L12: Functional Programming OOP

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

February 20th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Sorry about the Question Scrambling

- Apparently Canvas scramble the order of the questions when you take the test ☹

Let's Create a Rational Number

↳ 1, 0.5, -0.5 (Fraction a/b)

- Idea 1: Use a pair — Easy way to do Rational Number
  - type Rational = (Int, Int)
  - def add(p: Rational, q: Rational) = (p, q) match {
    case ((np, dp), (nq, dq)) => (np*dq + nq*dp, dp*dq)
    }
  - def toString(p: Rational) = (p, q) match {
    p.toString + "/" + q.toString
    }
- Idea 2: Use a record
  - case class Rational(n: Int, d: Int)
  - def add(p: Rational, q: Rational) = Rational(p.n*q.d + q.n*p.d, p.d*q.d)
  - def toString(p: Rational) = p.n.toString + "/" + p.d.toString

$\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd}$

↳ Print on Screen

get N

get d

Using the Class

- Let's use a class to define a rational number
- class Rational(n: Int, d: Int) {
  def numer = n
  def denom = d
  }

.numer = n
.denom = d

- Instantiation in Scala
  - new
- val r = new Rational(3, 4)
- Accessing r can be done by using *r.numer* and *r.denom*

3 4

Implementing an Add

- This can be done so it becomes a class method
  - def add(that: Rational) = new Rational(this.numer*that.denom + that.numer*this.denom, this.denom*that.denom)

Argument in Rational (Numer) (Denom)

- def mult(that: Rational) = new Rational(numer*that.numer, denom*that.denom)
- override def toString = numer + "/" + denom

```
class Rational(n: Int, d: Int) {
    def numer = n
    def denom = d
    def add(r: Rational) = (this.numer * r.denom) + (this.denom * r.numer) /
                           (this.denom * r.denom)
```

Numer

Denom

$\frac{a}{b} + \frac{c}{d}$

This r

Public and Private

- private def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
  private val g = gcd(n, d)
  def numer = n/g
  def denom = d/g
  require(d>0, "denominator must be positive")

  If Less than 0

- By default, def is public

Constructor - construct object

- We can define a constructor by adding aux. constructors
- def this(n:Int) = this(n, 1)

- Notice how we use “this” to self-reference
- Example:
- def less(that: Rational) =
  numer * that.denom < that.numer * denom
- // This could have been: this.numer * that.denom < that.numer * this.denom

Overloading Operators - Same function name, different behavior

- Unlike an Integer, adding a rational class is different
  - You cannot just call x+y [because it doesn't know how to add an rational number]
  - You ended up having to define r.add
- Alternative: overloading the "+" operator
  - Operators are treated like a function, you can define it
- def + (r: Rational) = … (Overload + behavior) $R_1 + R_2$ (this, Input) $R_1.add(R_2)$, $R_1 + R_2 \Rightarrow R_3$
- def – (r: Rational) = …
- def unary_- = … (only 1 input) (Treat - as an operator)
- Keep in mind that operators have precedence rule
  - Overloaded operators keep the same rule

a + b * c / d + e

① ② ③ ④

Abstract Class

- What if we want to make an abstract class?
  - Q: What is an abstract class?
- Let's say we want to create the following things:
- An IntSet, where it collects a set of Integers
  - add(x: Int): IntSet – produce a new set taking the union of this set and {x}.
  - has(x: Int): Boolean — ask if x is a member of this set
- How can we specify the interface?

Abstract Class: Interface

- Create an abstract class
- abstract class IntSet {
  def add(x: Int): IntSet
  def has(x: Int): Boolean
  }

- Then, we can implement this class later

Abstract Class: Implementation

- Example: Implement this IntSet using a linked list
- class Empty extends IntSet {
  def has(x: Int): Boolean = false
  def add(x: Int): IntSet = ... // new NonEmpty(x, new Empty)
  }
- class NonEmpty(elt: Int, other: IntSet) extends IntSet {
  def has(x: Int): Boolean = (x==elt) || (other has x)
  def add(x: Int): IntSet = new NonEmpty(x, this)
  }

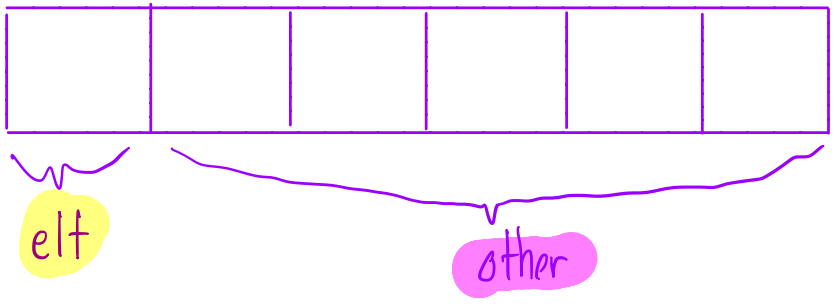
element adding

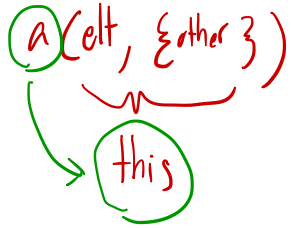
current set

(else)

or other.has(x)

Include both x and other





Abstract Class: Implementation

- Empty and NonEmpty extend the class IntSet
- Both conforms to IntSet
- IntSet is the superclass to Empty and NonEmpty
  - Vice versa, Empty and NonEmpty are the subclasses
- Everything has an Object as a superclass
  - This includes your REPL statements
  - This means you can override the implementation
    - val on top on existing variables

Limit Copies to One - Replace class with object (static)

- From our example, the Empty set should really have one copy, right?
  - This is pretty easy to fix using static class in other languages
- In Scala, this is also an easy fix using a singleton ***object***
- object Empty extends IntSet {
  ```
  def has(x: Int) = false
  def add(x: Int) = new NonEmpty(x, Empty)
  ```
  }
- This define an object called Empty, no other instances of this object can be created
- Empty evaluate to itself (it is a value)

In-Class Exercise 12

- Recreate an object for the Expression type with we been using, with the following traits
  - trait Expr {
    def +(that: Expr) = [Fill in this blank]
    def *(that: Expr) = [Fill in this blank]
    def unary_- = [Fill in this blank]
    def toVal(implicit ctx: Map[String, Double]): Double
    }
- It should have the following methods
  - case class Var(name: String) extends Expr
  - case class Constant(n: Double) extends Expr
  - case class Negate(e: Expr) extends Expr
  - case class Sum(e1: Expr, e2: Expr) extends Expr
  - case class Prod(e1: Expr, e2: Expr) extends Expr
  - Each of these should implement its version of ***toVal***