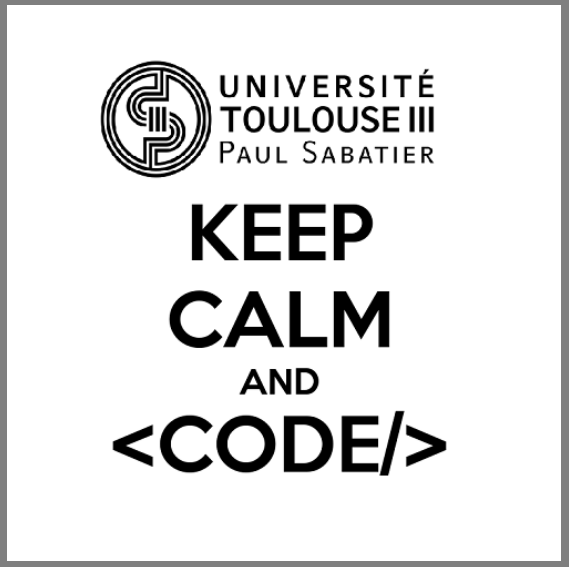

Programmation orientée objet – Application en Kotlin

Polymorphisme et liaison dynamique

Héritage versus délégation

Sylvie TROUILHET

Retour sur le polymorphisme

Surcharge : définir une fonction avec le même nom qu'une autre fonction déjà définie dans la classe, mais avec une nombre et/ou des types de paramètres différents

Redéfinition : modifier le corps d'une méthode héritée. La redéfinition permet le polymorphisme : une même méthode s'applique à des objets de types différents.

Retour sur le polymorphisme

Exemple de la méthode equals() :

- Elle est définie dans la classe Any : elle compare la référence de 2 objets
- Elle est redéfinie dans des sous-classes de Any (par exemple par la classe String)
- Elle peut être redéfinie dans toutes les classes

Exemple

```
class Ville(_nom : String){
    private var nom:String=_nom
}

val v1=Ville("Castres")
val v2=Ville("Castres")

print(v1==v2)
```

Qu'est-ce qui est affiché ?

```
"C:\Program Files\JetBrains\IntelliJ I
false
Process finished with exit code 0
```

Pourquoi ?

```
Ville@7cc355be
Ville@6e8cf4c6
```

Exemple

Quelle(s) méthodes de Any faut-il redéfinir ?

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    if (other !is Ville) return false

    if (nom != other.nom) return false

    return true
}

override fun hashCode(): Int {
    return nom.hashCode()
}
}
```

Retour sur le polymorphisme

En tant qu'utilisateur on peut appeler **`toString()`** pour n'importe quel objet.

- soit l'objet appelé a redéfini **`toString()`**
- sinon la méthode **`toString()`** de **`Any`** est appelée

Retour sur le polymorphisme

- L'affichage fait appel aux méthodes `toString()` des objets concernés .
- Exemple :

```
open class Ville {
    private var nom:String
    constructor(nom:String) {
        this.nom=nom }
    override fun toString(): String {
        return "$nom" }

val v=Ville("Castres")
print("$v")  // Castres
```

Cas de la méthode constructeur

Le constructeur de la super classe est TOUJOURS appelé avant d'exécuter le constructeur de la classe fille : on parle de chaînage des constructeurs

Visibilité : mode protégé (1)

Pour la classe `Capitale` :

```
class Capitale:Ville {
   private var pays
   private var commentaire

   constructor(nom:String, pays: String):super(nom){
        this.pays=pays
        this.commentaire="capitale ${this.nom}"
        }
  }
```

La compilation signale une erreur…

Visibilité : mode protégé (2)

Erreur 2 : `cannot acces nom : it is invisible in Capitale`

Comment rendre `nom` accessible depuis une sous classe mais pas depuis d'autres classes ?

Visibilité : mode protégé (3)

- Réponse : grâce au mode protégé (`protected`)
- Exemple :

```
open class ville {
     protected var nom : String
     }
```

- L'attribut `nom` est accessible depuis ses sous classes

Modificateur final

Le modificateur `final` peut s'appliquer :

- à des méthodes : on parle de méthodes finalisées

- à des classes : on parle de classes finalisées

→ en Kotlin : par défaut classes et méthodes sont `final`

Méthodes finalisées

- Une méthode finalisée ne peut pas être redéfinie dans une sous classe.
- L'intérêt est double :
  - Éviter la redéfinition : s'assurer qu'une méthode donnée n'est pas redéfinie en contradiction avec la définition de départ.
  - Pour de meilleures performances (cela évite la liaison dynamique : cf plus loin).

Classes finalisées

- Aucune classe ne peut hériter d'une classe finalisée.

- Cela permet d'accroître la sécurité :
  - cela évite l'utilisation malveillante d'une classe en définissant une sous classe aux comportements différents.

Transtypage (1)

L'intérêt principal de l'héritage est le transtypage.

Exemple : un objet de type `Capitale` peut être considéré comme une ville

```
val v : Ville
v = Capitale("Paris", "France")
```

Transtypage (2)

- On peut manipuler des objets de type `Ville` et des objets de type `Capitale` dans un tableau unique :

```
val tab:MutableList<Ville> = mutableListOf()

tab.add(Ville("Toulouse"))
tab.add(Capitale("Paris", "France"))
```

- On peut obtenir la description de tous les éléments du tableau :

```
for (v in tab) print(v)
```

Liaison dynamique (1)

Que se passe-t-il quand on appelle la méthode description() ?

- `tab[0].description()` : appelle la méthode de la classe `Ville` (c'est normal)
- `tab[1].description()` : appelle la méthode de la classe `Capitale`

→ c'est la liaison dynamique

Liaison dynamique (2)

- à l'exécution, l'interpréteur détermine le vrai type (type dynamique) de l'objet manipulé

- à partir de ce type dynamique, il cherche en remontant dans l'héritage la méthode à appliquer,

- une méthode finalisée est plus rapide à exécuter car aucun mécanisme de liaison dynamique n'est mis en place.

Classe abstraite (1)

- Supposons que nous ayons défini deux classes `Carre` et `Cercle` qui chacune définit une méthode `perimetre()`

- On souhaite rendre polymorphe cette méthode : c'est-à-dire pouvoir considérer à la fois des objets de type Cercle ou Carre et appliquer la méthode `perimetre()`

Classe abstraite (2)

- On veut factoriser la méthode `perimetre()` dans une super classe. On crée la classe `Figure`.
- Cependant, la classe `Figure` ne peut pas mettre en œuvre le calcul du périmètre car ce calcul est différent pour un carré et un cercle.
- La méthode `perimetre()` est dite abstraite : Seule l'entête est déclarée avec le mot clé `abstract`

La mise en œuvre sera faite par redéfinition dans ses sous classes

Classe abstraite (3)

- Une classe qui contient au moins une méthode abstraite est dite **abstraite**.
- Une classe abstraite n'est pas instantiable car il lui manque des éléments de mise en œuvre de ses méthodes abstraites.

Récapitulatif des modificateurs d'accès

	Méthode attribut
Public (par défaut)	La méthode ou l'attribut est accessible par le code, en dehors de la classe
private	La méthode ou l'attribut est accessible seulement au sein de la même classe
protected	La méthode ou l'attribut est accessible seulement au sein de la même classe ou de sa sous-classe
internal	La méthode ou l'attribut est accessible au sein du même module

Programmation orientée objet – Application en Kotlin

Héritage versus délégation

LPRO Dream

Sylvie TROUILHET

Préférer la délégation à l’héritage

Soit A la classe à écrire :
- plutôt que de faire hériter A d’une classe B, mettre une instance de B dans A et lui déléguer des comportements

Kotlin : créer un interface contenant les méthodes de délégation, les implémenter dans la classe A puis constituer la sur-classe au moyen du mot-clé by

Interface

- Une interface se déclare comme une classe mais utilise le mot clé interface à la place de class
- Une classe n'hérite pas d'une interface mais elle la met en œuvre
- L'interface n'est qu'une déclaration de signatures de méthodes

→ L'interface est un moyen de donner un surtype commun à des classes sans lien d'héritage

Interface

- Exemple : un smartphone combine un téléphone et une caméra, On pourrait dire que c'est un objet qui hérite de téléphone et de caméra.

- Si l'on préfère la délégation, on dira que c'est un objet qui englobe un téléphone et une caméra.
  - L'interface Numerotable décrira les services offerts par le téléphone
  - L'interface Capturable, ceux d'une camera

Interface

- Si l’on préfère la délégation, on dira que c’est un objet qui englobe un téléphone et une caméra.
  - L’interface Numerotable décrira les services offerts par le téléphone
  - La classe Téléphone implémente l’interface

```
interface Numerotable {
    fun numeroter(numTel:String): String
}

class Telephone : Numerotable {
    override fun numeroter(numTel: String)
                = "composition de $numTel"
}
```

Délégation : mot-clé by

- Smartphone délègue à son instance de type téléphone :

```
class Smartphone (private val telephone: Numerotable = Telephone()) : Numerotable by telephone
```

- Une intance de Smartphone peut appeler toutes les méthodes de Telephone

```
val smart=Smartphone()
smart.numeroter("06 01 32 45 11")
```

Exercice

Faire la même chose pour l’interface Capturable (services d’une camera)