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

```python
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 in a rational number in a data structure.

In Scala, we do this by defining a class:

```scala
class Rational(x: Int, y: Int) {
  def numerator = x
  def denominator = 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 definitions 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, `numerator` and `denominator`.

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

Exemple:

```scala
scala> val x = new Rational(1, 2)
scala> x.numerator
1
scala> x.denominator
2
```

Working with objects

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

\[
\begin{align*}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2} \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} &= \frac{n_1 d_2}{d_1 n_2} \\
\frac{n_1}{d_1} &= \frac{n_2}{d_2} \text{ iff } n_1 d_2 = d_1 n_2
\end{align*}
\]
Exemple :

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

Exemple :

```scala
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
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:

```scala
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)
val y = new Rational(5, 7)
val z = new Rational(3, 2)
x.add(y).mul(x)
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
}

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

For example,

```scala
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`.

```scala
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.

```scala
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 auxiliary constructors named `this`.

**Exemple** : Add an auxiliary constructor to the class `Rational`.

```scala
class Rational(x: Int, y: Int) {
  def this(x: Int) = this(x, 1)
  //...
}
```

With this definition, we obtain:

```scala
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 \( \text{new } C(e_1, \ldots, e_m) \) evaluated?

Answer: The expression arguments \( e_1, \ldots, e_m \) are evaluated like the arguments of a normal function. That’s it. The resulting expression, say, \( \text{new } C(v_1, \ldots, v_m) \), is already a value.

Now suppose that we have a class definition,

\[
\text{class } C(x_1, \ldots, x_m) \{ \ldots \text{def } f(y_1, \ldots, y_n) = b \ldots \}
\]

where

- The formal parameters of the class are \( x_1, \ldots, x_m \).
- The class defines a method \( f \) with formal parameters \( y_1, \ldots, y_n \).

Examples of Rewriting

\[
\text{new Rational}(1, 2).\text{numerator} \rightarrow 1
\]

\[
\text{new Rational}(1, 2).\text{denominator} \rightarrow 2
\]

\[
\text{new Rational}(1, 2).\text{less}(\text{new Rational}(2, 3)) \rightarrow \text{new Rational}(1, 2).\text{numerator} \ast \text{new Rational}(2, 3).\text{denominator} < \text{new Rational}(2, 3).\text{numerator} \ast \text{new Rational}(1, 2).\text{denominator}
\]

\[
\ldots \rightarrow 1 \ast 3 < 2 \ast 2
\]

\[
\ldots \rightarrow \text{true}
\]

Operators

In principle, the rational numbers defined by \( \text{Rational} \) 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 proceed in two steps.

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

It is therefore possible to write

\[
r.add(s) \quad \text{in place of} \quad r.add(s)
\]

\[
r.less(s) \quad \text{in place of} \quad r.less(s)
\]

\[
r.max(s) \quad \text{in place of} \quad r.max(s)
\]

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:

```scala
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 numerator = x / g
  def denominator = y / g
  def + (r: Rational) =
    new Rational(
      numerator * r.denominator + r.numerator * denominator,
      denominator * r.denominator
    )
  def − (r: Rational) =
    new Rational(
      numerator * r.denominator − r.numerator * denominator,
      denominator * r.denominator
    )
  def ∗ (r: Rational) =
    new Rational(
      numerator * r.numerator,
      denominator * r.denominator
    )
  override def toString() = numerator + "/" + denominator
}
```

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

```scala
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.

```scala
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`. 

---

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

```scala
class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}
```

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 :

```scala
abstract class Base {
  def foo = 0
  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

```scala
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

```scala
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:**

(new Empty).contains(7)
→ 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.

```scala
package scala

trait Boolean {
  def ifThenElse[a](t: ⇒ a)(e: ⇒ a) : a
  def && (x : ⇒ Boolean) : Boolean = ifThenElse[Boolean](x)(false)(true)
  def | (x : ⇒ Boolean) : Boolean = ifThenElse[Boolean](true)(x)(false)
  def != (x : Boolean) : Boolean = ifThenElse[Boolean](true)(false)(false)
  def < (x : Boolean) : Boolean = ifThenElse[Boolean](false)(true)(false)
  def > (x : Boolean) : Boolean = ifThenElse[Boolean](false)(false)(true)
  def ≤ (x : Boolean) : Boolean = ifThenElse[Boolean](true)(false)(true)
}

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
  def < < (cnt: Int): Int
  def & (that: Long): Long
  def & (that: Int): Int
  def == (that: Double): Boolean
  def == (that: Float): Boolean
  def == (that: Long): Boolean
  def == (that: Int): Boolean

  // idempotent - , /, %, */
  def < (that: Int): Int
  def < (that: Float): Float
  def < (that: Long): Long
  def < (that: Double): Double
  def <= (that: Int): Int
  def <= (that: Float): Float
  def <= (that: Long): Long
  def <= (that: Double): Double
  def > (that: Int): Int
  def > (that: Float): Float
  def > (that: Long): Long
  def > (that: Double): Double
  def >= (that: Int): Int
  def >= (that: Float): Float
  def >= (that: Long): Long
  def >= (that: Double): Double

  // idempotent !=, <, >, <=, >=
}

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 ⇒ B` is treated like an abbreviation for objects that have a method for application:

```scala
  def apply(x: A): B
```
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(par1, par2) extends B { def1, def2 }

Definitions def1 in a class can be preceded by modifiers private, protected or override.