Une meilleure alternative 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) with {
  def numer = x
  def denom = y
}
```

La définition précédente introduit deux entités :

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

La définition précédente est à peu près équivalente à :

```
type Rational = ...
def Rational(x: Int, y: Int): Rational = ...
```

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

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*.

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 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.

Travailler avec les Objets

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

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

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 = Rational(1, 2)
> x.numer
1
> x.denom
2
```

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 pourrait avoir l'implantation suivante :

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


Par exemple

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

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

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

Abstraction de donnée

L'exemple précédent a montré que les nombres rationnels ne sont pas toujours représentés sous leur forme la plus courte. (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 :


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

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

Voici un client de la nouvelle abstraction des rationnels :

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

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

Par ex.

```
class Rational(x: Int, y: Int) with {
  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 *numer* et *denom* soit appelés 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 des données.

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


```
class Rational(x: Int, y: Int) with {
  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 = Rational(1, 3)
> val y = Rational(5, 7)
> val z = Rational(3, 2)
> x.add(y).mul(z)
33/21
```

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

Classes et Substitutions

Précédemment, on a définit 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 $C(e_1, ..., e_m)$ 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 $C(v_1, ..., v_m)$, est déjà une valeur.

Maintenant supposons qu'on a une définition de classe

$$\textbf{\textit{class}}\ C(x_1, ..., x_m)\ \textbf{\textit{with}}\ \{\ ...\ \textbf{\textit{def}}\ f(y_1, ..., y_n) = b\ ...\ \}$$

où

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

Auto-référence

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

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

```
class Rational(x: Int, y: Int) with {
  ...
  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 abbréviation 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
```

Exemples de réécriture

Exemple :

$$
\begin{array}{ll}
 & \mathit{Rational}(1,\,2).\mathit{numer} \\
\rightarrow & \\
 & 1 \\
 & \mathit{Rational}(1,\,2).\mathit{denom} \\
\rightarrow & \\
 & 2 \\
 & \mathit{Rational}(1,\,2).\mathit{less}(\mathit{Rational}(2,\,3)) \\
\rightarrow & \\
 & \mathit{Rational}(1,\,2).\mathit{numer} * \mathit{Rational}(2,\,3).\mathit{denom} < \\
 & \mathit{Rational}(2,\,3).\mathit{numer} * \mathit{Rational}(1,\,2).\mathit{denom} \\
\rightarrow \ldots \rightarrow & \\
 & 1 * 3 < 2 * 2 \\
\rightarrow \ldots \rightarrow & \\
 & \mathit{True}
\end{array}
$$

(Chacune des listes de paramètres peut être absente. Pour simplifier, on a omis le type des paramètres.)

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

Réponse : L'expression se réécrit en :

$$
\begin{array}{l}
[w_1/y_1,\,\ldots,\,w_n/y_n] \\
\quad [v_1/x_1,\,\ldots,\,v_m/x_m] \\
\quad\quad [C(v_1,\,\ldots,\,v_m)/\textbf{\textit{this}}]\ b
\end{array}
$$

Il y a trois substitutions à l'oeuvre ici :

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

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 :

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.

... et les nombres rationnels peuvent être utilisés comme *Int* ou *Float* :

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



```
class Rational(x: Int, y: Int) with {
  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) =
    Rational(
      numer * r.denom + r.numer * denom,
      denom * r.denom)
  def - (r: Rational) =
    Rational(
      numer * r.denom - r.numer * denom,
      denom * r.denom)
  def * (r: Rational) =
    Rational(
      numer * r.numer,
      denom * r.denom)
  ...
  override def toString() = numer + "/" + denom;
}
```

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 with {
  def contains(x: Int): Boolean = False
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
```


Classes Abstraites

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

```
abstract class IntSet with {
  abstract def incl(x: Int): IntSet
  abstract def contains(x: Int): Boolean
}
```

La classe *IntSet* et ses opérations sont marquées ***abstract***.

Cela signifie qu'aucune implantation n'est donnée, et qu'aucun objet de la classe *IntSet* ne peut être instancié.

Classes de Bases 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 :

```
class Base with {
  def foo = 1
  abstract def bar: Int
}
```

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



```
class NonEmpty(elem:Int, left:IntSet, right:IntSet) extends IntSet with {
  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) NonEmpty(elem, left incl x, right)
    else if (x > elem) NonEmpty(elem, left, right incl x)
    else this
}
```

Remarques :

- *Empty* et *NonEmpty* étendent tous deux la classe *IntSet*.
- Cela signifie que
  - Les types *Empty* et *NonEmpty* se conforment au type *IntSet* : un objet de type *Empty* ou *NonEmpty* peut être utilisé partout où un objet de type *IntSet* est requis.

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 :

$$\begin{array}{ll} & \mathit{Empty.contains}(7) \\ \rightarrow & \\ & \mathit{False} \end{array}$$

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

Exercice : Ajouter une méthode

$$\textbf{def}\ \mathit{excl}(x\colon \mathit{Int})$$

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

$$\textbf{def}\ \mathit{isEmpty}\colon \mathit{Boolean}$$

pour les ensembles.

Classes Standards

En fait, les types tels que *Int* ou *Boolean* n'ont pas besoin d'un traitement particulier, en Scala ; on peut les définir comme n'importe quelle autre classe Scala.

Exemple : Voici comment on peut définir la classe *Boolean*.

(Dans l'implantation actuelle, les objets *Boolean* sont effectivement associés au type *boolean*, mais c'est transparent pour l'utilisateur.)

Exemple :

$$
\begin{array}{ll}
 & \mathit{NonEmpty}(7, \mathit{Empty}, \mathit{Empty}).\mathit{contains}(1) \\
\rightarrow & \\
 & \textbf{if } (1 < 7)\ \mathit{Empty\ contains\ 1} \\
 & \textbf{else if } (1 > 7)\ \mathit{Empty\ contains\ 1} \\
 & \textbf{else } \mathit{True} \\
\rightarrow & \\
 & \mathit{Empty\ contains\ 1} \\
\rightarrow & \\
 & \mathit{False}
\end{array}
$$

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 ?

La classe Int

Voici une spécification partielle de la classe *Int*.

```
class Int extends Long with {
  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 |, ^ */

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


Class Boolean

```
abstract class Boolean with {
  abstract def ifThenElse[a](def t: a)(def e: a): a

  def && (def x: Boolean): Boolean  =  ifThenElse[Boolean](x)(False)
  def || (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.!)
}
class True extends Boolean with {
  def ifThenElse[a](def t: a)(def e: a): a = t }
class False extends Boolean with {
  def ifThenElse[a](def t: a)(def e: a): a = e }
```

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 $(A)B$ est traité comme une abbréviation pour les objets possédant une méthode d'application :

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


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

```
abstract class Nat with {
  abstract def isZero(): Boolean
  abstract def predecessor: Nat
  abstract def successor: Nat
  abstract def +(that: Nat): Nat
  abstract 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.

Éléments du Langage Introduits Cette Semaine

Types :

Type = ... | *ident*

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

Expressions :

Expr = ... | *Expr* '.' *ident*

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

Résumé

- Nous avons vu comment implanter des structures de données avec des classes.
- Une classe définit une 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.

Définitions :

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

Une définition peut maintenant être une définition de classe telle que

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

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