Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 12, 2019

Announcements

- Homework 0 is "due" today.
  (You should now know how to use the SVN repo.)

- Homework 1 assignment has been moved to Tuesday, and the due date will shift likewise.

Class Methods

- Methods are functions defined in the body of a class definition. They have direct access to the members of a class instance

- Syntactically, they are placed between braces, after the class parameters

Class Methods

```
case class C(field1: Type1, ..., fieldN: TypeN) {
def m1(x11: TypeP11, ..., xK1: TypePk1): TypeR11 =
    expr
...
def mJ(x1J: TypeP1J, ..., xKJ: TypePkJ): TypeR1J =
    expr
}
```

Method Definitions

```
case class Coordinate(x: Int, y: Int) {
  def magnitude() = x*x + y*y
}
```

Applying a Class Method

- Given a class definition:

```
class C(p₁: T₁, ..., pₖ: Tₖ) { ...
    def m(param₁: T₁₁, paramₙ: T₁ₙ): T = e
    ...
}
```

- To reduce the application of a method:

$$\texttt{C}(v_1, \ldots, v_k).\texttt{m}(arg_1, \ldots, arg_N)$$

  - Reduce the receiver and arguments, left to right
  - Reduce the body of `m`, replacing constructor parameters with constructor arguments and method parameters with method arguments

Applying a Class Method

```
Coordinate(5,3).magnitude() ↦

5*5 + 3*3 ↦

25 + 9 ↦

34
```

Compound Value Patterns

```
def dotProduct(c1: Coordinate, c2: Coordinate) = {
  (c1, c2) match {
    case (Coordinate(x1,y1), Coordinate(x2,y2)) =>
      x1*x2 + y1*y2
  }
}
```

Patterns in Assignments

Patterns in Scala may also be used for destructuring assignments:

```
def dotProduct(c1: Coordinate, c2: Coordinate) = {
  val Coordinate(x1, y1) = c1
  val Coordinate(x2, y2) = c2
  x1*x2 + y1*y2
}
```

Symbols in Patterns: Binding or Constant?

- A symbol with a *lower-case* first character is a binding symbol
- A character with an *upper-case* first character is a value
- You can make a variable a constant using `backticks`

```
val pi = 3.14
val One = 1.0
expr match {
  case `pi` => "Pi"
  case One => "One"
}
```

Singleton Objects

Singleton Objects

- Also, we often would like to organize identifiers and functions together into a single entity

- When *compiling* a Scala file, it is *required* that all constant and function definitions are placed inside a class or object

- For this purpose, we can make use of *singleton objects*

Singleton Objects

```
object IncomeTax {

  val cutoff0 = 0
  val bracket0 = 0

  val bracket1 = 100
  val cutoff1 = 9075
  ...

  def incomeTaxForBracket(income: Int, cutoff: Int, bracket: Int) = {
    require(income >= 0)
    (income - cutoff) * bracket / divisor + incomeTax(cutoff)
  } ensuring (_ >= 0)
}
```

Syntax for Singleton Objects

```
object Name {

  valDefs*
  functionDefs*
}
```

We Can Refer to the Constants and Functions in the Object Using Dot Notation

```
IncomeTax.bracket1
↦
100
```

We Can Refer to the Constants and Functions in the Object Using Dot Notation

```
IncomeTax.incomeTax(100000)
```

↦

21174

Case Objects

- Declaring a *case object* denotes your intent:
  You will use this object as a *value* and use it as a pattern in *match* expressions.

- Using a normal *object* denotes a container for “static” methods declarations, or a value that won’t be *matched*.

```
case object Name {

  valDefs*

  functionDefs*
}
```

Homework

Homework Grading Criteria

- Style: 50%

- Correctness: 50%

Style of Program Code and Test Code

- Clarity
- Comments
- Contracts
- Design Principles

Clarity: Is the Program Easy to Read?

- Is the program concise?

  "Make every word say."
  (Strunk and White, *The Elements of Style*)

- Are functions kept relatively small, with sub-parts broken up according to the problem domain?

  Think of the *profit*, *revenue*, and *cost* example from Lecture 2

Clarity: Is the Program Easy to Read?

- Are the names of functions and variables syntactically consistent?

- For instance, do they all use `camelCase`?

- Are similar functions given names of similar length?

Clarity: Is the Program Easy to Read?

- Are names adequately descriptive and appropriate?
  - For example, using single letter names for public functions is not appropriate
  - Are consistent metaphors used for functions that work together?

Clarity: Is the Program Easy to Read?

- Is the program consistent in its indentation and whitespace?
    - This can affect readability
- Is there appropriate spacing?
    - Code that is too close together can be hard to read

Comments

- Does each function include a statement of purpose?
- Are the comments excessive?
- Comments embedded in program should be used only for cases where it is not clear locally why the program is doing what it does
- The reader should be expected to know the language the text is written in

Contracts

- Do the parameter types and return types of all functions and variables make sense?
- Are `require` and `ensuring` clauses included when necessary?
- Are the included `require` and `ensuring` clauses defined appropriately?
- Are requirements that cannot be expressed in `require` and `ensuring` clauses defined as documentation?

Design Principles

- Does the program stick to the constructs covered in class so far?

- Is the program purely functional?

Design Principles

- Does the program follow templates provided in class when appropriate?
    - For instance, is the function body a simple algebraic expression?
- Is it a series of `if-else` expressions breaking up sub-ranges?
- Is it a `match` expression breaking up an abstract datatype?

Design Principles

- Does the program include abstractions to factor out common code? (DRY)
  - Copy-and-paste coding should be strongly avoided
- Does the program avoid unnecessary complexity? (KISS)

Correctness

- Does the program compile?
- Do all student submitted tests pass?
- Does the program include all entry points required by the assignment?
- Are all tests automated? Tests should indicate on their own that either they pass or fail

Correctness

- Example Tests: Are simple examples included in the tests showing how the function behaves under usually circumstances?
- Stress Tests: Are there additional tests ensuring that the function behaves appropriately when given extreme data values

`0, 1, -1, PositiveInfinity, NegativeInfinity, NaN, etc.`

Correctness

- Persuasive Tests: Is there adequate coverage to convince the reader that the program behaves as expected?

- Does the program perform correctly when subjected to additional testing provided by the course staff?

Expected Test Structure

- All tests in a program should be captured in a *test suite*
- For each component of a program, there should be a corresponding test class
- For each function, there should be a corresponding test function
- For each test function, there should be multiple tests, checking both common and extreme cases

Example: Testing Our Theater Profit Calculator

```
import org.scalatest._

class TheaterProfitTest extends FunSuite {
  test("attendance") {
     ...
  }
  test("cost") {
     ...
  }
  test("profit") {
     ...
  }
  test("revenue") = {
     ...
  }
  test("max") {
      ...
  }
}
```

Example: Testing Our Theater Profit Calculator

```
import org.scalatest._

class TheaterProfitTest extends FunSuite {
  test("attendance") {
    assert(120 == attendance(500))
    assert(135 == attendance(490))
    assert(165 == attendance(470))
    assert(0 == attendance(1000))
    assert(0 == attendance(580))
    assert(2 == attendance(579))
    assert(870 == attendance(0))
  }
  ...
}
```

Example: Testing Our Theater Profit Calculator

```
import org.scalatest._

class TheaterProfitTest extends FunSuite {
  test("attendance") {
    assert(120 === attendance(500))
    assert(135 === attendance(490))
    assert(165 === attendance(470))
    assert(0 === attendance(1000))
    assert(0 === attendance(580))
    assert(2 === attendance(579))
    assert(870 === attendance(0))
  }
  ...
}
```

ScalaTest equality operator provides better error messages, but only valid in tests

Example: Testing Our Theater Profit Calculator

```
import org.scalatest._

class TheaterProfitTest extends FunSuite {
  test("attendance") {
    assert(120 === attendance(500))
    assert(135 === attendance(490))
    assert(165 === attendance(470))
    assert(0 === attendance(1000))
    assert(0 === attendance(580))
    assert(2 === attendance(579))
    assert(870 === attendance(0))
    intercept[IllegalArgumentException] {
      attendance(-1)
    }
  }
  ...
}
```

Use ScalaTest intercept[T] construct to assert for errors in your tests

Example: Testing Our Theater Profit Calculator

```
import org.scalatest._

class TheaterProfitTest extends FunSuite {
  ...
  test("revenue") {
    assert(0 === revenue(0))
    assert(0 === revenue(1000))
    assert(53550 === revenue(510))
  }
  ...
}
```