Synthèse TD

[Une classe vide et non abstraite 1](#_Toc98114799)

[Une classe vide et abstraite 1](#_Toc98114800)

[Une classe avec attributs 2](#_Toc98114801)

[Solution 1 : valeur par défaut 2](#_Toc98114802)

[Solution 2 : constructeur principal 2](#_Toc98114803)

[Attributs public et private 2](#_Toc98114804)

[Association unidirectionnelle de cardinalité 1 4](#_Toc98114805)

[Association unidirectionnelle de cardinalité 0..1 5](#_Toc98114806)

[Association unidirectionnelle de cardinalité 0..n 6](#_Toc98114807)

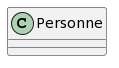
[Les interfaces 7](#_Toc98114808)

[Héritage 9](#_Toc98114809)

Les exemples qui suivent ne sont pas exhaustifs mais illustrent des situations que vous avez rencontré en TD et TP.

Les diagrammes UML ici, mais aussi tous ceux des cours et des TDs/TPs sont réalisés grâce à PlanUML : http://www.plantuml.com/.

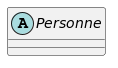
# Une classe vide et non abstraite



class Personne {  
}

val personne : Personne *//personne est de type Personne*personne = Personne() *//personne est une instance de Personne*

# Une classe vide et abstraite

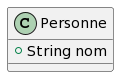


La notation en italique n’est pas très lisible, préférez le stéréotype <<abstract>>

abstract class Personne {  
}

val personne : Personne *//personne est de type Personne*personne = Personne() *//* =>*Cannot create an instance of an abstract class*

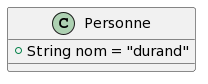
# Une classe avec attributs



class Personne {  
 ~~public var nom : String~~  
}

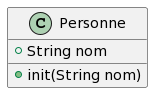
*//* => Property must be initialized or be abstract

## Solution 1 : valeur par défaut



class Personne {  
 public var nom : String = "durand"  
}

## Solution 2 : constructeur principal



class Personne(nom: String) {  
 public var nom : String  
 init {  
 this.nom = nom  
 }  
}

personne = Personne("Neuneu") *//personne est une instance de Personne*

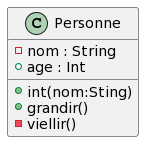
*//construite avec comme valeur du paramètre nom Neuneu*

~~personne = Personne()~~

*//* => No value passed for parameter 'nom'

## Attributs public et private

Une mauvaise conception



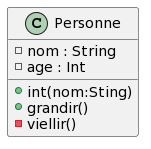
Les visiblités sont plutôt données par

+ pour public – pour private et # pour protected

class Personne(nom : String) {  
 private var nom : String  
 public var age : Int *//peut-être modifié hors de la classe* init {  
 this.nom = nom  
 age =0  
 }  
 public fun grandir() {  
 viellir()  
 }  
 private fun viellir(){  
 age++  
 }  
}

val personne: Personne  
personne = Personne("Durand")  
personne.grandir()  
personne.age= -10 *//comportement non souhaité*personne.viellir() *//Cannot access 'viellir': it is private in 'Personne'*

Une meilleure conception

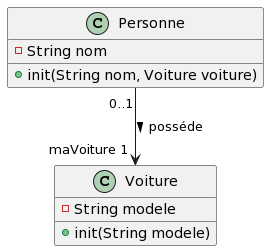


class Personne(nom : String) {  
 private var nom : String  
 private var age : Int *~~//peut-être modifié hors de la class~~e* init {  
 this.nom = nom  
 age =0  
 }  
 public fun grandir() {  
 viellir()  
 }  
 private fun viellir(){  
 age++  
 }  
}

val personne: Personne2  
personne = Personne2("Durand")  
personne.grandir()  
personne.age= -10 *//Cannot access 'age': it is private in 'Personne2'*

# Association unidirectionnelle de cardinalité 1

Toute personne possède une voiture et une seule. Une voiture à au plus un propriétaire et peut ne pas en avoir.



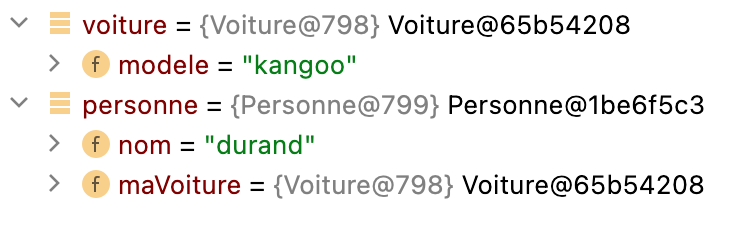
Attention, le rôle doit également renseigné sur la visibilité

class Voiture(modele:String) {  
 private var modele: String  
 init {  
 this.modele = modele  
 }  
}

class Personne(nom: String, voiture: Voiture) {  
 private var nom : String  
 private var **maVoiture** : Voiture *//induit par l'assiciation* init {  
 this.nom = nom  
 maVoiture = voiture  
 }  
}

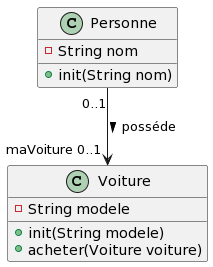
val voiture : Voiture  
val personne : Personne  
voiture = Voiture("kangoo") *//voiture doit être instanciée avant Personne*personne = Personne("durand",voiture)

En mémoire :



# Association unidirectionnelle de cardinalité 0..1

Une personne peut posséder ou non une unique voiture. Une voiture à au plus un propriétaire et peut ne pas en avoir.



class Voiture(modele:String) {  
 private var modele: String  
 init {  
 this.modele = modele  
 }  
}

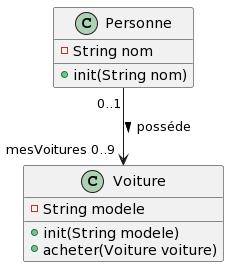
class Personne(nom: String) {  
 private var nom : String  
 private var maVoiture : Voiture? *//induit par l'association* init {  
 this.nom = nom  
 maVoiture = null  
 }  
 public fun achete(voiture: Voiture){  
 maVoiture = voiture  
 }  
}

val voiture : Voiture  
val personne : Personne  
  
voiture = Voiture("kangoo")  
personne = Personne("durand")  
  
personne.achete(voiture)

# Association unidirectionnelle de cardinalité 0..n

Une personne peut posséder jusqu’à 10 voitures. Une voiture à au plus un propriétaire et peut ne pas en avoir.

Rem : Ne disposant pour le moment que des tableaux il est difficile d’exprimer 0..\*



class Voiture(modele:String) {  
 private var modele: String  
 init {  
 this.modele = modele  
 }  
}

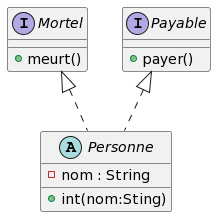
*//une approche qui consomme du temps sans attribut supplémentaire*class Personne(nom: String) {  
 private var nom : String  
 private var mesVoitures : Array<Voiture?> *//induit par l'association* init {  
 this.nom = nom  
 mesVoitures = *arrayOfNulls*(9)  
 }  
 public fun achete(voiture: Voiture){  
 var pos : Int;  
 pos = mesVoitures.*indexOf*(null) *//un parcours est caché dernière* if (pos != -1)  
 mesVoitures[pos]=voiture  
  
 }  
}

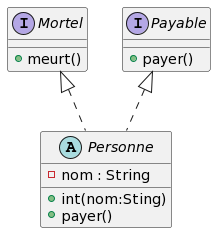
*//une approche qui ne consomme pas de temps mais qui utilise un attribut supplémentaire*class Personne(nom: String) {  
 private var nom : String  
 private var mesVoitures : Array<Voiture?> *//induit par l'assiciation* private var nbVoiture : Int  
 init {  
 this.nom = nom  
 this.mesVoitures = *arrayOfNulls*(9)  
 nbVoiture = 0;  
 }  
 public fun achete(voiture: Voiture){  
 if (nbVoiture < mesVoitures.size) {  
 mesVoitures[nbVoiture] = voiture  
 nbVoiture++  
 }  
 }  
}

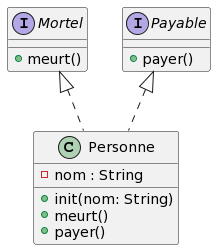
var voiture : Voiture  
val personne : Personne  
  
  
personne = Personne("durand")  
for (i in 1..11) { *//la derniere sera perdue* voiture = Voiture("kangoo")  
 personne.achete(voiture)  
}

# Les interfaces

On implémente une ou plusieurs interfaces, si l’ensemble des méthodes n’est pas implémenté alors la classe doit-être abstraite







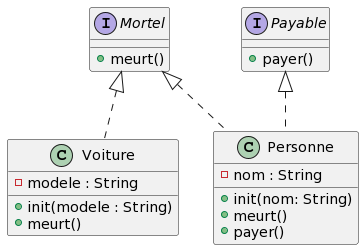
class Personne(nom: String):Payable,Mortel{  
 private var nom : String  
  
 init {  
 this.nom = nom  
 }

override fun meurt() {  
 *print*("${nom} est morte")  
}  
  
override fun payer() {  
 *print*("${nom} est riche")  
}

}

val personne1 : Personne  
personne1 = Personne("Durand")  
personne1.meurt() *//Durand est morte*val mortel : Mortel  
mortel = Personne("Dupond") *//A découvrir plus tard*mortel.meurt()*//Dupond est morte*

Une interface peut être implémentée par plusieurs classes

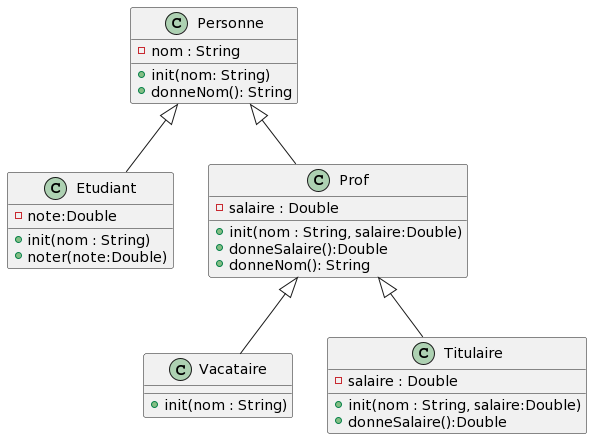


val personne : Personne  
personne = Personne("Durand")  
personne.meurt() *//Durand est morte*val voiture: Voiture  
voiture = Voiture("Kangoo")  
voiture.meurt()*//Kangoo est morte*val mortels : Array<Mortel>  
mortels = *arrayOf*(personne, voiture)  
for (mortel in mortels)  
 mortel.meurt()*//Durand est morte Kangoo est morte*

# Héritage

On ne peut hériter que d’une classe mère, les classes filles héritent des attributs et des méthodes de la classe mère et peuvent les redéfinir.

Une personne a un nom, un étudiant est une personne avec une note, un prof est une personne avec un salaire, le nom d’un prof est préfixé de M., un vacataire est un prof avec un salaire unique de 1000 Euros, un titulaire est un prof dont le salaire affiché est double d’un prof.



open class Personne(nom: String){ *//open pour pouvoir etre hérité* private var nom : String  
 init {  
 this.nom = nom  
 }  
 open public fun donneNom():String { *//open pour pouvoir être redéfinie* return nom;  
 }  
}

class Etudiant(nom:String, note:Double):Personne(nom) {  
 private var note :Double  
 init{  
 this.note = note  
 }  
 public fun donneNote():Double{  
 return note  
 }  
}

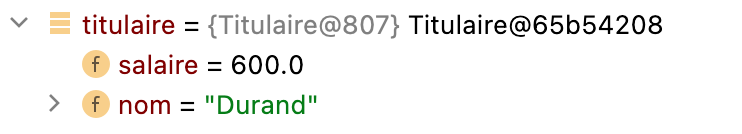
open class Prof(nom:String, salaire:Double):Personne(nom) {  
 private var salaire : Double  
 init{  
 this.salaire = salaire  
 }  
 open public fun donneSalaire():Double{  
 return salaire  
 }  
 public override fun donneNom():String { *//overide pour indiquer la redéfinition* return "M. "+super.donneNom() *//donneNom() de Personne* }  
  
}

class Vacataire(nom: String):Prof(nom,1000.0) {  
}

class Titulaire(nom: String,salaire:Double):Prof(nom, salaire) {  
 public override fun donneSalaire():Double {  
 return super.donneSalaire()\*2;  
 }  
}

val titulaire = Titulaire("Durand",600.0)  
*println*(titulaire.donneNom()) *//M. Durand  
println*(titulaire.donneSalaire()) *//12000.0*

En mémoire :



Rem : un attribut ou une méthode peuvent-être protected (#) et donc accessible dans les classes filles.