Functions and Data

In this section we will learn how functions create and encapsulate data structures.

Example: Rational Numbers

We want to design a package to do rational arithmetic.

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

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

Let's say we want to implement addition of two rational numbers.

We could define 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 hard to keep track of all the numerators and denominators.

A better alternative packs the numerator and denominator of a rational in one data structure.

In Scala, this is done by defining a class:

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

The previous definition introduces two entities:

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

Scala keeps names for types and values in different name spaces. Hence there is no conflict between the two defintions of *Rational*.

Elements of a class type are called objects.

Objects are created by prefixing a class constructor application with ***new***, e.g. ***new*** *Rational(1, 2)*.

Object Members

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

Members of an object are selected using infix '.' (i.e. as in Java).

Example:

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


Working with Objects

We can now define arithmetic functions on rationals which implement the standard rules:

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

For instance

```
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
```


Methods

We can go further and also package functions operating on a data abstraction with the data abstraction itself.

Such functions are also called methods.

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

An implementation could be 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;
    }
```

Note: the ***override*** modifier declares that *toString* overrides another method (i.e. the one in *java.lang.Object*).

Here is a client of the new *Rational* abstraction:

```
? 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
```


Data Abstraction

The previous example showed that rational numbers are not always represented in their shortest form. (Why?)

One would expect that rationals are reduced to smallest possible numerators and denominators by dividing with their common divisor.

This could be implemented in all operations on rationals. But then it is easy to forget the division in some operation.

A better alternative is to normalize the representation by the class at the point where 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; they can be accessed only from within class *Rational*.

With this definition, one gets:

```
? val x = new Rational(1, 3)
? val y = new Rational(5, 7)
? val z = new Rational(3, 2)
? x.add(y).mul(z)
33/21
```

In this example, we computed *gcd* immediately because we anticipated that *numer* and *denom* would be called often.

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

E.g.

```
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 would be advantageous if we expected *numer* and *denom* to be called infrequently.

The clients would in each case observe exactly the same behaviour.

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

It is one of the cornerstones of good software engineering.

Self References

Within a class definition, the name ***this*** stands for the object whose method is executing.

Example: Add *less* and *max* functions to class *Rational* as follows.

```
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 a class is just a shorthand for ***this***.x. That is, *less* could have equivalently been formulated as follows.

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


Classes and Substitution

Previously, the meaning of function application was defined using the substitution model of computation.

We now extend this model to classes and objects.

Question: How is a class instantiation ***new*** $C(e_1, ..., e_m)$ evaluated?

Answer: The argument expressions $e_1, ..., e_m$ are evaluated as normal function arguments. That's all. The resulting expression, say, ***new*** $C(v_1, ..., v_m)$, is already a value.

Now assume 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 function f with formal parameters $y_1, ..., y_n$.

(The function's parameter list may be absent. For brevity, we have omitted parameter types.)

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

Answer: The expression rewrites to:

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

There are three substitutions at work here:

- the substitution of actual arguments $w_1, ..., w_n$ for the formal parameters of function f,
- the substitution of actual class arguments $v_1, ..., v_m$ for the formal parameters of class C,
- the substitution of the object value $C(v_1, ..., v_n)$ for the self reference ***this***.

Rewriting Examples

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, *Rational* numbers are just as “natural” as integers.

But for users of these abstractions, there is an apparent difference:

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

In Scala, this difference can be made to disappear. We do that in two steps.

Step 1 Every method with one parameter can be used as an infix operator.

Hence, it is possible to write

r add s		*r.add (s)*
r less s	*as alternatives for*	*r.less (s)*
r max s		*r.max (s)*

Step 2 Operators can be used as normal identifiers.

I.e. an identifier can be one of the following:

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

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

The following table lists characters from lower to higher-precedence:

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

Hence, we can define *Rational* more naturally as follows...

```
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 *float*:

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

Abstract Classes

Consider the task of writing a class for sets of integer numbers with operations as follows.

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

IntSet is an abstract class.

Abstract classes may contain members whose implementation is missing (in our case: *incl* and *contains*).

Therefore, no objects of an abstract class can be instantiated with ***new***.

Class Extensions

We plan to implement sets as binary trees.

There are two possible forms of trees: A tree for the empty set, and a tree consisting of an integer and two subtrees.

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;
}
```

Notes:

- Both *Empty* and *NonEmpty* extend class *IntSet*.
- This means that
  - The types *Empty* and *NonEmpty* conform to type *IntSet*.

    I.e. 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 base class of *Empty* and *NonEmpty*.
- *Empty* and *NonEmpty* are subclasses of *IntSet*.
- Every user-defined class in Scala extends some other class.
- If no ***extends*** clause is given, class *scala.Object* is assumed.
- Subclasses inherit all members of their base class.
- The definitions of *contains* and *incl* in classes *Empty* and *NonEmpty* implement the abstract functions in the base class *IntSet*.
- It is also possible to override an existing non-abstract definition in a subclass, but then the overriding definition must be labelled ***override***.

Example:

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

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

Exercise: Write methods *union* and *intersection* to form the union and intersection between two sets.

Exercise: Add a method

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

to return the given set without the element *x*. To accomplish this, it is useful to also implement a test method

```
def isEmpty: boolean
```

for sets.

Dynamic Binding

- Object-oriented languages (Scala included) implement dynamic method dispatch.
- That is, the code invoked for a method call depends on the run-time type of the object which contains the method.

Example:

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

Example:

```
        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
```

Dynamic method dispatch is analogous to higher-order function calls.

Question:

Can either be implemented in terms of the other?

Standard Classes

In fact, types such as *int* or *boolean* are not treated treated specially in Scala;

They are defined as type aliases of Scala classes in module *Predef*:

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

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

But this is just an optimization, it has no effect on the meaning of a program.

Here is a hypothetical implementation of class *Boolean*.

Class Boolean

```
package scala;
trait Boolean {
  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.!);
}
val true = new Boolean { def ifThenElse[a](def t: a)(def e: a) = t }
val false = new Boolean { def ifThenElse[a](def t: a)(def e: a) = e }
```


Class Int

Here is a partial specification of class *Int*.

```
package scala;
class Int extends Long {
  def + (that: Double): Double;
  def + (that: Float): Float;
  def + (that: Long): Long;
  def + (that: Int): Int;        /* analogous for -, *, /, % */

  def << (cnt: Int): Int;        /* analogous for >>, >>> */

  def & (that: Long): Long;
  def & (that: Int): Int;        /* analogous for |, ^ */

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

Exercise: Give an implementation of the following class of 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 any of Scala's standard numeric classes in the implementation.

Implement instead two subclasses

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

one for the number zero; the other for positive numbers.

Pure Object-Orientation

A pure object-oriented language is a language where every value is an object.

If the language is class-based, this means that every value type should be a class.

Is Scala a pure object-oriented language?

We have seen that Scala's numeric types and type *boolean* can be implemented as normal classes.

We will see next week that functions can be regarded as objects as well.

The function type $A \Rightarrow B$ is treated as a shorthand for objects with an apply method:

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

Summary

- 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 members which are selected using infix '.'.
- Classes and members can be abstract, i.e. given without concrete implementation.
- A class can extend another class.
- If class A extends B then type A conforms to B.
  That is, objects of type A can be used wherever objects of type B are required.

Language Elements Introduced This Week

Types:

Type = ... | *ident*

Types can now be arbitrary identifiers which represent classes.

Expressions:

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

An expression can now be an object creation, or a selection $E.m$ of a member m from an object-valued expression E.

Definitions:

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

A definition can now be a class definition such as

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

The definitions *defs* in a class may be preceded by modifiers ***private*** or ***override***.