Functions and Data

Principles of Functional Programming

Functions and Data

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

Example

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.

Rational Addition

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

```
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.

Classes

In Scala, we do this by defining a *class*:

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

This definition 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 entities named `Rational`.

Objects

We call the elements of a class type *objects*.

We create an object by calling the constructor of the class:

Example

```
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).

Example

```
val x = Rational(1, 2)   > x: Rational = Rational@2abe0e27
x.numer                  > 1
x.denom                  > 2
```

Rational Arithmetic

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

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

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

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

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

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

Implementing Rational Arithmetic

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

def makeString(r: Rational): String =
  s"${r.numer}/${r.denom}"
```

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

Note: s"..." in `makeString` is an *interpolated string*, with values `r.numer` and `r.denom` in the places enclosed by `${...}`.

Methods

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

Such functions are called *methods*.

Example

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

Methods for Rationals

Here's a possible implementation:

```
class Rational(x: Int, y: Int):
  def numer = x
  def denom = y
  def add(r: Rational) =
    Rational(numer * r.denom + r.numer * denom,
             denom * r.denom)
  def mul(r: Rational) = ...
  ...
  override def toString = s"$numer/$denom"
```

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

Calling Methods

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

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

Exercise

1. In your worksheet, add a method `neg` to class Rational that is used like this:

```
x.neg           // evaluates to -x
```

2. Add a method `sub` to subtract two rational numbers.
3. With the values of `x`, `y`, `z` as given in the previous slide, what is the result of

```
x - y - z
```

?