Object-Oriented Programming in Scala

Jacek Wasilewski

Rational Numbers Example

- Rational number is represented by two integers

$$\frac{x}{y}$$

  - x is the numerator
  - y is the denominator
- Problem: design a package for rational arithmetic

Multiplying Rational Numbers – Naïve way

- Multiplying two rational numbers
  - calculate the numerator based on two numerators
  - calculate the denominator based on two denominators
- Now: we store numerators and denominators separately, as normal variables.
- Code:

```
def multNumerator(n1: Int, d1: Int, n2: Int, d2: Int) = ...
def multDenominator(n1: Int, d1: Int, n2: Int, d2: Int) =
...
```

Class

- Data structure of things that should be stored together
- ```
  class Rational(x: Int, y: Int) {
      def numerator = x
      def denominator = y
  }
  ```
- This code defines
  - type
  - constructor
  ```

Instances

- Class – just a **data structure** with **methods/behaviours**
- We work on **instances**
- Creating same way as in Java:

```
class Rational(x: Int, y: Int) {
    def numerator = x
    def denominator = y
}

new Rational(1, 2)
```

Members

- Two members:
  - `numerator`,
  - `denominator`.
- Members access – like in Java:
  `.` (the infix operator)
  ```
  x.numerator
  x.denominator
  ```

```
class Rational(x: Int,
               y: Int) {
    def numerator = x
    def denominator = y
}

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

Methods (1)

- Rational multiplication:

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

- Using classes:

```
def multRationals(a: Rational, b: Rational): Rational = {
    new Rational(
        a.numerator * b.numerator,
        a.denominator * b.denominator)
}
```

- We also would like to print out the representation:

```
def toString(r: Rational) = r.numerator + "/" + r.denominator
```

Methods (2)

- More sense in putting these as methods:

```
class Rational(x: Int, y: Int) {
    def numerator = x
    def denominator = y
    def mult(r: Rational): Rational = {
        new Rational(
            numerator * r.numerator,
            denominator * r.denominator)
    }
    override def toString = numerator + "/" + denominator
}
```

- `toString` is a method of `java.lang.Object`

Methods (3)

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> val z = x.mult(y).mult(y)
z: Rational = 4/18
```

Access Modifiers (1)

- Our rational numbers are not always stored in the simplest way
- We can simplify them – using GCD
-
```
class Rational(x: Int, y: Int) {
  def gcd(a: Int, b: Int) = if (b == 0) a else gcd(b, a % b)
  def g = gcd(x, y)
  def numerator = x / g
  def denominator = y / g
  ...
}
```

Access Modifiers (2)

- **`gcd`** method can be called by anyone
  - not a property of rational
  - maybe should be hidden?

```
private def gcd(a: Int, b: Int) =
  if (b == 0) a else gcd(b, a % b)
private def g = gcd(x, y)
```

Access Modifiers (3)

- Modifiers
  - public
  - private
  - protected

```
class Example {
    val example1 = 1
    private val example2 = 2
    protected val example3 = 3
}
```

Self Reference

- `this` references the current object

```
class Rational(x: Int, y: Int) {
    ...
    def mult(r: Rational): Rational = {
        new Rational(
            this.numerator * r.numerator,
            this.denominator * r.denominator)
    }
    ...
}
```

Constructors (1)

- Code:
```
class Rational(x: Int, y: Int) {
    ...
}
```
- Primary constructor
  - takes the parameters of the class,
  - executes the body of the class.

Constructors (2)

- Auxiliary constructors

```
class Rational(x: Int, y: Int) {
    def this(x: Int) = this(x, 1)
    def this() = this(1, 1)
}
new Rational(2)
```

Constructors (3)

- Private constructor
  - `private` keyword after the class name and before the parameter list

```
scala> class SecretRational private(x: Int, y: Int) { ... }
defined class SecretRational

scala> new SecretRational(1, 1)
error: constructor SecretRational in class SecretRational
cannot be accessed in object $iw
```

Preconditions

- All public methods should validate parameters:
- Use `require`:

```
class Rational(x: Int, y: Int) {
    require(y > 0, "denominator must be positive")
    ...
}
```

- If condition fails, `IllegalArgumentException` is thrown.
- Message is optional.

Operators (1)

- To multiply two rational numbers
```
val x = new Rational(1)
val y = new Rational(2)
x.mult(y)
```
- If these were integers:
```
val a = 5
val b = 5
a * b
```
- Rational numbers are like integers
  - want to use `*` as well

Operators (2)

- Assume `Rational` class has methods: `add`, `div`, `max`, of one parameter
- Normal calls

```
r.add(1)
r.div(2)
r.max(3)
```

- But also possible

```
r add 1
r div 2
r max 3
```

Operators (3)

- Operators (`+, - , *, /`) can be used as identifiers
- Identifiers
  - alphanumeric
  - symbolic
- Now:

```
class Rational(x: Int, y: Int) {
    ...
    def *(r: Rational): Rational = ...
    ...
}
new Rational(1) * new Rational(2)
```

Abstract Classes

- Like in Java
- Missing implementation is possible:
```
abstract class Shape {
    def area(): Double
}
```
- Instances CANNOT be created

Traits (1)

- Like interfaces in Java
- Like abstract classes
- Trait can be partially implemented
- Main difference is in intent
  - abstract classes – object modeling through inheritance
  - traits – properties and common behaviors
- Traits do not have constructors.

Traits (2)

- Example:

```
trait Equality {
    def isEqual(x: Any): Boolean
    def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

Static Objects (1)

```
abstract class Point {
    val x: Double
    val y: Double
    def isOrigin = (x == 0.0 && y == 0.0)
}

val origin = new Point() {
    val x = 0.0
    val y = 0.0
}
val origin2 = new Point() { ... }
...
```

Static Objects (2)

- Creating it every single time feels inappropriate
- No static members in Scala
- Accompanying object
  - same name as class'

- Code:

```
abstract class Point {
    val x: Double
    val y: Double
    def isOrigin =
        (x == 0.0 && y == 0.0)
}
object Point {
    val origin = new Point() {
        val x = 0.0
        val y = 0.0
    }
}
```

Standalone Applications

- Object can be created not only as accompanying object.
- Useful for standalone apps.
- ```
  object HelloWorld {
      def main(args: Array[String]) = {
          println("Hello World!")
      }
  }
  ```
  ```

References

- “Scala By Example”, Martin Odersky, EPFL
- “Functional Programming Principles in Scala”, Martin Odersky, Coursera