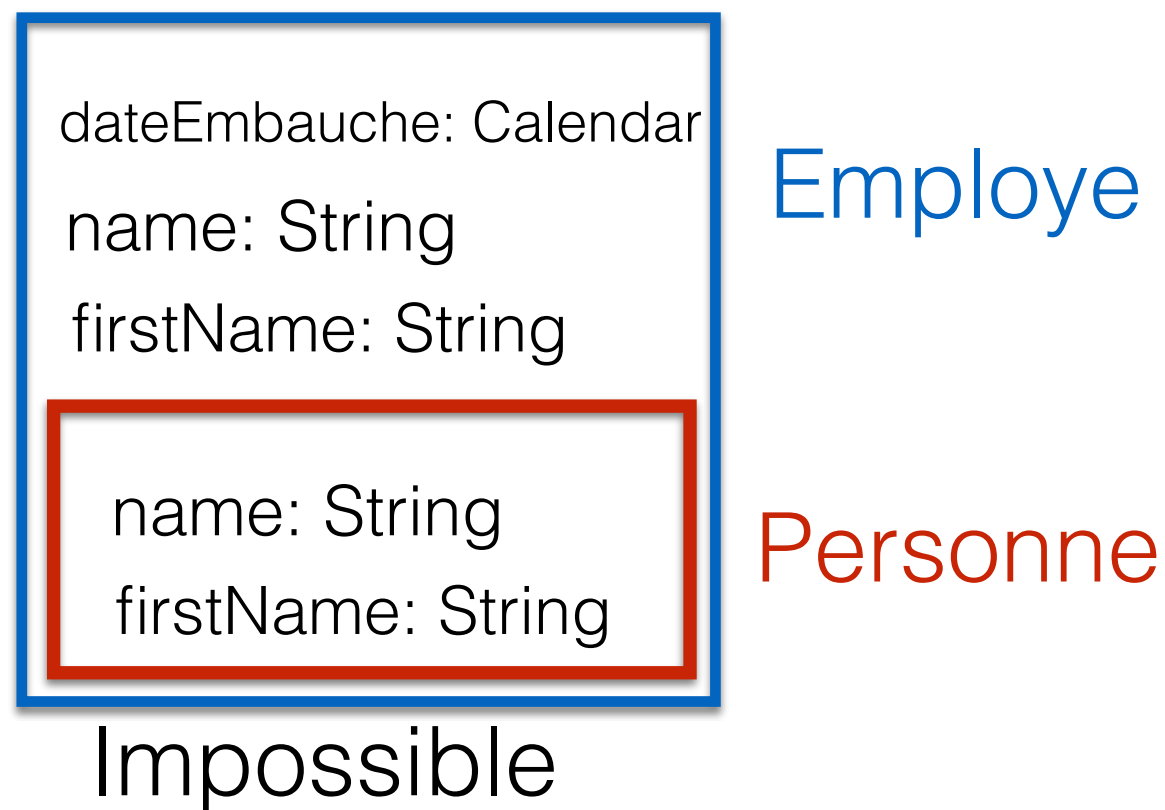
Un constructeur Kotlin avec un paramètre avec une valeur par défaut permet de remplacer deux constructeur java.

classes dérivée

Ajouter val devant les deux premiers paramètres :

```
classe Employe( val name: String, val firstName: String, val dateEmbauche: Calendar .....):  
    Personne( name, firstName)
```

donnerait une classe Employe avec les propriétés name et firstName en double, une fois dans la classe mère Personne et deuxième fois dans la classe Employe



classes dérivée

En générale

```
class B(paramètres_B): A(paramètres_A)
```

définit une classe B dérivée de la classe A. La partie

```
A(paramètres_A)
```

dit que B est dérivé de A mais en plus cela remplace

```
super(paramètres_A)
```

dans le constructeur de la classe B en java.

programme kotlin complet en mode mode terminal

```
open class Personne( val name: String, val firstName: String){
    override fun toString(): String = "${name}, ${firstName}"
}

class Employe( name: String, firstName: String, val dateEmbauche:
Calendar = Calendar.getInstance()):
    Personne3(name, firstName){
        override fun toString(): String {
            return "${super.toString()} "
                "${dateEmbauche.get(Calendar.YEAR)}/" +
                "${dateEmbauche.get(Calendar.MONTH)}/" +
                "${dateEmbauche.get(Calendar.DAY_OF_MONTH)} "
        }
    }
}

fun main{
    val p3 = Personne(firstName= "Janusz", name = "Antoni")
    val nom = p3.name
    val date = Calendar.getInstance()
    date.set( 2020, 7, 22)
    val employe3 = Employe("Durand", "Maxime", date)
    println( p3, employe3 )
}
```

Function types

Il est possible d'avoir les variables de type fonction (qui contiennent une référence vers une fonction) et il est possible de passer une fonction comme paramètre d'une autre fonction.

```
fun appl( a: List<Int>, b: List<Int>,
         f: (Int,Int)-> Int ) : MutableList<Int> {

    var r = mutableListOf<Int>()

    val s = if (a.size < b.size) a.size else b.size

    for( i in 0..s-1 )
        r.add( f( a[i], b[i] ) )

    return r
}
```

`appli()` prend comme paramètres deux listes de `Int` et le troisième paramètre
`f: (Int,Int)-> Int`

est une fonction qui prend deux `Int` et retourne un `Int`. La fonction `appli()` construit une nouvelle liste obtenue en appliquant `f` aux couples correspondants de listes `a` et `b`.

Fonctions comme paramètres, lambda-expressions

```
fun fu ( i: Int, j: Int) : Int { return i*i + j*j }
```

```
val q = listOf( 2, 6, 8 , 99)
val p = listOf<Int>( -1, 7, 90, 14)
```

//passer une fonction en paramètre

```
val res = appl(p, q, ::fu ) //le nom de fonction précédé par :: ou par NomClasse:: si la fonction
                          //définie dans une classe
```

//passer une lambda expression en paramètre

```
val res2 = appl( p, q , {a , b -> a*b })
```

{ a, b -> a * b }

est une lambda expression qui désigne une fonction sans nom avec les arguments *a*, *b*. Le compilateur déduit le type de paramètres de la définition de `appl ()`

Lambda expression avec le type de paramètres :

```
{a : Int ,   b : Int ->  val x = a-b
                        x * x
}
```

La valeur de lambda expression est la dernière valeur calculée dans le corps de la fonction.

Fonctions comme paramètres, lambda-expressions

```
val q = listOf( 2, 6, 8 , 99)
val p = listOf<Int>( -1, 7, 90, 14)

val res2 = appl( p, q ) {a , b -> (a-b)*(a-b) }
```

Si lambda expression est le dernier argument d'une fonction alors on peut la mettre en dehors de parenthèses.