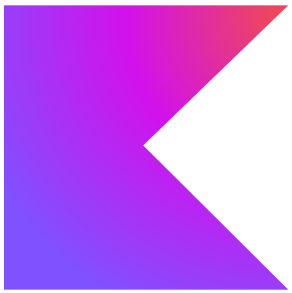

Kotlin

IUT Clermont Auvergne

Janvier 2024

L'Objet en Kotlin

Classes et objets

- Squelette de classe standard en Java (POJO)

```java
class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public String getAge() { return age; }
  public void setAge(int age) { this.age = age; }

  public int hashCode() { … }
  public boolean equals() { … }

  @Override
  public String toString() {
    return "Person(name=" + name + ", age=" + age + ")";
  }
}
```

Classes et objets

- L'équivalent en Kotlin

```
class Person(var name: String, var age: Int) {
  override fun int hashCode() : Int { … }
  override fun equals() : Boolean { … }

  override fun toString() = "Person(name=$name, age=$age)"
}
```

- Instanciation : pas besoin de `new`

```
val moi = Person("Laurent Provot", 27)
```

Visibilités

Modifieur	Package (top level)	Classe
`public`	partout	à tout ceux qui voient la classe
`private`	au sein du fichier contenant la déclaration	au sein de la classe uniquement
`internal`	dans le même module	ceux dans le même module qui voient la classe
`protected`	—	comme `private` + dans les classes dérivées

- Module : ensemble de fichiers Kotlin compilés ensemble
  - pour IntelliJ IDEA = module
  - pour Android = Gradle source set

Constructeurs (Ctor)

1. Constructeur principal

- On ne peut spécifier de code tout de suite

```
class Person constructor(name: String) {…}
class Person(name: String) {…}
```

- Bloc d'initialisation
- On peut utiliser les paramètres du Ctor principal dans les blocs d'initialisation et les initialisations d'attributs
- Initialisations effectuées dans l'ordre déclaré

```
class Person(name: String) {
  private val nameUpper = name.uppercase()
  init {
    println("My name is: $name. What? I said $nameUpper")
  }
}
```

Constructeurs (Ctor)

2 Constructeurs secondaires

```
class Person(name: String) {
  private val upperName = name.uppercase()
  private var age = 0

  constructor(name: String, age: Int) : this(name) {
    this.age = age
  }

  constructor(codename: Int) : this(decipher(codename), 42)
}
```

- Délégation au Ctor principal *obligatoire*
- Tous les blocs d'initialisation sont appelés d'abord

Constructeurs (Ctor)

- Si annotations ou modificateur (visibilité, ...)

```
class Person private constructor(name: String) {
  …
}

class Person @Inject constructor(name: String) {
  …
}
```

Propriétés

- Attribut déclaré avec `val` = propriété en lecture seule
- Attribut déclaré avec `var` = propriété en lecture / écriture

```
class Person {
  var name = "John Doe";
}
john = Person();
```

- Accès : `john.name` (utilisation du getter)
- Modification : `john.name = "johnny"` (utilisation du setter)
- Pour initialisation de propriétés dans Ctor

```
class Person(val firstName: String, var age: Int) { … }
```

Propriétés & getter / setter

- Syntaxe complète

```
var <propertyName>[: PropertyType] [= <initializer>]
    [<getter>]
    [<setter>]
```

- Propriété personnalisée : accès à l'attribut avec `field`

```
var speed: Int
    get() = field * 100;
    set(value) {
      if (value >= 0) field = value;
    }
```

Propriétés & getter / setter

- Propriétés calculées
- Attribut (*backing field*) fourni seulement si nécessaire

```
val isEmpty: Boolean
    get() = this.size == 0
```

- Changement de visibilité ou ajout d'annotation en conservant l'implémentation par défaut

```
var devOnly: String
    @NotNull get
    private set
```

Classes de données

- Rappel : squelette de classe standard en Java (POJO)

```java
class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public String getAge() { return age; }
  public void setAge(int age) { this.age = age; }

  public int hashCode() { … }
  public boolean equals() { … }

  @Override
  public String toString() {
    return "Person(name=" + name + ", age=" + age + ")";
  }
}
```

Classes de données

- L'équivalent (encore plus court) en Kotlin

```kotlin
data class Person(var name: String, var age: Int)
```

- \o/ sympa… mais contraintes :
  - Ne peut pas être `abstract`, `open`, `sealed`, `inner`
  - Ctor principal doit avoir au moins un paramètre
  - Les paramètres doivent tous être marqués `val` ou `var`
  - `equals()`, `hashCode()`, `toString()` générées automatiquement sauf si présentes explicitement

Classes de données

```kotlin
data class Person(val name: String, val age: Int)
```

- En plus des propriétés, `toString()` et `equals()` / `hashCode()`
  - `copy()`
  - `component1()`, …, `componentN()`

- Méthode `copy` :

```kotlin
val john = Person(name = "John", age = 42)
val youngJohn = john.copy(age = 22)
```

Classe de données

- Méthodes `component` : pour déstructuration de la classe
- Une pour chaque propriété du Ctor principal, dans l'ordre de déclaration
- Underscore possible si paramètre non utilisé

```
println(john.component1()) // affiche "John"
val (name, age) = john
println("$name, $age years old") // affiche "John, 42 years old"
```

- Sympa pour les retours de fonctions, Map, itérations avec index

Héritage

- Hiérarchie basée sur la classe `Any` ($\neq$ `java.lang.Object`)
- Autorisation explicite de l'héritage avec `open` (l'opposé de `final` en Java)

```
open class Base(arg: String)
class Derived(num: Int) : Base(num.toString())
```

- Ctor secondaires : appel obligatoire de `super` ou délégation à un autre Ctor

```
class Derived : Base {
  constructor(arg: String, num: Int) : super(arg) {…}
  constructor(arg: String) : this(arg, 42)
}
```

Redéfinition

- Redéfinition de méthode : explicite avec `override`

```
open class Base {
  open fun fo() {…}
  fun fc() {…}
}
class Derived() : Base() {
  override fun fo() {…}
}
```

- Un membre déclaré `override` est automatiquement `open`
- Si non désiré : le marquer `final`

Redéfinition

- Redéfinition de propriété : pareil que pour les méthodes
- On peut redéfinir un `val` en `var`
- `override` peut être utilisé dans le constructeur principal

```
open class Base {
  open val x: Int = 0
}

class Derived : Base() {
  override var x: Int = 42
}

class AnotherDerived(override val x: Int) : Base()
```

Redéfinition

- Comme en Java, le code de la classe dérivée peut appeler des méthodes/propriétés de sa classe de base grâce au mot clé `super`
- Classe, méthode et propriété abstraites : déclarées avec le mot clé `abstract`
- `abstract` implique `open`

```
abstract class Base {
  abstract val arg: String
}

class Derived : Base {
  override val arg: String
    get() = "Something"
}
```

Sealed class

- Permet de représenter une hiérarchie contrainte de classes
- On connaît l'ensemble complet des classes dérivées à la compilation du module
- Classe abstraite déclarée avec le mot-clé `sealed`
- Les classes dérivées doivent être définies dans le même package

```
sealed class Feuillage
class Caduque(val date: LocalDate) : Feuillage()
class Persistant(val info: String) : Feuillage()

fun info(arbre: Arbre) {
    val type = arbre.feuillage
    when(type) {
        is Caduque -> println("Perds ses feuilles le ${type.date}")
        is Persistant -> println("Infos : ${type.info}")
        // pas besoin de else, tous les cas sont couverts
    }
}
```

Classe imbriqués

- Possibilité de déclarer des classes dans des classes
- Améliore l'encapsulation

1. *Nested class*

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

Classe interne

- Si la classe imbriqué à besoin d'accéder aux attributs de sa classe externe

2. *Inner class*

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Interfaces

- Comme en Java : entité abstraite, ne peut pas contenir d'état
- Méthodes abstraite ou avec implémentation
- Peut contenir des propriétés sans *backing field*
  - ▷ soit abstraites
  - ▷ soit `val` avec getter implémenté

```
interface Named {
  val name: String
}
interface FullNamed : Named {
  val firstName: String
  val lastName: String
  override val name: String get() = "$firstName $lastName"
}
class Person(
  override val firstName: String
  override val lastName: String
) : FullNamed
```

Interfaces

- Désambiguïsation grâce à `super<...>` si implémentation dans l'interface

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

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

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

Interface fonctionnelle

- Interface avec une seule méthode abstraite (*SAM = Single Abstract Method*)
- Permet d'utiliser des lambdas au lieu d'objets dont le type implémente l'interface
- Introduite avec le mot-clé `fun`

```
fun interface IntPredicate {
    fun test(value: Int): Boolean
}

val isNegative = IntPredicat { it < 0 }
isNegative.test(-3);  // renvoie true
```

- Évidemment le type de la lambda doit correspondre au prototype de la fonction abstraite

object

- Généralisation des classes anonymes de Java
- Expression objet = création d'une instance unique avec certains comportements
- Utile pour un besoin unique ponctuel

```
interface MouseListener {
  fun mouseClicked(e: MouseEvent)
  fun mouseEntered(e: MouseEvent)
}
class Button {
  …
  fun addMouseListener(listener: MouseListener)
}
…
button.addMouseListener(object : MouseListener() {
  override fun mouseClicked(e: MouseEvent) { … }
  override fun mouseEntered(e: MouseEvent) { … }
})
```

Singleton

- Déclaration d'objet = singleton

```
object DatabaseManager {
  fun connect(conn: Connector) { … }
  fun fetchData() : Data { … }
}

fun main(args: String[]) {
  DatabaseManager.connect(…)
  println(DatabaseManager.fetchData())
}
```

- Déclaration global
- Initialisation thread-safe
- Ne peut pas être du côté droit d'une affectation

companion object

- `object` attaché à une classe
- Déclaration dans une classe possible avec le mot clé `companion`

```
class MyClass {
  companion object Factory {
    fun create(): MyClass = MyClass()
  }
}
val instance = MyClass.create()
```

- Utilisation « à la `static` » de Java (mais attention instancie un objet)
- Utilisation de `@JvmStatic` si on veut que cela génère des membres `static` dans le bytecode Java

`object`

- Les expressions objet sont exécutées *immédiatement*
- Les `companion object` sont créés au moment du chargement de leur classe d'attachement

Surcharge d'opérateur

- Il est possible de surcharger les opérateurs
- On redéfinit des méthodes spécifiques de la classe
- cf. Doc pour la liste exhaustive
- Elles sont marquées `operator`
- +, -, *, /, %, ..
  - ▷ `a + b` équivalent à `a.plus(b)`
  - ▷ `a..b` équivalent à `a.rangeTo(b)`
- `in`, `!in`
  - ▷ `a.contains(b)`

Surcharge d'opérateur

- Accès indexé `[]`
  - `a[i]` équivalent à `a.get(i)`
  - `a[i] = b` équivalent à `a.set(i, b)`
- Appel de méthode
  - `a(i, j)` équivalent à `a.invoke(i, j)`
- a == b
  - `a?.equals(b) ?: (b === null)`
- a > b, a < b, a >= b, a <= b
  - obtenus à partir de `a.compareTo(b)`