Semaine 3 : Fonctions et données

Dans cette section nous allons apprendre comment les fonctions créent et encapsulent des structures de données.

Exemple : Les nombres rationnels

Nous voulons concevoir un paquetage pour faire de l'arithmétique rationnelle.

On représente un nombre rationnel $\frac{x}{y}$ par deux entiers :

- son *numérateur* x, et
- son *dénominateur* y.

Mettons que nous voulions implanter l'addition de deux rationnels.

On pourrait définir les deux fonctions

```
def addRationalNumerator(n1: int, d1: int, n2: int, d2: int): int;
def addRationalDenominator(n1: int, d1: int, n2: int, d2: int): int;
```

mais il serait alors difficile de gérer tous ces numérateurs et dénominateurs.

Un meilleur choix consiste à regrouper le numérateur et le dénominateur d'un rationnel dans une structure de données.

En Scala, on réalise cela en définissant une classe:

```
class Rational(x: int, y: int) {
  def numer = x;
  def denom = y
}
```

La définition précédente introduit deux entités :

- Un nouveau type, nommé *Rational*.
- Un constructeur *Rational* pour créer des éléments de ce type.

Scala garde les noms des types et des valeurs dans des espaces de noms différents. Il n'y a donc pas de conflit entre les deux définitions de *Rational*.

On appelle les éléments d'un type classe des objets.

On crée un objet en préfixant une application de constructeur de classe avec **new**, par ex. **new** *Rational*(1, 2).

Membres d'un objet

Les objets de la classe *Rational* ont deux membres, *numer* et *denom*.

On sélectionne les membres d'un objet avec l'opérateur infixe '.' (c.-à-d. comme en Java).

Exemple :

```
? val x = new Rational(1, 2)
? x.numer
1
? x.denom
2
```


Travailler avec les objets

On peut maintenant définir les fonctions arithmétiques qui implantent les règles standards.

$$
\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2} \\
\frac{n_1}{d_1} \Big/ \frac{n_2}{d_2} &= \frac{n_1 d_2}{d_1 n_2} \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} &\quad \text{iff} \quad n_1 d_2 = d_1 n_2
\end{aligned}
$$

Par exemple

```
def addRational(r: Rational, s: Rational): Rational =
  new Rational(
    r.numer * s.denom + s.numer * r.denom,
    r.denom * s.denom);

def makeString(r: Rational) =
    r.numer + "/" + r.denom;

? makeString(addRational(new Rational(1, 2), new Rational(2, 3)))
7/6
```


Méthodes

On pourrait aller plus loin et empaqueter aussi les fonctions opérant sur une abstraction de donnée dans l'abstraction de donnée elle-même.

De telles fonctions sont appelées des méthodes.

Exemple : Les nombres rationnels auraient maintenant, en plus des fonctions *numer* et *denom*, les fonctions *add*, *sub*, *mul*, *div*, *equal*, *toString*.

On aurait par exemple l'implantation suivante :

```
class Rational(x: int, y: int) {
  def numer = x;
  def denom = y;
  def add(r: Rational) =
    new Rational(
      numer * r.denom + r.numer * denom,
      denom * r.denom);
  def sub(r: Rational) =
```

```
      ...
    ...
    override def toString() = numer + "/" + denom;
}
```

Remarque: le modificateur ***override*** déclare que *toString* redéfinit une autre méthode (celle dans la classe *java.lang.Object*).

Voici un client de la nouvelle abstraction *Rational* :

```
? val x = new Rational(1, 3)
? val y = new Rational(5, 7)
? val z = new Rational(3, 2)
? x.add(y).mul(z)
66/42
```


Abstraction de donnée

L'exemple précédent a montré que les nombres rationnels ne sont pas toujours représentés sous leur forme la plus simple. (Pourquoi ?)

On s'attendrait à ce que les rationnels soient réduits à leurs plus petits numérateurs et dénominateurs en divisant par leur diviseur commun.

On pourrait implanter cela dans chaque opération des rationnels. Mais il serait alors facile d'oublier cette division dans une opération.

Une meilleure alternative consiste à normaliser la représentation dans la classe, au moment où les objets sont construits :

```
class Rational(x: int, y: int) {
  private def gcd(a: int, b: int): int = if (b == 0) a else gcd(b, a % b);
  private val g = gcd(x, y);
  def numer = x / g;
  def denom = y / g;
  ...
}
```

gcd et *g* sont des membres privés ; on ne peut y accéder que depuis l'intérieur de la classe *Rational*.

Avec cette définition, on obtient :

```
? val x = new Rational(1, 3)
? val y = new Rational(5, 7)
? val z = new Rational(3, 2)
? x.add(y).mul(z)
11/7
```

Dans cet exemple, on calcule *gcd* immédiatement, car on s'attend à ce que les fonctions *numer* et *denom* soient appelées souvent.

Il est aussi possible d'appeler *gcd* dans le code de *numer* et *denom* :

Par ex.

```
class Rational(x: int, y: int) {
  private def gcd(a: int, b: int): int = if (b == 0) a else gcd(b, a % b);
  def numer = x / gcd(x, y);
  def denom = y / gcd(x, y);
}
```

Cela peut être avantageux si on s'attend à ce que les fonctions *numer* et *denom* soient appelées peu souvent.

Les clients observent dans chaque cas exactement le même comportement.

Cette faculté de pouvoir choisir différentes implantations des données sans affecter les clients est appelée abstraction de donnée.

C'est l'un des pilliers du génie logiciel.

Auto-référence

A l'intérieur d'une classe, le nom ***this*** représente l'objet dont on exécute la méthode.

Exemple : Ajout des fonctions *less* et *max* à la classe *Rational*.

```
class Rational(x: int, y: int) {
  ...
  def less(that: Rational) =
    numer * that.denom < that.numer * denom;
  def max(that: Rational) = if (this.less(that)) that else this;
}
```

Remarquez qu'un nom simple *x*, qui fait référence à un autre membre de la classe, n'est qu'une abréviation pour ***this***.*x*. Ainsi, on aurait pu formuler *less* de façon équivalente comme suit.

```
  def less(that: Rational) =
    this.numer * that.denom < that.numer * this.denom;
```


Classes et substitutions

On a défini précédemment la signification d'une application de fonction en utilisant le modèle de calcul basé sur la substitution. On étend maintenant ce modèle aux classes et objets.

Question : Comment une instanciation de classe ***new*** $C(e_1, ..., e_m)$ est-elle évaluée ?

Réponse : Les expressions arguments $e_1, ..., e_m$ sont évaluées comme les arguments d'une fonction normale. C'est tout. L'expression résultante, disons ***new*** $C(v_1, ..., v_m)$, est déjà une valeur.

Maintenant supposons qu'on ait une définition de classe

```
  class C(x1, ..., xm) { ... def f(y1, ..., yn) = b ... }
```

où

- Les paramètres formels de la classe sont $x_1, ..., x_m$.
- La classe définit une fonction f avec paramètres formels $y_1, ..., y_n$.

(La liste de paramètres de la fonction peut être absente. Pour simplifier, on a omis le type des paramètres.)

Question : Comment l'expression ***new*** $C(v_1, ..., v_m).f(w_1, ..., w_n)$ est-elle évaluée ?

Réponse : L'expression se réécrit en :

$$[w_1/y_1, ..., w_n/y_n]\ [v_1/x_1, ..., v_m/x_m]\ [\textbf{\textit{new}}\ C(v_1, ..., v_m)/\textbf{\textit{this}}]\ b$$

Il y a trois substitutions à l'oeuvre ici :

- la substitution des paramètres formels $y_1, ..., y_n$ de la fonction f par les arguments effectifs $w_1, ..., w_n$,
- la substitution des paramètres formels $x_1, ..., x_m$ de la classe C par les arguments de classe effectifs $v_1, ..., v_m$,
- la substitution de l'auto-référence ***this*** par la valeur de l'objet ***new*** $C(v_1, ..., v_n)$.

Exemples de réécriture

new *Rational*(*1*, *2*).*numer*
→
1

new *Rational*(*1*, *2*).*denom*
→
2

new *Rational*(*1*, *2*).*less*(***new*** *Rational*(*2*, *3*))
→
new *Rational*(*1*, *2*).*numer* ∗ ***new*** *Rational*(*2*, *3*).*denom* <
new *Rational*(*2*, *3*).*numer* ∗ ***new*** *Rational*(*1*, *2*).*denom*
→ ... →
$1 * 3 < 2 * 2$
→ ... →
true

Opérateurs

En principe, les nombres rationnels définis par *Rational* sont aussi "naturels" que les entiers.

Mais pour l'utilisateur de ces abstractions, il y a une différence apparente :

- On écrit $x + y$, si x et y sont des entiers, mais
- on écrit $r.add(s)$ si r et s sont des nombres rationnels.

En Scala, on peut éliminer cette différence. On procède en deux étapes.

Etape 1 Toute méthode avec un paramètre peut être utilisée comme un opérateur infixe.

Il est donc possible d'écrire

r add s		*r.add(s)*
r less s	*à la place de*	*r.less(s)*
r max s		*r.max(s)*

Etape 2 Les opérateurs peuvent être utilisés comme des identificateurs.

Ainsi, un identificateur peut être :

- Une lettre, suivie d'une séquence de lettres ou de chiffres
- Un symbole d'opérateur, suivi par d'autres symboles d'opérateurs.

La priorité d'un opérateur est déterminée par son premier caractère.

Le tableau suivant liste les caractères par ordre croissant de priorité :

(*toutes les lettres*)
|
^
&
< >
= !
:
+ −
* / %
(*tous les autres caractères spéciaux*)

Par conséquent, on peut définir *Rational* plus naturellement ainsi :

```
class Rational(x: int, y: int) {
    private def gcd(a: int, b: int): int = if (b == 0) a else gcd(b, a % b);
    private val g = gcd(x, y);
    def numer = x / g;
    def denom = y / g;
    def + (r: Rational) =
        new Rational(
            numer * r.denom + r.numer * denom,
            denom * r.denom);
    def - (r: Rational) =
        new Rational(
            numer * r.denom - r.numer * denom,
            denom * r.denom);
    def * (r: Rational) =
        new Rational(
            numer * r.numer,
            denom * r.denom);
    ...
    override def toString() = numer + "/" + denom;
}
```


... et les nombres rationnels peuvent être utilisés comme *int* ou *float* :

```
? val x = new Rational(1, 2)
? val y = new Rational(1, 3)
? x * x + y * y
13/36
```

Classes abstraites

Considérons la tâche d'écrire une classe pour les ensembles de nombres entiers avec les opérations suivantes.

```
abstract class IntSet {
    def incl(x: int): IntSet;
    def contains(x: int): boolean
}
```

IntSet est une classe abstraite.

Les classes abstraites peuvent contenir des membres dont il manque l'implantation (dans notre cas *incl* et *contains*).

Par conséquent aucun objet d'une classe abstraite ne peut être instancié avec *new*.

Extensions d'une classe

On envisage d'implanter les ensembles comme des arbres binaires.

Il y a deux sortes d'arbres possibles : Un arbre pour l'ensemble vide, et un arbre consistant en un entier et deux sous-arbres.

Voici leurs implantations.

```
class Empty extends IntSet {
    def contains(x: int): boolean = false;
    def incl(x: int): IntSet = new NonEmpty(x, new Empty, new Empty);
}
```

```
class NonEmpty(elem:int, left:IntSet, right:IntSet) extends IntSet {
  def contains(x: int): boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: int): IntSet =
    if (x < elem) new NonEmpty(elem, left incl x, right)
    else if (x > elem) new NonEmpty(elem, left, right incl x)
    else this;
}
```

Remarques :

- *Empty* et *NonEmpty* étendent toutes deux la classe *IntSet*.
- Cela signifie que les types *Empty* et *NonEmpty* sont conformes au type *IntSet* : un objet de type *Empty* ou *NonEmpty* peut être utilisé partout où un objet de type *IntSet* est requis.

Classes de base et sous-classes

- *IntSet* est appelée classe de base de *Empty* et *NonEmpty*.
- *Empty* et *NonEmpty* sont des sous-classes de *IntSet*.
- En Scala toute classe définie par l'utilisateur étend une autre classe.
- En l'absence de clause ***extends***, la classe *scala.Object* est implicite.
- Les sous-classes héritent de tous les membres de leur classe de base.
- Les définitions de *contains* et *incl* dans les classes *Empty* et *NonEmpty* implantent les fonctions abstraites de la classe de base *IntSet*.
- Il est aussi possible de redéfinir une définition existante, non-abstraite, dans une sous-classe, en utilisant ***override***.

Exemple :

```
abstract class Base {
  def foo = 1;
  def bar: int;
}
```

```
class Sub extends Base {
  override def foo = 2;
  def bar = 3;
}
```

Exercice : Ecrire des méthodes *union* et *intersection* pour former l'union et l'intersection de deux ensembles.

Exercice : Ajouter une méthode

```
def excl(x: int)
```

qui retourne l'ensemble donné sans l'élément x. Pour y parvenir, il est utile d'implanter aussi une méthode de test

```
def isEmpty: boolean
```

pour les ensembles.

Liaison dynamique

- Les langages orientés-objets (Scala y compris) implantent la sélection dynamique de méthodes.
- Cela signifie que le code invoqué par un appel de méthode dépend du type à l'exécution de l'objet qui contient la méthode.

Exemple :

```
  (new Empty).contains(7)
→
  false
```

Exemple :

```
    (new NonEmpty(7, new Empty, new Empty)).contains(1)
→
    if (1 < 7) new Empty contains 1
    else if (1 > 7) new Empty contains 1
    else true
→
    new Empty contains 1
→
    false
```

La sélection dynamique de méthodes est analogue aux appels de fonctions d'ordre supérieur.

Question :

Peut-on implanter un concept en termes de l'autre ?

Classes standards

En fait, les types tels que *int* ou *boolean* ne sont pas traités de façon particulière en Scala.

Ils sont définis dans le module *Predef* comme des alias pour des classes Scala:

```
type boolean = scala.Boolean;
type int = scala.Int;
type long = scala.Long;
...
```

Pour des raisons d'efficacité, le compilateur représente habituellement les valeurs de type *scala.Int* par des entiers de 32 bits, et les valeurs de type *scala.Boolean* par des booléens Java, etc.

Mais c'est juste une optimisation, cela n'a aucun effet sur la signification d'un programme.

Voici une implantation possible de la classe *Boolean*.

La classe Boolean

```
package scala;
trait Boolean {
  def ifThenElse[a](t: => a)(e: => a): a;

  def && (x: => Boolean): Boolean  =  ifThenElse[Boolean](x)(false);
  def || (x: => Boolean): Boolean  =  ifThenElse[Boolean](true)(x);
  def !                 : Boolean  =  ifThenElse[Boolean](false)(true);

  def == (x: Boolean): Boolean  =  ifThenElse[Boolean](x)(x.!);
  def != (x: Boolean): Boolean  =  ifThenElse[Boolean](x.!)(x);
  def <  (x: Boolean): Boolean  =  ifThenElse[Boolean](false)(x);
  def >  (x: Boolean): Boolean  =  ifThenElse[Boolean](x.!)(false);
  def ≤  (x: Boolean): Boolean  =  ifThenElse[Boolean](x)(true);
  def ≥  (x: Boolean): Boolean  =  ifThenElse[Boolean](true)(x.!);
}

val true = new Boolean { def ifThenElse[a](t: => a)(e: => a) = t }
val false = new Boolean { def ifThenElse[a](t: => a)(e: => a) = e }
```


La classe Int

Voici une spécification partielle de la classe *Int*.

```
class Int extends Long {
  def + (that: Double): Double;
  def + (that: Float): Float;
  def + (that: Long): Long;
  def + (that: Int): Int;         /* idem pour −, ∗, /, %  */

  def << (cnt: Int): Int;         /* idem pour >>, >>>  */

  def & (that: Long): Long;
  def & (that: Int): Int;         /* idem pour |, ˆ */

  def == (that: Double): Boolean;
  def == (that: Float): Boolean;
  def == (that: Long): Boolean;
          /*  idem pour !=, <, >, ≤, ≥ */
}
```

Exercice : Donner une implantation de la classe suivante qui représente les entiers non-négatifs.

```
abstract class Nat {
    def isZero(): Boolean;
    def predecessor: Nat;
    def successor: Nat;
    def + (that: Nat): Nat;
    def − (that: Nat): Nat;
}
```

Ne pas utiliser les classes numériques standards dans cette implantation.

Implanter plutôt deux sous-classes

```
class Zero extends Nat;
class Succ(n: Nat) extends Nat;
```

l'une pour le nombre zéro ; l'autre pour les nombres strictement positifs.

Orientation objet pure

Un langage orienté-objet pur est un langage dans lequel chaque valeur est un objet.

Si le langage est basé sur les classes, cela signifie que le type de chaque valeur est une classe.

Scala est-il un langage orienté-objet pur ?

Nous avons vu que les types numériques Scala et le type *boolean* peuvent être implantés comme des classes normales.

Nous verrons la semaine prochaine que les fonctions peuvent aussi être vues comme des objets.

Le type fonctionnel $A \Rightarrow B$ est traité comme une abréviation pour les objets possédant une méthode d'application :

```
def apply(x: A): B
```

Résumé

- Nous avons vu comment implanter des structures de données avec des classes.
- Une classe définit un type et une fonction pour créer des objets de ce type.
- Les objets ont pour membres des fonctions qu'on sélectionne en utilisant '.' (infixe).
- Les classes et les membres peuvent être abstraits, c.-à-d. donnés sans implantation concrète.
- Une classe peut étendre une autre classe.
- Si la classe A étend B alors le type A se conforme au type B. Autrement dit des objets de type A peuvent être utilisés partout où des objets de type B sont requis.

Éléments du langage introduits cette semaine

Types :

$$\mathit{Type} \quad = \ldots \quad | \quad \mathit{ident}$$

Un type peut maintenant être un identificateur, c.-à-d. le nom d'une classe.

Expressions :

$$\mathit{Expr} \quad = \ldots \quad | \quad \mathit{Expr}\ \text{'.'}\ \mathit{ident} \quad | \quad \textbf{\textit{new}}\ \mathit{Expr}$$

Une expression peut maintenant être une création d'objet ou une sélection $E.m$ d'un membre m d'une expression E dont la valeur est un objet.

Définitions :

Def	= *FunDef* | *ValDef* | *ClassDef*
ClassDef	= [***abstract***] ***class*** *ident* ['(' [*Parameters*] ')']
	[***extends*** *Expr*] ['{' {*TemplateDef*} '}']
TemplateDef	= [*Modifier*] *Def*
Modifier	= ***private*** | ***override***

Une définition peut maintenant être une définition de classe telle que

class *C*(*params*) ***extends*** *B* { *defs* }

Les définitions *defs* dans une classe peuvent être précédées des modificateurs ***private*** ou ***override***.