

Initiation au Génie Logiciel

Cours 5

— Concepts avancés de programmation fonctionnelle et de programmation objet

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Fonctions/méthodes d'ordre supérieur

Définition 1 (Fonction d'ordre supérieur)

Une fonction *d'ordre supérieur* prend en paramètre une ou plusieurs fonctions.

Exemple 2 (La fonction `map` sur les listes)

La fonction `map` applique une fonction (un *traitement*) à tous les éléments d'une liste.

```
val l= List(1,2,3)
def f(x:Int)= x+1
val l2= l.map (f)           // l2= List(2,3,4)

// ou plus simplement en utilisant l'application partielle sur +
val l3= l.map (_ + 1)     // l3= List(2,3,4)
```

Remarque 1 : A comparer avec la version impérative (avec `for`)!

Fonctions/méthodes d'ordre supérieur (II)

Exemple 3 (La fonction `filter` sur les listes)

La fonction `filter` filtre les éléments d'une liste en fonction d'un **prédicat**, i.e. une fonction dont le résultat est booléen. `filter` conserve les éléments pour lesquels le prédicat est vrai.

```
val l= List(1,2,3,4)
def f(x:Int)= x>2
val l2= l.filter (f)      // l2= List(3,4)

// ou plus simplement en utilisant l'application partielle sur >
val l3= l.filter (_ > 2) // l3= List(3,4)
```

Fonctions/méthodes d'ordre supérieur (III)

Exemple 4 (La fonction `reduce` sur les listes)

La fonction `reduce` réduit une liste d'éléments à une valeur à l'aide d'une fonction à 2 arguments. La liste doit comporter au moins 1 élément !

```
val l= List(1,2,3,4)
def f(x:Int,y:Int)= x+y
val i= l.reduce(f)      // i= 10

// ou plus simplement en utilisant l'application partielle sur +
val l3= l.reduce (_ + _) // i= 10
```

Remarque 1 : toutes ces fonctions existent pour les `Set`, `Array`, `Map`, ...

Remarque 2 : il existe beaucoup d'autres fonctions d'ordre supérieur : `exists`, `forall`, `foldLeft`, `foldRight`, ...

Fonctions/méthodes d'ordre supérieur (IV)

Exercice 1

A partir de l'ensemble d'entiers $\{1, 2, 3\}$, construisez l'ensemble de chaînes $\{"1", "2", "3"\}$.

Exercice 2

Définissez la fonction

`remove(x:String,l>List[String]):List[String]` qui supprime toutes les occurrences de l'élément `x` dans `l`.

Exercice 3

Calculez la somme des éléments d'un tableau. Calculez la factorielle de 10.

Exercice 4

Calculez l'ensemble des carrés de l'ensemble `Set(1,2,3,4)`.

Fonctions anonymes

Notation de fonction classique

$$f : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$$
$$f(x, y) = x + y$$

Lambda-notation

$$f : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$$
$$f = \lambda x y. x + y$$

$\lambda x y. x + y$ est une fonction anonyme ajoutant deux naturels

Cette fonction s'écrirait `(lambda (x y) (+ x y))` en APF/Racket

- La fonction anonyme Scala ajoutant deux entiers s'écrit :

$$((x:Int,y:Int) => x + y)$$

Le type inféré par Scala pour cette fonction est : `(Int,Int) => Int`

Exercice 5

Refaire l'exercice 4 à l'aide de `map` et d'une fonction anonyme.

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Abstraction de type et polymorphisme

Quel est le problème ? Comment factoriser ce code ?

```
// Fonction retournant le premier élément d'un couple d'entiers
def first(p:(Int,Int)):Int= p match case (x,y) => x
val e= first((10,20)) // e=10
// Fonction retournant le premier élément d'un couple de chaînes
def first(p:(String,String)):String= p match case (x,y) => x
```

On peut paramétrer les fonctions par des **variables de type**

```
// Fonction retournant le premier élément d'un couple
def first[T1,T2](p:(T1,T2)):T1= p match case (x,y) => x
```

- pour appeler la fonction on doit donner les valeurs des types

```
val e= first[String,String](("titi","toto"))// e="titi"
val e2= first[Int,String]((10,"toto")) // e2=10
```
- ... mais ceux-ci peuvent généralement être inférés !!

```
val e3= first((10,"toto")) // e3=10
```

Abstraction de type et polymorphisme (II)

Quel est le problème ? Comment factoriser ce code ?

```
trait IntQueue{//files d'entiers   trait StringQueue{//files
  def get:Int                       def get:String //de chaînes
  def put(x:Int):Unit              def put(x:String):Unit}
```

On peut paramétrer les classes/objets/traits par des types

```
trait Queue[T]{ // Interface des files génériques
  def get:T
  def put(x:T):Unit}

// Classe de files génériques
class MyQueue[T](init:List[T]) extends Queue[T]{
  private var b= init
  def get={val h=b(0); b=b.drop(1); h}
  def put(x:T):Unit= {b=b:+x}}

val f= new MyQueue[String](List()) // valeur de T non inferée
val f2= new MyQueue(List("toto")) // valeur de T EST inferée
```

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Plus d'abstraction avec les traits : les fabriques d'objets

Connaître le trait **X** ne permet pas de créer un objet de type **X**

- Il faut connaître une classe **C** implantant **X**
- Dans le code, `new C` apparaît partout où un nouvel objet de type **X** est nécessaire
- \implies On ne peut pas facilement changer d'implantation de **X** sans reprendre tout le code

Un patron de conception : les fabriques d'objets

- En complément du trait **X**, fournir une fabrique d'objets

```
object Xfactory{
  def get:X= new C
}
```
- Dans le reste du code, remplacer `new C` par `Xfactory.get`

Voir exemple 4 dans le projet Eclipse du cours

Plus d'abstraction avec les traits : les fabriques d'objets (II)

Exemple 5 (Changer d'implantation de **X** sans fabrique)

Dans le code, remplacer **toute** occurrence de **C** par **C2**

```
val q1= new C      | val q1= new C2
val q2= new C      | val q2= new C2
...                | ...
val qn= new C      | val qn= new C2
```

Exemple 6 (Changer d'implantation de **X** avec une fabrique)

Dans le code, remplacer **une** occurrence de **C** par **C2**

```
object Xfactory{   | object Xfactory{
  def get:X= new C |   def get:X= new C2}

val q1= Xfactory.get | val q1= Xfactory.get // ne change pas
val q2= Xfactory.get | val q2= Xfactory.get
...                  | ...
val qn= Xfactory.get | val qn= Xfactory.get
```

Polymorphisme et fabriques

Un **multi-ensemble** est un ensemble où chaque valeur peut apparaître plusieurs fois.

Exercice 6

Définir le trait pour un multi-ensemble polymorphe (de valeurs de type **T**).
Donnez le type des opérations d'ajout d'un élément et de test d'appartenance qui donne le nombre d'occurrences de l'élément dans le multi-ensemble.

Exercice 7

Définir une classe polymorphe implantant le trait multi-ensemble.

Exercice 8

Définir une fabrique pour le trait des multi-ensembles.

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Héritage et visibilité

Si la classe **A** hérite de **B**, tous les *membres* de **B**...

- déclarés **public** sont accessibles aux objets de classe **A**

```
class B {
  val x=0
  def f(x:Int)=x+1}
class A extends B {
  val y=18}
```

```
scala> val a= new A
scala> a.x
Int = 0
scala> a.y
Int = 18
scala> a.f(10)
Int = 11
```
- déclarés **private** sont invisibles pour tous les objets (y compris **A**)
- déclarés **protected** sont accessibles aux objets des classes héritant de **B** (dont **A**), **depuis le code de ces classes**.

Remarque 1 (Liaison dynamique et superclasse)

Si **o** est un objet de classe **A**, **o.x** fait référence au membre **x** de la classe **A**, s'il existe. Sinon, **x** est recherché dans la classe dont hérite **A**. Ce processus est itéré jusqu'à trouver une classe implantant **x**. Dans la classe **A**, **super.x** fait référence à l'implantation de **x** dans la classe dont hérite **A**. De la même façon, ceci est itéré jusqu'à trouver une classe implantant **x**.

Héritage et visibilité (II)

Quiz 1 (Ces programmes sont-ils valides?)

```
class B {
  protected val x=0}
class A {
  val y=18}
val a= new A
a.x
```

V Oui R Non

```
class B {
  private val x=0
  def f=x}
class A extends B {
  val y=18}
(new A).f
```

V Oui R Non

```
class C {
  val x=0}
class B extends C
class A extends B
val a= new A
a.x
```

V Oui R Non

```
class C {
  val x=0}
class B extends C
class A extends B{
  private val x=1}
(new A).x
```

V Oui R Non

Héritage et redéfinition

Tous les champs/méthodes hérités peuvent être redéfinis avec **override**

Exemple 7 (Redéfinition de champs/méthodes avec **override**)

```
class B {
  val x=0
  def f(y:Int)=y+1
}
class A extends B{
  override val x=18
  override def f(z:Int)=20
}
```

```
scala> val a=new A
scala> a.x
Int = 18
scala> a.f(18)
Int = 20
```

Héritage et redéfinition (II)

Quiz 2 (Quels sont les résultats attendus pour ces programmes?)

```
class B {
  def f(y:Int)=y+1
}
class A extends B {
  override def f(z:Int)=18
}
val a= new A
val b= new B
println(a.f(10)+b.f(10))
```

V 29 R 36

```
class B {
  val x=0
  def f(y:Int)=x
}
class A extends {
  override val x=18
}
val a= new A
println(a.f(10))
```

V 18 R 0

Quiz 3 (Ce programme rend V 2 R "11" ?)

```
class C {
  def f(x:Int)=x+1
}
class B extends C{
  def f(x:String)=x+1
}
class A extends B{
  val a= new A
  println(a.f(1))
}
```

Héritage et redéfinition... le cas de toString et equals

Tout objet hérite de la classe `Any` de Scala qui définit un certain nombre de méthodes par défaut : `toString`, `equals`, `hashCode`, ...

Pourquoi redéfinir toString?

Par défaut, `toString` affiche la référence de l'objet.

Exemple 8

Dans la classe `Rational` du CM1, on a redéfini `toString` (CM5.scala).

Pourquoi redéfinir equals?

Par défaut, `equals` compare les références sur les objets.

Exemple 9

A partir du code de la classe `Rational`, construire des ensembles de rationnels. Que se passe-t-il si on ajoute des objets rationnels différents mais avec de même valeur ?

Redéfinition de equals

Comment redéfinir equals, == et !=?

- Toute égalité de 2 objets Scala (`==` et `!=`) est réalisée par `equals`
- Par défaut, `equals` compare les références des objets (sauf `case class` où il compare les valeurs de tous les champs du constructeur)
- La méthode à redéfinir est `equals(o:Any):Boolean`

```
class Rational(x:Int,y:Int){ val n=x; val d=y ; ...
  override def equals(o:Any)=
    o match {
      case e:Rational => e.n==this.n && e.d==this.d
      case _ => false }}
```

Exercice 9

Redéfinissez `equals` dans `Rational` et observez l'impact sur les ensembles de nombres rationnels.

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Traits étendus

- Les traits ne sont pas nécessairement *abstracts* et peuvent contenir une implantation
- Dans les traits, les déclarations de méthodes peuvent contenir du code et les champs peuvent contenir des valeurs

Exercice 10

Définissez un trait `Aire` avec deux champs `unite:String` et `taille:Int` ainsi qu'une méthode `toString`. Ensuite, définissez une classe `Rectangle` et une classe `Cercle` implantant `Aire`.

Remarque 2 (Implantation multiple)

Si une classe `A` implante deux traits `B` et `C` : `A extends B with C`. Si `B` et `C` définissent un même champ/méthode `f`, c'est la définition de `C` qui l'emporte.

Traits étendus (II)

Quizz 4 (Qu'affichent les programmes suivants ?)

```
trait A{
  override def toString="toto"
}
trait B{
  override def toString="tutu"
}
```

```
object C extends A with B
println(C)
```

"tutu" "toto"

```
trait A{
  def f(x:Int)="toto"
}
trait B{
  def f(x:Any)="tutu"
}
```

```
class C extends A with B
val c=new C
println(c.f(10))
```

"tutu" "toto"

Plan

- 1 Fonctions d'ordre supérieur et fonctions anonymes
- 2 Polymorphisme
- 3 Traits et fabriques d'objets
- 4 Héritage, visibilité et redéfinition
- 5 Traits étendus
- 6 Exceptions

Des objets spécifiques : les exceptions

- Les exceptions de Scala se comportent comme celles de Java
- Déclenchement par `throw` sur un objet de type exception
e.g. `throw new IllegalArgumentException` (Exception Java)
- Tant qu'elle n'est pas capturée une exception interrompt l'exécution de la méthode courante et remonte la pile des méthodes appelantes
- La capture se fait par une construction `try-catch-case` :

```
try (m(3)=="ok") catch {
  case e:RuntimeException => throw e
  case _:NoSuchElementException => false }
}
```

- On définit de nouvelles exceptions en étendant la classe `Throwable` :

```
class monException(x:String) extends Throwable{
  val content=x
}
```