

Initiation au Génie Logiciel

Cours 3

— Architecture logicielle

Ce que nous allons voir dans ce cours

Quelques principes de base d'Architecture logicielle (orientée objet)

- Encapsulation
- Relation d'association
- Relation d'héritage
- Interfaces/Traits
- Le découpage en packages

L'objectif commun de ces notions est d'*abstraire* un programme

... pouvoir l'utiliser ou le concevoir en s'abstrayant des détails

Ces principes seront approfondis en L3 (UE BMO)

Nous verrons la mise en oeuvre de ces principes en Scala ainsi que :

- Les case classes

Plan

- 1 Principes de base d'architecture logicielle orientée objet
 - Encapsulation
 - Héritage et association
 - Diagrammes de classes
 - Héritage et association en Scala
 - Traits
- 2 Principes avancés de programmation orientée objet
 - Case classes et match-case
 - Packages
 - Importation de code

Plan

- 1 Principes de base d'architecture logicielle orientée objet
 - Encapsulation
 - Héritage et association
 - Diagrammes de classes
 - Héritage et association en Scala
 - Traits
- 2 Principes avancés de programmation orientée objet
 - Case classes et match-case
 - Packages
 - Importation de code

Encapsulation

L'encapsulation abstrait un code du point de vue de son utilisateur :

- masque les détails inutiles (les rend **invisibles**)
- améliore la **visibilité** des éléments importants

Exemple 1

```
class Vehicule(m:String){
  def vidange(g:String,n:Int){...}
  def pressionPneus(i:Int){...}
  def demarrer={...}
  val modele=m
  val consommation=
    calculerConso(cylindree(m))
  val cylindree=cylindree(m)
}

class Vehicule(m:String){
  // ...
  def demarrer={...}
  val modele=m
  // ...
}
```

Rmq : c'est ce qui manquait à la classe **Graphe** du package **C** du TP2!

Encapsulation (II)

En Scala (comme en Java), en déclarant un **membre private**, on le rend **invisible en dehors de l'objet/classe**. Membre=(champ/méthode/classe/objet)

Remarque 1 (Visibilité par défaut : public)

Si on ne déclare pas un membre **private** il est implicitement visible (comme déclaré **public**).

Remarque 2 (Protège les objets contre des altérations extérieures)

```
class Compte{
  var m=0
  def déposer(x:Int){m=m+x}
  def retirer(x:Int){m=m-x}
}

val c1= new Compte
c1.déposer(1000)

class Compte{
  private var m=0
  def déposer(x:Int){m=m+x}
  def retirer(x:Int){m=m-x}
}

val c2= new Compte
c2.déposer(1000)
```

`c1.m = c1.m - 200 // réussi! ☺`

`c2.m = c2.m - 200 // échoue! ☹`

Encapsulation (III)

Quiz 1 (Ces codes Scala sont-ils corrects?)

```
class Element(s:String){
  private val c= s
  private def f= s+s
}

val e= new Element("toto")
```

V Oui R Non

```
object Element{
  private def f={print("toto")}
}

Element.f
```

V Oui R Non

```
class Element(s:String){
  private val c= s
  def f= c
}

val e= new Element("toto")
println(e.f)
```

V Oui R Non

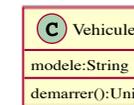
```
object Element{
  private var c=0
  def f(x:Int)={c=x}
}

Element.f(4)
```

V Oui R Non

Et après l'encapsulation ?

- On peut voir un objet/classe encapsulé comme un « composant » disposant de valeurs (ici, **modele**) et offrant des opérations (ici, **demarrer**) :



- On peut obtenir des « composants » similaires pour d'autres classes :

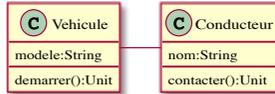


- Pour les combiner on utilise deux relations : l'**association** et l'**héritage**

Association d'objets/classes

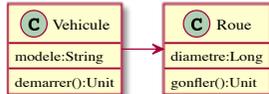
Définition 2 (Relation d'association - association réciproque)

L'association entre une classe A et une classe B, notée  signifie que les objets de classe A « ont des » objets de classe B, et réciproquement.



Remarque 3 (Association orientée)

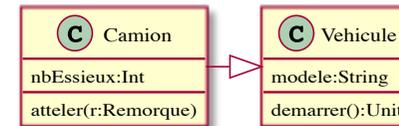
Si la relation n'est pas réciproque, on utilise l'association orientée .



Héritage d'objets/classes

Définition 3 (Relation d'héritage)

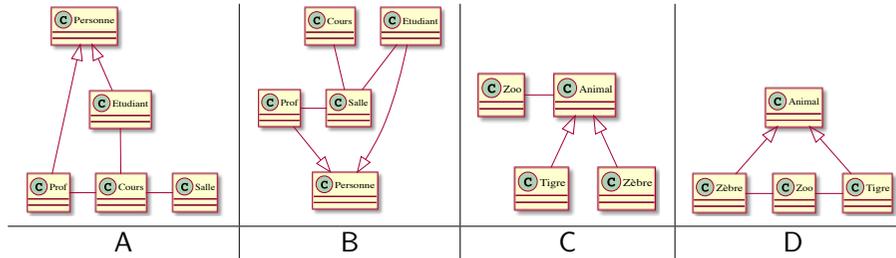
L'héritage entre une classe A et une classe B, noté , signifie que les objets de classe A « sont des » objets de classe B. Ainsi, les objets de la classe A ont (au moins) tous les champs et opérations de la classe B.



Remarque 4 (Lien avec le sous-typage (par exemple en Scala))

La relation d'héritage coïncide avec la relation de sous-typage. Si une classe A hérite d'une classe B, alors A est un sous-type de B. Par exemple, `Int` hérite de `Any`.

Diagrammes de classes, le quizz !



Quiz 2

1 Les diagrammes A,B,C et D sont-ils corrects ?

V Oui || R Non

2 Entre A et B, quel est le diagramme le plus pertinent ?

V A || R B

3 Entre C et D, Quel est le diagramme le plus pertinent ?

V C || R D

Implantation des associations et héritages en Scala

Implantation d'une association orientée

On ajoute à la classe A un champ contenant une (ou des) références vers des objets de classe B. Par exemple :

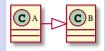
- Un A a **exactement** un B : `class A{val objB: B ...}`
- Un A a 0, 1, 2,..., n B : `class A{val objB: Set[B]...}`
- Un A a 0, 1, 2,..., n B, et l'ordre des B est important :
`class A{val objB: List[B]...}`

Implantation d'une association réciproque

Pour une association réciproque, en plus du code précédent, on ajoute à la classe B un champ contenant une (ou des) références vers des objets de A. Par exemple, un A a **exactement** un B qui a 0, 1, 2,..., n A :

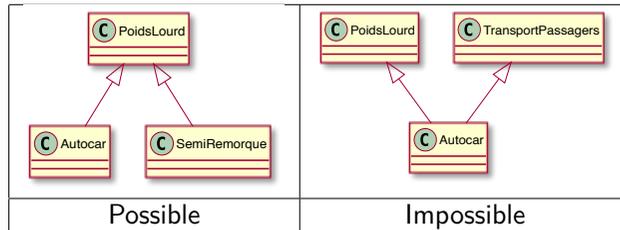
- `class A{val objB: B ...}` et `class B{val objA: Set[A]...}`

Implantation des associations et héritages en Scala (II)

Implantation d'une relation d'héritage  avec **extends**

```
class B { ... code de la classe B }
class A extends B { ... code de la classe A }
```

Remarque : En Scala on ne peut hériter que d'une classe au plus !



On en reparlera dans le cours de concepts objets avancés...

Traits

Définition 4 (Traits)

Un trait définit des types d'objets, sans donner leur implantations. On parle aussi de **types abstraits** (en S12) ou d'**interfaces** (en PO/Java).

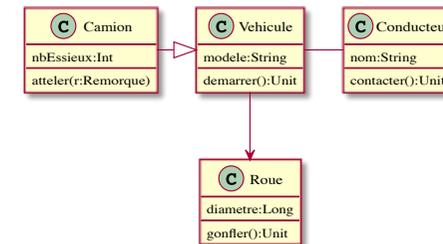
« Une interface est la couche limite entre deux éléments par laquelle ont lieu des échanges et des interactions »

Remarque 6 (Les traits facilitent le développement collaboratif)

Un trait déclare un ensemble de champs/méthodes avec leurs types. Il sépare un logiciel en deux parties : une partie utilisatrice des champs/méthodes et une partie implantant ces champs/méthodes.

Association et héritage en Scala

Les diagrammes obtenus sont nommés *diagrammes de classe*



Exercice 1

Implantez succinctement en Scala le diagramme de classe précédent.

Remarque 5 (Un diagramme de classe peut contenir plus d'informations)

Les types de relations sont plus nombreux, elles peuvent présenter des cardinalités, etc. Vous les étudierez plus en détail en L3 (UE BMO)

Traits (II)

Exemple 5 (Trait pour un logiciel manipulant des files d'entiers)

Utilisateurs	Interface	Implanteurs
Une équipe réalise un logiciel utilisant des <code>IntQueue</code>	<pre>trait IntQueue { def get: Int def put(x: Int): Unit def empty: Boolean }</pre>	Une équipe implante le trait <code>IntQueue</code> dans une classe <code>MyQueue</code>

```
def vider(q: IntQueue): Unit =
  while (!q.empty) q.get

def incAll(q: IntQueue): Unit = {
  var l = List[Int]()
  while (!q.empty) l = q.get :: l
  for (e <- l.reverse) q.put(e+1)
}

class MyQueue extends IntQueue {
  private var b = List[Int]()
  def get = { val h = b(0); b = b.drop(1); h }
  def put(x: Int): Unit = { b = b:+x }
  def empty = b.isEmpty
}
```

Traits (III)

- Le mot clé `extends` précise qu'une classe implante un trait

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0); b=b.drop(1); h}
  def put(x:Int):Unit= {b=b:+x}
  def empty=b.isEmpty
}
```

Remarque 7

Pour implanter un trait, une classe doit implanter tous les champs et méthodes qui n'ont pas d'implantation (en respectant les types de ceux-ci).

- Les utilisateurs de `IntQueue` n'ont pas besoin de connaître l'implantation `MyQueue` pour développer leur partie :
`def vider(q: IntQueue):Unit= while(!q.empty) q.get`
- Sauf pour créer un objet de ce type : `val q= new MyQueue`

Traits (IV)

Remarque 8

Abstraire le fonctionnement d'une partie d'un logiciel par un trait, le rend plus robuste à l'évolution de l'implantation de cette partie.

Exemple 6

On peut remplacer l'implantation `MyQueue` de `IntQueue` par une autre sans toucher au code de la partie utilisateur : `vider`, `incAll`.

Dans le TP2, les implantations D, E et F sont interchangeables car elles respectent le même **trait**. Ce n'est pas le cas pour B et C.

Exercice 2

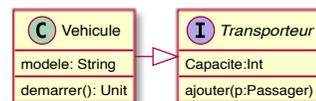
Quel pourrait être le trait abstrayant le fonctionnement de la classe `Graphe` du TP1, pour pouvoir facilement changer d'implantation de graphes sans changer le code de l'objet `Canalisations`.

Représentation des traits dans les diagrammes

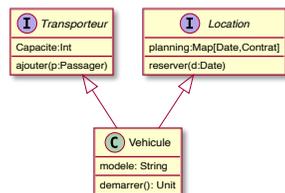
Représentation des **traits**/**I**nterfaces



Représentation de la relation d'implantation



Une classe/objet peut implanter plusieurs traits :



```
class Vehicule
  extends Transporteur
  with Location {
    ... code de la classe
    Vehicule ...
  }
```

Exercice 3 (Complétez la déclaration des classes Vehicule, etc.)

Faites apparaître les traits `Location` et `Transporteur`.

Traits (III)

Quiz 3 (Ces codes Scala sont-ils corrects ?)

```
trait A{
  val c:String
}
```

```
val a= new A
```

V Oui R Non

```
class A{
  val c:String
}
```

```
val a= new A
```

V Oui R Non

Quiz 4 (Ces codes Scala sont-ils corrects ?)

```
trait A{
  val c:String
}
```

```
object B extends A{
  val c="toto"
}
```

V Oui R Non

```
trait A{
  def f(x:Int):String
}
```

```
class B extends A{
  def f(x:Any):String="oui!"
}
```

V Oui R Non

Traits (IV)

Exercice 4 (Développement collaboratif d'un navigateur Web)

Le *navigateur* parcourt des *pages* localisées par des *URLs*. Les pages contiennent des *éléments* clicables ou non (du *texte* et des *images*). Le navigateur est muni d'un *historique* des pages visitées.

- Quels traits proposeriez-vous (pour au moins 6 équipes) ? avec quels champs/opérations ?
- Quelles relations définiriez-vous entre ces traits ? (association/héritage)

Quiz 5 (Pour le navigateur, quelle est la solution la plus pertinente?)

```
trait Navigateur{ ... }  
class MonNav  
  extends Navigateur {  
  ... }  
V
```

```
trait Navigateur{ ... }  
object MonNav  
  extends Navigateur {  
  ... }  
R
```

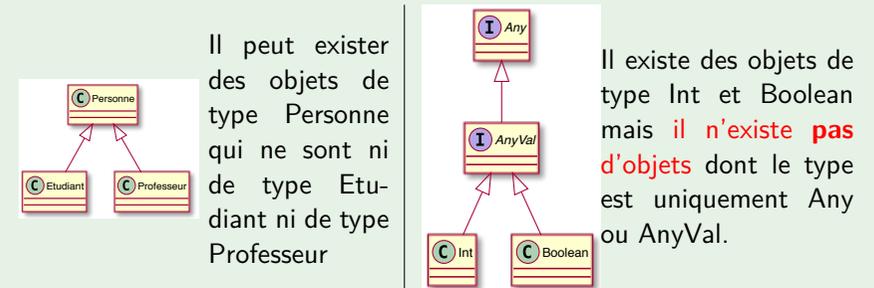
Traits (V)

Les traits facilitent le développement collaboratif mais aussi le sous-typage

Remarque 9 (Les traits pour le sous-typage)

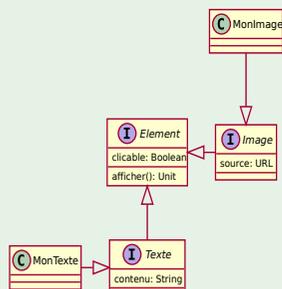
Un trait permet de définir quelles sont les champs/opérations communes entre plusieurs classes (c'est un super-type). Cela remplace l'héritage quand le super-type n'a pas d'objets propres.

Exemple 7 (Différence entre héritage et implantation de traits)



Traits (VI)

Exemple 8



- On peut créer des objets de type *MonImage* ou *MonTexte*
- Un objet de type *MonTexte* sera aussi de type *Texte* et *Element*
- On ne peut pas créer d'objets de type *Image*, *Texte* ou *Element* qui ne soient ni des *MonTexte* ou *MonImage*

Plan

- 1 Principes de base d'architecture logicielle orientée objet
 - Encapsulation
 - Héritage et association
 - Diagrammes de classes
 - Héritage et association en Scala
 - Traits
- 2 Principes avancés de programmation orientée objet
 - Case classes et match-case
 - Packages
 - Importation de code

Case class/object définissent des types algébriques

« Un type algébrique de données est un type de données dont chacune des valeurs est une donnée d'un autre type enveloppée dans un des constructeurs du type. **Toutes les données enveloppées sont des arguments du constructeur.** La seule manière d'opérer sur les données est d'enlever le constructeur en utilisant le filtrage par motif. »

- `case class Rect(x1:Int,y1:Int,x2:Int,y2:Int) // Rectangles`
- Les instances de `case class` sont initialisées sans `new` (c-à-d que les objets de `case class` sont obtenus sans `new`)

```
scala> val r1= Rect(0,0,5,7)
```
- L'accès aux données de l'objet peut se faire par `match-case`

```
scala> r1 match {case Rect(_,_,x,y) => (x,y)}
(Int,Int) = (5,7)
```
- Par défaut, `toString` et `equals` respectent la structure des objets

```
scala> println(r1) | scala> val r2= Rect(0,0,5,7)
Rect(0,0,5,7)      | scala> r1==r2
                    | Boolean = true
```

Case classes et case objects, premier quizz

Quiz 6 (Qu'affichent ces programmes ?)

```
class P(x:Int)
val p= new P(1)
val p2= new P(1)
println(p==p2)
```

V true R false

```
case class P(x:Int)
val p= P(1)
val p2= P(1)
println(p==p2)
```

V true R false

Quiz 7 (Qu'affichent ces programmes ?)

```
class P(x:Int)
val p= new P(1)
val p2= new P(2)
println(p==p2)
```

V true R false

```
case class P(x:Int)
val p= P(1)
val p2= P(2)
println(p==q)
```

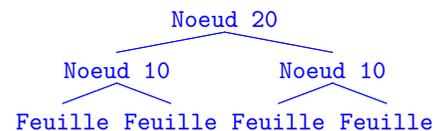
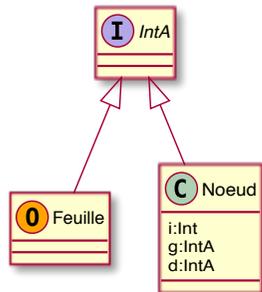
V true R false

Case classes et case objects : applications

Les case classes permettent de définir **simplement** et de manipuler **efficacement** tout type d'objet non mutable

Exemple 9 (Définition des arbres binaires d'entiers par case class)

```
trait IntA // Arbres binaires d'entiers
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```



```
val a1= Noeud(10,Feuille,Feuille)
val a2= Noeud(20,a1,a1)
```

Case classes et le pattern matching par match-case

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```

- `match-case` permet de décomposer les objets issus de case classes

```
def sommetPositif(e:IntA):Boolean= {
  e match {
    case Feuille => false
    case Noeud(e,_,_) => e>0
  }
}
```

Exercice 5 (Définissez la fonction (récursive) suivante)

`dans(i: Int, a: IntA): Boolean` qui teste si un entier est dans un arbre.

Case classes et le pattern matching par match-case (II)

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```

- match-case peut plonger dans plusieurs niveaux de case classes
- ```
def filsPositif(e:IntA):Boolean= {
 e match {
 case Noeud(_,Noeud(e,_,_),Feuille) => e>0 //fils gauche
 case Noeud(_,Feuille,Noeud(e,_,_)) => e>0 //fils droit
 case Noeud(_,Noeud(e1,_,_),Noeud(e2,_,_)) => e1>0 || e2>0
 case _ => false
 }
}
```

## Case classes et le pattern matching par match-case (III)

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```

```
val a= Noeud(2,Noeud(1,Feuille,Feuille),Feuille)
```

### Quiz 8 (Ces pattern-matchings sont-ils corrects?)

|                                                                  |                                       |     |
|------------------------------------------------------------------|---------------------------------------|-----|
| a match {<br>case 2 => false<br>case _ => true<br>}              | <input checked="" type="checkbox"/> V | Oui |
|                                                                  | <input type="checkbox"/> R            | Non |
| a match {<br>case Noeud(,_,_) => false<br>case _ => true<br>}    | <input checked="" type="checkbox"/> V | Oui |
|                                                                  | <input type="checkbox"/> R            | Non |
| a match {<br>case _(2,_,Feuille) => false<br>case _ => true<br>} | <input checked="" type="checkbox"/> V | Oui |
|                                                                  | <input type="checkbox"/> R            | Non |

## Case classes et le pattern matching par match-case (IV)

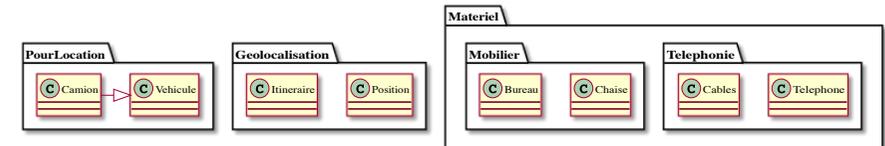
```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
val a= Noeud(2,Noeud(1,Feuille,Feuille),Feuille)
```

### Quiz 9 (Quels sont les résultats de ces pattern-matchings?)

|                                                                                                   |                                       |                          |
|---------------------------------------------------------------------------------------------------|---------------------------------------|--------------------------|
| a match {<br>case Noeud(,_,Feuille) => 2<br>case Noeud(2,_,_) => 3<br>}                           | <input checked="" type="checkbox"/> V | 2                        |
|                                                                                                   | <input type="checkbox"/> R            | 3                        |
| a match {<br>case Noeud(x,Noeud(y,Feuille,_),Feuille) => x+y<br>case Noeud(x,Feuille,_) => x<br>} | <input checked="" type="checkbox"/> V | 2                        |
|                                                                                                   | <input type="checkbox"/> R            | 3                        |
| a match {<br>case Noeud(,x,_) => x<br>case Noeud(,_,x) => Feuille<br>}                            | <input checked="" type="checkbox"/> V | Noeud(1,Feuille,Feuille) |
|                                                                                                   | <input type="checkbox"/> R            | Feuille                  |

## Packages

Pour structurer le code, on introduit des **packages** pour rassembler des classes/objets/traits ayant un lien logique.



### Définition 10 (Notation pointée, les chemin d'accès pour les packages)

Comme pour les répertoires dans un système de fichiers, un package **p2** peut être inclus dans un package **p1**, etc. Dans ce cas, depuis le package **p1**, le chemin d'accès au package **p2** sera simplement : **p2**. Depuis le package *contenant* **p1**, le chemin d'accès sera **p1.p2**.

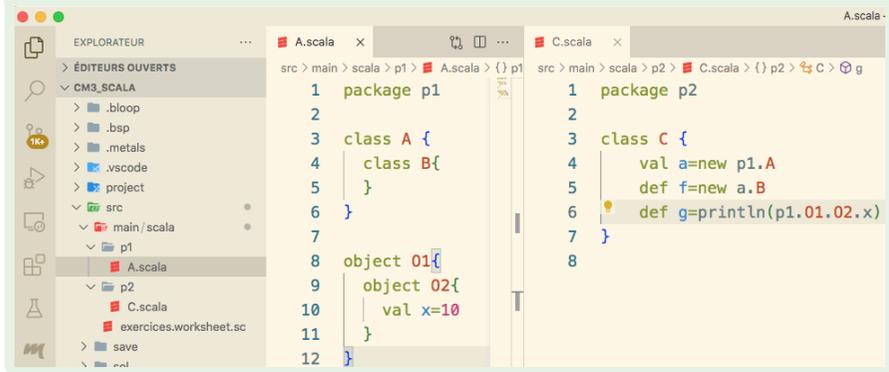
### Remarque 10 (Définition du package d'un code Scala avec package)

Si **p** est un chemin d'accès à un package, pour placer un code dans le package de chemin **p**, celui-ci doit débiter par **package p**.

## Packages et chemins d'accès aux objets/champs/méthodes

- On peut aussi inclure des classes/objets dans d'autres classes/objets
- Pour accéder à un objet/classe/champ/méthode `x` d'un objet `an` on utilise un chemin d'accès `a1.a2. ... an.x` (où `a1 a2 ... an` sont des noms de packages ou d'objets (pas de noms de classes!))

### Exemple 11 (Exemple de chemin d'accès (package/objets/classes))



```

1 package p1
2
3 class A {
4 class B {
5 }
6 }
7
8 object O1 {
9 object O2 {
10 val x=10
11 }
12 }

```

```

1 package p2
2
3 class C {
4 val a=new p1.A
5 def f=new a.B
6 def g=println(p1.O1.O2.x)
7 }
8

```

## Packages et chemins d'accès (II)

### Quiz 10 (Comment accéder au champ `x`?)

```

package p0.p1.p2
object A {
 val x=10
}

```

```

package p0.p3
object C {
 val lx= ??? x
}

```

`p1.p2.A.x` ||  `p0.p1.p2.A.x`

```

package p0.p1
class A {
 val x=10
}

```

```

package p0.p2
object C {
 val lx= ??? x
}

```

`val lx= (new p0.p1.A).x`  
 `val lx= p0.p1.A.x`

## Packages et importation de code

Les chemins sont simplifiés en important des classes/objets/champs/méthodes

### Définition 12 (import `p.x`)

Importe la classe/objet/champ/méthode `x` de chemin d'accès `p`. Le chemin `p` peut contenir des noms de packages et d'objets mais pas de classes. Si `x` est `'_'`, on importe tous les objets membres du package `p` ou tous les champs/méthodes de l'objet désigné par `p`.

- `import p.C` importe la classe/objet `C` du package de chemin `p`
- `import p._` importe toutes les classes/objets du package de chem. `p`
- `import p.O._` importe les champs/méthodes de l'objet `O` de chem. `p`

### Remarque 11 (Classes/objets/méthodes importés par défaut)

```

import java.lang._ // tout le package java.lang
import scala._ // la librairie de base Scala: List,Set,...

import scala.Predef._ // des méthodes prédéfinies: println,...

```

## Packages et importation de code (II)

### Quiz 11 (Comment accéder au champ `x`?)

```

package p0.p1.p2
object A {
 val x=10
}

```

```

package p0.p3
import p0.p1._
object C {
 val lx= ??? x
}

```

`p1.p2.A.x` ||  `p2.A.x`

```

package p0.p1
object A {
 object B {
 val x=10
 }
}

```

```

package p0.p2
import p0.p1.A._
object C {
 val lx= ??? x
}

```

`x` ||  `B.x`