

# Kotlin

Les bases et un peu +



# Menu

<b>Introduction</b> Kotlin, c'est quoi, pourquoi, pour qui ?	<b>Variables</b> Types, mutables/immutables, lateinit, constantes, nullables	<b>Structures conditionnelles</b> Boucles, conditions, opérateurs ternaires	<b>Fonctions</b> Paramètres, retours, fonctions d'extension	<b>Null safety</b> Où la fin du null pointer exception	<b>Fonctions d'extension</b> Customisez des classes existantes
<b>Lambdas et Higher-Order functions</b> Fonctions anonymes et fonctions d'ordre supérieur	<b>Scope functions</b> Accéder à ses objets avec let, run, with, apply et also	<b>Classes</b> Créer des modèles en une ligne, c'est possible!	<b>Interfaces</b> Implémentez vos interfaces	<b>Any, Unit et Nothing</b> Ces classes particulières...	<b>Collections</b> List, Map, Set, etc...
<b>Opérations sur les collections</b> Extensions et opérations possibles sur les collections	<b>Cast et alias de type</b> Castez proprement et retrouvez plus facilement vos types grâce à Kotlin	<b>Propriétés déléguées</b> Delegated Properties	<b>Déclaration de déstructuration</b> Destructuring declarations	<b>Operator overloading</b> Implémentez vos propres opérateurs	<b>Réflexion</b> Introspecter la structure de votre propre programme au moment de l'exécution
<b>Coroutines</b> Exécuter du code asynchrone allégé	<b>Interopérabilité Java/Kotlin</b> Kotlin a été conçu avec pour base l'interopérabilité Java	<b>Kotlin et Android</b> Jetbrain a aussi pensé aux développeurs Android			



# Introduction

Kotlin, c'est quoi, pourquoi, pour qui ?



# Historique

- **2010**
  - début du développement du langage
- **2011**
  - JetBrains dévoile le projet Kotlin, un nouveau langage pour la JVM
- **2012**
  - open source du projet sous la licence Apache 2
- **2016**
  - *Kotlin v1.0*: première version officiellement stable
- **2017**
  - lors de la Google/IO, il est annoncé le support officiel de Kotlin sur Android.
  - *Kotlin v1.2*: le partage de code entre les plates-formes JVM et Javascript a été récemment ajouté à cette version.
- **2018**
  - *Kotlin v1.3*: apportant des coroutines pour la programmation asynchrone.
- **2019**
  - Google annonce que le langage de programmation Kotlin est désormais son langage préféré pour les développeurs d'applications Android (Kotlin first).
- **2020**
  - *Kotlin v.1.4*: temps de build réduit de plus moitié



# Kotlin/JS

- En cours de développement, **Kotlin for JS** permet de générer du code JS à partir de Kotlin.
- Cette transpilation de code permet de cibler la machine virtuelle Javascript côté client et serveur.



# Kotlin Native

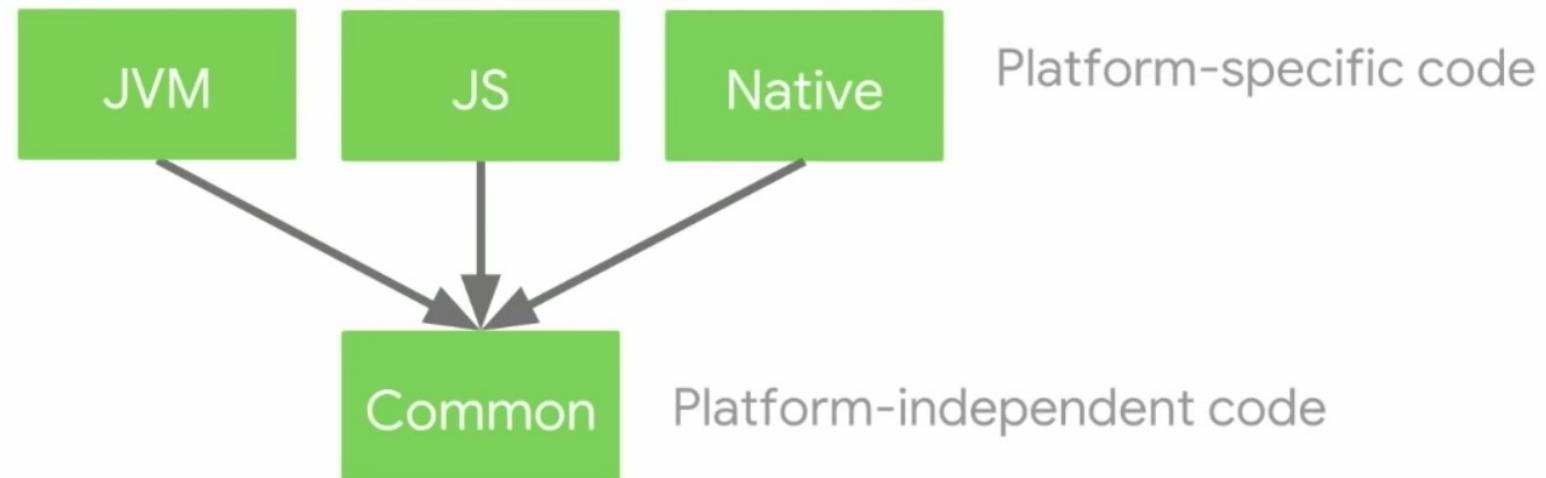
- Un développement est en cours (en beta) pour générer du code natif à partir de Kotlin (Kotlin/Native).
  - Pour le moment Kotlin/Native supporte les plateformes suivantes :
    - Windows (x86\_64)
    - Linux (x86\_64, arm32, MIPS, MIPS little endian)
    - MacOS (x86\_64)
    - iOS (arm64 only)
    - Android (arm32 and arm64)
    - WebAssembly (wasm32 only)
  - + d'infos : <https://github.com/JetBrains/kotlin-native>



# Kotlin multiplatform

Travailler sur toutes les plateformes est l'objectif de Kotlin, c'est aussi la prémissse d'un objectif beaucoup plus important: le partage de code entre les plateformes. Avec la prise en charge de JVM, Android, JavaScript, iOS, Linux, Windows, Mac et même des systèmes embarqués comme STM32, Kotlin peut gérer tous les composants d'une application moderne.

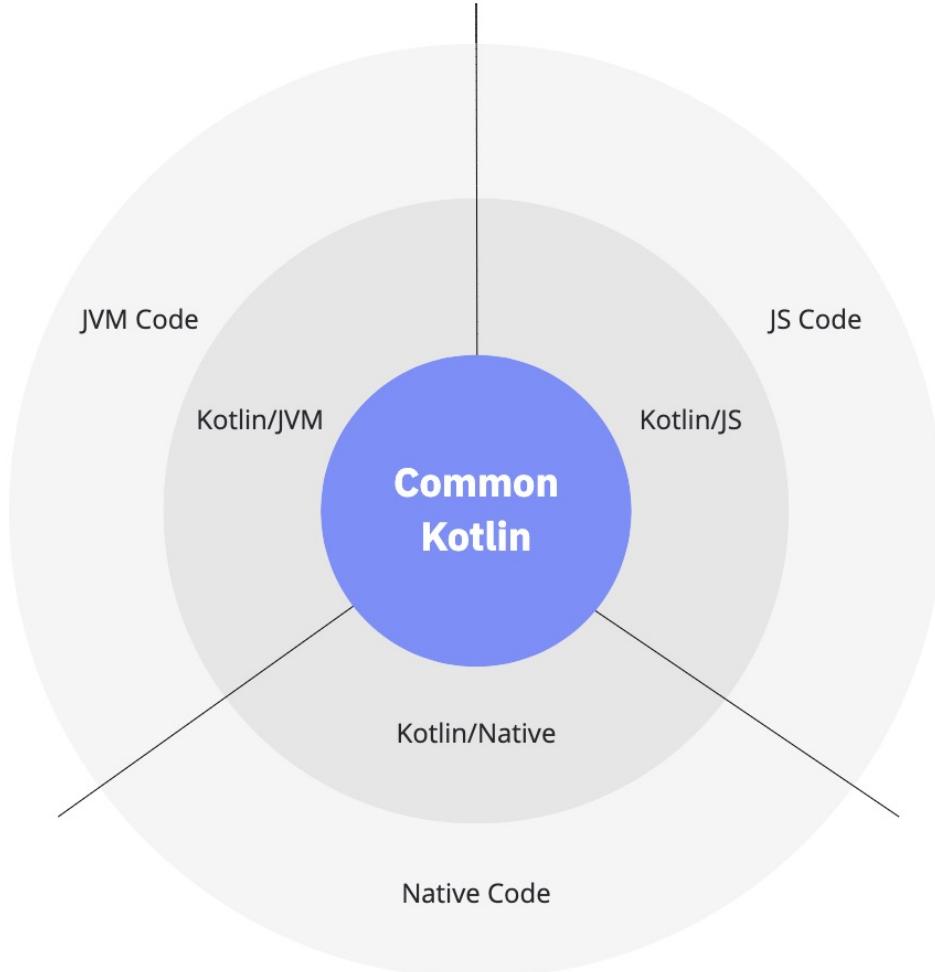
Cela apporte l'avantage inestimable de la réutilisation pour le code, évitant ainsi de recoder la même chose plusieurs fois.



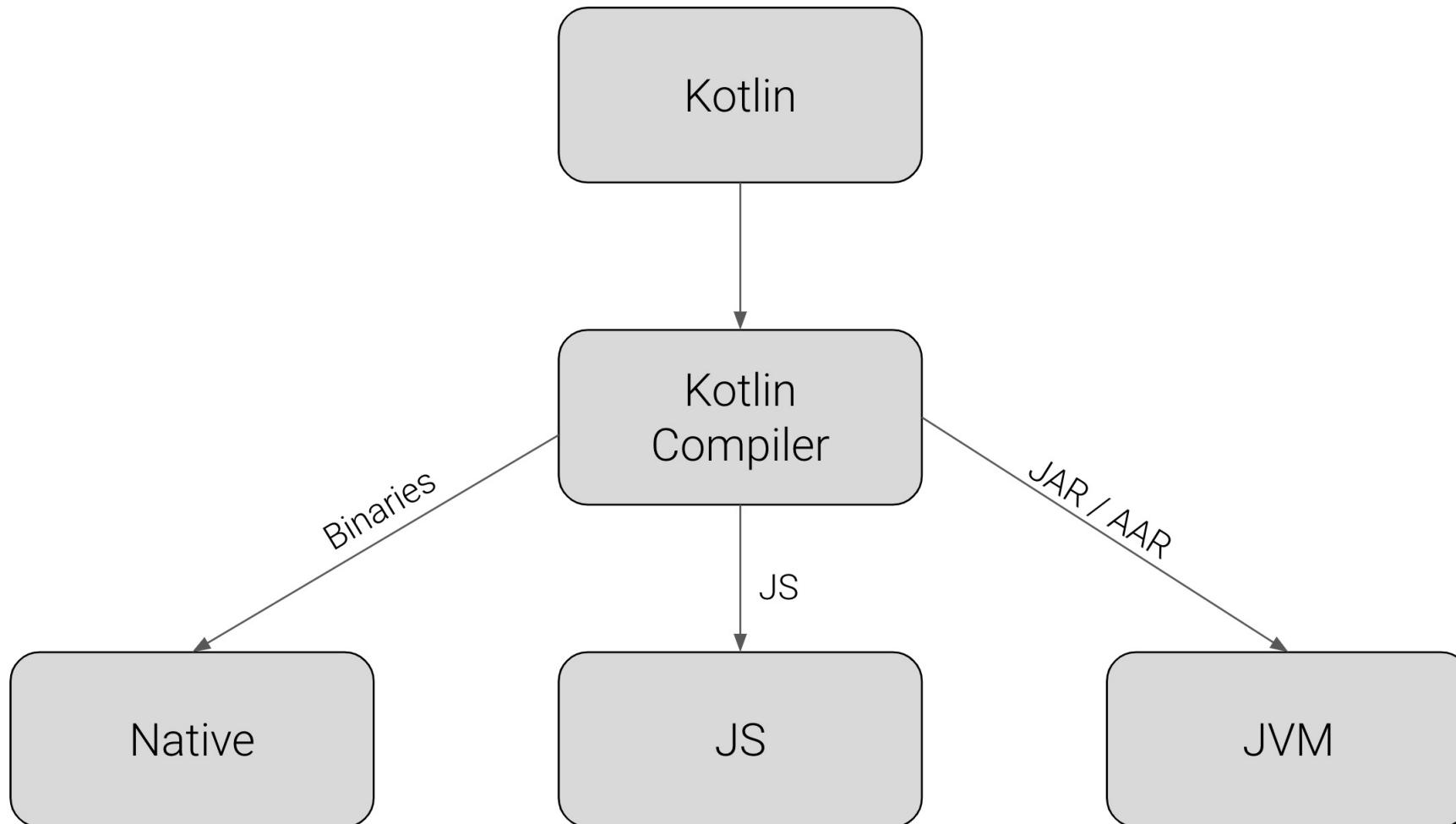
Exemple avec un jeu de Sudoku pour Android/Web ->



# Solutions Kotlin



# Kotlin Compiler



# Avantages de Kotlin

- **Open Source**
  - Distribué sous licence Apache 2.0
- **Interopérable**
  - 100% interopérable avec Java
- **Tool-Friendly**
  - Développé par JetBrains, une compagnie réputée pour ses outils de développement
- **Concis**
  - Une réduction des lignes de code d'environ 40% par rapport à Java
- **Simple**
  - Simple à apprendre, d'autant plus si vous connaissez Java
- **Safe**
  - Évite les erreurs de type *null exceptions* bien connues en Java



# Kotlin features

- Null safety
- No checked exceptions
- Extension functions
- Higher-order functions
- Function types & lambdas
- Default & named arguments
- Properties
- Type inference
- Operator overloading
- Smart casts
- Data classes
- Immutable collections
- Enhanced switch-case
- String interpolation
- Ranges
- Inline functions
- Infix notation
- Tail recursion
- Coroutines (async/await)
- Great Standard Library

Et d'autres...



# Envie d'apprendre ? De s'informer ?

- Google Code Labs propose des tutos pour apprendre à coder en Kotlin avec Android :
  - À voir sur <https://codelabs.developers.google.com/> comme par exemple :
    - Taking Advantage of Kotlin
    - Build Your First Android App in Kotlin
    - Building Beautiful Apps Faster with Material Components on Android (Kotlin)
    - Notification Channels and Badges (Kotlin)
- Vous pouvez faire des exercices fournis par l'équipe Kotlin :
  - Grâce à un plugin pour Android Studio :
    - [kotlinlang.org/docs/tutorials/edu-tools-learner.html](https://kotlinlang.org/docs/tutorials/edu-tools-learner.html)
  - En ligne :
    - [try.kotlinlang.org](https://try.kotlinlang.org)
- Actualité
  - Kotlin blog : <https://blog.jetbrains.com/kotlin/>



# KotlinConf

- Conférence annuelle mettant en avant les nouveautés liées à Kotlin
- Les conférences sont rediffusées sur YouTube
- + d'infos : [www.kotlinconf.com](http://www.kotlinconf.com)



# Quelques raccourcis clavier pour IntelliJ

Keyboard Shortcut	Windows	Mac
Inspect expression type	Alt + =	Ctrl + Shift + P
Compile modified files	Ctrl + F9	Cmd + F9
Run application	Shift + F10	Ctrl + R
Debug	Shift + F9	Cmd + D
Debugging: Step over	F8	F8
Debugging: Step into	F7	F7
Debugging: Step out	Shift + F8	Shift + F8
Debugging: Resume	F9	Cmd + Alt + F
Quick doc	Ctrl + Q	Ctrl + Shift + P



# Conventions de codage

- Disponible ici :
  - <https://kotlinlang.org/docs/reference/coding-conventions.html>
- Pour configurer le formateur IntelliJ selon le guide style Kotlin officiel, allez dans **Settings | Editor | Code Style | Kotlin**, puis **Set from...** et selectionnez **Kotlin style**
- Pour vérifier que votre code est formaté conformément au guide de style, allez dans **Settings | Editor | Inspections and enable the Kotlin | Style issues | File is not formatted according to project settings**. Des inspections supplémentaires qui vérifient les autres problèmes décrits dans le guide de style (comme les conventions de dénomination) sont activées par défaut.



# Trailing comma

- Depuis Kotlin 1.4, il est possible de laisser une virgule à la fin d'une série d'éléments.
- C'est valable pour :
  - Enums
  - Propriétés de classe
  - Paramètres de fonction
  - Paramètres de lambdas
  - Etc...

```
fun main() {  
    val x = {  
        x: Comparable<Number>,  
        y: Iterable<Number>, // trailing comma  
        ->  
        println("1")  
    }  
  
    println(x)  
}
```

```
class Customer(  
    val name: String,  
    val lastName: String, // trailing comma  
)  
  
class Customer(  
    val name: String,  
    lastName: String, // trailing comma  
)
```

```
enum class Direction {  
    NORTH,  
    SOUTH,  
    WEST,  
    EAST, // trailing comma  
}
```



# Mots-clefs réservés

- Il est interdit d'utiliser des mots-clefs réservés comme identifiant (pour ses variables par exemple)
- La liste des mots-clefs réservés est disponible ici :  
<https://kotlinlang.org/docs/reference/keyword-reference.html>
- Pour utiliser un mot-clef réservé, vous pouvez néanmoins l'entourer le nom de votre identifiant par ce caractère : « ` ». Exemple :

```
val for = "Hello"    // interdit
val `for` = "hello" // possible
print(`for`)
```



# Variables

Types, mutables/immutables, lateinit, constantes, nullables



# Déclarer une variable

```
// Java  
int a = 1;  
String b = "xyz";
```

```
// Kotlin  
val a: Int = 1  
val b: String = "xyz"
```



# Number types

Type	Size (bits)
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Underscores avec les numeric literals :

```
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val pi = 3.14_15F
val hexBytes = 0xFF_EC_DE_5E
val hexWords = 0xCAFE_BABE
val maxval = 0xffff_ffff_ffff_ffffL
val bytes = 0b11010010_01101001_10010100_10010010;
```

```
val pi1 = 3_.1415F      // Invalid cannot put underscores adjacent to a decimal point
val pi2 = 3._1415F      // Invalid cannot put underscores adjacent to a decimal point
val socialSecurityNumber1
    = 999_99_9999_L      // Invalid cannot put underscores prior to an L suffix

val x1 = _52             // This is an identifier, not a numeric literal
val x2 = 5_2              // OK (decimal literal)
val x3 = 52_
    // Invalid cannot put underscores at the end of a literal
val x4 = 5_____2        // OK (decimal literal)

val x5 = 0_x52            // Invalid cannot put underscores in the 0x radix prefix
val x6 = 0x_52             // Invalid cannot put underscores at the beginning of a number
val x7 = 0x5_2              // OK (hexadecimal literal)
val x8 = 0x52_              // Invalid cannot put underscores at the end of a number
```



# (Im)mutability

- **var** permet de créer une variable **mutable** (peut-être réassignée)
- **val** permet de créer une variable **immutable** comme avec le mot clef final en Java (ne peut pas être réassignée)

```
// Assign-once (read-only) variable
val x = 1
x = 2          // Compile-time error
```

```
// Mutable variable
var y = 5
y = 10         // OK
```



# Inférence de type

```
// Java  
int a = 1;  
String b = "xyz";
```

```
// Kotlin  
val a = 1          // Inferred type is Int  
val b = "xyz"      // Inferred type is String
```



# Null safety avec variables

En Kotlin, une variable ne peut pas être null par défaut.

```
var str: String = "xyz"  
  
str = null // Compile-time error
```

Pour qu'une variable soit nullable, il faut spécifier cela en ajoutant un point d'interrogation à la fin du type déclaré :

```
var str: String? = "xyz"  
  
str = null // OK
```



# Constantes

En Java, on utilise les variables statiques pour les constantes.

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

En Kotlin, **static** n'existe pas. On utilise le mot-clef **const** pour déclarer une constante :

```
const val SERVER_URL: String = "https://my.api.com"
```



# Initialiser plus tard avec lateinit

Pour spécifier qu'on initialisera une variable plus tard, on utilise **lateinit** :

```
private lateinit var name: String

fun main(args: Array<String>) {
    answer = "John"
}
```

Si vous essayez d'accéder à une variable lateinit qui n'a jamais été initialisée, vous aurez une erreur de type *UninitializedPropertyAccessException: lateinit property yourVariable has not been initialized.*



# Lazy

Java

```
class JavaClass {  
    Sdk mySdk;  
  
    final Sdk getSdk() {  
        if (mySdk == null) {  
            mySdk = new Sdk();  
        }  
        return mySdk;  
    }  
}
```

Kotlin

```
val mySdk : Sdk by lazy { getSdk() }  
  
protected fun getSdk() = Sdk()  
  
val mySdk : Sdk by lazy { Sdk() }
```

```
val lazyValue: String by lazy {  
    println("appel du get")  
    "Hello"  
}  
  
fun main() {  
    println(lazyValue)  
    println(lazyValue)  
}
```

```
appel du get  
Hello  
Hello
```

**lazy()** est une fonction qui prend un lambda et retourne une instance de `Lazy<T>` qui peut servir de délégué pour implémenter une propriété « paresseuse »: le premier appel à `get()` exécute le lambda passé à `lazy()` et se souvient du résultat, les appels ultérieurs à `get()` renvoient simplement le résultat mémorisé.

*Exemple d'utilisation avec Android :*

```
class MainActivity : AppCompatActivity() {  
    private val messageView : TextView by lazy {  
        // runs on first access of messageView  
        findViewById(R.id.message_view) as TextView  
    }  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
    }  
  
    fun onSayHello() {  
        // Initialization would be run at here!!  
        messageView.text = "Hello"  
    }  
}
```



# String interpolation

Kotlin inclut un support qui permet l'utilisation de templates au sein des chaînes de caractères.

```
var a = 1
val s1 = "a is $a"

a = 2
val s2 = "${s1.replace("is", "was")}, but now is $a"
```





# Exercice (10 minutes)

Lateinit/Lazy

- Créer deux variable de type String en dehors de **main()** :
  - **name (lateinit)**
  - **email (lazy)**
- La valeur de email se fera en fonction de **name** : « name@kotlin.com »
- Dans main(), donnez une valeur à **name** et affichez **email** dans le terminal.



# Structures conditionnelles

Boucles, conditions, opérateurs ternaires



# Opérateurs d'égalité

- L'égalité référentielle (identité)
  - Opérateur : `==`
  - Signifie que les pointeurs de deux objets sont les mêmes. C'est-à-dire que les objets sont contenus dans le même emplacement mémoire ce qui nous conduit au fait que les pointeurs font référence au même objet.
- L'égalité structurelle
  - Opérateur : `==`
  - Signifie que deux objets ont un contenu équivalent.
  - Il faut surcharger la méthode `equals()` (ou utiliser le mot clef `data`) pour comparer deux objets.



# If/else

- Le if/else standard (hors opérateur ternaire) est similaire à celui de la plupart des langages
- Les opérateurs logiques de base sont similaires.

```
if (testExpression) {  
    // codes to run if testExpression is true  
}  
else {  
    // codes to run if testExpression is false  
}
```



# Opérateur ternaire

```
val max = if (a > b) a else b
```

En Kotlin, l'opérateur ternaire n'existe plus, on utilisera if/else :



```
int result = a > b ? a : b;
```

... équivaut à ...



```
val result = if (a > b) a else b
```



# Opérateur ternaire avec bloc d'instruction

Il est possible d'ajouter un bloc d'instruction dans un opérateur ternaire.  
La dernière valeur sera celle qui sera retournée pour l'affectation :

```
val max = if (a > b) {  
    print("Choose a")  
    a  
} else {  
    print("Choose b")  
    b  
}
```



# When



```
int apiResponse = 200;

switch (apiResponse) {
    case 200: System.out.print("OK");
    break;
    case 404: System.out.print("NOT FOUND");
    break;
    case 401: System.out.print("UNAUTHORIZED");
    break;
    case 403: System.out.print("FORBIDDEN");
    break;
    default: System.out.print("UNKNOWN");
    break;
}
```

En Java



```
var apiResponse = 404;

when (apiResponse) {
    200 -> print("OK")
    404 -> print("NOT FOUND")
    401 -> print("UNAUTHORIZED")
    403 -> print("FORBIDDEN")
    else -> print("UNKNOWN")
}
```

En Kotlin



# Autres applications du When

```
val apiResponse = 200

when (apiResponse) {
    200, 201, 202 -> print("SUCCESS")
    300, 301, 302 -> print("REDIRECTION")
    400, 404 -> print("ERROR")
    else -> print("UNKNOWN")
}
```

<- avec possibilités multiples

```
fun printResponse(apiResponse: Int) = when (apiResponse) {
    200 -> print("OK")
    404 -> print("NOT FOUND")
    401 -> print("UNAUTHORIZED")
    403 -> print("FORBIDDEN")
    else -> print("UNKNOWN")
}
```

<- dans la définition d'une fonction



# Boucle while

Concernant le while, rien n'a changé c'est comme en Java 😊



```
List<String> names = Arrays.asList("Jake Wharton", "Joe Birch", "Robert Martin");

for (String name : names) {
    System.out.println("This developer rocks : "+name);
}
```



```
val names = listOf("Jake Wharton", "Joe Birch", "Robert Martin")

for (name in names) {
    println("This developer rocks : $name")
}
```

```
for (i in names.indices) {
    println("This developer with number $i rocks : ${names[i]}")
}
```

```
for ((index, value) in names.withIndex()) {
    println("This developer with number $index rocks : $value")
}
```

▶ This developer with number 0 rocks : Jake Wharton  
This developer with number 1 rocks : Joe Birch  
This developer with number 2 rocks : Robert Martin



# Boucle for

```
val names = listOf("Anne", "Peter", "Jeff")
for (name in names) {
    println(name)
}
```

Ici, on parcours la liste de **names** et à chaque itération, on stock l'item en cours dans la variable **name**, le tout sans se soucier du type 😊

Il est possible de récupérer aussi l'index de chaque itération, pour cela, on utilisera **withIndex()** :

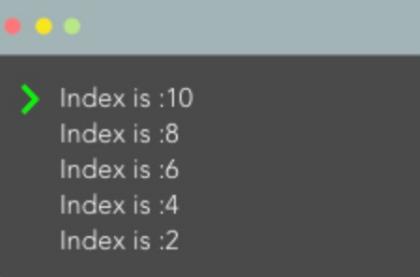
```
for (name in names.withIndex()){
    println("index ${name.index}")
    println("index ${name.value}")
}
```



# Les intervalles (1/2)

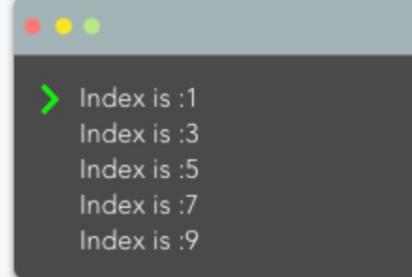
```
for (i in 1..42) {  
    println("I love this number : $i")  
}
```

```
for (i in 10 downTo 1 step 2) {  
    println("Index is :$i")  
}
```



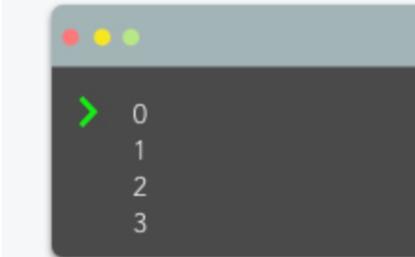
Terminal window showing the output of the first code example. The window has three colored dots (red, yellow, green) at the top. The text area shows the following output:  
Index is :10  
Index is :8  
Index is :6  
Index is :4  
Index is :2

```
for (i in 1..10 step 2) {  
    println("Index is :$i")  
}
```



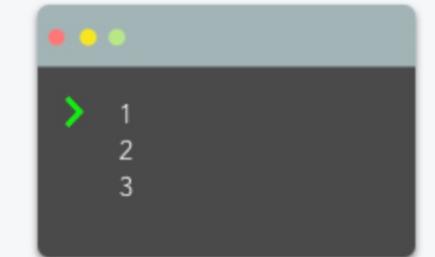
Terminal window showing the output of the second code example. The window has three colored dots (red, yellow, green) at the top. The text area shows the following output:  
Index is :1  
Index is :3  
Index is :5  
Index is :7  
Index is :9

```
for (i in 0..3)  println(i)
```



Terminal window showing the output of the third code example. The window has three colored dots (red, yellow, green) at the top. The text area shows the following output:  
> 0  
1  
2  
3

```
for (i in 1..3)  println(i)
```



Terminal window showing the output of the fourth code example. The window has three colored dots (red, yellow, green) at the top. The text area shows the following output:  
> 1  
2  
3





# Exercice (10 minutes)

Rectangle

- A partir d'un entier, dessiner :
  - une ligne d'étoile
  - un carré d'étoiles
  - un triangle d'étoiles
- Exemple  $n = 5$  :

*****	*
*****	**
*****	***
*****	****
*****	*****



# Les intervalles (2/2)

- Spécifie l'objet en cours d'itération dans une boucle for
- Est utilisé comme opérateur infixé pour vérifier qu'une valeur appartient à une plage (range), une collection ou une autre entité qui définit la méthode 'contains'
- Est utilisé dans des expressions dans le même but

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

```
val letter = 'a'  
print("$letter is in ${ when(letter) {  
    in "arthur" -> "arthur"  
    in "john" -> "john"  
    else -> "NOTHING"  
}}")
```



# Exercice (5 minutes)



When

- Créer une variable age: Int avec une valeur de test
- Afficher à l'écran :
  - « mineur » si l'utilisateur a moins de 18 ans
  - « majeur en France » si l'utilisateur a entre 18 et 21 ans
  - « majeur dans le monde » si l'utilisateur a plus de 21 ans



# Labels : break et continue

```
myLoop@ for (i in 1..100) {  
    for (j in 1..100) {  
        println("$j")  
        if (i == 50 && j == 25)  
            break@myLoop  
    }  
}
```

```
I/System.out: i=1 j=1  
I/System.out: i=1 j=2  
I/System.out: i=1 j=3  
I/System.out: i=1 j=4  
I/System.out: i=1 j=5  
I/System.out: i=2 j=1  
I/System.out: i=2 j=2  
I/System.out: i=2 j=3
```

<- **break**: termine la boucle englobante  
la plus proche

```
myLoop@ for (i in 1..5) {  
    for (j in 1..5) {  
        println("i=$i j=$j")  
        if (j == 2)  
            continue@myLoop  
    }  
}
```

```
I/System.out: i=1 j=1  
I/System.out: i=1 j=2  
I/System.out: i=2 j=1  
I/System.out: i=2 j=2  
I/System.out: i=3 j=1  
I/System.out: i=3 j=2  
I/System.out: i=4 j=1  
I/System.out: i=4 j=2  
I/System.out: i=5 j=1  
I/System.out: i=5 j=2
```

<- **continue**: Passe à l'étape suivante de la boucle  
englobante la plus proche.



# Labels : return et lambda

Supposons le contenu d'une fonction :

```
listOf(1, 2, 3, 4, 5).forEach { it: Int
    if (it == 3)
        return
    println("number $it")
}
println("I will never be printed :( ")
```

```
I/System.out: number 1
I/System.out: number 2
```

```
listOf(1, 2, 3, 4, 5).forEach lambda@{ it: Int
    if (it == 3)
        return@lambda
    println("number $it")
}
println("I will be printed :) ")
```

```
I/System.out: number 1
I/System.out: number 2
I/System.out: number 4
I/System.out: number 5
I/System.out: I will be printed :) 
```

**<- return:** Retour de la fonction englobante la plus proche ou de la fonction anonyme.



# Fonctions

Paramètres, retours, fonctions d'extension



# Déclarer une fonction

```
// Java
public int sum(int a, int b) {
    return a + b;
}
```

```
// Kotlin
public fun sum(a: Int, b: Int): Int {
    return a + b
}
```



# Simplification de fonction

```
// Kotlin  
public fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

=

```
// Kotlin  
public fun sum(a: Int, b: Int): Int = a + b
```

=

```
// Kotlin  
public fun sum(a: Int, b: Int) = a + b
```



# Retours de fonction

```
fun minOf(a: Int, b: Int ): Int = if (a < b) a else b
```

=

```
fun minOf(a: Int, b: Int ) = if (a < b) a else b
```



# Paramètres de fonction

```
// Java
public void repeat(String str, int count, String separator) {
    ...
}

repeat("abc", 5, "");
```

```
// Kotlin
fun repeat(str: String, count: Int, separator: String) {
    ...
}

repeat("abc", 5, "")
```



# Paramètres par défaut

Dans cet exemple, la valeur de **count** sera **5** et **separator** une **chaine de caractères vide**.

```
// Kotlin
fun repeat(str: String, count: Int = 3, separator: String = "") {
    ...
}

repeat("abc", [5])
```

On peut aussi définir une variable dans l'ordre que l'on souhaite en spécifiant son nom :

```
// Kotlin
fun repeat(str: String, count: Int = 3, separator: String = "") {
    ...
}

repeat("abc", separator = ",")
```



# Exercice (5 minutes)



One for X, one for me

- Créez une fonction qui prend un String **name** en paramètre et retourne un String avec ce message : « **One for \$name, one for me** »
- **name** aura « **you** » comme valeur par défaut



# Fonctions locales

Il est possible de créer une fonction dans une autre fonction.  
Dans ce cas, elle ne pourra être utilisée qu'à l'intérieur de celle-ci.

```
fun saveUser(user: User){  
    fun validate(value: String){  
        if (value.isEmpty()) throw IllegalArgumentException("Field must not be empty !")  
    }  
  
    validate(user.email)  
    validate(user.password)  
    validate(user.urlImage)  
  
    //... save user in database  
}
```



# Top-level functions

Les Top-level functions sont des fonctions statiques, détachés de toute classe, qui peuvent s'utiliser partout dans votre projet

```
// MyFunctions.kt
package com.kotlin.demo

fun warning(msg: String) {
    print("WARNING: $msg")      // package kotlin.io
}

// Demo.kt
import com.kotlin.demo.warning

fun doSomething() {
    warning("I'm warning you!")
}
```



# Vararg

On peut prendre plusieurs arguments dans une fonction grâce au mot-clef **vararg** :

```
fun foo(vararg strings: String) {  
    val listStrings = strings.toList()  
    println(listStrings)  
}
```

Deux manières d'appeler sa fonction :

```
foo(...strings: "a", "b", "c")  
foo(strings = *arrayOf("a", "b", "c"))
```



# Fonction avec pré-requis

Ici, cette fonction doit avoir **count** supérieur à 0 pour ne pas provoquer une erreur d'exception :

```
fun getIndices(count: Int): List<Int> {
    require( value: count >= 0) { "Count must be non-negative, was $count" }
    // ...
    return List(count) { it + 1 }
}

// getIndices(-1) // will fail with IllegalArgumentException

println(getIndices( count: 3)) // [1, 2, 3]
```

Si la condition du **require** est à false on aura un **IllegalArgumentException**.



# Null safety

Ou la fin du null pointer exception



# Question



Safe call avec fonction

Est-ce que ce code compilera ?

```
fun getLength(str: String): Int? {  
    return str.length // ???  
}
```



# Question



Safe call avec fonction

Est-ce que ce code compilera ?

```
fun getLength(str: String?): Int? {  
    return str.length // ???  
}
```



# Solution 1



Vérifier que str n'est pas null

```
fun getLength(str: String?): Int? {  
    if (str != null) {  
        return str.length  
    }  
    return 0  
}
```



# Solution 2



Mettre un safe call sur str avant d'appeler length

Si **str** est null, la fonction retournera null sans se donner la peine d'appeler la méthode length.

```
fun getLength(str: String?): Int? {  
    return str?.length  
}
```



# Solution 3



Mettre un safe call avec Elvis operator

Si **str** est null, la fonction retournera 0.

```
fun getLength(str: String?): Int {  
    return str?.length ?: 0  
}
```



# Safe call

```
// Java
public ZipCode getZipCode(User user) {
    if (user != null) {
        if (user.getAddress() != null) {
            return user.getAddress().getZipCode();
        }
    }
    return null;
}

// Kotlin
fun getZipCode(user: User?) = user?.address?.zipCode
```



# Pour résumer (Null Safety)

```
val x: Int = null      -> impossible (compiler error)
```

```
val x: Int? = null     -> possible ☺
```

## Opérateur Safe Call (?.)

```
val y = x.toDouble()    -> impossible si x peut être null (compiler error)
```

```
val y = x?.toDouble()  -> possible ☺ y est un Double? et si x est null, y sera null
```

## Opérateur Elvis (?:)

```
val y = x?.toDouble() ?: 0.0  -> si x?.toDouble == null, y = 0.00
```

## L'opérateur !!

```
val x: Int? = null      -> permet de ne pas vérifier si la variable peut-être null.  
val y = x!!.toDouble()  mauvaise idée car autorise le NullPointerException
```





# Exercice (10 minutes)

One function in Kotlin vs 4 functions in Java

Supposons ces 4 fonctions en Java :

```
public String foo(String name, int number, boolean toUpperCase) {  
    return (toUpperCase ? name.toUpperCase() : name) + number;  
}  
public String foo(String name, int number) {  
    return foo(name, number, false);  
}  
public String foo(String name, boolean toUpperCase) {  
    return foo(name, 42, toUpperCase);  
}  
public String foo(String name) {  
    return foo(name, 42);  
}
```

Toutes ces surcharges Java peuvent être remplacées par une seule fonction dans Kotlin. Comment ?

Créez la fonction et testez la en affichant ce qu'elle retourne avec ses 4 possibilités.



# Fonctions d'extension

Customisez des classes existantes



# Exemple de fonction utilitaire Java

```
// Java
public class StringUtils {
    public static String toCamelCase(String str) {
        return str.replaceAll...
    }
}
```

```
String camelStr = StringUtils.toCamelCase("lorem ipsum");
```



# Exemple de fonction utilitaire Kotlin

```
// Kotlin
fun toCamelCase(str: String): String {
    return str.replace...
```



# Exemple de fonction d'extension en Kotlin

```
// Kotlin
fun String.toCamelCase(): String {
    return this.replace...
}
```

```
"lorem ipsum".toCamelCase()
```



# Exemple de fonction d'extension isNull{}

```
3 fun Any?.isNull(f: ()-> Unit){  
4     if (this == null)  
5         f()  
6 }  
7  
8 fun main(args: Array<String>) {  
9  
10    var name: String? = null  
11  
12    name.isNull{  
13        println("is null")  
14    }  
15  
16    name?.let{  
17        println("is not null")  
18    }  
19  
20 }
```



# Pour résumer (fonctions d'extensions)

Les extensions servent à étendre le comportement d'une classe ou un ensemble de classes.



Exemple :

```
fun Button.disableButton() {  
    isEnabled = false  
    alpha = 0.7f  
}
```

->

Résultat:

```
loginButton.disableButton()
```





# Exercice (10 minutes)

Fonction d'extension swap

- Créez une fonction d'extension de **MutableList<Int>** nommée **swap**.
- Elle doit prendre en paramètre deux indexes et les échanger dans le corps de la fonction
- Voici comment nous pourrions tester votre code :

```
val myMutableList = mutableListOf(3, 5, 7, 8)
println("before swap $myMutableList")
myMutableList.swap( index1: 0, index2: 1)
println("after swap $myMutableList")
```



# Extensions et accesseurs/mutateurs

On ajoute un attribut firstLetter et ses accesseurs

```
StringExtensions.kt ×

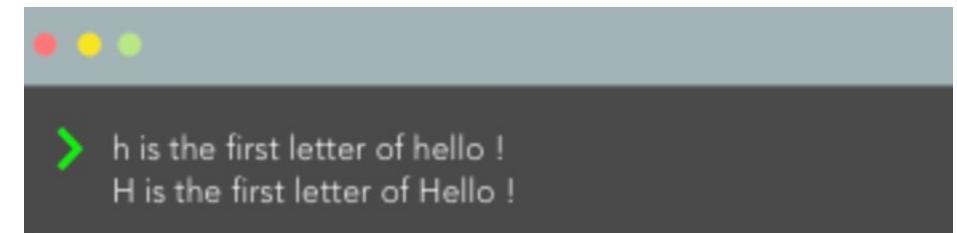
1 package extensions

2

3 var StringBuilder.firstLetter: Char
4     get() = get(0)
5     set(value) = this.setCharAt(index: 0, value)
```

```
Main.kt ×

1 import extensions.firstLetter
2
3 ▶ 4 fun main(args: Array<String>) {
5     val message = StringBuilder(str: "hello !")
6     println("${message.firstLetter} is the first letter of $message")
7     message.firstLetter = 'H'
8     println("${message.firstLetter} is the first letter of $message")
```



# Lambdas et Higher-Order functions

Fonctions anonymes et fonctions d'ordre supérieur



# Lambdas

```
val lambdaName : Type = { argumentList -> codeBody }
```

- Une lambda est une expression permettant de décrire une fonction anonyme définie par des paramètres d'entrée et un type de retour.
- Elle est entourée de {} et une -> sépare les paramètres de la fonction de sa définition.

```
{ x: Int, y: Int -> x + y }
```

- paramètres : x: Int, y: Int  
- définition : x + y

```
val addition = {a:Int, b:Int -> a + b }
println("${addition(1, 4)}")
```



# De classe anonyme à lambda

Création d'une classe anonyme

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        System.out.println("User has clicked on button !");
    }
});
```

```
button.setOnClickListener ( { v: View -> println("User has clicked on button !") } )
```

```
button.setOnClickListener { v: View -> println("User has clicked on button !") }
```

```
button.setOnClickListener { v -> println("User has clicked on button !") }
```

```
button.setOnClickListener { println("User has clicked on button !") }
```

On peut transformer une classe anonyme en lambda que si la classe n'a qu'une seule méthode.

Dans cet exemple, l'interface **View.OnClickListener** n'a qu'une seule méthode : **onClick()**



# Fonction anonyme

La différence est très mince entre lambda et fonction anonyme. La plupart du temps vous utiliserez des lambdas.

- L'avantage du lambda réside dans sa simplicité.
- L'avantage de la fonction anonyme vient du contrôle de valeur de retour (cas très rare)

```
val lambda = { input : String ->
    "received $input"
}

val anonymousFunction = fun (input : String): String {
    return "received $input"
}
```

Notez que 99,99% des cas vous utiliserez des lambdas.

Autre cas spécifique où la fonction anonyme pourrait-être utile : <https://stackoverflow.com/a/48112360/5364583>



# Traitement des lambdas par le compilateur

```
class Operator {  
    val addition = { x: Int, y: Int -> x + y }  
}
```



Lambda en Kotlin

Paramètres



```
public class Addition implements Function2<Integer, Integer, Integer> {  
    @Override  
    public Integer invoke(Integer integer, Integer integer2) {  
        return integer+integer2;  
    }  
}
```

Corps

```
public class Operator {  
    public Addition getAddition() { return new Addition(); }  
}
```



Classes générées pour la JVM  
(simplifiées et écrites en Java)



# Priorités entre fonction et lambdas

Si vous avez dans un même scope un lambda et une fonction avec le même nom, la priorité ira à la fonction :

```
fun sum(x: Int, y:Int) = x + y
val sum = { x: Int, y: Int -> x + y }
val result = sum( x: 1, y: 2)
println(result)
```

Dans ce cas, on peut utiliser invoke (permet d'invoquer la lambda) pour différencier les deux :

```
fun sum(x: Int, y:Int) = x + y
val sum = { x: Int, y: Int -> x + y }
val result = sum.invoke(1, 2)
println(result)
```



# Exercice (5 minutes)



Calculate grade with lambda

- Stockez une lambda dans une variable calculateGrade
- Celle-ci devra renvoyer une String en fonction d'un grade:Int :
  - De 0 à 40 -> « fail »
  - De 41 à 70 -> « pass »
  - De 71 à 100 -> « distinction »
  - Sinon -> « other »
- Testez en affichant des résultats



# Higher-Order Functions

Supposons une classe Operator qui contient :

```
val addition = { x: Int, y: Int -> x + y }
val subtraction = { x: Int, y: Int -> x - y }
val division = { x: Int, y: Int -> x / y }
```

Pour retenir :  
**higherOrderFunction(lambda)**

**Lambda** = fonction définie comme une valeur

**Higher order function** = fonction contenant une fonction en paramètre

Higher-Order Functions (fonction pouvant contenir une fonction en paramètre) :



```
executeOperation( x: 10, y: 5, addition) // Result = 15
executeOperation( x: 5, y: 2, subtraction) // Result = 3
executeOperation( x: 10, y: 2, division) // Result = 5
```

```
print(executeOperation( x: 10, y: 5, { x, y -> x * y })) // Result = 50
```



# Expression lambda à un seul paramètre (it)

- Il est très courant qu'une expression lambda n'ait qu'un seul paramètre.
- Si le compilateur peut comprendre lui-même la signature, il est autorisé de ne pas déclarer le seul paramètre et d'omettre `->`.
- Le paramètre sera implicitement déclaré sous le nom **it**

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```



# Unused variables

- Depuis la version 1.1 de Kotlin, il est possible de changer le nom d'un paramètre de lambda non-utilisé par un simple underscore '\_'
- L'IDE vous proposera de faire ce choix en surlignant le paramètre en question.

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





# Exercice (15 minutes)

Has A

- Créez une higher order fonction d'extension de String :
  - hasA(nbA: (Int) -> Unit)
- Si la String contient au moins une lettre A, invoquer **nbA()** et lui envoyer le nombre de A en paramètre. Pour cela, on utilisera la méthode de Collection **count{}**
- Testez

```
val nbF = "abcdef".count { it == 'f' }
```



# Inline function

Dans Kotlin, les fonctions sont first class citizen (objet de première classe), afin que nous puissions les transmettre ou les renvoyer comme d'autres types normaux en paramètre.

Cependant, la représentation de ces fonctions au moment de l'exécution peut parfois entraîner quelques limitations ou complications de performances.

Si nous exécutons le même code directement, au lieu d'utiliser des lambdas, notre implémentation serait plus efficace.

Faut-il choisir entre abstraction et efficacité?

Avec les fonctions en ligne de Kotlin, nous pouvons avoir les deux! **Nous pouvons écrire nos lambdas agréables et élégants**, et le compilateur génère le code intégré et direct pour nous. Tout ce que nous avons à faire est de mettre un **inline** dessus.

## Fonction d'ordre supérieur non optimisée



```
fun nonInlined(lambda: () -> Unit) {  
    println("Before lambda execution")  
    lambda()  
    println("After lambda execution")  
}
```

Déclaration

```
nonInlined { println("Hello students !") }
```

Usage

## Fonction d'ordre supérieur optimisée

```
inline fun inlined(lambda: () -> Unit) {  
    println("Before lambda execution")  
    lambda()  
    println("After lambda execution")  
}
```

Déclaration

```
inlined { println("Hello students !") }
```

Usage

Le mot magique!



Interprétation et traitement par le compilateur  
(Simplifié et écrit en Java)



```
private void nonInlined(Function0 lambda) {  
    System.out.println("Before lambda execution");  
    lambda.invoke();  
    System.out.println("After lambda execution");  
}
```

Déclaration

```
nonInlined(new Function0() {  
    @Override  
    public Object invoke() {  
        System.out.println("Hello students !");  
        return null;  
    }  
});
```

Usage

Création d'une classe anonyme, donc d'un objet en mémoire!

```
private void inlined(Function0 lambda) {  
    System.out.println("Before lambda execution");  
    lambda.invoke();  
    System.out.println("After lambda execution");  
}
```

Déclaration

```
System.out.println("Before lambda execution");  
System.out.println("Hello students !");  
System.out.println("After lambda execution");
```

Usage

Aucune création d'objet et l'appel de la fonction `inlined()` a disparu. À la place, nous trouvons le contenu du corps de la fonction `inlined()`!



# Limites de l'inlining

Généralement, nous pouvons utiliser des fonctions en ligne avec des paramètres lambda uniquement si le lambda est appelé directement ou transmis à une autre fonction en ligne. Sinon, le compilateur empêche l'inlining.

Par exemple, regardons **replace()** dans la bibliothèque standard Kotlin: avec une erreur du compilateur.

```
inline fun CharSequence.replace(regex: Regex, noinline transform: (MatchResult) -> CharSequence  
    regex.replace(this, transform)//passing to a normal function
```

L'extrait ci-dessus transmet le lambda, **transform**, à une fonction normale, **replace()** , d'où le **noinline**.



# Scope functions

Accéder à ses objets avec let, run, with, apply et also



# Appliquer sur une instance avec apply()

apply() : Groupe des actions  
sur un même objet

```
TextView textView = new TextView( context: this);
textView.setVisibility(View.VISIBLE);
textView.setText("Hello world !");
textView.setTextSize(19f);
textView.setMaxLines(3);
```



En Java

```
val textView = TextView( context: this)
textView.apply { this: TextView
    visibility = View.VISIBLE
    text = "Hello world !"
    textSize = 19f
    maxLines = 3
}
```



En Kotlin

Autre exemple :

```
val adam = Person("Adam").apply {
    age = 32
    city = "London"
}
```



# Traitements additionnels avec also()

Est utilisé pour ajouter du code supplémentaire en retournant l'objet appelé.

Comme **apply**, **also** peut surcharger des propriétés de l'objet.

Généralement utilisé pour ajouter des traitements additionnels à une chaîne de fonctions **let**, **apply** ou **run** (exemple pour ajouter du logging debug).

Lorsque vous enchainez plusieurs fonctions, n'hésitez-pas à renommer le paramètre **it** pour éviter la confusion.

*Exemple 1 :*

```
val person: Person = getPerson().also {  
    print(it.name)  
    print(it.age)  
}
```

*Exemple 2 :*

```
var a = 1  
var b = 2  
  
a = b.also { b = a }  
  
println(a) // print 2  
println(b) // print 1
```



# Exécuter du code (if != null) avec let()

```
arguments?.let { it: Bundle  
    param1 = it.getString(ARG_PARAM1)  
    param2 = it.getString(ARG_PARAM2)  
}
```



# Différence entre run, apply...

**run** retourne ce que l'on souhaite et re-scope la variable utilisée dans **this** :

```
val password: Password = PasswordGenerator().run {  
    seed = "someString"  
    hash = {s -> someHash(s)}  
    hashRepetitions = 1000  
  
    generate()  
}
```

Le générateur de mot de passe est maintenant rescopé à **this** et nous pouvons setter **seed**, **hash**, et **hashRepetitions** sans utiliser de variable. De plus, **generate()** retourne une instance de **Password**.

**apply** est similaire mais retourne **this** :

```
val generator = PasswordGenerator().apply {  
    seed = "someString"  
    hash = {s -> someHash(s)}  
    hashRepetitions = 1000  
}  
val password = generator.generate()
```

C'est particulièrement utile en remplacement du Builder pattern, aussi si l'on souhaite réutiliser certaines configurations.



# Ainsi que let et also

**let** est surtout utilisé afin d'éviter les vérifications de null mais peut aussi être utilisé en remplacement de **run**. La différence est que **this** n'est pas remplacé par l'instance et qu'on accède à la variable re-scopé en utilisant **it** :

```
val fruitBasket = ...  
  
apple?.let {  
    println("adding a ${it.color} apple!")  
    fruitBasket.add(it)  
}
```

Ce code ajoute **apple** à **fruitBasket** seulement si il n'est pas null. On notera aussi que **it** n'est pas nullable contrairement à **apple** (plus besoin d'utiliser un safe call avec **?** pour accéder aux attributs)

**also** est utile quand on souhaite utiliser **apply** mais qu'on ne souhaite pas surcharger **this** :

```
class FruitBasket {  
    private var weight = 0  
  
    fun addFrom(appleTree: AppleTree) {  
        val apple = appleTree.pick().also { apple ->  
            this.weight += apple.weight  
            add(apple)  
        }  
        ...  
    }  
    ...  
    fun add(fruit: Fruit) = ...  
}
```

En utilisant **apply** ici, **this** aura fait référence à **apple** et non à **fruitBasket**.



# Recevoir, argument et result avec let{}

```
class MyClass {  
    fun test() {  
        val str: String = "..."  
  
        val result = str.let {  
            print(this) // Receiver  
            print(it) // Argument  
            42 // Block return value  
        }  
    }  
}
```

Dans cet exemple avec let, nous avons :

- **this** : le receiver
  - > instance de MyClass (this@MyClass) car test() est une méthode de MyClass. Si test() avait été une « free function » (non attachée à une classe), nous aurions eu une erreur de compilation)
- **it** : l'argument
  - > c'est le String "..." avec lequel nous avons appellé let{}
- **result** : le résultat (retour)
  - > c'est 42, ce que nous avons retourné

Pour résumer, nous avons donc ceci :

Function	Receiver (this)	Argument (it)	Result
let	this@MyClass	String("...")	Int(42)



# Comparaison receiver, argument et result

Voici un comparatif de receiver, argument, et result en fonction des différentes scope functions :

Function	Receiver (this)	Argument (it)	Result
let	this@MyClass	String("...")	Int(42)
run	String("...")	N\A	Int(42)
run*	this@MyClass	N\A	Int(42)
with*	String("...")	N\A	Int(42)
apply	String("...")	N\A	String("...")
also	this@MyClass	String("...")	String("...")

\* ces fonctions ne sont pas des extensions



# Classes

Créer des modèles en une ligne, c'est possible!



# Exemple de classe en Java

- Supposons une classe en Java composée de :
  - Trois attributs
  - Un constructeur
  - Des assesseurs et mutateurs
  - Une surcharge des méthodes equals(), hashCode() et toString()



# Exemple de classe en Java

```
// Java
public class User {

    String firstName;
    String lastName;
    int age;

    public User(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        User user = (User) o;

        if (age != user.age) return false;
        if (firstName != null ? !firstName.equals(user.firstName) : user.firstName != null)
            return false;
        return lastName != null ? lastName.equals(user.lastName) : user.lastName == null;
    }

    @Override
    public int hashCode() {
        int result = firstName != null ? firstName.hashCode() : 0;
        result = 31 * result + (lastName != null ? lastName.hashCode() : 0);
        result = 31 * result + age;
        return result;
    }

    @Override
    public String toString() {
        return "User{" +
            "firstName='" + firstName + '\'' +
            ", lastName='" + lastName + '\'' +
            ", age=" + age +
            '}';
    }
}
```

Java  
<- 53 lignes de code!!!



# Classe en Kotlin

```
// Kotlin
class User {

    var firstName: String?
    var lastName: String?
    var age: Int

    constructor(firstName: String?, lastName: String?, age: Int) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other?.javaClass != javaClass) return false

        other as User

        if (firstName != other.firstName) return false
        if (lastName != other.lastName) return false
        if (age != other.age) return false

        return true
    }

    override fun hashCode(): Int {
        var result = firstName?.hashCode() ?: 0
        result = 31 * result + (lastName?.hashCode() ?: 0)
        result = 31 * result + age
        return result
    }

    override fun toString(): String {
        return "User{" +
            "firstName='" + firstName + '\'' +
            ", lastName='" + lastName + '\'' +
            ", age=" + age +
            '}'
    }
}
```

Kotlin  
<- 32 lignes de code

On va simplifier `toString()` en utilisant le template Kotlin...



# Classe en Kotlin

```
// Kotlin
class User {

    var firstName: String?
    var lastName: String?
    var age: Int

    constructor(firstName: String?, lastName: String?, age: Int) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other?.javaClass != javaClass) return false

        other as User

        if (firstName != other.firstName) return false
        if (lastName != other.lastName) return false
        if (age != other.age) return false

        return true
    }

    override fun hashCode(): Int {
        var result = firstName?.hashCode() ?: 0
        result = 31 * result + (lastName?.hashCode() ?: 0)
        result = 31 * result + age
        return result
    }

    override fun toString() = "User{firstName='$firstName', lastName='$lastName', age=$age}"
}
```

**Kotlin**  
**<- 26 lignes de code**

On va simplifier le constructeur en le remplaçant par un « primary constructor »...



# Classe en Kotlin

```
// Kotlin
class User(var firstName: String?, var lastName: String?, var age: Int) {

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other?.javaClass != javaClass) return false

        other as User

        if (firstName != other.firstName) return false
        if (lastName != other.lastName) return false
        if (age != other.age) return false

        return true
    }

    override fun hashCode(): Int {
        var result = firstName?.hashCode() ?: 0
        result = 31 * result + (lastName?.hashCode() ?: 0)
        result = 31 * result + age
        return result
    }

    override fun toString() = "User{firstName='$firstName', lastName='$lastName', age=$age}"
}
```

Kotlin

<- 18 lignes de code

On laisser le soin au compilateur Kotlin d'implémenter les @overidde (souvent répétitifs) de la classe : equals(), hashCode(), toString()... grâce au mot-clef **data**.



# Classe en Kotlin

```
// Kotlin  
data class User(var firstName: String?, var lastName: String?, var age: Int)
```

Kotlin  
← 1 ligne de code! ☺

```
// data = equals() + hashCode() + toString() + copy()
```



# Data

Dans beaucoup de classes nous avons à gérer des actions répétitives comme :

- **Afficher son contenu** sous forme de texte via la méthode `toString()` (dans une console de log par exemple)
- **Comparer** deux objets de même type (via la méthode `equals()` de sa superclasse)
- **Duplicer** un objet de manière distincte en un second objet (via la méthode `copy()` ou `clone()` de sa superclasse)

En Java, cela nous **obligeait à redéfinir** les méthodes `toString()`, `hashCode()`, `equals()` et `copy()` dans le corps de notre classe.

En Kotlin, nous allons pouvoir définir une classe comme étant destinée à contenir des modèles de données grâce au mot-clé `data`.

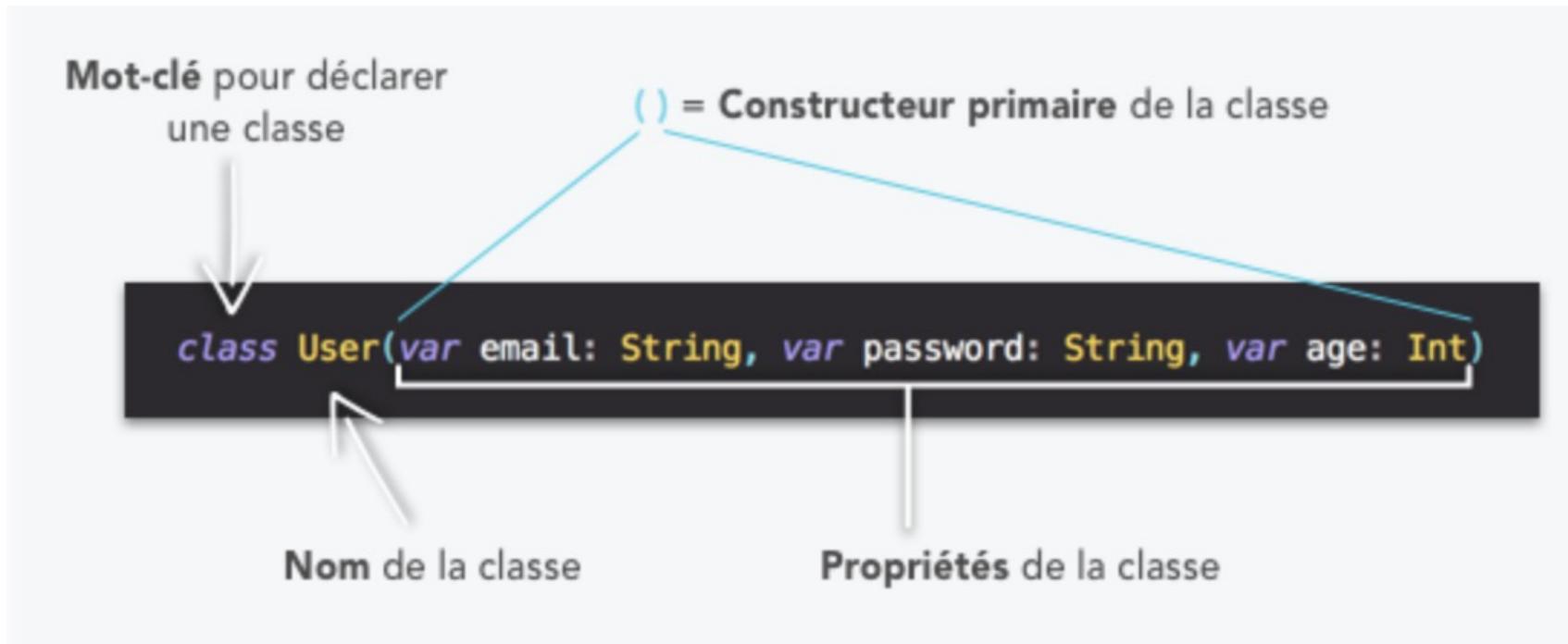
Grâce à `data`, le compilateur Kotlin implémentera pour nous les principales méthodes utilisées pour "comparer" et "décrire" un objet contenant des données comme les méthodes `toString()` , `hashCode()` , `equals()` ou encore `copy()`

Classe contenant  
des modèles de données

```
data class User(var email: String,  
               var password: String,  
               var isConnected: Boolean)
```



# Pour résumer (classes)



# Créer et accéder à un objet

En Java :

```
Contact contact = new Contact("John", "Doe", "mail@ippon.fr");
contact.setEmail("nouveau_mail@ippon.fr");
String myEmail = contact.getEmail();
```

En Kotlin :

```
var contact = Contact("John", "Doe", "mail@ippon.fr")
contact.email = "nouveau_mail@ippon.fr"
var myEmail = contact.email
```



# Attributs et setter/getter d'une classe

Vous pouvez définir les assesseurs et mutateurs d'une classe en même temps que ses attributs avec **val** (getter) / **var** (getter/setter), mais aussi avec le keyword de visibilité **private** (pas de getter/setter).

```
class User {  
    var firstName: String? = ...      // mutable (getter/setter)  
    var lastName: String = ...        // mutable  
    val age: Int = ...                // read-only (getter only)  
}
```



# Exercice (5 minutes)



Première classe en Kotlin

- Modélez cette classe Java en Kotlin ->

```
public class User {  
    private final String name;  
    private final int age;  
    private String email;  
  
    public User(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public User(String name, int age, String email) {  
        this.name = name;  
        this.age = age;  
        this.email = email;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```



# init{...}

- Le constructeur principal ne peut contenir aucun code. Le code d'initialisation peut être placé dans des blocs d'initialisation, qui sont préfixés avec le mot-clé **init**.
- Lors d'une initialisation d'instance, les blocs d'initialisation sont exécutés dans le même ordre qu'ils apparaissent dans le corps de classe.
- **init{}** est utilisée pour tous les constructeurs afin que vous puissiez avoir plusieurs constructeurs (sans primaire) qui initialisent les valeurs de base et ensuite ne vous souciez pas de savoir si vous avez réellement invoqué votre code init partagé.

```
class MyClass(val nb: Int){  
  
    init {  
        println("init MyClass with nb=$nb")  
    }  
  
}  
  
val myClass = MyClass( nb: 42)
```

```
class InitOrderDemo(name: String) {  
    val firstProperty = "First property: $name".also(::println)  
  
    init {  
        println("First initializer block that prints ${name}")  
    }  
  
    val secondProperty = "Second property: ${name.length}".also(::println)  
  
    init {  
        println("Second initializer block that prints ${name.length}")  
    }  
}
```

Résultat :

```
init MyClass with nb=42
```



# Getter/setter customisés d'un attribut

Si vous voulez définir des assesseurs/mutateurs moins basiques, vous pouvez les redéfinir avec **get()** et **set(value)** :

```
class User {  
    var firstName: String? = ""  
        get() = "abc"  
        set(value) {  
            field = value + "xyz"  
        }  
  
    var lastName: String = ""  
    val age: Int = 0  
}
```

**field** = la valeur de votre attribut





# Exercice (10 minutes)

Changement d'adresse

- Créez une class User composé des attributs :
  - name:String
  - address:String
- Le constructeur ne doit prendre que name en paramètre.
- À chaque changement d'adresse d'un User, on doit afficher :
  - "Address was changed for NAME: ANCIENNE\_ADRESSE -> NOUVELLE ADRESSE"
- Testez avec un code dans ce style :

```
val user = User( name: "Alice")
user.address = "48 rue Dupond, 75001 Paris"
user.address = "2 rue Coeur, 68000 Colmar"
```



# Héritage et modificateurs d'accès

- **final** : Classe/Méthode/Propriété ne **pouvant pas** être redéfinie. C'est l'état par défaut de tous les éléments en Kotlin.
- **open** : Classe/Méthode/Propriété **pouvant être** redéfinie. Ce modificateur d'accès doit être indiqué explicitement.
- **abstract** : Classe/Méthode/Propriété **devant** être redéfinie. Ce modificateur d'accès peut être utilisé uniquement dans des classes abstraites.

On ouvre (autorise) l'héritage sur cette classe

```
open class Button {  
    fun show(){ }  
    fun hide(){ }  
}
```

Equivalent du mot-clé « **extends** » en Java

Appel du **constructeur** de la classe **Button**

```
class CircularButton : Button() {  
    override fun show(){ }  
    override fun hide(){ }  
}  
'hide' in 'Button' is final and cannot be overridden
```

Classe **Button**

Classe **CircularButton**

<- par defaut, les classes et méthodes sont **final**



# Constructeurs multiples

Idéalement il vaut mieux privilégier les paramètres par défaut et les paramètre nommés.

Les constructeurs multiples permettent surtout de gérer l'interopérabilité avec les classes Java qui ont plusieurs constructeurs.

The diagram illustrates the definition and usage of the `Button` class in Kotlin. The class has a primary constructor that takes a `color` parameter and a secondary constructor that initializes the color to "#FFF". The usage section shows how to create a button with both constructors.

```
open class Button(var color: String) {  
    constructor(): this( color: "#FFF")  
    // ...  
}  
  
val myButton = Button()  
val myRedButton = Button( color: "#FF0000")
```

Rappel : la superclasse de toute classe est **Object** en Java et **Any** en Kotlin.



# Nested class (classe imbriquée)

Une nested class est une classe dans une classe

```
class Outer {  
    class NestedClass {  
        fun myFunction() = 2  
    }  
}  
  
private val demo = Outer.NestedClass().myFunction()
```



# Nested class : cas impossible

Impossible d'accéder à notre NestedClass depuis une instance de la classe Outer :

```
private val demo = Outer().NestedClass().myFunction()
```

Impossible d'accéder aux attributs/méthodes de la classe Outer depuis une NestedClass :

```
class Outer {  
    private val a: Int = 1  
    class NestedClass {  
        fun myFunction() = a  
    }  
}  
  
private val demo = Outer.NestedClass().myFunction()
```



# Inner class

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 a: Int = 1  
    inner class InnerClass {  
        fun myFunction() = a  
    }  
  
    private val demo = Outer().InnerClass().myFunction()
```

```
println("a=$demo")
```

```
I/System.out: a=1
```



# Companion Object

En Kotlin, le mot-clé **static** n'existe plus. En effet, nous ne créons plus de méthodes ou de propriétés statiques publiques, mais plutôt des propriétés/fonctions de premier niveau (ou "top-level") créées à l'intérieur d'un fichier plutôt que d'une classe (comme en Java).

Imaginons que nous souhaitons créer l'équivalent "à l'intérieur" d'une classe afin **d'accéder à l'ensemble de ses membres** (propriétés, fonctions, constructeurs) qu'ils soient publics ou même privés : la notion de "top-level" ne pourra pas fonctionner dans ce cas-là !

En Kotlin il existe la notion de Companion Object qui permet de créer des propriétés ou des méthodes à l'intérieur d'une classe, **accessibles même si aucune instance** de cette classe n'existe.

```
data class User(var email: String,  
               var password: String,  
               var isConnected: Boolean) {  
  
    companion object {  
        fun newInstanceAfterSignUp(email: String, password: String) = User(email, password, isConnected: true)  
    }  
}
```

Déclaration  
d'un « **objet compagnon** »

```
// First syntax  
val user = User.newInstanceAfterSignUp( email: "toto@gmail.com", password: "azerty")  
// Second syntax  
val secondUser = User.Companion.newInstanceAfterSignUp( email: "tata@gmail.com", password: "youhou")
```

Syntaxe d'utilisation  
d'un « **objet compagnon** »





# Exercice (10 minutes)

Create user with Companion Objects

- Créez une classe User avec un primary constructor privé contenant l'attribut nickname:String
- Cette classe devra pouvoir créer deux types d'utilisateurs à l'aide de deux méthodes dans un companion object :
  - newSubscribingUser(email: String)
    - créé un User dont le nickname est composé des caractères du mail avant le @ (on utilisera `substringBefore("@")`)
  - newFacebookUser(accountId: Int)
    - créé un User dont le nickname = « fb:ACCOUNTID »

```
fun main(args: Array<String>) {  
    val subscribingUser = User.newSubscribingUser( email: "bob@gmail.com")  
    val facebookUser = User.newFacebookUser( accountId: 4)  
    println(subscribingUser.nickname)  
    println(facebookUser.nickname)  
}
```

->

bob  
fb:4



# Exemple de Companion Object

Sans Companion Object :

```
object CarFactory {  
    val cars = mutableListOf<Car>()  
  
    fun makeCar(horsepowers: Int): Car {  
        val car = Car(horsepowers)  
        cars.add(car)  
        return car  
    }  
  
    val car = CarFactory.makeCar(150)  
    println(CarFactory.cars.size)
```

Avec Companion Object :

```
class Car(val horsepowers: Int) {  
    companion object Factory {  
        val cars = mutableListOf<Car>()  
  
        fun makeCar(horsepowers: Int): Car {  
            val car = Car(horsepowers)  
            cars.add(car)  
            return car  
        }  
  
        val car = Car.makeCar(150)  
        println(Car.Factory.cars.size)
```

*L'objet companion devra principalement être utilisé dans le cas où vous souhaitez accéder à des éléments internes d'une classe ou stocker des variables/fonctions ayant un lien logique très fort avec la classe.*



# Singleton

```
object RetrofitClient {  
  
    private var instance: Api? = null  
    private val BASE_URL = "https://jsonplaceholder.typicode.com/"  
  
    fun getInstance(): Api? {  
        if (instance == null) {  
            val retrofit = Retrofit.Builder()  
                .baseUrl(BASE_URL)  
                .addConverterFactory(GsonConverterFactory.create())  
                .build()  
            instance = retrofit.create(Api::class.java)  
        }  
        return instance  
    }  
}
```

Un objet Kotlin (object) est comme une classe qui ne peut pas être instanciée. Il doit donc être appelé par son nom (une classe statique en soi).

```
public class UserDao {  
  
    private static volatile UserDao instance;  
  
    private UserDao() {}  
  
    public static UserDao getInstance() {  
        if (instance == null) {  
            synchronized (UserDao.class) {  
                if (instance == null) instance = new UserDao();  
            }  
        }  
        return instance;  
    }  
  
    // ...  
  
    public User getUser() { ... }  
    public void saveUser(User user) { ... }  
}
```



## Déclaration d'un Singleton



```
UserDao.getUser()  
UserDao.saveUser(User( email: "toto@gmail.com", password: "azerty", isConnected: true))
```



```
UserDao.getInstance().getUser();  
UserDao.getInstance().saveUser(new User( email: "toto@gmail.com", password: "azerty", isConnected: true));
```

## Utilisation d'un Singleton



# Multiples classes dans un même fichier

---

Il est possible de mettre plusieurs classes dans un même fichier :

```
// Models.kt
data class User(val firstName: String, val lastName: String, val age: Int)
data class Address(val street: String, val zipCode: ZipCode)
data class ZipCode(val prefix: String, val postfix: String)
```



# Qualified this

- Pour accéder à "this" depuis un scope externe, il faut écrire **this@nomDuScope** :

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this

            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int

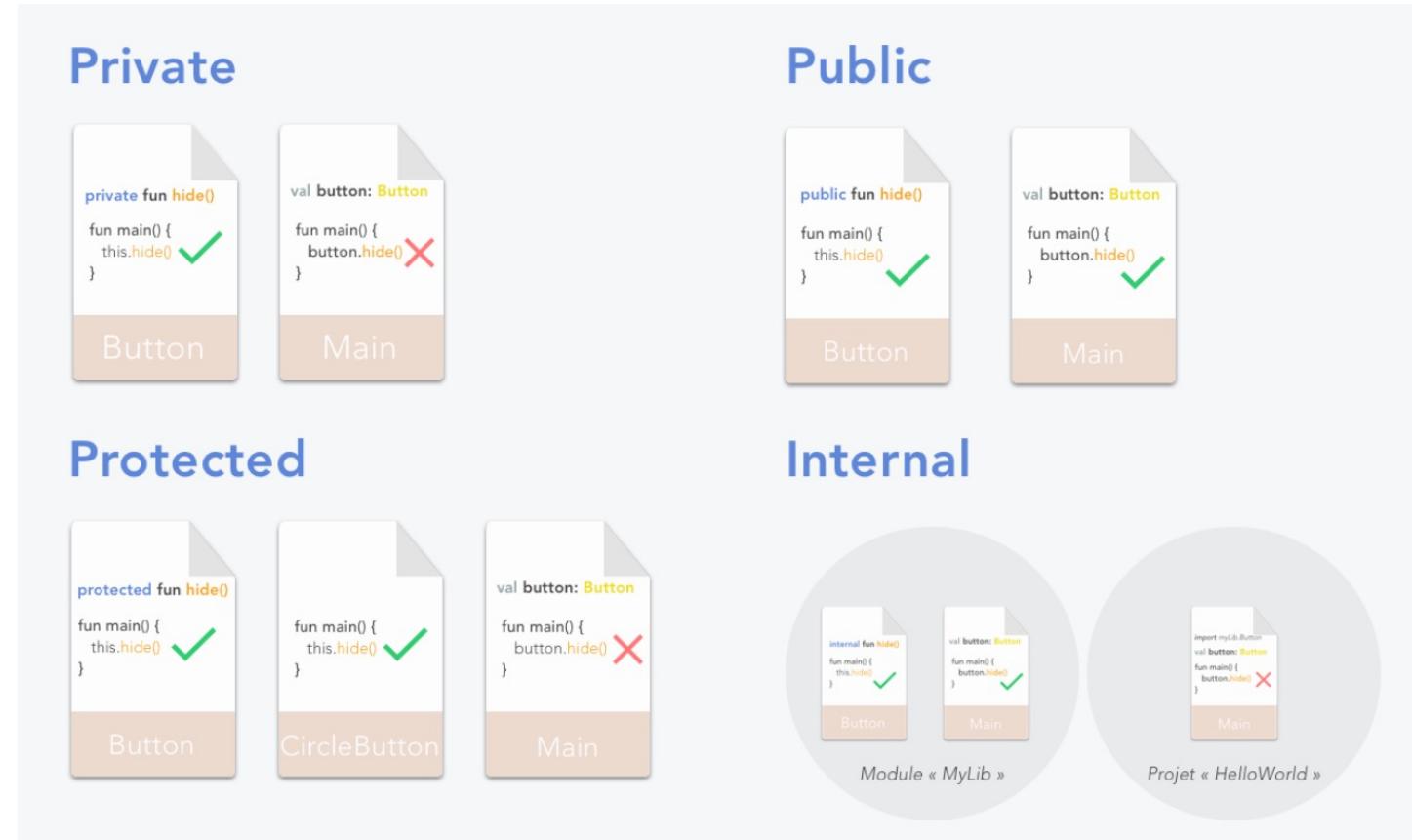
            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver
            }

            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}
```



# Visibilité des membres d'une classe

- **Private** : Un membre déclaré comme private sera visible uniquement dans la classe où il est déclaré
- **Protected** : Un membre déclaré comme protected sera visible uniquement dans la classe où il est déclaré ET dans ses sous-classes (via l'héritage).
- **Internal** : Un membre déclaré comme internal sera visible par tous ceux du même module. Un module est un ensemble de fichiers compilés ensemble (comme une librairie Gradle ou Maven, par exemple).
- **Public (par défaut)** : Un membre déclaré comme public sera visible partout et par tout le monde.



# Classes Enum

Définissons une énumération comme ayant trois constantes décrivant les types de cartes de crédit à l'aide du mot clef **enum** :

```
enum class CardType {  
    SILVER, GOLD, PLATINUM  
}
```

- Les énumérations dans Kotlin, tout comme dans Java, peuvent avoir un constructeur.
- Étant donné que les constantes enum sont des instances d'une classe Enum, les constantes peuvent être initialisées en passant des valeurs spécifiques au constructeur.

```
enum class CardType(val color: String) {  
    SILVER("gray"),  
    GOLD("yellow"),  
    PLATINUM("black")  
}
```

```
val color = CardType.SILVER.color
```



# Interfaces

Implémentez vos interfaces



# Créer une interface

- Les interfaces dans Kotlin peuvent contenir des déclarations de méthodes abstraites, ainsi que des implémentations de méthodes.
- Ce qui les différencie des classes abstraites, c'est que les interfaces ne peuvent pas stocker d'état. Ils peuvent avoir des propriétés mais celles-ci doivent être abstraites ou fournir des implémentations d'accesseur.
- Une interface est définie en utilisant le mot-clef **interface** :

```
interface MyInterface {  
    fun bar()  
    fun foo() {  
        // optional body  
    }  
}
```



# Implémenter une interface

- Une classe ou un objet peut implémenter une ou plusieurs interfaces :

```
class Child : MyInterface {  
    override fun bar() {  
        // body  
    }  
}
```

- L'appel à un constructeur différencie l'héritage de l'implémentation :

```
open class OtherClass  
interface MyInterface1 {}  
interface MyInterface2 {}  
interface MyInterface3 {}  
  
class MyClass : MyInterface1, MyInterface2, OtherClass(), MyInterface3
```





# Exercice (20 minutes)

ApiManager with interface and Higher Order Functions/Lambdas

Créez...

- Une interface **Network** composée de trois Higher Order Functions :
  - **connect(success: (message: String) -> Unit, fail: (errorMessage: String) -> Unit)**
  - **disconnect(done : () -> Unit)**
  - **sendHello(success: (message: String, code: Int) -> Unit, fail: (errorMessage: String, code :Int) -> Unit)**
- Un object **ApiManager** implémentant **Network** et composé des attributs suivant :
  - **baseUrl: String?** (null par défaut)
  - **isConnected: Boolean** (false par défaut)
- Fonctionnement :
  - Avec **connect()**, si **baseUrl** est **null** -> invoquer **fail()** avec un message de type « Not found », sinon on invoque **success()** avec le message « Success » et **isConnected** devient **true**
  - Avec **disconnect()** -> **isConnected** devient **false** puis on appelle **done()**
  - Avec **sendHello()** -> **success** si on est connecté, sinon **fail**.
  - Si on change la **baseUrl**, on se déconnecte



# Déclarer des propriétés dans une interface

- Vous pouvez déclarer des propriétés dans les interfaces.
- Une propriété déclarée dans une interface peut être soit abstraite, soit fournir des implémentations pour les accesseurs.
- Les propriétés déclarées dans les interfaces ne peuvent pas avoir de champs de sauvegarde, et donc les accesseurs déclarés dans les interfaces ne peuvent pas les référencer.

```
interface MyInterface {  
    val prop: Int // abstract  
  
    val propertyWithImplementation: String  
        get() = "foo"  
  
    fun foo() {  
        print(prop)  
    }  
}  
  
class Child : MyInterface {  
    override val prop: Int = 29  
}
```





# Exercice (10 minutes)

Interface with properties

- Créez une interface **Information** contenant deux propriétés :
  - **email: String**
  - **nickname: String** -> retourne la chaîne de caractères précédant le @ de email
- Créez une data classe **User** qui implémente l'interface **Information**. Son constructeur doit prendre un **id:Int** ainsi que l'**email** de l'interface.
- Testez comme ceci :

```
val user = User( id: 42, email: "jerome@kotlin.com")
println("$user :: nickname ${user.nickname}")
```



# Héritage des interfaces

- Une interface peut dériver d'autres interfaces et ainsi fournir à la fois des implémentations pour leurs membres et déclarer de nouvelles fonctions et propriétés.
- Naturellement, les classes implémentant une telle interface ne sont nécessaires que pour définir les implementations manquantes.

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```



# Résoudre des conflits d'implémentation

- Lorsque nous implémentons des interfaces contenant des nom de méthodes similaires, il peut sembler que nous héritons de plusieurs implémentations de la même méthode.
- Pour résoudre ces conflits, on spécifiera à quelle interface se réfère la méthode appellée grâce à l'utilisation de ***super<NomInterface>.methode()***

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```



# Exercice (10 minutes)



Button implement interfaces with same function

- Deux interfaces :
  - **Clickable** - composée des méthodes :
    - **click()**
    - **showOff()** -> affiche "I'm clickable!"
  - **Focusable** – composée des méthodes :
    - **setFocus(Boolean)** -> affiche "I got focus" si le Boolean est true, sinon "I lost focus"
    - **showOff()** -> affiche "I'm focusable!"
- Une classe **Button** qui implémente les deux interfaces
- Dans la classe **Button**, on surchargera les méthodes :
  - **click()** -> affiche "I was clicked »
  - **showOff()** -> appelle les deux méthodes showOff de chaque interface
- Dans la fonction **main()**, on utilisera notre instance de **Button()** comme dans l'exemple codé plus haut.

```
fun main(args: Array<String>) {  
    val button = Button()  
    button.showOff()  
    button.setFocus(true)  
    button.click()  
}
```



# Any, Unit et Nothing

Ces classes particulières...



# Any

- *Object* est la racine de la hiérarchie des classes en Java, chaque classe a *Object* comme superclasse. Dans Kotlin, le type **Any** représente le supertype de tous les types non nullables.
- Il diffère de l'objet Java sur deux points principaux:
  - En Java, les types primitifs (int, long,...) ne sont pas du type de la hiérarchie et vous devez les encadrer implicitement, tandis qu'en Kotlin, **Any** est un supertype de tous les types.
  - **Any** ne peut pas contenir la valeur **null**, si vous avez besoin que **null** fasse partie de votre variable, vous pouvez utiliser le type **Any?**

Les méthodes **toString()**, **equals()** et **hashCode()** de **java.lang.Object** sont héritées de **Any** tandis que pour utiliser **wait()** et **notify()**, vous devrez convertir votre variable en **Object** pour les utiliser.



# Unit

- En Java, si nous voulons qu'une fonction ne renvoie rien, nous utilisons **void** : Unit est l'équivalent Kotlin.
- Les principales caractéristiques de **Unit** par rapport au **void** de Java :
  - **Unit** est un type et peut donc être utilisée comme argument de type.
  - Une seule valeur de ce type existe.
  - Il est renvoyé implicitement. Pas besoin d'une déclaration de retour.

```
interface DataProcessor<T> {  
    fun processData(): T  
}  
  
class NoResultDataProcessor : DataProcessor<Unit> { // Use "no value" as a type argument  
    override fun processData() {  
        ...  
        // No need of a explicit return  
    }  
}
```



# Nothing

- Ce type n'existe pas en Java. Il est utilisé lorsqu'une fonction ne se terminera jamais normalement et qu'une valeur de retour n'a donc aucun sens.
- Il est très utile lors de l'analyse de ce type de code, de savoir que la fonction ne se terminera jamais.
- Un exemple de ce type de fonctions est la fonction d'échec dans les systèmes de test, ou la boucle principale dans un moteur de jeu.

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}

val address = employee.address ?: fail("${employee.name} has no address defined")
println(address)

// > java.lang.IllegalStateException: John has no address defined
```

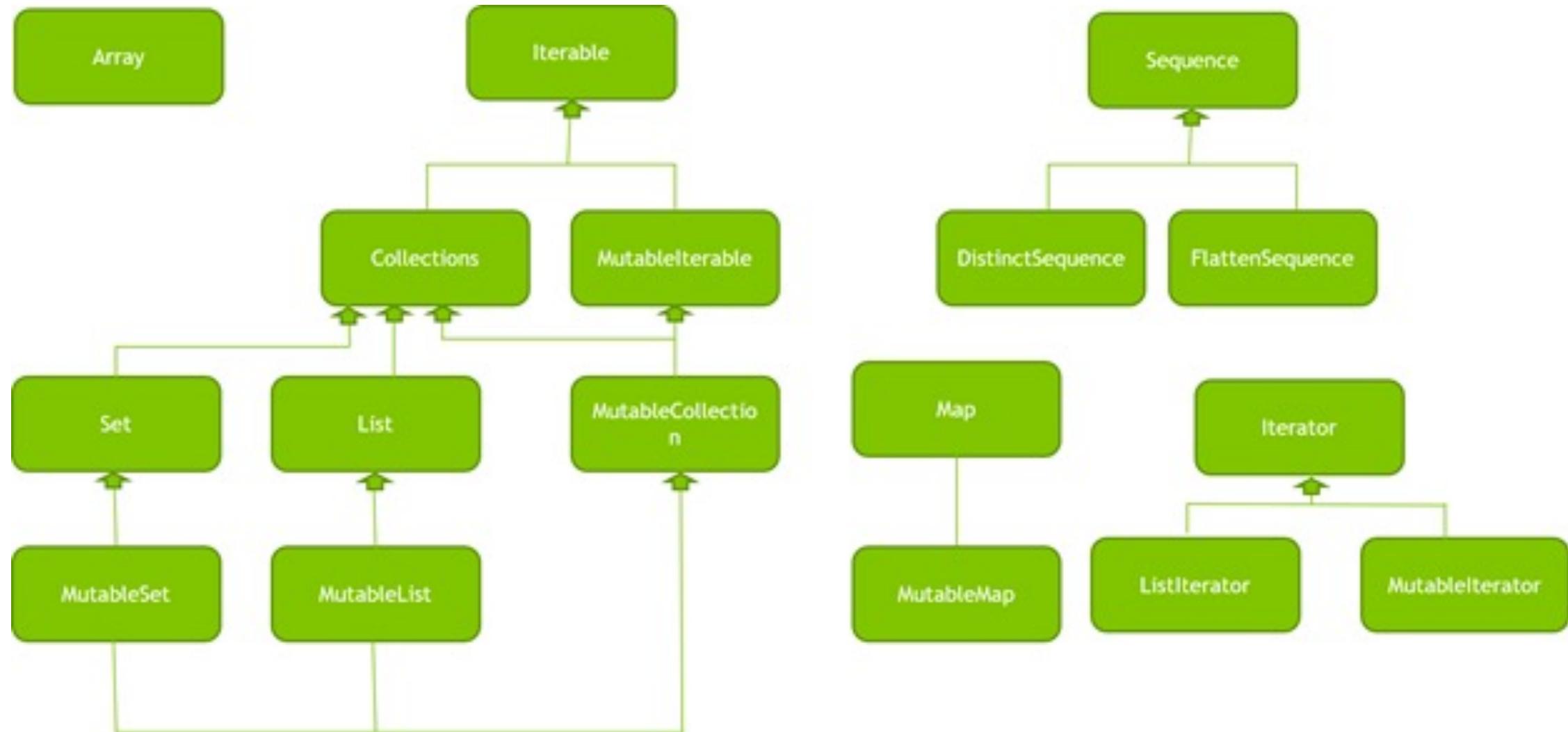


# Collections

List, Map, Set, etc...



# Types de collections



# Listes

```
val listOfNames = listOf("Jake Wharton", "Joe Birch", "Robert Martin")
listOfNames[0] // => Jake Wharton
listOfNames[0] = "Mathieu Nebra" // ERROR ! List is immutable
```

## listOf

```
val listOfNames = mutableListOf("Jake Wharton", "Joe Birch", "Robert Martin")
listOfNames[0] // => Jake Wharton
listOfNames[0] = "Mathieu Nebra" // SUCCESS !
```

## mutableListOf

```
val setOfNames = setOf("Jake Wharton", "Joe Birch", "Robert Martin")
setOfNames.first() // => Jake Wharton
setOfNames.add("Mathieu Nebra") // ERROR ! Set is immutable
```

## setOf

```
val setOfNames = mutableSetOf("Jake Wharton", "Joe Birch", "Robert Martin")
setOfNames.first() // => Jake Wharton
setOfNames.add("Mathieu Nebra") // SUCCESS !
```

## mutableSetOf

- Une liste de type **list** contiendra des éléments **uniques** et/ou **différents** de manière ordonnée. Ses éléments seront accessibles grâce à un index.
- À l'inverse, une liste de type **set** contiendra des éléments **uniques** et **distincts** de manière non ordonnée. Ses éléments ne pourront pas être accessibles via un index.

- **listOf** : Permet de créer une liste d'éléments **ordonnée** et **immutable**.
- **mutableListOf** : Permet de créer une liste d'éléments **ordonnée** et **muable**.
- **setOf** : Permet de créer une liste d'éléments **désordonnée** et **immutable**.
- **mutableSetOf** : Permet de créer une liste d'éléments **désordonnée** et **muable**.

Il existe d'autres méthodes comme **arrayOf** pour créer un tableau de valeurs et même **mapOf** pour créer un dictionnaire de valeurs.



# Map<K,V>

Renvoie une Map en lecture seule avec le contenu spécifié, sous la forme d'une liste de paires où la première valeur est la clé et la seconde la valeur.

Si plusieurs paires ont la même clé, la Map résultante contiendra la valeur de la dernière de ces paires.

Les entrées de la Map sont itérées dans l'ordre dans lequel elles ont été spécifiées.

```
val myMap = mapOf(1 to "x", 2 to "y", -1 to "zz")
```

Résultat :

```
{1=x, 2=y, -1=zz}
```

Pour une Map mutable, on utilisera : mutableMapOf()

```
val map = mutableMapOf<Int, Any?>()
map[1] = "x"
map[2] = 1.05
```





# Exercice (5 minutes)

## Resistor color

- Les résistances ont des bandes codées par couleur, où chaque couleur correspond à un nombre. Les 2 premières bandes d'une résistance ont un schéma de codage simple: chaque couleur correspond à un numéro unique.
- Ces couleurs sont codées comme ceci :
  - Noir: 0
  - Marron: 1
  - Rouge: 2
  - Orange: 3
  - Jaune: 4
  - Vert: 5
  - Bleu: 6
  - Violet: 7
  - Gris: 8
  - Blanc: 9
- Vous remarquerez que l'on peut :
  - Soit mapper les couleurs aux nombres
  - Soit stocker les couleurs sous forme de tableau et faire correspondre les nombres à l'index
- Créez une fonction capable de retourner le nombre (Int) lié à une couleur rentrée en paramètre (String)



# Opérations sur les collections

Extensions et opérations possibles sur les collections



# Opérateurs plus et minus

Les opérateurs plus (+) et moins (-) sont définis pour les collections.

Ils prennent une collection comme premier opérande. Le deuxième opérande peut être un élément ou une autre collection.

La valeur de retour est une nouvelle collection en lecture seule.

Le résultat de plus contient les éléments de la collection d'origine et du deuxième opérande.

Le résultat de moins contient les éléments de la collection d'origine, moins les éléments du deuxième opérande.

Si c'est un élément, moins supprime sa première occurrence;  
s'il s'agit d'une collection, toutes les occurrences de ses éléments sont supprimées.

```
val numbers = listOf("one", "two", "three", "four")
val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")
println(plusList)
println(minusList)
```

Résultat :

```
I/System.out: [one, two, three, four, five]
I/System.out: [one, two]
```



# maxByOrNull(), minByOrNull()

```
fun main() {  
  
    val products = mutableListOf<Product>()  
    products.add(Product(name: "Beurre", price: 2))  
    products.add(Product(name: "Lait", price: 3))  
    products.add(Product(name: "Chevre", price: 5))  
    products.add(Product(name: "Eau", price: 3))  
  
    val produitLePlusChere = products.maxByOrNull { it.price }  
  
    println(produitLePlusChere)  
  
}  
  
data class Product(  
    val name: String,  
    val price: Int  
)
```

```
Product(name=Chevre, price=5)
```



# SortBy / SortedBy

Permet de trier une liste.

Supposons une liste mutable de User :

```
class User(val name: String, val position: Int)
```

Pour trier cette liste on pourra faire (ne fonctionne qu'avec une collection mutable) :

```
users.sortBy { it.position }
```

Pour retourner liste à partir d'un tri :

```
val newList = users.sortedBy { it.position }
```





# Exercice (5 minutes)

Oldest Person

Définir quelle **Person** de la liste **persons** est la plus âgée.

Si **age** est **null**, considérez une valeur à 0:

```
data class Person(  
    val name: String,  
    val age: Int? = null  
)  
  
val persons = listOf(  
    Person(name: "Alice"),  
    Person(name: "Bob", age: 29),  
    Person(name: "Marc", age: 42),  
    Person(name: "Christine")  
)  
  
val oldest = // TODO  
    println("The oldest is: $oldest")
```



# GroupBy

Retourne une **Map<K, List<Any>** à partir d'une condition de regroupement :

```
val words = listOf("a", "abc", "ab", "def", "abcd")
val byLength = words.groupBy { it.length }

println(byLength.keys) // [1, 3, 2, 4]
println(byLength.values) // [[a], [abc, def], [ab], [abcd]]
```



# Filtres

Avec **filter{}** il est possible de filtrer une liste de sorte à récupérer uniquement les éléments correspondants à la condition exprimée :

```
val filtered = items.filter { item.startsWith('o') }
```

Exemple :

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

Autre exemple :

```
for (person in listPerson.filter { it.age >= 18 }.sortedBy { it.name }) {
    Log.d( tag: "debug", person.toString())
}
```



# All, any, count, find

all

=> Vérifie si *chaque élément* de la collection *respecte* la condition

any

=> Vérifie si *au moins UN* élément de la collection *respecte* la condition

count

=> Retourne le *nombre d'éléments* de la collection qui *respectent* la condition

find

=> Retourne le *premier élément* de la collection *respectant* la condition

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

val users = listOf(
    User( email: "toto@gmail.com",   age: 20),
    User( email: "hello@gmail.com",  age: 18),
    User( email: "oc@gmail.com",    age: 35))

// Is all users >= 20 ?
users.all { it.age >= 20 }           // false

// Is any user >= 35 ?
users.any { it.age >= 35 }           // true

// How many user are >= 20 ?
users.count { it.age >= 20 }          // 2

// Find the first user who is >= 20
users.find { it.age >= 20 }           // User(email=toto@gmail.com, age=20)
```





# Exercice (10 minutes)

Nombre pair dans ma collection ?

- Créez une fonction d'extension de **Collection<Int>**.
- Celle-ci doit renvoyer **true** si la collection contient au moins un nombre pair et **false** si ce n'est pas le cas.
- Pour vérifier si un nombre est pair : `nb % 2 == 0`
- Tester.



## .map{...}

Renvoie une liste contenant les retours de la fonction de transformation donnée à chaque élément du tableau d'origine.

```
val numbers = listOf(1, 2, 3)
println(numbers.map { it * it }) // [1, 4, 9]
```



# Sum et SumBy

*sum() ne prend que des collections de Int ou Double*

Supposons une classe Product :

```
class Product(val name: String, var price: Int)
```

Et une liste de Product :

```
val products = listOf(  
    Product(name: "A", price: 200),  
    Product(name: "B", price: 400),  
    Product(name: "C", price: 600)  
)
```

Pour avoir le total de tous les prix, on pourrait faire :

```
val totalPrice = products.map { it.price }.sum()
```

Ou

```
val totalPrice = products.sumBy { it.price }
```





# Exercice (20 minutes)

Scrabble score

- Créez une fonction qui prend une String en paramètre et nous retourne son score (Int) en se basant sur ce tableau :

<i>lettres</i>	<i>valeur</i>
A, E, I, O, U, L, N, R, S, T	1
D, G	2
B, C, M, P	3
F, H, V, W, Y	4
K	5
J, X	8
Q, Z	10



# flatMap{} et flatten()

```
data class MyClass(val listStr: List<String>)

val list1 = MyClass(listOf("a", "b", "c"))
val list2 = MyClass(listOf("1", "2", "3"))

val data = listOf(list1, list2)

val flattenData = data.flatMap { it.listStr }

println("data : $data")
println("flattenData $flattenData")
```

->

**flatMap** merge deux collections en une seule.

```
data : [MyClass(listStr=[a, b, c]), MyClass(listStr=[1, 2, 3])]
flattenData [a, b, c, 1, 2, 3]
```

Avec **map** on aurait eu une simple liste de listes :

```
val mapData = data.map { it.listStr }
println("mapData $mapData")
```

```
mapData [[a, b, c], [1, 2, 3]]
```

```
val list1 = listOf("a", "b", "c")
val list2 = listOf("1", "2", "3")

val data = listOf(list1, list2)

// val flattenData: List<String> = data.flatMap { it }
// dans ce cas on préfèrera utiliser directement flatten() pour le même résultat :
val flattenData: List<String> = data.flatten()

println("data : $data")
println("flattenData $flattenData")
```

Il est possible d'utiliser **flatten()** pour des listes simples.

->

```
data : [[a, b, c], [1, 2, 3]]
flattenData [a, b, c, 1, 2, 3]
```





# Exercice (10 minutes)

Flatten function

- Construire une fonction qui prend en paramètre une liste imbriquée et renvoi une seule liste aplatie (flatten) avec toutes les valeurs sauf null.
- Exemple
  - input: [1,[2,3,null,4],[null],5, null, 6, [[null, 7], 8, [9, 10], null], null]
  - output: [1,2,3,4,5,6,7,8,9,10]
- Petit coup de pouce :
  - Smart-cast pour vérifier si on a Collection de n'importe quel type :
    - if (value **is Collection<\*>**)



# Zip

Retourne une séquence de valeurs construite à partir des éléments de cette séquence et de l'autre séquence avec le même index en utilisant la fonction de transformation fournie appliquée à chaque paire d'éléments. La séquence résultante se termine dès la fin de la séquence d'entrée la plus courte.

L'opération est intermédiaire et sans état.

```
val a = "abcd"  
val b = "efgh"  
  
val c = a.zip(b)  
  
println("$c")
```

```
[(a, e), (b, f), (c, g), (d, h)]
```

[List<Pair<T, R>>](#)

Pour accéder à un élément d'une Pair, on utilise **first** ou **second**.



# Partition

Retourne deux collections à partir d'une seule

Dans cet exemple, on aura deux collections (**positive** et **negative**) :

```
val numbers = listOf(1, 3, -4, 2, -11)
val (positive, negative) = numbers.partition { it > 0 }
```



# Exercice (15 minutes)

Hamming distance sur ADN



Hamming distance = 3 —

A	1	0	1	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0	0	1	1

Créer une fonction qui calcule la distance de Hamming pour deux brins d'ADN (chaînes de caractères) entrés en paramètre. Cette fonction retourne donc un **Int**.

Exemple (ici la distance de Hamming = 7) :

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
^ ^ ^  ^ ^      ^^
```

On utilisera **zip** pour faire une séquence de **Pair** à comparer.

Si les chaînes de caractère ne font pas la même taille, on aura une `IllegalArgumentException`.



# Associate

Renvoie une Map contenant des paires clé-valeur fournies par la fonction de transformation appliquée aux éléments de la séquence donnée.

Si l'une des paires possède la même clé, la dernière est ajoutée à la carte.

La Map renvoyée conserve l'ordre d'itération des entrées de la séquence d'origine.

```
val map = "13579".associate {it to "impair"}
```

```
{1=impair, 3=impair, 5=impair, 7=impair, 9=impair}
```

Une variante est `associateBy`, qui ne transforme pas les valeurs d'origine mais utilise une fonction de sélection de clé.



# Exemple avec associate et opérateur plus

```
val listGoodCode = listOf(200, 201, 202, 203)
val listBadCode = listOf(400, 404, 500)

val mapOfResponses = listGoodCode.associateWith { "OK" } +
    listBadCode.associateWith { "BAD" }
```



# Exercice (30 minutes)

---



Shop

- Compléter les fonctions d'extensions fournies avec l'exercice et testez



# Cast et alias de type

Castez proprement et retrouvez plus facilement vos types grâce à Kotlin



# Smart Cast

```
private int getDefaultSize(Object object){  
    if (object instanceof String) {  
        return ((String) object).length();  
    }  
    else if (object instanceof List) {  
        return ((List) object).size();  
    }  
    return 0;  
}
```

instanceof  
Vérification du type  
(Classe)  
Cast



Vérification et conversion  
d'une variable

Java

```
private fun getDefaultSize(anyObject: Any): Int {  
    if (anyObject is String) {  
        return anyObject.length  
    } else if (anyObject is List<Any>) {  
        return anyObject.size  
    }  
    return 0  
}
```

is  
Vérification du type



Vérification et conversion  
d'une variable

Kotlin

Autre exemple ->

```
when (x) {  
    is Int -> print(x + 1)  
    is String -> print(x.length + 1)  
    is IntArray -> print(x.sum())  
}
```



# Unsafe Cast

Dans cet exemple, si y n'est pas castable, on aura un bug:

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

Il vaut mieux le mettre dans un **try/catch** ou rendre nullable la variable :

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





# Exercice (10 minutes)

Smart casts et when

Ré-écrivez la fonction Java suivante en utilisant **smart casts** et expression **when** :

```
public int eval(Expr expr) {  
    if (expr instanceof Num) {  
        return ((Num) expr).getValue();  
    }  
    if (expr instanceof Sum) {  
        Sum sum = (Sum) expr;  
        return eval(sum.getLeft()) + eval(sum.getRight());  
    }  
    throw new IllegalArgumentException("Unknown expression");  
}
```

Vous aurez besoin de ces classes et cette interface (Kotlin) pour tester votre code :

```
interface Expr  
class Num(val value: Int) : Expr  
class Sum(val left: Expr, val right: Expr) : Expr
```



# Alias de type

Attention : un **typealias** doit être déclaré en dehors de toute classe

Lorsque l'on a deux classes avec le même nom mais dans des packages différents, il peut être pratique de mettre en place des alias de type.

Supposons une déclaration de View qui se traduit de cette manière :

```
val demoView =  
    com.agrawalsuneet.demoapp.common.customviews.View(context)
```

Avec un **typealias** on aurait :

```
typealias DemoCustomView =  
    com.agrawalsuneet.demoapp.common.customviews.View
```

```
val demoView = DemoCustomView(context)
```



# Cas plus fréquents d'alias de type

```
typealias MapIntToList = HashMap<Int, List<String>>
```

```
val map = MapIntToList()
```

On peut faire des typealias génériques 😊 :

```
typealias MapIntToTemplate<T> = HashMap<Int, T>
```

```
val stringMap = MapIntToTemplate<String>()
val mapOfLists = MapIntToTemplate<List<Int>>()
```

Autres exemples avec des interfaces ou inner class :

```
class DemoClass {

    interface ViewHolderCallback
    inner class CustomViewHolder
}

typealias ViewHolderCallbackInner =
com.agrawalsuneet.demoapp.common.DemoClass.ViewHolderCallback

typealias CustomViewHolderInner =
com.agrawalsuneet.demoapp.common.DemoClass.CustomViewHolder
```

Ou avec des ressources :

```
typealias AndroidColors = android.R.color
typealias ProjectColors = R.color
```

```
ContextCompat.getColor(this, ProjectColors.colorPrimary)
ContextCompat.getColor(this, AndroidColors.black)
```



# Alias de type sur fonctions

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

```
val fourDigitFilter : Conditional<String> = { it.length == 4}
```

```
print(listOf("abc", "abcd", "abcde").filter(fourDigitFilter))
```



# Propriétés déléguées

Delegated Properties



# C'est quoi ?

- Il existe certains types communs de propriétés qui, bien que nous puissions les implémenter manuellement chaque fois que nous en avons besoin, seraient très agréables à implémenter une fois pour toutes.
- Par exemple :
  - **lazy properties**: la valeur est calculée uniquement lors du premier accès;
  - **propriétés observables**: les écouteurs sont informés des modifications apportées à cette propriété.
- Pour couvrir ces cas (et d'autres), Kotlin prend en charge les **delegated properties**.



# Syntaxe

```
val/var <property name>: <Type> by <expression>
```

L'expression après **by** est le délégué, car **get()** et **set()** correspondant à la propriété seront délégués à ses méthodes **getValue()** et **setValue()**.

Les délégués de propriété n'ont pas à implémenter d'interface, mais ils doivent fournir une fonction **getValue()** et **setValue()**.



# Exemple

```
class Delegate {  
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {  
        return "$thisRef, you are delegating '${property.name}' to me!"  
    }  
  
    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
        println("$value has been assigned to '${property.name}' in $thisRef.")  
    }  
}  
  
class Example {  
    var p: String by Delegate()  
}  
  
fun main() {  
    val e = Example()  
    println(e.p)  
    e.p = "test"  
}
```

Résultat :

```
Example@1eb44e46, you are delegating 'p' to me!  
test has been assigned to 'p' in Example@1eb44e46.
```



# Déclaration de déstructuration

Destructuring declarations



# C'est quoi ?

- Parfois, il est pratique de déstructurer un objet en un certain nombre de variables, par exemple:

```
val (name, age) = person
```

- Cette syntaxe est appelée une déclaration de déstructuration car elle crée plusieurs variables à la fois. Effectivement, nous avons déclaré deux nouvelles variables: le nom et l'âge, et pouvons les utiliser indépendamment:

```
println(name)  
println(age)
```



# Structure

- Une déclaration de déstructuration est compilée dans le code suivant:

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

- Les fonctions **component1()** et **component2()** sont un autre exemple du principe des conventions largement utilisées dans Kotlin. Et, bien sûr, il peut y avoir component3 () et component4 () et ainsi de suite.
- Notez que les fonctions **componentN()** doivent être marquées avec le mot-clé operator pour permettre leur utilisation dans une déclaration de déstructuration.



# Exemple

```
class User(  
    val id: Int,  
    val name: String,  
    val age: Int  
) {  
    operator fun component1() = id  
    operator fun component2() = name  
    operator fun component3() = age  
}
```

```
val user = User( id: 1, name: "John", age: 42)  
val(id, name, age) = user
```



# Operator overloading

Implémentez vos propres opérateurs



# Qu'est ce que l'Operator overloading ?

- Kotlin nous permet de fournir les implémentations d'un ensemble prédéfini d'opérateurs sur nos types.
- Ces opérateurs ont une représentation symbolique fixe (comme + ou \*) et une priorité fixe.
- Pour implémenter un opérateur, nous fournissons une fonction membre ou une fonction d'extension avec un nom fixe, pour le type correspondant, c'est-à-dire le type de gauche pour les opérations binaires et le type d'argument pour les opérations unaires.
- Les fonctions qui surchargent les opérateurs doivent être marquées avec le mot-clef **operator**.
- De plus, nous décrivons les conventions qui régissent la surcharge des opérateurs pour différents opérateurs.



# Sans mot-clef operator, pas d'operator

```
data class Point(val x: Int, val y: Int)

fun main(args: Array<String>) {
    val p1 = Point( x: 0, y: 1)
    val p2 = Point( x: 1, y: 2)
    println(p1 + p2)
}

'operator' modifier is required on 'plus' in ''
```

**💡** `fun Point.plus( Add 'operator' modifier ↴ ↵ More actions... ↴ ↵ y + other.y)`



# Arithmetic operators

```
data class Point(val x: Int, val y: Int)

fun main(args: Array<String>) {
    val p1 = Point( x: 0,  y: 1)
    val p2 = Point( x: 1,  y: 2)
    println(p1 + p2)
}

operator fun Point.plus(other: Point) = Point( x: this.x + other.x,  y: this.y + other.y)
```

Résultat : Point(x=1, y=3)

Expression	Traduite par
a+b	a.plus(b)
a-b	a.minus(b)
a*b	a.times(b)
a/b	a.div(b)
a..b	a.rangeTo(b)



# Unary prefix operators

```
data class Point(val x: Int, val y: Int)

fun main(args: Array<String>) {
    val point = Point( x: 10, y: 20)
    println(-point) // prints "Point(x=-10, y=-20)"
}

operator fun Point.unaryMinus() = Point(-x, -y)
```

Résultat : **Point(x=-10, y=-20)**

Expression	Traduite par
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

<- x et y de la fonction d'extension correspondent au x et y de l'instance du Point (this).

**Point(-x, -y) = Point(-this.x, -this.y)**



# 'In' operator

```
data class Point(val x: Int, val y: Int)
data class Rectangle(val upperLeft: Point, val lowerRight: Point)

operator fun Rectangle.contains(p: Point): Boolean {
    return p.x in upperLeft.x until lowerRight.x &&
           p.y in upperLeft.y until lowerRight.y
}

fun main(args: Array<String>) {
    val rect = Rectangle(Point(x: 10, y: 20), Point(x: 50, y: 50))
    println(Point(x: 20, y: 30) in rect)
    println(Point(x: 5, y: 5) in rect)
}
```

Résultat :

```
true
false
```

Expression	Traduite par
a in b	b.contains(a)
a !in b	!b.contains(a)





# Exercice (10 minutes)

DateRange with operator in

Codez la classe DateRange et son operator function :

```
data class MyDate(val year: Int, val month: Int, val dayOfMonth: Int): Comparable<MyDate>{
    override fun compareTo(other: MyDate) = when {
        year != other.year -> year - other.year
        month != other.month -> month - other.month
        else -> dayOfMonth - other.dayOfMonth
    }
}

fun main(args: Array<String>) {
    val date1 = MyDate( year: 2000, month: 11, dayOfMonth: 22)
    val date2 = MyDate( year: 2021, month: 5, dayOfMonth: 1)
    val date3 = MyDate( year: 2022, month: 2, dayOfMonth: 12)

    println(checkInRange(date = date1, first = date2, last = date3))
    println(checkInRange(date = date2, first = date1, last = date3))
    println(checkInRange(date = date3, first = date1, last = date2))
}

fun checkInRange(date: MyDate, first: MyDate, last: MyDate) = date in DateRange(first, last)
```

**Résultat attendu :**

```
false
true
false
```



# Réflexion

Introspecter la structure de votre propre programme au moment de l'exécution



# Qu'est-ce que la réflexion ?

- La réflexion est le nom pour la capacité d'**inspecter, de charger et d'interagir avec les classes, les champs et les méthodes lors de l'exécution.**
- Nous pouvons le faire même quand nous ne savons pas ce qu'ils sont au moment de la compilation.
- **Cette prise en charge est intégrée à la machine virtuelle Java** et est donc implicitement disponible pour tous les langages basés sur la JVM.
- Cependant, certains langages de la JVM bénéficient d'un support supplémentaire par rapport à ce qui est déjà disponible.



# Réflexion Java

- Toutes les constructions standard Java Reflection fonctionnent parfaitement avec Kotlin. Cela inclut la classe ***java.lang.Class*** ainsi que tout le contenu du package ***java.lang.reflect***.
- Si nous voulons utiliser les API Java Reflection standard, nous pouvons le faire exactement de la même manière qu'en Java.
  - Exemple, pour obtenir une liste de toutes les méthodes publiques dans une classe Kotlin : **MyClass::class.java.methods**
  - Cela se décompose dans les constructions suivantes:
    - ***MyClass::class*** nous donne la représentation de la classe Kotlin ***MyClass.class***
    - ***.java*** nous donne l'équivalent de ***java.lang.Class***
    - ***methods*** est un appel à l'accesseur ***java.lang.Class.getMethods()***



# Exemple de réflexion Java

```
data class ExampleDataClass(val name: String, var enabled: Boolean)
```

```
ExampleDataClass::class.java.methods.forEach(::println)
```

```
public boolean ExampleDataClass.equals(java.lang.Object)
public java.lang.String ExampleDataClass.toString()
public int ExampleDataClass.hashCode()
public final java.lang.String ExampleDataClass.getName()
public final ExampleDataClass ExampleDataClass.copy(java.lang.String,boolean)
public final boolean ExampleDataClass.getEnabled()
public final void ExampleDataClass.setEnabled(boolean)
public final java.lang.String ExampleDataClass.component1()
public final boolean ExampleDataClass.component2()
public static ExampleDataClass ExampleDataClass.copy$default(ExampleDataClass,java.lang.String,boolean,int,java.lang.Object)
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

Cela fonctionnera exactement de la même manière, qu'il soit appelé depuis Java ou Kotlin, et qu'il soit appelé sur une classe Java ou Kotlin. Cela inclut les constructions spécifiques à Kotlin, telles que les classes de données (**data**).



# Références de classe Kotlin

- L'API Kotlin Reflection permet d'accéder à la référence de classe Java (l'objet *java.lang.Class*) mais également à tous les détails spécifiques à Kotlin.
- L'API Kotlin pour les détails de classe est centrée sur la classe *kotlin.reflect.KClass*.

```
data class MyClass(val name: String, var enabled: Boolean)
```

```
// Kotlin Reflection :  
val myClass = MyClass::class  
println("qualifiedName -> ${myClass.qualifiedName}")  
println("isData -> ${myClass.isData}")  
println("isCompanion -> ${myClass.isCompanion}")  
println("isAbstract -> ${myClass.isAbstract}")  
println("isFinal -> ${myClass.isFinal}")  
println("isSealed -> ${myClass.isSealed}")  
println("constructors -> ${myClass.constructors}")
```

```
// Java Reflection :  
myClass.java.methods.forEach(::println)
```



# Référence de méthode Kotlin

- En plus de pouvoir interagir avec les classes, **nous pouvons également interagir avec méthodes et propriétés** .
- Cela inclut les propriétés de classe (définies avec `val` ou `var`) , les méthodes de classe standard et les fonctions de niveau supérieur.
- Cela fonctionne aussi bien sur du code écrit en Java standard que sur du code écrit en Kotlin.

```
val str = "Hello"  
val lengthMethod = str::length  
println("length ${lengthMethod.get()}")
```

-> length 5



# Coroutines

Exécuter du code asynchrone allégé

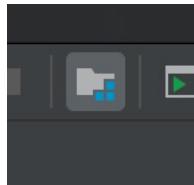


# Une coroutine, qu'est-ce que c'est ?

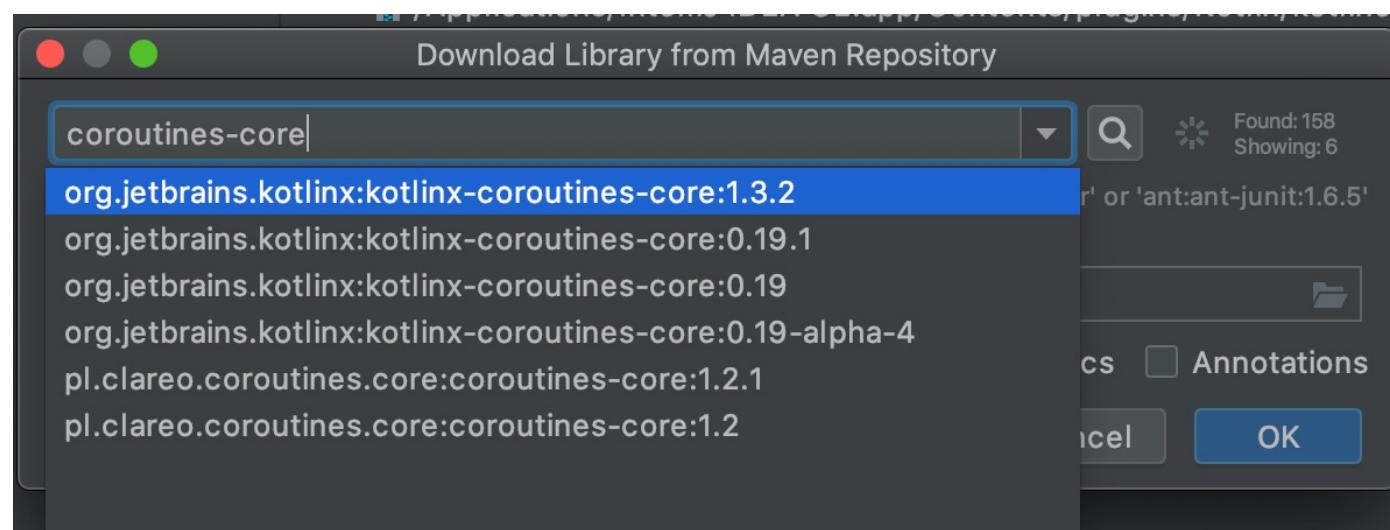
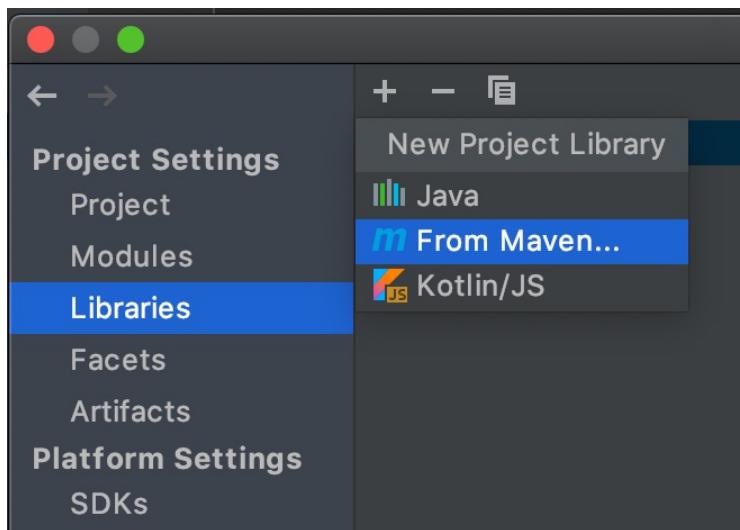
- Une coroutine est une **unité de traitement** permettant d'**exécuter du code non-bloquant et asynchrone**.
- Sur le principe, il s'agit d'un **Thread “allégé”**.
  - Il est possible d'en créer des centaines de milliers en parallèle sur un poste classique sans problèmes d'**out of memory**.
- **Elle peut être suspendue et reprise plus tard**
  - Suspendue dans un Thread et être reprise dans un autre. Elle ne dépend pas d'un Thread en particulier.
- Il est aussi possible de faire communiquer les coroutines entre elles.
- 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
  - des traitements n'ayant pas le besoin de manipuler l'interface utilisateur (ou seulement lorsque le traitement est terminé)
- Les coroutines sont fournies par la librairie **kotlinx.coroutines** :
  - <https://github.com/Kotlin/kotlinx.coroutines>



# Ajouter la librairie Coroutines sur IntelliJ



Ouvrez **Project Structure**, allez dans **Project Settings > Libraries**, appuyez sur "+" et sélectionnez "**From Maven...**". Quand la fenêtre s'affiche, entrez le nom et version de la librairie dont vous avez besoin (vous pouvez la trouver dans la documentation).



# Votre première coroutine

Supposons ce code :

```
import kotlinx.coroutines.*\n\nfun main() {\n    GlobalScope.launch { // launch a new coroutine in background and continue\n        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)\n        println("World!") // print after delay\n    }\n    println("Hello,") // main thread continues while coroutine is delayed\n    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive\n}
```

Et son résultat :

Hello,  
World!

Ici nous lançons une nouvelle coroutine dans le GlobalScope, ce qui signifie que la durée de vie de celle-ci n'est limité que par la durée de l'application entière.



# Et avec Thread ?

```
import kotlinx.coroutines.*  
import kotlin.concurrent.thread  
  
fun main() {  
    thread {  
        delay(1000L)  
        println("World!")  
    }  
    println("Hello,")  
    Thread.sleep(2000L)  
}
```

- ! Suspend function 'delay' should be called only from a coroutine or another suspend function

```
import kotlinx.coroutines.*  
import kotlin.concurrent.thread  
  
fun main() {  
    thread {  
        Thread.sleep(1000L)  
        println("World!")  
    }  
    println("Hello,")  
    Thread.sleep(2000L)  
}
```

Hello,  
World!

**delay()** est une fonction de suspension spéciale qui ne bloque pas un Thread, mais suspend la coroutine et ne peut être utilisée qu'à partir d'une coroutine.

Vous pouvez obtenir le même résultat en remplaçant **GlobalScope.launch{...}** par le **thread{...}** et **delay(...)** par **Thread.sleep{...}**.

N'oubliez pas d'importer **kotlin.concurrent.thread**.

*NOTE : n'utilisez pas Thread dans une coroutine!*



# Pont entre mondes bloquants et non bloquants

Le premier exemple mélange un **delay non bloquant** et un **Thread.sleep bloquant** dans le même code. Il est facile de perdre la trace de celui qui est bloquant et de celui qui ne l'est pas.

Soyons explicites sur le blocage à l'aide du générateur de coroutine **runBlocking** :

```
fun main() {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L)
        println("World!")
    }
    println("Hello,") // main thread continues here immediately
    runBlocking { // but this expression blocks the main thread
        delay(2000L) // ... while we delay for 2 seconds to keep JVM alive
    }
}
```

Le résultat est le même mais ce code utilise seulement des **delay()** non bloquant.

Le thread principal invoquant **runBlocking{...}** est bloqué jusqu'à ce que l'execution de cette coroutine soit terminée.



# runBlocking appliqué à main()

Il est possible d'écrire la même instruction afin d'appliquer **runBlocking{...}** sur l'ensemble de la fonction **main()** :

```
fun main() = runBlocking<Unit> { // start main coroutine
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L)
        println("World!")
    }
    println("Hello,") // main coroutine continues here immediately
    delay(2000L)      // delaying for 2 seconds to keep JVM alive
}
```

Ici, **runBlocking<Unit>{...}** fonctionne comme un adaptateur qui est utilisé pour démarrer la coroutines principale de niveau supérieur.

Nous spécifions explicitement son type de retour **Unit** car une fonction **main()** bien formée dans Kotlin doit retourner **Unit**.

*Il existe aussi un moyen d'écrire vos tests unitaires pour vos fonctions suspendues :*

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking<Unit> {
        // nous pouvons utiliser nos fonctions suspendues ici
    }
}
```





# Exercice (5 minutes)

First launch

L'ensemble du code de votre main() doit être dans un scope bloquant (runBlocking)

Dans votre main :

- **Lancer une coroutine qui :**
  - Attend 2 secondes
  - Affiche "Tasks from some blockingMethod"
- **Créer une coroutine scope et dedans :**
  - Lancer une coroutine qui :
    - Attend 2 secondes
    - Affiche "Task from nested launch"
  - Attendre 1 seconde
  - Afficher "Task from coroutine scope"
- **Afficher "Coroutine scope is over"**



# Rejoindre une tâche : join() a Job

Retarder pendant un certain temps alors qu'une autre coroutine fonctionne n'est pas une bonne approche. Attendons explicitement (de manière non bloquante) que la tâche d'arrière-plan que nous avons lancé soit terminé:

```
val job = GlobalScope.launch { // run et garde la référence du travail
    delay(1000L)
    println("World!")
}
println("Hello,")
job.join() // attend que le job soit terminé
```

Ici, la coroutine démarre en même temps que la référence de son travail est stockée dans **job** (de type **Job**).

Ici, la classe **Job** est utilisée pour représenter un travail d'une Coroutine et est également utilisée pour gérer l'exécution de ladite Coroutine. Vous pouvez annuler l'exécution d'une Coroutine si vous annulez un Job. La méthode **join()** permet d'attendre que la tâche de la coroutine **job** soit terminée.

Le résultat est toujours le même mais le code de la coroutine principale ne fait plus référence à la durée du travail en arrière-plan.

L'attente est explicite, c'est plus compréhensible.



# Arrêter une tâche : cancel() a job

Pour arrêter une tâche, on utilisera la méthode **cancel()** de notre job :

```
job.cancel()
```

Dans cet exemple, on appelle `cancel` directement dans notre job après la 5<sup>ème</sup> itération de `repeat()` :

```
val job = launch { this: CoroutineScope
    repeat( times: 1000) { i ->
        println("job: I'm sleeping $i ...")
        delay( timeMillis: 500L)
        if (i == 5)
            cancel()
    }
    job.join()
    println("main: Now I can quit.")
```

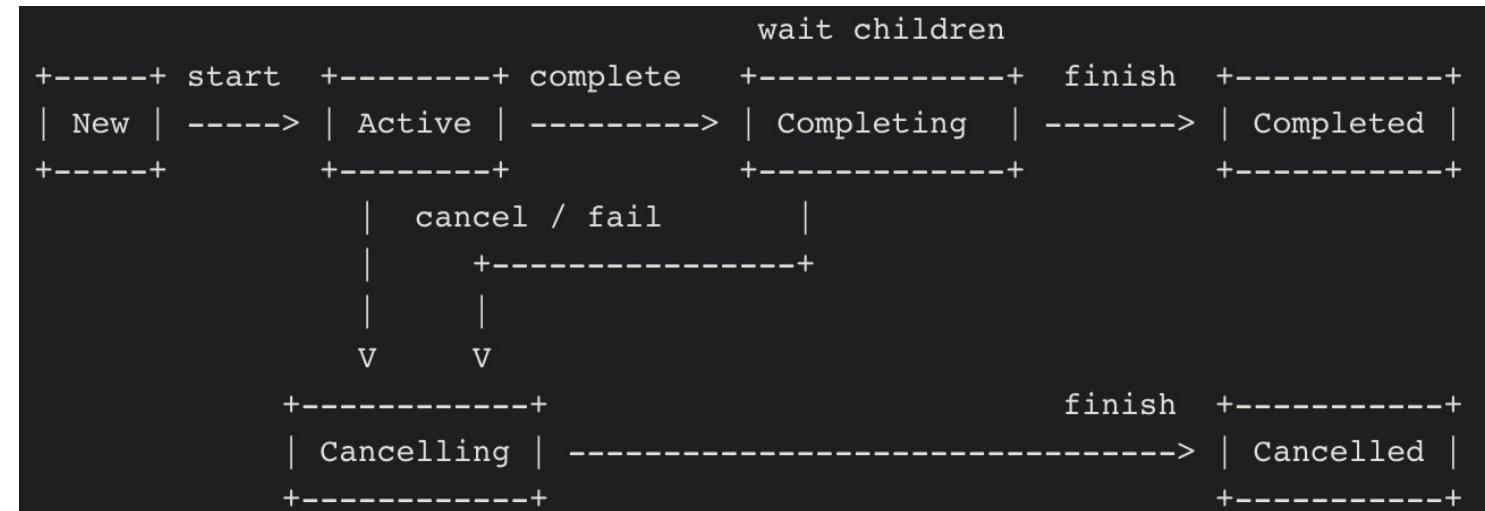
```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
job: I'm sleeping 3 ...
job: I'm sleeping 4 ...
job: I'm sleeping 5 ...
job: I'm sleeping 6 ...
main: Now I can quit.
```



# Cycle de vie j'un Job

- **NEW**
  - Lors de la création du Job
- **ACTIVE**
  - Lorsque le Job commence à s'exécuter
- **COMPLETING**
  - Lorsque le Job attend que ses Job enfants se terminent
- **COMPLETED**
  - Lorsque le Job et les Job enfants sont terminés
- **CANCELLING**
  - Si le Job ou l'un de ses enfants échoue ou est annulé
- **CANCELLED**
  - Lorsque le Job a terminé son annulation

```
fun Job.status(): String = when {  
    isActive && !isCompleted && !isCancelled -> "Active/Completing"  
    !isActive && !isCompleted && isCancelled -> "Cancelling"  
    !isActive && isCompleted && isCancelled -> "Cancelled"  
    !isActive && isCompleted && !isCancelled -> "Completed"  
    else -> "New"  
}
```



Déduire le status d'un Job grâce à ses méthodes :

State	<u>isActive</u>	<u>isCompleted</u>	<u>isCancelled</u>
New (optional initial state)	false	false	false
Active (default initial state)	true	false	false
Completing (transient state)	true	false	false
Cancelling (transient state)	false	false	true
Cancelled (final state)	false	true	true
Completed (final state)	false	true	false



# Scope builder

```
fun main() = runBlocking { //this: CoroutineScope
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope { //creates a coroutine scope
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope") //printed before nested launch
    }

    println("Coroutine scope over") //no printed until nested launch completes
}
```

En plus du scope de la coroutine fournie par différents builders, il est possible de déclarer votre propre scope à l'aide du builder **coroutineScope**. Il crée le scope de la coroutine et ne se termine que lorsque tous les enfants lancés sont terminés.

**runBlocking** et **coroutineScope** peuvent se ressembler car ils attendent tous les deux la fin de leurs corps et de tous ses enfants.

La principale différence entre ces deux est que la méthode **runBlocking** bloque le thread actuel en attente, tandis que **coroutineScope** se suspend simplement, libérant le thread sous-jacent pour d'autres utilisations.

En raison de cette différence, **runBlocking** est une fonction régulière et **coroutineScope** est une fonction de suspension.

Résultat :

```
Task from coroutine scope
Task from runBlocking
Task from nested launch
Coroutine scope over
```

Notez que juste après le message "**task from coroutine scope**" en attendant le lancement imbriqué, "**task from runBlocking**" est exécuté et imprimé, bien que **coroutineScope** ne soit pas encore terminé.



# Démonstration de légèreté des coroutines

Testez le code suivant :

```
fun main() = runBlocking {
    repeat(100_000) { // launch a lot of coroutines
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

Il lance 100000 coroutines et, après une seconde, chaque coroutine imprime un point à l'écran.

Maintenant, essayez cela avec des threads.

Que se passerait-il?

Très probablement, votre code produira des erreurs de mémoire insuffisante.



# Suspending function

Extrayons le bloc de code de `launch{...}` dans une fonction distincte...

Si vous effectuez une refactorisation de type "**extract function**" sur ce code avec IntelliJ, vous obtenez une nouvelle fonction avec le mot-clé **suspend**.... c'est votre première fonction de suspension!

Les fonctions de suspension peuvent être utilisées à l'intérieur des coroutines tout comme les fonctions normales, mais leur caractéristique supplémentaire est qu'elles peuvent, à leur tour, utiliser d'autres fonctions de suspension (comme le délai dans cet exemple) pour suspendre l'exécution d'une coroutine.

```
fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Suspend permet de mettre en suspension une méthode ou une lambda. Après leurs exécutions dans un **CoroutineScope**, elles peuvent être arrêtées à tout moment .

```
// fonction suspendue
suspend fun maMethode(): String {
    // ...
    return value
}

suspend () -> {
    // ...
}
```



# Comment utiliser mes fonctions suspendues ?

- Pour cela il faut utiliser les **coroutines builders**.
- Les coroutines builders sont des fonctions qui prennent en paramètre une lambda suspendue afin d'en créer une coroutine. Il existe une multitude de builder, les plus connus étant les suivants :
  - `async(...)`
  - `launch(...)`
  - `buildSequence(...)`
- *Rappel : ces fonctions retournent un objet de type **Job***



# launch

Le launch Coroutine Builder lance une nouvelle coroutine sans bloquer le thread actuel et renvoie une référence à la coroutine en tant que Job.

```
fun CoroutineScope.launch(  
    context: CoroutineContext = EmptyCoroutineContext,  
    start: CoroutineStart = CoroutineStart.DEFAULT,  
    block: suspend CoroutineScope.() -> Unit  
) : Job (source)
```

<- À noter ici que le lancement de la Coroutine Builder est en fait une fonction d'extension sur **CoroutineScope**

Voyons les 3 paramètres de la Coroutine Builder **lauch** :

- **context: CoroutineContext**

- Spécifie le contexte dans lequel la Coroutine doit s'exécuter.
    - Par défaut, la valeur passée ici est **EmptyCoroutineContext**, qui fournit simplement un CoroutineContext vide à votre Coroutine.

- **start: CoroutineScope**

- Ce paramètre est utilisé pour définir la façon dont vous souhaitez lancer votre Coroutine.
    - Par défaut, il s'agit de **CoroutineStart.DEFAULT**. Voici toutes les valeurs énumérées pour **CoroutineStart**:
      - **DEFAULT**: cette valeur commence à exécuter immédiatement la Coroutine.
      - **ATOMIQUE**: similaire à DEFAULT, sauf que la coroutine ne peut pas être annulée avant de commencer son exécution.
      - **LAZY**: démarre la Coroutine uniquement lorsque cela est nécessaire. Ceci est communément appelé initialisation paresseuse.
      - **UNDISPATCHED**: démarre immédiatement la Coroutine mais se suspendra à l'atteinte d'un point de suspension dans le thread actuel.

- **block: suspend CoroutineScope. () -> Unit**

- C'est là que va le code qui doit être exécuté. Il prend un bloc de suspension, qui est une fonction d'extension sur CoroutineScope, et retourne une unité.



# async

La Coroutine Builder `async` est la même que `launch` à la différence qu'elle retourne un `Deferred<T>`.

```
fun <T> CoroutineScope.async(  
    context: CoroutineContext = EmptyCoroutineContext,  
    start: CoroutineStart = CoroutineStart.DEFAULT,  
    block: suspend CoroutineScope.() -> T  
) : Deferred<T> (source)
```

`Deferred<T>` renvoie une valeur particulière de type `T` une fois la Coroutine a terminé son exécution, contrairement à `Job`.

*Pour résumer: `launch` est plus un Coroutine Builder à lancer et oublier, tandis que `async` renvoie en fait une valeur une fois que votre Coroutine est terminée.*

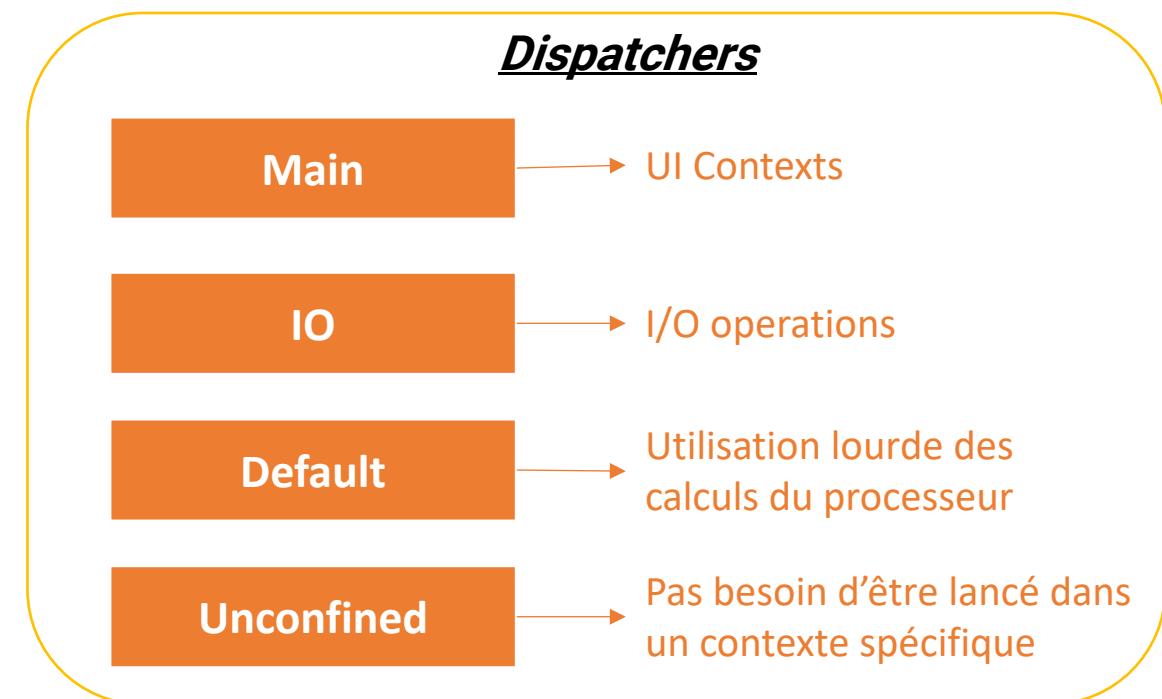
Remarque: Un `Deferred<T>` étend `Job`. Vous pouvez donc y appeler `cancel()` comme vous le feriez normalement pour annuler votre Coroutine. Il n'a pas la méthode `join()` mais a la méthode `await()` à la place. Il s'agit d'un type de retour de `T`, c'est-à-dire qu'il attend que votre Coroutine ait terminé son exécution et renvoie la variable résultante.

Étant donné que `Deferred<T>` étend `Job`, le cycle de vie reste le même.



# CoroutineContext et Dispatchers

- Les coroutines s'exécutent toujours dans un contexte représenté par une valeur de type **CoroutineContext**, définie dans la bibliothèque standard de Kotlin.
- Le **CoroutineContext** est un ensemble de divers éléments. Les principaux éléments sont :
  - le **Job** de la coroutine
  - son **dispatcher** (répartiteur)



# Dispatchers et threads

- Le contexte de la coroutine comprend un répartiteur de coroutine (`CoroutineDispatcher`) qui détermine le ou les threads que la coroutine correspondante utilise pour son exécution.
- Le `CoroutineDispatcher` peut :
  - limiter l'exécution de la coroutine à un thread spécifique
  - le répartir dans un pool de threads
  - le laisser s'exécuter de manière non confinée.
- Tous les générateurs de coroutine (`CoroutineBuilder`) comme **launch** et **async** acceptent un paramètre `CoroutineContext` facultatif qui peut être utilisé pour spécifier explicitement le répartiteur pour la nouvelle coroutine et d'autres éléments de contexte.



# Exemple de launch avec Dispatchers

```
fun main() = runBlocking<Unit> { this: CoroutineScope

    launch { // context du parent : coroutine du main runBlocking
        println("main runBlocking      : I'm working in thread ${Thread.currentThread().name}")
    }

    launch(Dispatchers.Unconfined) { // not confiné -- fonctionnera avec le main thread
        println("Unconfined          : I'm working in thread ${Thread.currentThread().name}")
    }

    launch(Dispatchers.Default) { // sera expédié vers le Dispatchers.Default
        println("Default            : I'm working in thread ${Thread.currentThread().name}")
    }

}
```

Unconfined	: I'm working in thread main
main runBlocking	: I'm working in thread main
Default	: I'm working in thread DefaultDispatcher-worker-1

Lorsque `launch{}` est utilisé sans paramètres, il hérite du contexte (et donc du répartiteur) du `CoroutineScope` à partir duquel il est lancé. Dans ce cas, il hérite du contexte de la coroutine `runBlocking` principale qui s'exécute dans le thread principal.

`Dispatchers.Unconfined` est un répartiteur spécial qui semble également s'exécuter dans le thread principal, mais il s'agit en fait d'un mécanisme différent (voir slide suivante).

Le répartiteur par défaut utilisé lorsque les coroutines sont lancées dans `GlobalScope` est représenté par `Dispatchers.Default` et utilise un pool d'arrière-plan partagé de threads, donc `launch(Dispatchers.Default){}` utilise le même répartiteur que `GlobalScope.launch{}`.

`newSingleThreadContext` crée un thread pour l'exécution de la coroutine. Un thread dédié est une ressource très coûteuse. Dans une application réelle, il doit être libéré, lorsqu'il n'est plus nécessaire, à l'aide de la fonction `close()`, ou stocké dans une variable de niveau supérieur et réutilisé dans toute l'application.





# Exercice (15 minutes)

launch, async, suspend

- Créer 3 fonctions qui retournent toute une **String** après un délai variable (de 1 à 3 secondes max):
  - downloadTask1()
  - downloadTask2()
  - downloadTask3()
- Créer 6 variables :
  - 3 = **launch** de chaque functions
  - 3 = **async** de chaque functions (sur le **Dispatcher IO**)
- Afficher le contenu de chaque variable en utilisant leur méthode **join** ou **await** (en fonction du type)



# Unconfined vs confined dispatcher

- Le répartiteur de coroutine **Dispatchers.Unconfined** démarre une coroutine dans le thread de l'appelant, mais uniquement jusqu'au premier point de suspension. Après la suspension, il reprend la coroutine dans le thread qui est entièrement déterminée par la fonction de suspension qui a été invoquée. Le répartiteur non confiné convient aux coroutines qui ne consomment pas de temps CPU ni ne mettent à jour les données partagées (comme l'interface utilisateur) confinées à un thread spécifique.
- De l'autre côté, le dispatcher est hérité du **CoroutineScope** externe par défaut. Le dispatcher par défaut pour la coroutine **runBlocking**, en particulier, est limité au thread invokeur, donc l'hériter a pour effet de limiter l'exécution à ce thread avec une planification FIFO prévisible.

```
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay( timeMillis: 500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
    delay( timeMillis: 1000)
    println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
}
```

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

Ainsi, la coroutine avec le contexte hérité de **runBlocking{}** continue de s'exécuter dans le thread principal, tandis que celle non confinée reprend dans le thread d'exécuteur par défaut que la fonction de délai utilise.

Le répartiteur non confiné est un mécanisme avancé qui peut être utile dans certains cas où l'envoi d'une coroutine pour son exécution ultérieure n'est pas nécessaire ou produit des effets secondaires indésirables, car certaines opérations dans une coroutine doivent être effectuées immédiatement. Le répartiteur non confiné ne doit pas être utilisé dans le code général.



# Enfant d'une coroutine

- Lorsqu'une coroutine est lancée dans le **CoroutineScope** d'une autre coroutine, elle hérite de son contexte via **CoroutineScope.coroutineContext** et le **Job** de la nouvelle coroutine devient un **enfant du Job** de la **coroutine parent**.
- Lorsque la **coroutine parent est annulée, tous ses enfants sont également annulés** récursivement.
- Cependant, lorsque **GlobalScope** est utilisé pour lancer une coroutine, il n'y a pas de parent pour le Job de la nouvelle coroutine. Il n'est donc pas lié au périmètre depuis lequel il a été lancé et fonctionne de manière indépendante.

```
val request = launch { this: CoroutineScope

    GlobalScope.launch { this: CoroutineScope
        println("job1: I run in GlobalScope and execute independently!")
        delay( timeMillis: 1000)
        println("job1: I am not affected by cancellation of the request")
    }

    launch { this: CoroutineScope
        delay( timeMillis: 100)
        println("job2: I am a child of the request coroutine")
        delay( timeMillis: 1000)
        println("job2: I will not execute this line if my parent request is cancelled")
    }

}

delay( timeMillis: 500)
request.cancel()
delay( timeMillis: 1000)
println("main: Who has survived request cancellation?")
```

job1: I run in GlobalScope and execute independently!  
job2: I am a child of the request coroutine  
job1: I am not affected by cancellation of the request  
main: Who has survived request cancellation?



# Responsabilité parentale

- Un parent coroutine attend toujours l'achèvement de tous ses enfants.
- Un parent n'a pas à suivre explicitement tous les enfants qu'il lance et il n'a pas besoin d'utiliser **Job.join()** pour les attendre à la fin.

```
val request = launch { this: CoroutineScope

    repeat( times: 3) { i -> // lance 3 jobs enfant
        launch { this: CoroutineScope
            delay( timeMillis: (i + 1) * 200L) // variable delay 200ms, 400ms, 600ms
            println("Coroutine $i est terminée")
        }
    }
    println("J'ai terminé et ne join() pas les jobs de mes enfants qui sont encores actifs")
}
request.join() // attendre la fin de la demande, y compris tous ses enfants
println("Le traitement de la demande est maintenant terminé")
```

J'ai terminé et ne join() pas les jobs de mes enfants qui sont encores actifs

Coroutine 0 est terminée

Coroutine 1 est terminée

Coroutine 2 est terminée

Le traitement de la demande est maintenant terminé

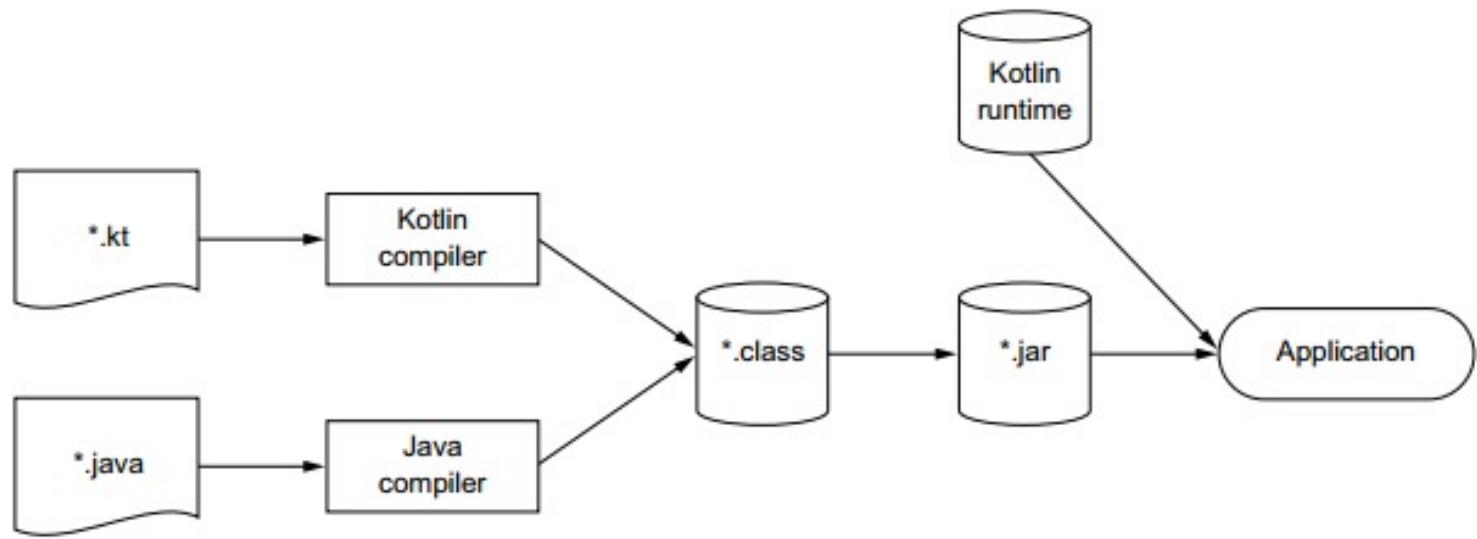


# Interopérabilité Java/Kotlin

Kotlin a été conçu avec pour base l'interopérabilité Java



# Rappel concernant la compilation



Que ce soit un fichier Kotlin ou Java, le compilateur générera des **.class** pour la VM Java.



# Appeler du code Java depuis Kotlin

- Le code Java existant peut être appelé à partir de Kotlin de manière naturelle, et le code Kotlin peut également être utilisé à partir de Java assez facilement.

```
fun demo(source: List<Int>) {  
    val list = ArrayList<Int>()  
    // 'for'-loops work for Java collections:  
    for (item in source) {  
        list.add(item)  
    }  
    // Operator conventions work as well:  
    for (i in source.indices) {  
        list[i] = source[i] // get and set are called  
    }  
}
```



# Java depuis Kotlin : propriétés

- Les méthodes qui suivent les conventions Java pour les getters et les setters sont représentées comme des propriétés dans Kotlin.
- Les méthodes d'accès booléennes (où le nom du getter commence par **is** et le nom du setter commence par **set**) sont représentées comme des propriétés qui ont le même nom que la méthode getter.
- Notez que si la classe Java n'a qu'un setter, elle ne sera pas visible en tant que propriété dans Kotlin, car Kotlin ne prend pas en charge les propriétés définies uniquement pour le moment.

```
val calendar = Calendar.getInstance()
if (calendar.firstDayOfWeek == Calendar.SUNDAY) // call getFirstDayOfWeek()
    calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
if (!calendar.isLenient) // call isLenient()
    calendar.isLenient = true // call setLenient()
```



# Java depuis Kotlin : void

- Si une méthode Java renvoie **void**, elle renverra **Unit** lorsqu'elle sera appelée depuis Kotlin.
- Si, par hasard, quelqu'un utilise cette valeur de retour, elle sera attribuée au site d'appel par le compilateur Kotlin, car la valeur elle-même est connue à l'avance (étant Unit).



# Java depuis Kotlin : Null-Safety

- Toute référence en Java peut être nulle, ce qui rend null-safety impraticables pour les objets provenant de Java.
- Les types de déclarations Java sont traités spécialement dans Kotlin et appelés types de plateforme. Les vérifications de **null** sont assouplies pour ces types, de sorte que les garanties de sécurité pour eux soient les mêmes qu'en Java.

Supposons que **item** provienne du Java :

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```



# Appeler du code Kotlin depuis Java

- 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.



# Kotlin depuis 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 **private**, avec le même nom que le nom de la propriété (uniquement pour les propriétés avec des backing fields)
- 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 aux valeurs booléennes.



# Kotlin depuis Java : package-level functions

- Toutes les fonctions et propriétés déclarées dans un fichier **app.kt** à l'intérieur d'un package **org.example**, y compris les fonctions d'extension, sont compilées dans des méthodes statiques d'une classe Java nommée **org.example.AppKt**.

```
// app.kt
package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```



# Kotlin et Android

Jetbrain a aussi pensé aux développeurs Android



# Android KTX

- Android KTX est un ensemble d'extensions Kotlin incluses avec Android Jetpack et d'autres bibliothèques Android.
- Pour ce faire, ces extensions exploitent plusieurs fonctionnalités du langage Kotlin, notamment les suivantes:
  - Fonctions d'extension
  - Propriétés d'extension
  - Lambdas
  - Paramètres nommés
  - Valeurs par défaut des paramètres
  - Coroutines
- Liste des extensions KTX :
  - <https://developer.android.com/kotlin/ktx/extensions-list>

