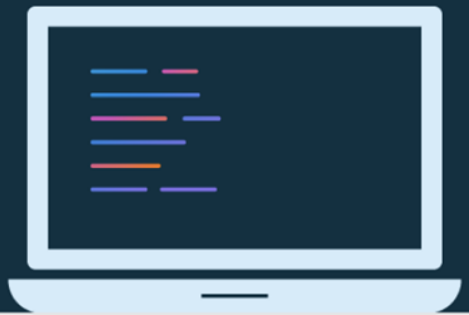
Kotlin 3 : Classes et objets

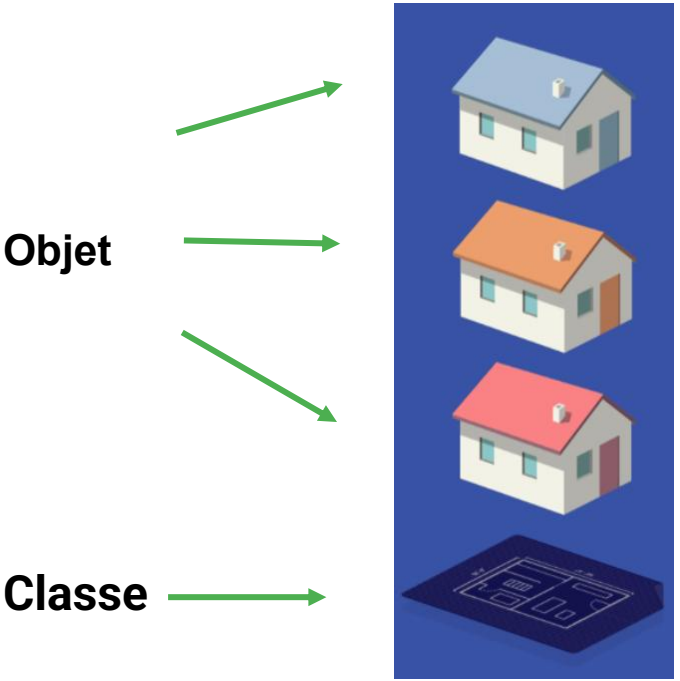


Sommaire

Classes

Classe

- Une classe est une sorte de blueprint, un patron de conception pour les objets
- Les classes définisses des méthodes qui s'opère sur les instances



Classe vs objet

Classe Maison

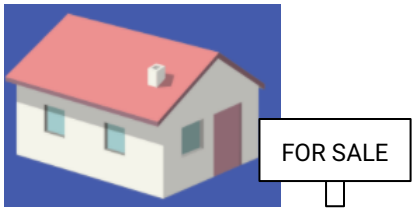
Données

- Couleur (String)
- Nombre de fenêtres (Int)
- Est en vente (Boolean)

Comportement

- `updateColor()`
- `putOnSale()`

Objets



Définir et utiliser une classe

Définition de classe

```
class House {
  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

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

Créer une nouvelle instance d'objet

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

Constructeurs

Quand un constructeur est défini dans le header de la classe, il peut contenir :

- Aucun paramètre

  ```
  class A
  ```

- Paramètres
  - Pas de `var` ou `val` → accessible seulement dans le scope du constructeur

    ```
    class B(x: Int)
    ```

  - Marqué avec `var` ou `val` → copie de la valeur accessible dans toute le classe

    ```
    class C(val y: Int)
    ```

Constructor examples

```
class A
```

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

```
class B(x: Int)
```

```
val bb = B(12)
println(bb.x)
=> compiler error unresolved
reference
```

```
class C(val y: Int)
```

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

Paramètres par défaut

Les instances de classes peuvent avoir des paramètres par défaut.

- Permet de réduire le nombre de constructeurs necessaire
- Permet d'être plus concis
- Comme les fonctions, on peut mélange paramètres requis et par défaut

```
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)
```

Constructeur principal

Déclare le constructeur principal dans le header de la classe.

```
class Circle(i: Int) {
    init {
        ...
    }
}
```

C'est techniquement equivalent à

```
class Circle {
    constructor(i: Int) {
        ...
    }
}
```

Bloc d’initialisation

- Tout code requis pour l’initialisation de la classe se passe dans le bloc `init`
- Plusieurs blocs `init` sont possible
- Le bloc `init` est le corps du constructeur principal

Bloc d’initialisation : exemple

Utiliser le mot clé `init` :

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

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

Plusieurs constructeurs

- Mot clé `constructor` pour définir des constructeurs secondaires
- Les constructeurs secondaires doivent :
  - Soit appelé le constructeur primaire avec le mot clé `this`
  - Soit un autre constructeur secondaire qui appelle le constructeur primaire
- Le corps de fonction du constructeur secondaire est optionnel

Plusieurs constructeurs

```
class Circle(val radius: Double) {
    constructor(name: String) : this(1.0)
    constructor(diameter: Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }
    init {
        println("Area: ${Math.PI * radius * radius}")
    }
}
val c = Circle(3)
```

Propriétés

- Définitions de propriétés avec `val` ou `var`
- Access these properties using dot . notation with property name
- Accès aux propiétés avec la notation nomInstance . nomDeLaPropriété
- Assignation avec la même notation (pour les `var` seulement)

Classe personne avec une propiété

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name)
    person.name = "Joey"
    println(person.name)
}
```

Accès avec .<property name>

Assigner avec .<property name>

Getters et setters personnalisés

Si besoin d'un autre comportement que le `get`/`set` par défaut :

- Surcharger (override) `get()`
- Surcharger `set()` (si c'est un `var`)

Format:

```
var propertyName: DataType = initialValue
        get() = ...
        set(value) {
            ...
        }
```

Getter personnalisé

```
class Person(val firstName: String, val lastName:String) {
    val fullName:String
        get() {
            return "$firstName $lastName"
        }
}

val person = Person("John", "Doe")
println(person.fullName)
=> John Doe
```

Setter personnalisé

```
var fullName:String = ""
    get() = "$firstName $lastName"
    set(value) {
        val components = value.split(" ")
        firstName = components[0]
        lastName = components[1]
        field = value
    }

person.fullName = "Jane Smith"
```

Méthodes

- Les classes peuvent avoir des fonctions appelées méthodes
- Se déclarent comme des fonctions classiques
- Elles s'appliquent sur l'objet avec lequel on fait l'appel

Héritage

Héritage

- Kotlin permet l'héritage que d'un seul parent
- La classe dont un autre classe hérite s'appelle la superclass ou classe mère
- Chaque sous-classe ou classe fille (subclass) hérite de tous les membres de la superclass, dont les membres hérités par la superclass elle-même

Si le fait d'hériter d'une seule classe est trop limitant, il est possible de passer par des interfaces car il n'y a pas de limites pour celles-ci.

Interfaces

- Permet de fournir un contrat qui doit être respecté par toutes les classes qui l'implémente
- Peut contenir des méthodes et des noms de propriétés
- Peut être dérivé d'autres interfaces

Format:

```
interface NomDeLinterface {
    ...
}
```

Exemple d'interface

```
interface Shape {
    fun computeArea() : Double
}

class Circle(val radius: Double) : Shape {
    override fun computeArea() = Math.PI * radius * radius
}


val c = Circle(3.0)
println(c.computeArea())
=> 28.274333882308138
```

Étendre une classe

Pour étendre une classe:

- Soit créer une classe qui hérite de la première
- Soit ajouté des fonctionnalités sans nouvelle classe mais grâce aux fonctions d’extensions

Créer une classe fille

- On ne peut pas hériter de classe par défaut
- Il faut utiliser le mot clé `open` pour le permettre
- Le mot clé `override` permet de redéfinir les propriétés et méthodes

Les classes sont finales par défaut

Déclarez une classe

```
class A
```

Tentez d'hériter de A

```
class B : A
```

```
=>Error: A is final and cannot be inherited from
```

Utilisation de `open`

Utilisez `open` pour declarer une classe héritable.

Déclarez une classe

```
open class C
```

Héritez de cette classe

```
class D : C()
```

Surcharger

- Le mot clé `open` doit être utilisé sur les propriétés et méthodes surchargeable

- Le mot clé `override` doit être utilisé quand on surchage une méthode ou prorpriété surchargeable

- Un membre marqué `override` peut être lui-même surchargé sauf si le mot clé `final` est utilisé en plus

Classe abstraites

- Une classe abstraite est signalé par le mot clé `abstract`
- Ne peut pas être instanciée, doit être hérité
- Ressemble à une interface, avec la possibilité de stocker un état
- Les propriétés et fonctions marquées `abstract` doivent être surcharges
- Peut avoir des members non-abstraits

Exemple de classe abstraite

```
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"
}
```

Quand les utiliser

- Pour définir un large spectres de comportements ou de types ? Interface
- Est-ce que le comportement sera spécifique à ce type ? Classe
- Besoin d'hériter de plusieurs classes ? Revoir le code pour déplacer certains comportements dans des interfaces
- Possibilité d'avoir des members hérités et d'autres définit dans les sous-classe ? Classe abstraite

Fonctions d’extension

Fonctions d’extension

Permet d’ajouter des fonctions à une classe qu’on ne peut pas modifier directement.

- Apparait comme si elle avait été ajouter à la classe
- La classe n’a pas été modifiée
- Ne peut pas accéder aux variables privées de l’instance

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

Pourquoi les fonctions d’extension ?

- Ajouter des fonctionnalités aux classes non héritables
- Ajouter des fonctionnalités aux classes non modifiables
- Séparer le coeur de l’API des méthodes d’aide pour les classes qu’on possède

Définissez les fonctions d’extension dans un endroit facilement trouvable, par exemple dans le même fichier ou avec un nom bien défini.

Exemple de function d'extension

Ajouter `isOdd()` à la classe `Int` :

```
fun Int.isOdd(): Boolean { return this % 2 == 1 }
```

Appelez `isOdd()` sur un `Int`:

```
3.isOdd()
```

Les fonctions d'extension de Kotlin sont très puissantes !

Classes spéciales

Data class

- Classe spéciale qui sert à stocker des données
- Utiliser le mot clé `data` avant `class`
- Génère des getters et setters (pour les var) automatiquement
- Génère les méthodes `toString()`, `equals()`, `hashCode()`, `copy()` et l'opérateur de destructuration

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

Exemple de data class

Définissez la data class :

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

Utilisez la data class :

```
val firstPlayer = Player("Lauren", 10)
println(firstPlayer)
=> Player(name=Lauren, score=10)
```

Les data class permettent de render le code plus concis

Pair et Triple

- `Pair` et `Triple` sont des classes prédéfinis qui permettent de stocké 2 ou 3 données respectivement
- Accès aux variables grâce à `.first`, `.second` et `.third` respectivement
- Préférer une data class (plus parlant grâce à son nom et celui des propriétés

Exemple pair et Triple

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")

println(bookAuthor)
=> (Harry Potter, J.K. Rowling)

val bookAuthorYear = Triple("Harry Potter", "J.K. Rowling", 1997)
println(bookAuthorYear)

println(bookAuthorYear.third)
=> (Harry Potter, J.K. Rowling, 1997)
    1997
```

Pair to

L'opérateur `to` de `Pair` permet d'omettre les virgules et parenthèses.
Cela permet un code plus lisible.
Permet de définir une relation clé / valeur

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K. Rowling)
```

Utilisé pour les map ou hashmap (clé/valeur)

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
=> map of Int to String {1=x, 2=y, 3=zz}
```

Classe Enum

Un type de donnée définie par l'utilisateur avec un ensemble de valeurs nommées possible

- Permet de restreindre les valeurs possible
- Défini par le mot clé `enum` suivi de `class`

Format: `enum class EnumName { NAME1, NAME2, … NAMEn }`

Référencé par `EnumName.<ConstantName>`

Exemple classe enum

Définissez une classe avec les valeurs rouge, vert, bleue

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255)
}

println("" + Color.RED.r + " " + Color.RED.g + " " + Color.RED.b)
=> 255 0 0
```

Object/singleton

- Parfois on a besoin de l'existance que d'une seule instance d'une classe
- Utilisez le mot clé `object` au lieu de `class`
- Les members sont accessibles avec la notation Classique `NameOfObject.<function or variable>`

Exemple Object/singleton

```
object Calculator {
    fun add(n1: Int, n2: Int): Int {
        return n1 + n2
    }
}

println(Calculator.add(2,4))
=> 6
```

Companion objects

- Permet de partager des variables et fonctions entre toutes les instance d'une classe
- Utilisez les mots clés `companion object`
- Accès via `ClassName.PropertyOrFunction`

Exemple companion object

```
class PhysicsSystem {
    companion object {
        val gravity = 9.8
        val unit = "metric"
        fun computeForce(mass: Double, accel: Double): Double {
            return mass * accel
        }
    }
}
println(PhysicsSystem.gravity)
println(PhysicsSystem.computeForce(10.0, 10.0))
=> 9.8100.0
```

Organiser son code

Un seul fichier, plusieurs entités

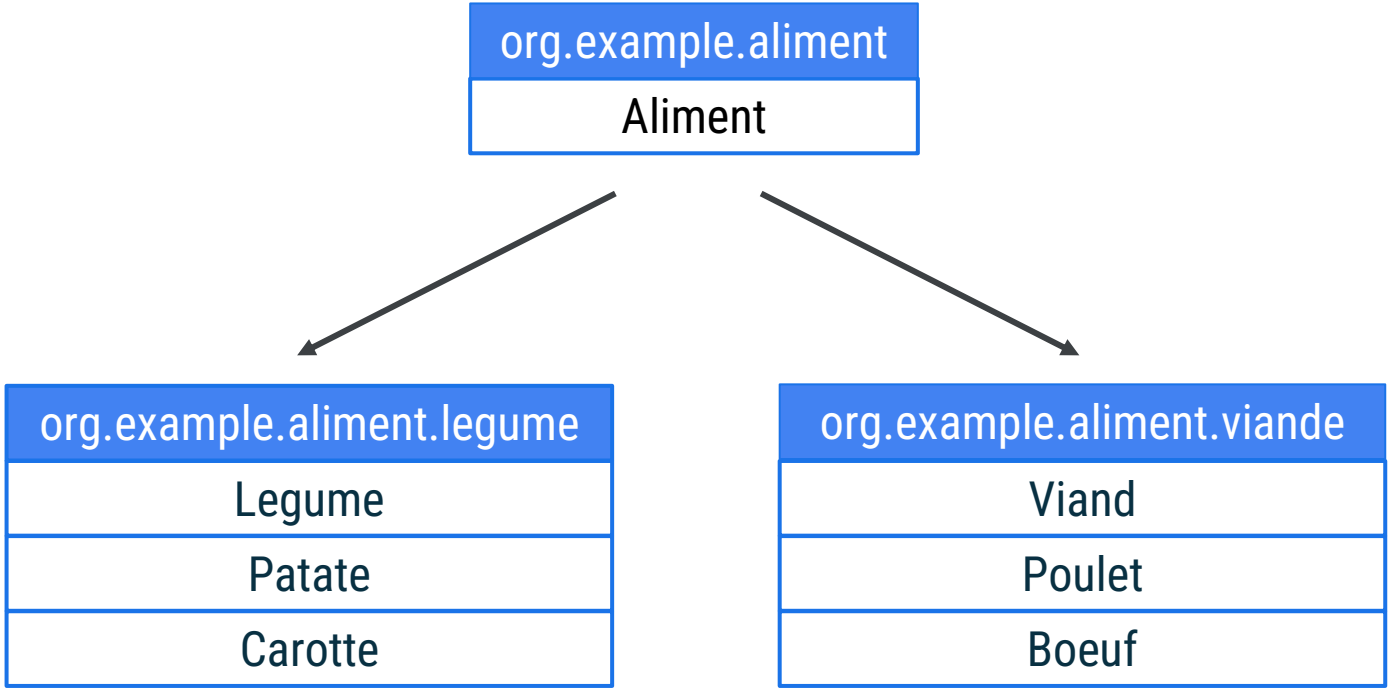
- Kotlin ne force pas d'avoir une seule entité (classe, interface…) par fichier
- Vous pouvez si possible regrouper les entités qui sont fortement liées
- Attention à la taille des fichiers et leurs complexités

Packages

- Provide means for organization
- Fourni une manière d'organizer les entités
- Utiliser des mots sans masjuscules séparés par des points
- Déclarer à la première ligne (qui n'est pas un commentaire) avec le mot clé `package`

```
package org.example.myapp
```

Exemple d'une hiérarchie de classe



Modificateurs de visibilité

Use visibility modifiers to limit what information you expose.
Utilisez les modificateurs de visibilité pour limiter l'accès aux informations exposées

- `public` veut dire que l'entité est visible en dehors de la classe. Tout est public par défaut (variables et méthodes).
- `private` signifie que l'élément est visible uniquement dans sa classe ou le fichier source où il est défini.
- `protected` fonctionne comme `private`, avec la possibilité en plus d'y accéder dans les sous-classes.

Pathway

Practice what you've learned by completing the pathway:

Lesson 3: Classes and objects