Lecture 09: Introduction to Scala

Computer Science Department, University of Crete

Multicore Processor Programming

Based on slides by D. Malayeri, S.D. Vick, P. Haller, and M. Madsen

Introduction

- Part 1: Introduction to Scala
- Part 2: Concurrency in Scala

What is Scala?

- Scala is a statically typed language
 - Combines Object-Oriented Programming and Functional Programming
 - Developed in EPFL, lead by Martin Odersky
 - Influenced by Java, ML, Haskell, Erlang, and other languages
- Many high-level language abstractions
 - Uniform object model
 - Higher-order functions, pattern matching
 - Novel ways to compose and abstract expressions
- Managed language runtime
 - Runs on the Java Virtual Machine
 - Runs on the .NET Virtual Machine

Goals of Scala

- Create a language with better support for component software
- Hypotheses:
 - Programming language for component software should be scalable
 - ★ The same concepts describe small and large parts
 - Rather than adding lots of primitives, focus on abstraction, composition, decomposition
 - Language that unifies OOP and functional programming can provide scalable support for components

Why use Scala?

- Runs on the JVM
 - Can use any Java code in Scala
 - Almost as fast as Java
- Much shorter code
 - Odersky reports 50% reduction in most code
 - Local type inference
- Fewer errors
 - No NullPointer errors
- More flexibility
 - As many public classes per source file as you want
 - Operator overloading
- All of the above, for .NET too

Why learn Scala?

Creating a trend in web service programming

- LinkedIn
- Twitter
- Ebay
- Foursquare
- List is growing

Features of Scala (1)

- Both functional and object-oriented
 - Every value is an object
 - Every function is a value (including methods)
- Scala is statically typed
 - Includes local type inference system

Java 1.5 Pair p = new Pair<Integer, String>(1, "Scala"); Scala val p = new Pair(1, "Scala");

Features of Scala (2)

- Supports lightweight syntax for anonymous functions, higher-order functions, nested functions, currying
- ML-style pattern matching
- Integration with XML
 - Can write XML directly in Scala program
 - Can convert XML DTD into Scala class definitions
- Support for regular expression patterns
- Allows defining new control structures without using macros, and while maintaining static typing
- Any function can be used as an infix or postfix operator
- Can define methods named +, <= or ::

Features of Scala (3)

- Actor-based programming, distributed, concurrent
- Embedded DSLs, usable as scripting language
- Higher-kinded types, first class functions, closures
- Delimited continuations
- Abstract Types, Generics
- Warning: Scala is the gateway drug to ML, Haskell, ...

An Example Class ...

```
Java

public class Person {

    public final String name;

    public final int age;

    Person(String name, int age) {

        this.name = name;

        this.age = age;

    }

}
```

Scala

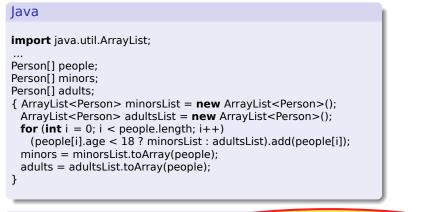
class Person(val name: String, val age: Int) {}

Java import java.util.ArrayList; ... Person[] people; Person[] minors; Person[] adults;

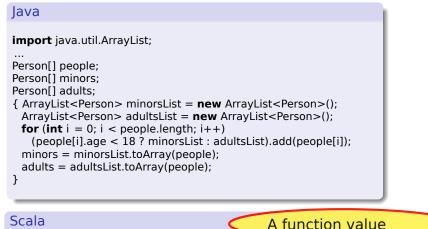
```
{ ArrayList<Person> minorsList = new ArrayList<Person>();
ArrayList<Person> adultsList = new ArrayList<Person>();
for (int i = 0; i < people.length; i++)
(people[i].age < 18 ? minorsList : adultsList).add(people[i]);
minors = minorsList.toArray(people);
adults = adultsList.toArray(people);
}
```

Scala

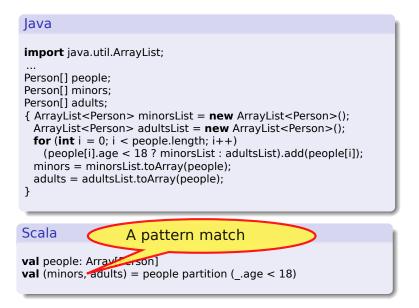
```
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)</pre>
```







val people: Array[Person] val (minors, adults) = people partition (_.age < 18)



Class Hierarchies and Abstract Data Types

- Scala unifies class hierarchies and abstract data types (ADTs)
- Introduces pattern matching for objects
- Uses concise manipulation of immutable data structures

Example: Pattern matching

Class hierarchy for binary trees

abstract class Tree[T] case object Empty extends Tree[Nothing] case class Binary[T](elem: T, left: Tree[T], right: Tree[T]) extends Tree[T]

In-order traversal

```
def inOrder[T](t: Tree[T]): List[T] = t match {
    case Empty =>
        List()
    case Binary(e, I, r) =>
        inOrder(I) ::: List(e) ::: inOrder(r)
}
```

- Extensibility
- Encapsulation: only constructor params exposed
- Representation independence

Functions and Collections

- First-class functions make collections more powerful
- Especially immutable ones

```
Container operations

people.filter(_.age >= 18)

.groupBy(_.surname)

.values

.count(_.length >= 2)
```

Functions and Collections

- First-class functions make collections more powerful
- Especially immutable ones



Functions and Collections

- First-class functions make collections more powerful
- Especially immutable ones



The Scala Object System

- Class-based
- Single Inheritance
- Can define singleton objects easily
- Subtyping is nominal: it is a subtype if declared to be a subtype
- Traits, compound types, views
 - Flexible abstractions

Classes and Objects

```
Classes and Objects
trait Nat;
object Zero extends Nat {
 def isZero: boolean = true;
 def pred: Nat =
 throw new Error("Zero.pred");
}
class Succ(n: Nat) extends Nat {
 def isZero: boolean = false:
 def pred: Nat = n;
}
```

Traits

- Similar to interfaces in Java
- They may have implementations of methods
- But cannot contain state
- Can have multiple inheritance

Example: Traits

```
trait Similarity {
    def isSimilar(x: Any): Boolean;
    def isNotSimilar(x: Any): Boolean = !isSimilar(x);
}
class Point(xc: Int, yc: Int) with Similarity {
    var x: Int = xc;
    var y: Int = yc;
    def isSimilar(obj: Any) =
        obj.isInstanceOf[Point] &&
        obj.asInstanceOf[Point].x == x;
}
```

Mixin Class Composition (1)

- Mixin: "A class which contains a combination of methods from other classes."
- Basic inheritance model is single inheritance
- But mixin classes allow more flexibility

```
class Point2D(xc: Int, yc: Int) {
 val x = xc;
 val y = yc;
 // methods for manipulating Point2Ds
}
class ColoredPoint2D(u: Int, v: Int, c: String) extends Point2D(u, v) {
 var color = c:
 def setColor(newCol: String): Unit = color = newCol;
class Point3D(xc: Int, yc: Int, zc: Int) extends Point2D(xc, yc) {
 val z = zc:
 // code for manipulating Point3Ds
class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col: String)
     extends Point3D(xc, yc, zc) with ColoredPoint2D(xc, yc, col);
```

Mixin Class Composition (2)

- Mixin composition adds members explicitly defined in ColoredPoint2D (members that were not inherited)
- Mixing a class C into another class D is legal only as long as D's superclass is a subclass of C's superclass.
- *i.e.*, D must inherit at least everything that C inherited
- Why?

Mixin Class Composition (2)

- Mixin composition adds members explicitly defined in ColoredPoint2D (members that were not inherited)
- Mixing a class C into another class D is legal only as long as D's superclass is a subclass of C's superclass.
- *i.e.*, D must inherit at least everything that C inherited
- Why?
- Remember that only members explicitly defined in ColoredPoint2D are mixin inherited
- So, if those members refer to definitions that were inherited from Point2D, they had better exist in ColoredPoint3D
 - They do, since ColoredPoint3D extends Point3D which extends Point2D

Views (1)

- Defines a *coercion* from one type to another
- Similar to conversion operators in C++ and C#

```
trait Set {
    def include(x: int): Set;
    def contains(x: int): boolean
}
def view(list: List) : Set = new Set {
    def include(x: int): Set = x prepend xs;
    def contains(x: int): boolean =
      !isEmpty && (list.head == x || list.tail contains x)
}
```

Views (2)

- Views are inserted automatically by the Scala compiler
- If e is of type T then a view is applied to e if:
 - Expected type of e is not T (or a supertype)
 - A member selected from e is not a member of T
- Compiler uses only views in scope

Variance Annotations (1)

```
class Array[a] {
    def get(index: int): a
    def set(index: int, elem: a): unit;
}
```

- Array[String] is not a subtype of Array[Any]
- If it were, we could do the following:

```
val x = new Array[String](1);
val y : Array[Any] = x;
y.set(0, new FooBar());
// just stored a FooBar in a String array!
```

Variance Annotations (2)

- Covariance is OK with functional data structures
- ... because they are immutable

```
trait GenList[+T] {
 def isEmpty: boolean;
 def head: T;
 def tail: GenList[T]
object Empty extends GenList[All] {
 def isEmpty: boolean = true;
 def head: All = throw new Error("Empty.head");
 def tail: List[All] = throw new Error("Empty.tail");
class Cons[+T](x: T, xs: GenList[T]) extends GenList[T] {
 def isEmpty: boolean = false;
 def head: T = x;
 def tail: GenList[T] = xs
```

Variance Annotations (3)

• Can also have contravariant type parameters

- Useful for an object that can only be written to
- Scala checks that variance annotations are sound
 - Covariant positions: Immutable field types, method results
 - Contravariant: method argument types
 - Type system ensures that covariant parameters are only used covariant positions
 - (similar for contravariant)
- If no variance specified, then Invariant
 - Neither superclass, nor subclass

Functions are Objects

- Every function is a value
 - Values are objects, so functions are also objects
- The function type S => T is equivalent to the class type scala.Function1[S, T]

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

For example, the anonymous successor function (x: Int)
 => x + 1 or in shorter code (_ + 1) expands to

```
new Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
}
```

Arrays are Objects

Arrays (mathematically): Mutable functions over integer ranges

Syntactic Sugar

a(i) = a(i) + 2 for a.update(i, a.apply(i) + 2)



```
final class Array[T](_length: Int)
    extends java.io.Serializable
    with java.lang.Cloneable {
    def length: Int = ...
    def apply(i: Int): T = ...
    def update(i: Int, x: T): Unit = ...
    override def clone: Array[T] = ...
}
```

Partial Functions

- Functions that are defined only for some objects
- Test using isDefinedAt

```
Example
trait PartialFunction[-A, +B] extends (A => B) {
    def isDefinedAt(x: A): Boolean
    def orElse[A1 <: A, B1 >: B]
    (that: PartialFunction[A1, B1]): PartialFunction[A1, B1]
}
```

- Blocks of pattern-matching cases are instances of partial functions
- This lets programmers write control structures that are not easy to express otherwise

Automatic Closure Construction

- Allows programmers to make their own control structures
- Can tag the parameters of methods with the modifier def
- When method is called, the actual def parameters are not evaluated and a no-argument function is passed

Example: Custom loop construct

```
object TargetTest1 with Application {
    def loopWhile(def cond: Boolean)(def body: Unit): Unit =
        if (cond) {
            body;
            loopWhile(cond)(body);
        }
    var i = 10;
    loopWhile (i > 0) {
        Console.println(i);
        i = i - 1;
    }
}
```

Types as Class Members

```
abstract class AbsCell {
  type T;
  val init: T;
  private var value: T = init;
  def get: T = value;
  def set(x: T): unit = { value = x }
}
def createCell : AbsCell {
  new AbsCell { type T = int; val init = 1 }
}
```

 Clients of createCell cannot rely on the fact that T is int, since this information is hidden from them

Next time

- Parallelism in Scala: actors and messages
- Message passing programming
- Event based programming
- Map-Reduce and BSP