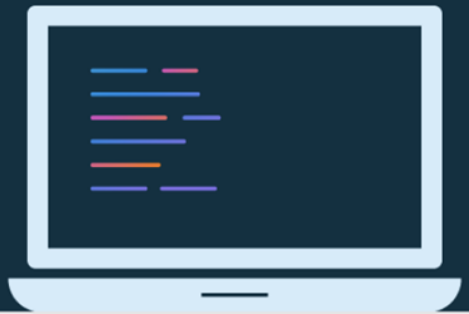
Les bases de Kotlin



Opérateurs

Opérateurs

- Mathématiques
- Incrémenter/décrémenter
- Comparaisons
- Assignement
- Égalité

```
+  -  *  /  %

++  --

<  <=   >  >=

=

==  !=
```

Opérations avec entier (Int)

```
1 + 1      =>      2

53 - 3     =>      50

50 / 10    =>      5

9 % 3      =>      0
```

Opérations avec des Doubles

```
1.0 / 2.0    =>    0.5

2.0 * 3.5    =>    7.0
```

Opérateurs mathématiques

```
1 + 1
⇒ kotlin.Int = 2
```

```
1.0 / 2.0
⇒ kotlin.Double = 0.5
```

```
53 - 3
⇒ kotlin.Int = 50
```

```
2.0 * 3.5
⇒ kotlin.Double = 7.0
```

```
50 / 10
⇒ kotlin.Int = 5
```

⇒ sortie du code et son type de sortie (`kotlin.Int`).

Méthodes numériques pour les nombres

Kotlin traite les nombres comme des primitives mais permet d'appeler des méthodes sur ceux-ci comme des objets.

```
2.times(3)
⇒ kotlin.Int = 6

3.5.plus(4)
⇒ kotlin.Double = 7.5

2.4.div(2)
⇒ kotlin.Double = 1.2
```

Types de données

Nombres entier

Type	Bits	Notes
Long	64	De -2^{63} to $2^{63}-1$
Int	32	De -2^{31} to $2^{31}-1$
Short	16	De -32768 to 32767
Byte	8	De -128 to 127

Autres types

Type	Bits	Notes
Double	64	16 - 17 significant digits
Float	32	6 - 7 significant digits
Char	16	16-bit caractère Unicode
Boolean	8	True or false -> vrai ou faux Opérations : || -> ou && -> et ! -> négation

Types des opérandes

Le résultat garde le type d'opérande

```
6*50
⇒ kotlin.Int = 300
```

```
1/2
⇒ kotlin.Int = 0
```

```
6.0*50.0
⇒ kotlin.Double = 300.0
```

```
1.0*2.0
⇒ kotlin.Double = 0.5
```

```
6.0*50
⇒ kotlin.Double = 300.0
```

Type casting

Assigner un `Int` à un `Byte`

```
val i: Int = 6
val b: Byte = i
println(b)
```

```
⇒ error: type mismatch: inferred type is Int but Byte was
expected
```

Convertir `Int` en `Byte` avec le casting

```
val i: Int = 6
println(i.toByte())
```

```
⇒ 6
```

Grands nombres

Utiliser un underscore pour faire des espaces dans les grands nombres.

```
val oneMillion = 1_000_000

val idNumber = 999_99_9999L

val hexBytes = 0xFF_EC_DE_5E

val bytes = 0b11010010_01101001_10010100_10010010
```

Strings (chaîne de caractères)

Les Strings sont une sequence de caractères entre guillemets.

```
val s1 = "Hello world!"
```

Les String literals peuvent contenir des caractères d'échapements.

```
val s2 = "Hello world!\n"
```

On peut écrire une chaîne sur plusieurs lignes avec triple guillements (""")

```
val text = """
    var bikes = 50
"""
```

Concaténation de String et templates

```
val numberOfDogs = 3
val numberOfCats = 2

”J’ai $numberOfDogs chiens" + " et $numberOfCats chats"

=> J’ai 3 chiens et 2 chats
```

String templates

Un template expression commence par un dollar ($) et un nom de variable:

```
val i = 10
println("i = $i")
=> i = 10
```

Ou une expression entre accolade :

```
val s = "abc"
println("$s.length égal ${s.length}")
=> abc.length égal 3
```

String template expressions

```
val numberOfShirts = 10
val numberOfPants = 5

”J’ai ${numberOfShirts + numberOfPants} vêtements"

=> J’ai 15 items vêtements
```

Variables

Variables

- Puissante inference de type
  - Le compilateur peut deviner le type
  - Sinon, on peut le specifier
- Muable ou immuable
  - Imuabilité pas force mais recommender

Kotlin est un language à typage statique (statically-typed language). Le type est résolue à la compilation et ne peut pas changer.

Conditionnel

Control flow

- if/else if/else
- when
- Boucle for
- Boucle while

if/else

```
val numberOfCups = 30
val numberOfPlates = 50

if (numberOfCups > numberOfPlates) {
    println("Too many cups!")
} else {
    println("Not enough cups!")
}
```

```
=> Not enough cups!
```

Expression if avec plusieurs cas

```
val guests = 30
if (guests == 0) {
    println("No guests")
} else if (guests < 20) {
    println("Small group of people")
} else {
    println("Large group of people!")
}
⇒ Large group of people!
```

Ranges

- Type de données contenant une suite de valeurs qui sont comparables entre eux (par exemple les entiers de 1 à 100)
- Les Ranges sont bornées (inclusive par défaut)
- Les objets dans une range peuvent être muables ou immuables

Ranges comme condition

```
val numberOfStudents = 50
if (numberOfStudents in 1..100) {
    println(numberOfStudents)
}
=> 50
```

Note: Pas d'espace autour de l'opérateur "range to" `(1..100)`

Déclaration when

```
when (results) {
    0  -> println("No results")
    in 1..39 -> println("Got results!")
    else -> println("That's a lot of results!")
}
⇒ That's a lot of results!
```

Tel que la declaration `if`, on peut aussi définir une expression `when` qui retourne une Valeur.

Boucle for

```
val pets = arrayOf("dog", "cat", "canary")
for (element in pets) {
    print(element + " ")
}
⇒ dog cat canary
```

Itère sur une liste d'éléments. Pas besoin de déclarer un itérateur ou d'incrémenter une variable.

Boucle for avec index et élément

```
for ((index, element) in pets.withIndex()) {
    println("Item at $index is $element\n")
}
⇒ Item at 0 is dog
Item at 1 is cat
Item at 2 is canary
```

Boucle for : pas, taille et range

```
for (i in 1..5) print(i)
⇒ 12345

for (i in 5 downTo 1) print(i)
⇒ 54321

for (i in 3..6 step 2) print(i)
⇒ 35

for (i in 'd'..'g') print (i)
⇒ defg
```

Boucle while

```
var bicycles = 0
while (bicycles < 50) {
    bicycles++
}
println("$bicycles bicycles in the bicycle rack\n")
⇒ 50 bicycles in the bicycle rack

do {
    bicycles--
} while (bicycles > 50)
println("$bicycles bicycles in the bicycle rack\n")
⇒ 49 bicycles in the bicycle rack
```

repeat loops

```
repeat(2) {
      print("Hello!")
}
```

```
⇒ Hello!Hello!
```

Listes et tableaux

Listes

- Les listes sont une collection d’éléments ordonnées
- Éléments accessibles programmatiquement par leurs index
- Les éléments ne sont pas forcéments uniques

An example of a list is a sentence: it's a group of words, their order is important, and they can repeat.

Liste immuable avec listOf()

Déclarer une liste avec `listOf()` et l'afficher

```
val instruments = listOf("trumpet", "piano", "violin")
println(instruments)
```

```
⇒ [trumpet, piano, violin]
```

Liste muable avec mutableListOf()

On peut créer une liste modifiable avec `mutableListOf()`

```
val myList = mutableListOf("trumpet", "piano", "violin")
myList.remove("violin")

⇒ kotlin.Boolean = true
```

Avec val, on ne peut pas changer la reference de la liste mais on peut quand même changer son contenu.

Arrays (tableaux)

- Arrays store multiple items
- On peut accéder au éléments grâce à leurs indices.

- On peut changer les éléments
- La taille est fixe

Array avec arrayOf()

Création d'un array avec `arrayOf()`

```
val pets = arrayOf("dog", "cat", "canary")
println(Arrays.toString(pets))
⇒ [dog, cat, canary]
```

Comme les listes, `val` ne permet pas de modifier la reference du tabkleau mais permet de changer les donneés.

Arrays avec des mélanges de types

Un array peut avoir des types de plusieurs éléments.

```
val mix = arrayOf("hats", 2)
```

Un array peut être limité à un type.

```
val numbers = intArrayOf(1, 2, 3)
```

Combiner des arrays

Utiliser l'opérateur +.

```
val numbers = intArrayOf(1,2,3)
val numbers2 = intArrayOf(4,5,6)
val combined = numbers2 + numbers
println(Arrays.toString(combined))

=> [4, 5, 6, 1, 2, 3]
```

Null safety

Null safety

- En Kotlin, les variables ne peuvent pas être null par défaut
- On peut specifier qu'une variable peut être null avec l'opérateur `?`
- Permettre les null-pointer exceptions avec l'opérateur `!!`
- On peut tester si une variable est null avec l'opérateur elvis `?:`

Les variables ne peuvent pas être null

Par défaut, la valeur `null` n'est pas premise.

Déclarer un entier `Int` et assigner `null` à celui-ci.

```
var numberOfBooks: Int = null
```

```
⇒ error: null can not be a value of a non-null type Int
```

L’opérateur Safe call

L’opérateur safe call `?` après le type permet d’indiquer qu’une variable peut être null.

Déclarer un `Int?` comme nullable

```
var numberOfBooks: Int? = null
```

En general, il vaut mieux éviter de définir une variable nullable quand c’est possible.

Tester pour un null

Vérifier que la variable `numberOfBooks` n'est pas `null`. Puis la décrémenter.

```
var numberOfBooks = 6
if (numberOfBooks != null) {
    numberOfBooks = numberOfBooks.dec()
}
```

Avec Kotlin en utilisant l'opérateur safe call :

```
var numberOfBooks = 6
numberOfBooks = numberOfBooks?.dec()
```

L’opérateur !!

Si on est sûr qu’une variable ne sera pas null, on peut utiliser `!!` Pour forcer la variable en non-null. Ce qui permet ensuite d’appeler des méthodes/propriétés.

```
val len = s!!.length
```

Si s est nulle lance une exception NullPointerException

Attention : Comme `!!` peut renvoyer une exception, à n’utilizer que quand on est sûr que ce ne sera pas null.

Opérateur Elvis

On peut mettre l'opérateur `?:` après un test de nullabilité ou l'opérateur safe call.

```
numberOfBooks = numberOfBooks?.dec() ?: 0
```

L'opérateur `?:` s'appelle Elvis car il ressemble à un smiley avec la coupe d'Elvis Presley avec sa coupe de cheveux.