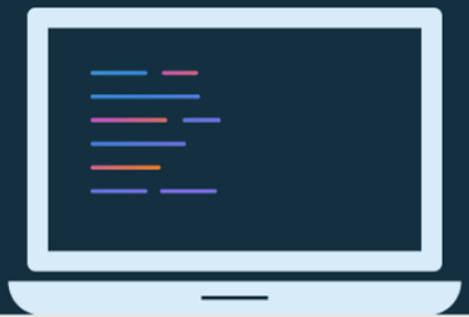


Kotlin 2 : les fonctions

(Presque) Tout à une valeur

(Preque) Tout à une valeur

En Kotlin, Presque tout est une expression et a une Valeur. Même un if !

```
val temperature = 20
val isHot = if (temperature > 40) true else false
println(isHot)
⇒ false
```

Valeur d’expressions

Des fois, la Valeur est `kotlin.Unit`.

```
val isUnit = println("This is an expression")

println(isUnit)
```

```
⇒ This is an expression
  kotlin.Unit
```

Fonctions en Kotlin

Les fonctions

- Un bloc de code qui permet de faire une tâche spécifique
- Permet de diviser le programme en plusieurs éléments plus petits
- Déclarer grâce au mot clé **fun**
- Peut recevoir des arguments qui peuvent être nommés ou avec Valeur par défaut

Éléments d’une fonction

Fonction simple qui utilise une autre fonction pour afficher « Hello World » dans la console

```
fun printHello() {
    println("Hello World")
}

printHello()
```

Fonction retournant Unit

Si une function ne retourne pas de Valeur utile, son type de retour est `Unit`.

```
fun printHello(name: String?): Unit {
    println("Hi there!")
}
```

`Unit` est un type avec une seule Valeur : `Unit`.

Fonction retournant Unit

Le type de retour `Unit` est optionnel.

```
fun printHello(name: String?): Unit {
    println("Hi there!")
}
```

est equivalent à :

```
fun printHello(name: String?) {
    println("Hi there!")
}
```

Arguments de fonctions

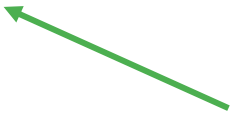
Les fonctions peuvent avoir :

- Paramètres par défaut
- Paramètres requis
- Paramètres nommés

Paramètre par défaut

Permet de définir une valeur par défaut si aucune valeur n'est passée à l'appel.

```
fun drive(speed: String = "fast") {
    println("driving $speed")
}
```



Utiliser = après le type pour définir la valeur

```
drive() ⇒ driving fast

drive("slow") ⇒ driving slowly

drive(speed = "turtle-like") ⇒ driving turtle-like
```

Paramètre requis

Si aucune Valeur par défaut n'est renseigné, l'argument est donc requis.


```
fun tempToday(day: String, temp: Int) {
    println("Today is $day and it's $temp degrees.")
}
```

Default versus required parameters

Les fonctions peuvent mélangés des paramètres par défaut et requis.

```
fun reformat(str: String,
             divideByCamelHumps: Boolean,
             wordSeparator: Char,
             normalizeCase: Boolean = true){
```

A une Valeur par défaut

Passer les arguments requis

```
reformat("Today is a day like no other day", false, '_')
```

Arguments nommés

Pour améliorer la lisibilité, on utilise les arguments nommés pour les arguments requis

```
reformat(str, divideByCamelHumps = false, wordSeparator = '_')
```

Placer les paramètres par défaut après les paramètres requis permet de n'avoir à specifier que ces derniers sans avoir à les nommer.

Fonctions compactes

Single-expression functions

Les fonctions compactes ou avec une seule expression (single-expression functions) permet de rendre le code plus concis.

```
fun double(x: Int): Int {
    x * 2
}
```

Version complète

```
fun double(x: Int):Int = x * 2
```

Version compacte

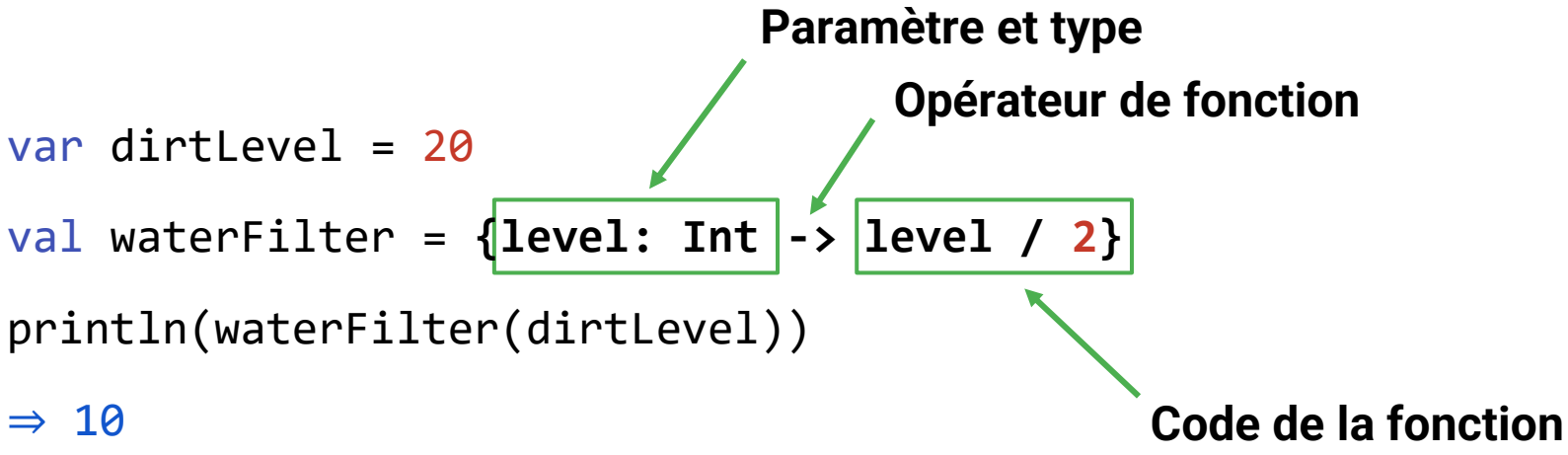
Lambdas et fonctions d’ordre supérieur

Les fonctions en Kotlin sont importantes

- Elles peuvent être stockées dans des variables ou structures de données
- Elles peuvent être passées en argument et/ou retournées d'autres fonctions
- Permet de redéfinir des fonctions à partir d'autres fonctions

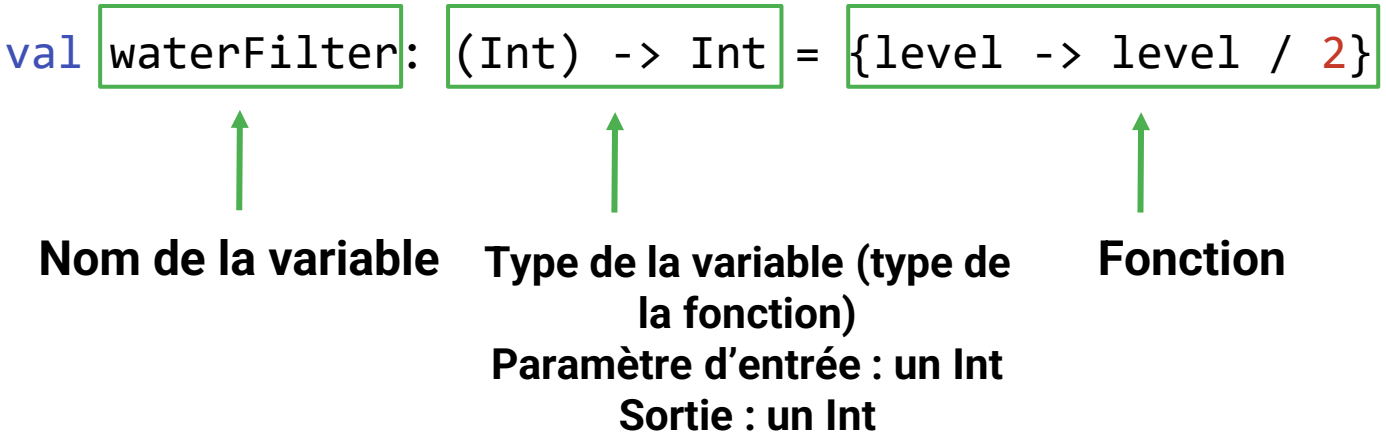
Fonctions lambdas

Une fonction lambda est une expression qui permet de définir une fonction sans nom



Syntaxe pour les types de fonctions

La syntaxe de Kotlin pour les types de fonctions est liée à comment sont declares les lambdas. Déclaration d'une variable qui contient une fonction.



Fonction d’ordre supérieur

Ce sont les fonctions qui prennent en paramètres ou retourne une function.

```
fun encodeMsg(msg: String, encode: (String) -> String): String {
    return encode(msg)
}
```

Le corps de cette fonction appelle la fonction passé en paramètre et lui passe sont premier argument.

Fonction d'ordre supérieur

Pour utiliser cette fonction, il faut lui passer un string et une fonction.

```
val enc1: (String) -> String = { input -> input.toUpperCase() }

println(encodeMsg("abc", enc1))
```

Utiliser un type de function permet de séparer l'implementation de l'usage. Ici déporter l'encodage du string.

Passer une référence de fonction

Utilisez l'opérateur :: pour passer une fonction nommée en argument à une autre fonction.

```
fun enc2(input:String): String = input.reversed()

encodeMessage("abc", ::enc2)
```

On passe une fonction nommé et pas une lambda

L'opérateur :: permet de définir qu'on veut passer la reference de la function et non pas l'appeler.

Syntaxe pour dernier paramètre d'appel

Kotlin préfère qu'un paramètre qui prend une fonction soir le dernier paramètre.

```
encodeMessage("acronym", { input -> input.toUpperCase() })
```

Cela permet de sortir le lambda des parenthèses.

```
encodeMsg("acronym") { input -> input.toUpperCase() }
```

Utiliser les fonctions d'ordre supérieur

La plupart des fonctions de bases de Kotlin utilise cette syntaxe.

```
inline fun repeat(times: Int, action: (Int) -> Unit)
```

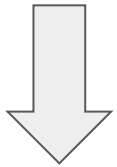
```
repeat(3) {
    println("Hello")
}
```

Filtres de listes

Filtres de listes

Permet de garder qu'une partie de la liste suivant une condition.

red	red-orange	dark red	orange	bright orange	saffron



Appliquer le `filter()` sur la liste.
Condition: l'élément contient "red"

red	red-orange	dark red

Itérer sur une liste

Si un litéral de fonction n'a qu'un seul paramètre, on peut omettre dans la déclaration le « -> ». Le paramètre aura pour nom par défaut **it** .

```
val ints = listOf(1, 2, 3)
ints.filter { it > 0 }
```

Le filtre s'applique sur une collection, **it** est la valeur de l'élément sur une iteration. C'est equivalent à :

```
ints.filter { n: Int -> n > 0 }   OR   ints.filter { n -> n > 0 }
```

Filtres de listes

La condition du filtre entre accolades `{}` est testé sur chaque élément de la liste. Si l'epression retourne `true`, l'élément est inclus.

```
val books = listOf("nature", "biology", "birds")
println(books.filter { it[0] == 'b' })
⇒ [biology, birds]
```

D’autres transformations de listes

- `map()` applique une transformation sur chaque item et renvoie la liste.

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
=> [3, 6, 9]
```

- `flatten()` retourne une liste à partir d’une liste de listes d’éléments.

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5), setOf(1, 2))
println(numberSets.flatten())
=> [1, 2, 3, 4, 5, 1, 2]
```