Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments
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:

```scala
class C(p1:T1, …, pk:Tk) { …
    def m(param1:T11, paramN:T1N):T = e
    …
}
```

• To reduce the application of a method:

```scala
C(v1, …, vk).m(arg1, …, argN)
```

• 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

\[ \text{Coordinate}(5,3).\text{magnitude}() \mapsto 5*5 + 3*3 \mapsto 25 + 9 \mapsto 34 \]
Nested Pattern Matching

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

```plaintext
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

\[
\text{IncomeTax}.\text{incomeTax}(100000) \\
\rightarrow \\
21174
\]
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, cost example in 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 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

\[ 0, 1, -1, \text{PositiveInfinity}, \text{NegativeInfinity}, \text{NaN}, \text{etc.} \]
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

class TheaterProfitTest(name: String) extends TestCase(name) {

    def testAttendance() = {
        ...
    }
    def testCost() = {
        ...
    }
    def testProfit() = {
        ...
    }
    def testRevenue() = {
        ...
    }
    def testMax() = {
        ...
    }
}
Example: Testing Our Theater Profit Calculator

```scala
class TheaterProfitTest(name: String) extends TestCase(name) {

def testAttendance() = {
    assertEquals(120, attendance(500))
    assertEquals(135, attendance(490))
    assertEquals(165, attendance(470))
    assertEquals(0, attendance(1000))
    assertEquals(0, attendance(580))
    assertEquals(2, attendance(579))
    assertEquals(870, attendance(0))

    ...
}
```
class TheaterProfitTest(name: String) extends TestCase(name) {
    ...
    def testRevenue() = {
        assertEquals(0, revenue(0))
        assertEquals(0, revenue(1000))
        assertEquals(53550, revenue(510))
    }
    ...
}
Using DrScala
DrScala

• Available from the course homepage
  http://comp311.rice.edu

• A lightweight development environment well-suited to the exercises we will do in this class
Welcome to DrScala. Working directory is /Users/ericeallen/tmp
Define your program in the definitions pane

```scala
object IncomeTax {
  val cutoff0 = 0
  val bracket0 = 0

  val bracket1 = 100
  val cutoff1 = 9075

  val bracket2 = 150
  val cutoff2 = 36900

  val bracket3 = 250
  val cutoff3 = 89350
}

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
```
A prompt to save your program after hitting the Compile button.
Successful compilation reported in the Compiler Output tab.

```scala
object IncomeTax {
    
    val cutoff0 = 0
    val bracket0 = 0

    val bracket1 = 100
    val cutoff1 = 9075

    val bracket2 = 150
    val cutoff2 = 36900

    val bracket3 = 250
    val cutoff3 = 89350

    Compilation completed. Output directory is: /Users/ericeallen/tmp
}
```
object IncomeTax {
    val cutoff0 = 0
    val bracket0 = 0

    val bracket1 = 100
    val cutoff1 = 9075

    val bracket2 = 150
    val cutoff2 = 36900

    val bracket3 = 250
    val cutoff3 = 89350
}
We can interact with the functions in our program directly in the interactions pane.
Create a new JUnit Test Case class.
We are prompted for a name. Let’s call it IncomeTaxTest.
A new test class is created

```scala
import junit.framework.TestCase
import junit.framework.Assert

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when running the test with JUnit.
 */
class IncomeTaxTest(name: String) extends TestCase(name) {

/**
 * A test method.
 * (Replace "X" with a name describing the test. You may write as

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
TESTING Nothing

>
import junit.framework.TestCase
import junit.framework.Assert._

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when running the test with JUnit.
 */

class IncomeTaxTest(name: String) extends TestCase(name) {

/**
 * A test method.
 * (Replace "X" with a name describing the test. You may write as

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
TESTING Nothing
import junit.framework.TestCase
import junit.framework.Assert_

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when running
 * the test with JUnit.
 */

class IncomeTaxTest(name: String) extends TestCase(name) {

    /**
     * A test method.
     * (Replace "X" with a name describing the test. You may write as
     */
Ignore the import statements for now

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when run
 * the test with JUnit.
 */

class IncomeTaxTest(name: String) extends TestCase(name) {

 /**
  * A test method.
  * (Replace "X" with a name describing the test. You may write as

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
TESTING Nothing
>
The provided tests don’t do very much.
All functions with names starting with "test" are treated as tests.
The `assertTrue` function is available to us in our tests.
The optional String is printed if the test fails.
The test fails if this argument does not reduce to true.

```scala
// many "testSomething" methods in this class as you wish, and each one will be called when running JUnit over this class.

def testX() {
}

/** Sample test method which tests no program code. */
def testNothing() {
  assertTrue("Dummy Test", true)
  println("TESTING Nothing")
}
```
assertEquals fails if its two arguments are not equal

Add many more test functions

```scala
/** *
 * Testing simple income tax computations.
 */
def testIncomeTax() {
  assuresEquals(100, IncomeTax.incomeTax(1000))
  assuresEquals(907, IncomeTax.incomeTax(9075))
  assuresEquals(907 + 138, IncomeTax.incomeTax(10000))
}

/** *
 * Sample test method which tests no program code. */
def testNothing() {
  assuresTrue("Dummy Test", true)
  println("TESTING Nothing")
}

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
TESTING Nothing
> |
```
Hitting the Test button prompts us to compile
Agreeing to compile prompts us to save
A green bar indicates that all tests passed.
A red bar indicates a test failure.
The failing test is highlighted in yellow.
To interact with our program, we use the Interactions Pane.
We can enter arbitrary Scala expressions
The value our expression reduces to is displayed
* Given an income in U.S. Dollars,
* returns the dollar value of tax
* owed for a single tax payer, using
* 2014-2015 IRS tax brackets.
*
```scala
def incomeTax(income: Int): Int = {
    require(income >= 0)
    if (income <= cutoff0) {
        bracket0
    }
```

As is its type
* Given an income in U.S. Dollars,
* returns the dollar value of tax
* owed for a single tax payer, using
* 2014-2015 IRS tax brackets.
*/

def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= cutoff0) {
    bracket0
  }

  // Other tax brackets and calculations
}

Welcome to DrScala. Working directory is /Users/ericeallen/tmp

> 2 + 2
res0: Int = 4
>
And the value is bound to a fresh identifier
The classes we have compiled in Definitions are in scope in Interactions.
We can refer to previously bound identifiers in subsequent expressions.
We can also bind new identifiers directly.

```scala
* Given an income in U.S. Dollars,
* returns the dollar value of tax
* owed for a single tax payer, using
* 2014-2015 IRS tax brackets.
*/

def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= cutoff0) {
    bracket0
  }
```
And compute with them
We can also define new functions.

* Given an income in U.S. Dollars,
* returns the dollar value of tax
* owed for a single tax payer, using
* 2014-2015 IRS tax brackets.
*
*def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= cutoff0) {
    bracket0

  }

*def square(x: Double) = x * x
square: (x: Double)Double

For definitions that are not syntactically complete, we are given a new line, indicated by a vertical bar.
The function is bound and an arrow type is displayed.
And we can refer to this function in subsequent expressions.
We can click on the file to appear in Definitions

```scala
// Brackets are in tenths of percentage points,
// thus percentage divisor must be 1000
val divisor = 1000

/**
 * Given an income in U.S. Dollars,
 * returns the dollar value of tax
 * owed for a single tax payer, using
 * 2014-2015 IRS tax brackets.
 */

def incomeTax(income: Int): Int = {
  require(income >= 0)

  if (income <= 0) 0
  else { /* code block */ }
}
```

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
Files that have not been saved include an asterisk.
Reset resets the Interactions session

```scala
/* Testing simple income tax computations. */
def testIncomeTax() {
    assertEquals(100, IncomeTax.incomeTax(1000))
    assertEquals(907, IncomeTax.incomeTax(9075))
    assertEquals(907 + 138, IncomeTax.incomeTax(10000))
}

/** Sample test method which tests no program code. */
def testNothing() {
    assertTrue("Dummy Test", true)
    println("TESTING Nothing")
}
```

Welcome to DrScala. Working directory is /Users/ericeallen/tmp
Run executes Definitions

```
* Testing simple income tax computations.
  */

def testIncomeTax() {
  assertEquals(100, IncomeTax.incomeTax(1000))
  assertEquals(907, IncomeTax.incomeTax(9075))
  assertEquals(907 + 138, IncomeTax.incomeTax(10000))
}

/** Sample test method which tests no program code.
  def testNothing() {
    assertTrue("Dummy Test", true)
    println("TESTING Nothing")
  }
```

Welcome to DrScala. Working directory is /Users/ericeallen/tmp

>