Week 3: Functions and Data

In this section, we'll learn how functions create and encapsulate data structures.

Exemple : Rational Numbers

We want to design a package for doing rational arithmetic.

A rational number $\frac{x}{y}$ is represented by two integers:

- its *numerator* x, and
- its *denominator* y.

Suppose we want to implement the addition of two rational numbers.

One could define the two functions

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

but it would be difficult to manage all these numerators and denominators.

A better choice is to combine the numerator and denominator of a rational number in a data structure.

In Scala, we do this by defining a class:

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

The definition above introduces two entities:

- A new type, named *Rational*.
- A constructor *Rational* to create elements of this type.

Scala keeps the names of types and values in different namespaces. So there's no conflict between the two defintions of *Rational*.

We call the elements of a class type objects.

We create an object by prefixing an application of the constructor of the class with the operator ***new***, for example ***new*** *Rational(1, 2)*.

Members of an object

Objects of the class *Rational* have two members, *numer* and *denom*.

We select the members of an object with the infix operator '.' (like in Java).

Exemple :

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


Working with objects

We can now define the arithmetic functions that implement the standard rules.

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} \Big/ \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \quad \text{iff} \quad n_1 d_2 = d_1 n_2$$

Exemple :

```
scala> def addRational(r: Rational, s: Rational): Rational =
         new Rational(
           r.numer * s.denom + s.numer * r.denom,
           r.denom * s.denom)
scala> def makeString(r: Rational) =
         r.numer + "/" + r.denom
scala> makeString(addRational(new Rational(1, 2), new Rational(2, 3)))
7/6
```


Methods

One could go further and also package functions operating on a data abstraction in the data abstraction itself.

Such functions are called methods.

Exemple : Rational numbers now would have, in addition to the functions *numer* and *denom*, the functions *add*, *sub*, *mul*, *div*, *equal*, *toString*.

One might, for example, implement this as follows:

```
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
}
```

Remark: the modifier ***override*** declares that *toString* redefines a method that already exists (in the class *java.lang.Object*).

Here is how one might use the new *Rational* abstraction:

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


Data Abstraction

The previous example has shown that rational numbers aren't always represented in their simplest form. (Why?)

One would expect the rational numbers to be reduced to their smallest numerator and denominator by dividing them by their divisor.

We could implement this in each rational operation, but it would be easy to forget this division in an operation.

A better alternative consists of normalizing the representation in the class when the objects are constructed:

```
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 and *g* are private members; we can only access them from inside the *Rational* class.

With this definition, we obtain:

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

In this example, we calculate *gcd* immediately, because we expect that the functions *numer* and *denom* are often called.

It is also possible to call *gcd* in the code of *numer* and *denom*:

For example,

```
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)
}
```

This can be advantageous if it is expected that the functions *numer* and *denom* are called infrequently.

Clients observe exactly the same behavior in each case.

This ability to choose different implementations of the data without affecting clients is called data abstraction.

It is a cornerstone of software engineering.

Self Reference

On the inside of a class, the name ***this*** represents the object on which the current method is executed.

Exemple : Add the functions *less* and *max* to the class *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

}
```

Note that a simple name *x*, which refers to another member of the class, is an abbreviation of ***this***.*x*. Thus, an equivalent way to formulate *less* is as follows.

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


Constructors

The constructor introduced with the new type *Rational* is called the primary constructor of the class.

Scala also allows the declaration of auxillary constructors named ***this***.

Exemple : Add an auxillary constructor to the class *Rational*.

```
class Rational(x: Int, y: Int) {
  def this(x: Int) = this(x, 1)

  //...
}
```

With this definition, we obtain:

```
scala> val x = new Rational(2)
scala> val y = new Rational(1, 2)
scala> x.mul(y)
1/1
```

Classes and Substitutions

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

Question: How is an instantiation of the class ***new*** $C(e_1, ..., e_m)$ evaluted?

Answer: The expression arguments $e_1, ..., e_m$ are evaluated like the arguments of a normal function. That's it. The resulting expresion, say, ***new*** $C(v_1, ..., v_m)$, is already a value.

Now suppose that we have a class definition,

class $C(x_1, ..., x_m)$ { ... ***def*** $f(y_1, ..., y_n) = b$ ... }

where

- The formal parameters of the class are $x_1, ..., x_m$.
- The class defines a method f with formal parameters $y_1, ..., y_n$.

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

Question: How is the expression ***new*** $C(v_1, ..., v_m).f(w_1, ..., w_n)$ evaluated?

Answer: The expression can be rewritten as:

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

There are three substitutions at work here:

1. the substitution of the formal parameters $y_1, ...,$ y_n of the function f by the arguments $w_1, ..., w_n$,
2. the substitution of the formal parameters $x_1, ...,$ x_m of the class C by the class arguments $v_1, ..., v_m$,
3. the substitution of the self reference ***this*** by the value of the object ***new*** $C(v_1, ..., v_n)$.

Examples of Rewriting

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

Operators

In principle, the rational numbers defined by *Rational* are as natural as integers.

But for the user of these abstractions, there is a noticeable difference:

- We write $x + y$, if x and y are integers, but
- We write $r.add(s)$ if r and s are rational numbers.

In Scala, we can eliminate this difference. We procede in two steps.

Step 1 Any method with a parameter can be used like an infix operator.

It is therefore possible to write

r add s		*r.add(s)*
r less s	*in place of*	*r.less(s)*
r max s		*r.max(s)*

Step 2 Operators can be used as identifiers.

Thus, an identifier can be:

- A letter, followed by a sequence of letters or numbers
- An operator symbol, followed by other operator symbols.

The priority of an operator is determined by its first character.

The following table lists the characters in ascending order of priority:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

Therefore, we can define *Rational* more naturally:


```
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
}
```


... and rational numbers can be used like *Int* or *Double*:

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


Abstract Classes

Consider the task of writing a class for sets of integers with the following operations.

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

IntSet is an abstract class.

Abstract classes can contain members which are missing an implementation (in our case, *incl* and *contains*).

Consequently, no object of an abstract class can be instantiated with the operator *new*.

Class Extensions

Let's consider implementing sets as binary trees.

There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub trees.

Here are their implementations:

```
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 }
}
```

Remarks:

- *Empty* and *NonEmpty* both extend the class *IntSet*.
- This means that the types *Empty* and *NonEmpty* conform to the type *IntSet*: an object of type *Empty* or *NonEmpty* can be used wherever an object of type *IntSet* is required.

Base Classes and Subclasses

- *IntSet* is called a base class of *Empty* and *NonEmpty*.
- *Empty* and *NonEmpty* are subclasses of *IntSet*.
- In Scala, any user-defined class extends another class.
- In the absence of ***extends***, the class *scala.ScalaObject* is implicit.
- Subclasses inherit all the members of their base class.
- The definitions of *contains* and *incl* in the classes *Empty* and *NonEmpty* implement the abstract functions in the base class *IntSet*.
- It is also possible to redefine an existing, non-abstract definition in a subclass by using ***override***.

Exemple :

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

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


Exercice : Write the methods *union* and *intersection* for forming the union and the intersection of two sets.

Exercice : Add a method

```
def excl(x: Int): IntSet
```

which returns the given set without the element x. To achieve this, it is also useful to implement a test method

```
def isEmpty: Boolean
```

Dynamic Binding

- Object-oriented languages (including Scala) implement dynamic dispatch of methods.
- This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Exemple :

$$(\textbf{\textit{new}}\ \mathit{Empty}).\mathit{contains}(7)$$
$$\rightarrow$$
$$\textbf{\textit{false}}$$

Exemple :

$$(\textbf{\textit{new}}\ \mathit{NonEmpty}(7,\ \textbf{\textit{new}}\ \mathit{Empty},\ \textbf{\textit{new}}\ \mathit{Empty})).\mathit{contains}(1)$$
$$\rightarrow$$
$$\begin{array}{l} \textbf{\textit{if}}\ (1 < 7)\ \textbf{\textit{new}}\ \mathit{Empty\ contains}\ 1 \\ \textbf{\textit{else if}}\ (1 > 7)\ \textbf{\textit{new}}\ \mathit{Empty\ contains}\ 1 \\ \textbf{\textit{else true}} \end{array}$$
$$\rightarrow$$
$$\textbf{\textit{new}}\ \mathit{Empty\ contains}\ 1$$
$$\rightarrow$$
$$\textbf{\textit{false}}$$

Dynamic dispatch of methods is analogous to calls to higher-order functions.

Question:

Can we implement one concept in terms of the other?

Standard Classes

In fact, types such as *Int* or *Boolean* do not receive special treatment in Scala. They are like the other classes, defined in the package *scala*.

For reasons of efficiency, the compiler usually represents the values of type *scala.Int* by 32-bit integers, and the values of type *scala.Boolean* by Java's Booleans, etc.

But this is just an optimization, this doesn't have any effect on the meaning of a program.

Here is a possible implementation of the class *Boolean*.

The class 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 }
```

The class Int

Here is a partial specification of the class *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 |, ^ */

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


Exercice : Provide an implementation of the abstract class below that represents non-negative integers.

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

Do not use standard numerical classes in this implementation.

Rather, implement two subclasses.

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

One for the number zero, the other for strictly positive numbers.

Pure Object Orientation

A pure object-oriented language is one in which each value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a pure object-oriented language?

We have seen that Scala's numeric types and the *Boolean* type can be implemented like normal classes.

We'll see next week that functions can also be seen as objects.

The function type $A \Rightarrow B$ is treated like an abbreviation for objects that have a method for application:

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


Recap

- We have seen how to implement data structures with classes.
- A class defines a type and a function to create objects of that type.
- Objects have functions as their members which can be selected using the '.' infix operator.
- Classes and members can be abstract, i.e., provided without a concrete implementation.
- A class can extend another class.
- If the class A extends B then the type A conforms to type B.
  This means that objects of type A can be used wherever objects of type B are required.

Language Elements Introduced This Week

Types:

Type	= ... | *ident*	

A type can now be an identifier, i.e., a class name.

Expressions:

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

An expression can now be an object creation or a selection $E.m$ of a member m of an expression E whose value is an object

Definitions:

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

A definition can now be a class definition such as

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

Definitions *defs* in a class can be preceeded by modifiers **private**, **protected** or **override**.