Introduction Kotlin

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

Organisation

android

- Objectifs pédagogiques
  - Acquérir les concepts du développement Android en Kotlin
  - Maîtriser les bonnes pratiques du développement d'une application mobile
- Supports de cours envoyés
- 3 Séances théories + TP
- Évaluation des TP + Examen final

Kotlin

- Langage de programmation **orienté objet** et **fonctionnel**
- Typage statique et inféré
- Compiler pour la JVM, JavaScript, et plusieurs plateformes en natif
- Langage officiel d'Android depuis 2019

Pourquoi Kotlin

android

- **Expressivité / concision** => ~30% de code en moins que Java
- **Typage sécurisé** => Null safety
- **Portabilité / compatibilité**
- **Communauté**
- **Android**

Hello, world!

```
fun main(args: Array<String>) {
    println("Hello, world!")
}

fun main() {
    println("Hello, world!")
}

fun main() = println("Hello, world!")
```

Les bases

android

Mutability

```
var mutableString: String = “Toto”
val immutableString: String = “Toto”
val inferredString = “Toto”
```

Numbers

```
var intNum = 10
val doubleNum = 10.0
val longNum = 10L
val floatNum = 10.0F
```

Booleans

```
var trueBoolean = true
val falseBoolean = false
```

Strings

```
var name = “Toto”
val greeting = “Hello, ” + name
val greetingTemplate = “Hello, $name”
val interpolated = “Hello, ${name.toUpperCase()}”
```

Null Safety

```
val cannotBeNull: String = null // Compile error
val canBeNull: String? = null // Valid

val cannotBeNull: Int= null // Compile error
val canBeNull: Int? = null // Valid
```

Safe Operator

```
val nullableLength: Int? = nullableString?.length
val chiefName: String? = person?.department?.head?.name
```

Elvis Operator

```
val nonNullLength: Int = nullableString?.length ?: 0
val chiefName: String? = person?.department?.head?.name
```

Control flow

If/Else

```
val guests = 30

if (guests == 0) {
        println(“No guests”)
} else if (guests < 20) {
        println(“Small group”)
} else {
        println(“Large group”)
}
```

```
val isEven = if (num % 2 == 0) true else false
```

When

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

```
val sign = when(x) {
    0 -> “Zero”
    in 1..4 -> “Four or less”
    else -> “Other numbers”
}
```

For

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

```
println(pets.joinToString(“ “))
```

Classes

android

Définir et utiliser une classe

```
class House {

  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}

  ...
}
```

```
val myHouse = House()
println(myHouse)
```

Exemples de constructeurs

```
class A
```

```
class B(x: Int)
```
⇒ x existe seulement dans le scope du constructeur

```
class C(val y: Int)
```
⇒ y existe dans toutes les instances de la classe

```
val aa = A()
```

```
val bb = B(12)
println(bb.x)
```
⇒ Erreur de compilation, x non résolu

```
val cc = C(42)
println(cc.y)
```
⇒ 42

Paramètres par défaut

Les instances peuvent avoir des valeurs par défaut.

- Permet de réduire le nombre de constructeurs
- Les valeurs par défaut peuvent être mélangés avec les paramètres obligatoires
- On peut nommer les paramètres lors de l'instanciation

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
val box4 = Box(length = 100, height = 40)
```

Bloc d’initialisation

Le code d’initialisation nécessaire est exécuté dans un bloc *init* spécial

- Plusieurs blocs *init* sont autorisés
- Représente le corps du constructeur primaire

```
class Square(val side: Int) {
    init {
        println(side * 2)
    }
}

val s = Square(10)
=> 20
```

Plusieurs constructeurs

```
class Circle(val radius:Double) { // Constructeur primaire

    constructor(name:String) : this(1.0) // Constructeur secondaire

    constructor(diameter:Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }

    init {
        ...
    }
}

val c = Circle(3)
```

Propriétés

```
class Person(var name: String)

fun main() {
    val person = Person("Jean")
    println(person.name)  // Getter
    person.name = "Vincent"  // Setter
    println(person.name)
}
```

Pour changer le comportement par défaut des propriétés :

- Override *get()*
- Override set() si la propriété est *mutable*

```
var name: String = “Toto”
    get() = “My name is $field”
    set(value) {
        field = value.toUpperCase()
    }
```

Héritage

- En kotlin, l’héritage des classes est monoparental.
- Chaque classe a exactement une classe parente, appelée superclasse.
- Chaque sous-classe hérite de tous les membres de sa superclasse, y compris ceux dont la superclasse a elle-même hérité.

- Il est possible d’implémenter autant d’interface que l’on souhaite.

Héritage

Mot clé *open* pour déclarer une classe héritable, par défaut les classes sont *“final”*.

```
open class C


class D : C()
```

```
abstract class Food {
    abstract val kcal : Int
    abstract val name : String
    fun consume() = println("I'm eating ${name}")
}

class Pizza() : Food() {
    override val kcal = 600
    override val name = "Pizza"
}

fun main() {
    Pizza().consume()    // "I'm eating Pizza"
}
```

Fonctions d’extension

- Donne l’impression d’avoir été ajouté par l’auteur de la classe.
- Ne modifie pas réellement la classe existante.
- Ne peut pas accéder aux variables d’instance privées.
- Permet d’ajouter des fonctionnalités aux classes qui ne sont pas ouvertes ou qui ne nous appartiennent pas

```
Format: fun ClassName.functionName( params ) { body }
```

```
fun String.removeFirstLastChar() : String =
    this.substring(1, this.length - 1)
```

Data class

- Classe spéciale qui existe uniquement pour stocker un ensemble de données
- Génère des getters pour chaque propriété (et des setters pour les propriétés mutables)
- Génère les méthodes *toString()*, *equals()*, *hashCode()*, *copy()* et les opérateurs de déstructuration
- Permet d'ajouter des fonctionnalités aux classes qui ne sont pas ouvertes ou qui ne nous appartiennent pas

Format: `data class <NameOfClass>( parameterList )`

```
data class Player(val name: String, val score: Int)
```

```
val (name, score) = player
```

→ Destructuration