Introduction to Object Oriented / Functional Programming 1

Object-Oriented Software Development
COMP4001
CSE UNSW Sydney

Lecturer: John Potter
Outline

- Functions as Objects in Java
- Overview of Scala based on Scala Tutorial
- Examples + Use of scala REPL interpreter
**OO Programming vs Functional Programming**

- **strengths of OO**
  - code extensibility via inheritance
  - dynamic binding
    - allows old client code to use new implementations of existing interfaces / classes
    - e.g. template method design pattern typical of application frameworks

- **weaknesses of OO**
  - imperative programming with object aliasing
    - shared mutable state
    - hard to achieve safety + high performance in concurrent programs
  - who's responsible for maintaining objects in valid state?
  - how or when are changes to objects notified to other objects that depend on them?
Functional Programming

- strengths
  - functions as data
  - allows higher order functions
    - functions being passed as arguments to and and returned from other functions
    - old higher order functions can use newly defined functions
  - a pure functional language requires that all data is immutable
    - safe to share existing values
    - safe to allow concurrent execution

- weaknesses
  - lack of mutability
    - changes to state must be modelled with new data
    - some kinds of problems may yield inefficient computations
Function Objects in Java

- we can easily model functions as objects

```java
interface Function<X,Y> {
    Y apply(X x);
}
```

- then bind this to a particular function via a class implementation

- e.g. using an anonymous class

```java
Function<Integer, Integer> addOne =
    new Function<Integer, Integer>() {
        public Integer apply(Integer x) {
            return x.add(1);
        }
    }
```
Use of Function Objects in Java

- function objects have not been too popular in Java
  - syntax is clunky
    - too much (dumb) information to declare
- to be useful, we need:
  - good support for defining new function objects from existing code
    - expressions and statement blocks form closures
      - unevaluated expressions that can be passed in calls
  - a library of higher order function types
    - to allow functional programming style of solutions
    - often much more concise than direct use of data objects
Function Objects in C++

- modern C++ is largely about functional programming!
- the metaprogramming style of generic programming
  - uses C++ class templates
  - relies on many concepts from functional programming
    - Standard Template Library and the Boost Library
    - provide models of functions and function composition for dealing with container objects
- currently C++ does not type check generic templates
  - type check occurs at use sites, only after generic instantiation
Function Objects in Scala

- Scala is a Java-like language
  - it is an OO language
  - but it directly supports programming with functions
- the trick behind Scala:
  - methods can be referenced as (function) objects
  - so code expressions and statement blocks can define objects
- other OO languages provide support for closures
  - e.g. Python and Ruby
  - in comparison
    - Scala is statically typed, more in the spirit of Java, with better syntax for writing in a functional style
    - OCaml is a functional language (in ML family) with support for OO
      - Clojure is another (in Scheme family)
- Scala borrows many ideas from Haskell, a functional programming language
Scala Background

- open source development
  - based in Programming Methods Lab at Lausanne Polytechnic, Switzerland
- main designer is Martin Odersky
  - he was a key contributor to the effort to introduce generics into Java
  - his work was based on his earlier research in a language design called Pizza
    - a precursor to Generic Java
  - he wrote the core part of Sun’s javac which introduced generics into Java
Scala Documentation

- see Scala Home at
  - http://www.scala-lang.org/
  - tutorials, examples
  - downloads including Eclipse plugin
Scala Tools

- similar to Java
  - Scala relies on JVM
  - Java library calls can be embedded in Scala programs
    - gives Scala direct access to huge Java library resources
- scalac compiles Scala source
  - scalac is similar to javac
- scala runs scala classfiles
  - like java command
- scala also provides an interactive (REPL) interpreter
  - unlike java
- these tools available as an Eclipse plugin
Scala in Summary

- everything is an object
  - even functions
  - higher order (FP-style) programming is possible
- object definitions
  - `class`
    - objects instances via `new`
    - principal constructor declared in class header
  - `object` (singleton pattern)
    - Review the Singleton design pattern in Java
  - `case class`
    - no need for `new`
    - pattern matching constructs used in case statements
Numbers are Objects

1 + 2 * 3 / x

- is equivalent to
1 + (2 * (3)) / (x)

clean syntax

- method call can be written without the dot ‘.’
  - c.f. Smalltalk, functional languages like Haskell
- any symbols allowed as method name
- all left associative, with precedence by first symbol
  - except symbolic names ending in ‘:’ are right associative
Functions are Objects

```java
object Ticker extends App {

  oncePerSecond(timeFlies)

  def oncePerSecond (callback: () => Unit) {
    while (true) {
      callback ()
      Thread.sleep 1000
    }
  }

  def timeFlies () {
    Console.println("Time flies like an arrow...")
  }
}
```

Compiler uses constructor code of Ticker as the main method inherited via App class

refers to the function object defined by the timeFlies method def below

callback : () => Unit is function argument to the oncePerSecond method

actual argument is not evaluated until it is called

function definition results in an an object
timeflies : () => unit
Anonymous Functions

```scala
package lect01

object TickerAnonymous extends App {

  oncePerSecond {
    println("Time flies like an arrow ... (anon)")
  }

  def oncePerSecond (callback: => Unit) {
    while (true) {
      callback
      Thread.sleep(1000)
    }
  }
}
```

A closure, or anonymous function definition constructs a new function object locally c.f. lambda abstraction in functional programming.

Call by name parameter passing. Argument is passed as an expression without evaluation.

Actual argument is evaluated whenever it is used.
Declarations

- use `:` syntax
  - `var x : Type` or `val x : Type`
  - var declares variables
  - val declares constants or "patterns"

- method declarations
  - `def eval (exp: Expr): Value`
  - `=>` is a synonym for the class of function objects
  - `ArgType => ResultType`
  - instances are just function objects
  - e.g. we can produce a function from a method:
    - `(exp : Expr) => eval (exp) : Expr => Value`
    - `eval (_) : Expr => Value`

- functions can be curried
  - a function of type 2 arguments of type X, Y and result type Z can be written as: `(X, Y) => Z`
  - its curried version is: `X => ( Y => Z )`
    - given an X argument yields a function from Y to Z
    - the parentheses are optional
class Complex(
    real: Double,
    imaginary: Double) {
    def re() = real
    def im() = imaginary
}

or

class Complex(
    real: Double,
    imaginary: Double) {
    def re = real
    def im = imaginary
}

- classes may have object parameters
- classes can also be generic, that is, have type parameters
- this defines the form of a constructor
- the methods define accessors for the parameters
- no-arg methods do not require ()
- a Complex object has no mutators
Inheritance

- all classes derive from a single superclass, as for Java classes
  - if not explicit, then it is scala.AnyRef
  - method overrides must be declared explicitly as overrides
class Point(xc: Int, yc: Int) {
  var x: Int = xc
  var y: Int = yc

  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }

  override def toString: String
    = "(" + x + ", " + y + ")";
}
Case Classes and Pattern Matching

- Case classes allow pattern matching on their structural form.
- New instances do not need `new`.
  - E.g. `Const(5)` denotes a new `Const` instance in an expression.
  - It also denotes a pattern for selecting such instances.
    - Patterns may introduce new variables which will be bound by a `match` expression.
- Case classes automatically provide:
  - Default `toString` and `hash` methods.
    - Based on the constructor of the case class.
  - Getter methods for the case class parameters.
    - Same name as the constructor args.
Case Class Example

- simple integer expressions
  - sums, variables and constants
- an expression is modelled as a tree
  - with an eval function
public abstract class Tree {
    abstract public int eval(Environment env);
}

class Sum {
    private final Tree l;
    private final Tree r;
    public Sum(Tree l, Tree r) {
        this.l = l;
        this.r = r;
    }
    public int eval(Environment env) {
        return l.eval(env) + r.eval(env);
    }
}

class Var {
    private final String name;
    public Var(String n) {
        name = n;
    }
    public int eval(Environment env) {
        return env.get(name);
    }
}

class Const {
    private final int value;
    public Const(int v) {
        value = v;
    }
    public int eval(Environment env) {
        return value;
    }
}

class Environment {
    public int get(String name);
}
Case Class Example in Scala

```scala
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree

type Environment = (String => Int)

def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n) => env(n)
  case Const(v) => v
}
```

- Scala code is much more concise
- but not equivalent to Java code
  - eval function has been defined externally case classes
    - c.f. a visitor method
    - pattern matching is used to select sub-expressions
  - hashCode and toString methods are generated automatically for Sum, Var and Const case classes
Member Types

- notice the type definition inside the `Tree` class
  - `type Environment = (String => Int)`
  - this type definition introduces a type alias
    - one type that stands for another
- *member type declaration* is a powerful feature of Scala
  - type declarations may be abstract
    - binding may be deferred to a subclass
    - or be simply constrained
    - `type Environment`
  - in OO type theory such types are often called *virtual types*
  - these provide even more expressiveness than generic classes
    - see more detail in later lecture
    - also read Abstract Types in Scala Tour
Comments on Use of Function Type

- refer to use of the Environment type in Java and Scala for the above example
  - its purpose is to allow us to look up bindings from variable names to values
- in Java we have to implement Environment as some kind of Map
  - perhaps as a decorator for HashMap
  - this requires extra wrapping of existing object types to provide the implementation of the Environment.get method
- in Scala, the Environment type is just String => Int
  - and any method of this type implements this type
    - with no further code required
Traits

trait Ord {
    def < (that: Any): Boolean

    def <=(that: Any): Boolean
        = (this < that) ||
        (this == that)

    def > (that: Any): Boolean
        = !(this <= that)

    def >=(that: Any): Boolean
        = !(this < that)
}

- traits are also called mixins
- similar to interfaces
  - but may include method definitions
  - role is to allow code reuse
  - traits do not define data fields
case class Date(year: Int, month: Int, date: Int) extends Ord {
  def <(that: Any): Boolean = {
    require (that.isInstanceOf[Date])
    val date = that.asInstanceOf[Date]
    year < date.year ||
    (year == date.year &&
     (month < date.month ||
      (month == date.month && day < date.day))
  }
}

Discussion of Date Example

- the `Ord` trait left the `< method abstract
  - with implementations for `<=, >` and `>=`
- constructor declaration of `Date` part of class header in class definition
- as a case class, `Date` automatically overrides `==`
- all fields must be equal for two Dates to be equal
- class extends a trait
  - the abstract method `<` is implemented
    - no need for override keyword with abstract methods
Class vs Object in Scala

- an object definition is a special case of a class definition
  - it represents a singleton object
Object Example

abstract class Bool {
    def && (x: => Bool): Bool
    def || (x: => Bool): Bool
}

object False extends Bool {
    def && (x: => Bool): Bool = this
    def || (x: => Bool): Bool = x
}

object True extends Bool {
    def && (x: => Bool): Bool = x
    def || (x: => Bool): Bool = this
}

- the => argument says pass the argument without evaluation
- this parameter passing mechanism is called pass-by-name
  - pass-by-name allows lazy evaluation
Closures in Scala

- any expression or block `{ ... }` of code has a value
  - possibly of type Unit, if no value is returned
    - like void in Java
- we can bind pass-by-name parameters to expressions or blocks
  - this forms a *closure*
  - an expression that can be passed around
    - and is only evaluated when it is used
Types in Scala

- everything is an object
  - defined via classes
- but there are two kinds of classes
  - value classes
    - predefined classes
      - like primitive types in Java, but values are objects
      - standard conversions apply between values
  - reference classes
    - user definable
    - autoboxing / reference between value and reference as appropriate
Homework

- Start reading Scala Tutorial and Scala By Example
  - via [http://www.scala-lang.org](http://www.scala-lang.org) web-site
- Start reading online textbooks
  - especially: Programming in Scala 1st ed by Odersky, Spoon, Venners
    - you may also buy the 2nd edition
      - updated for new libraries with Scala 2.8
- at home
  - install Scala compiler from Scala download site
  - you may use eclipse with the Scala plugin
- try out scala interpreter
- try out scalac + scala for compiling
  - examples from the tutorial and Wampler