CSC 347 - Concepts of Programming Languages

Scala Introduction

Instructor: Stefan Mitsch

Learning Objectives

❓ How to combine multiple programming paradigms in a single language?

- Understand Scala basic syntax
- Understand functional programming in Scala
- Understand lists in Scala

Scala

- Functional and object-oriented PL
- Java + ML + more
- Scalable Component Abstractions
- Compiles to JVM
- Interop: Scala calls Java; Java calls Scala
- Examples
  - Twitter/X Scala School
  - Apache Spark (Scala, Java, Python, R)
  - Chicago Scala Meetup

Scala

- Scala has a read-evaluate-print loop (REPL)
- Boolean literals: `false || true`
- Numeric literals: `1 + 2`
- String literals: `("hello" + " " + "world").length`
- Use of Java's libraries

```
1 val dir = java.io.File ("/tmp")
2 dir.listFiles.filter (f => f.isDirectory && f.getName.startsWith ("c"))
```

Everything is an Object

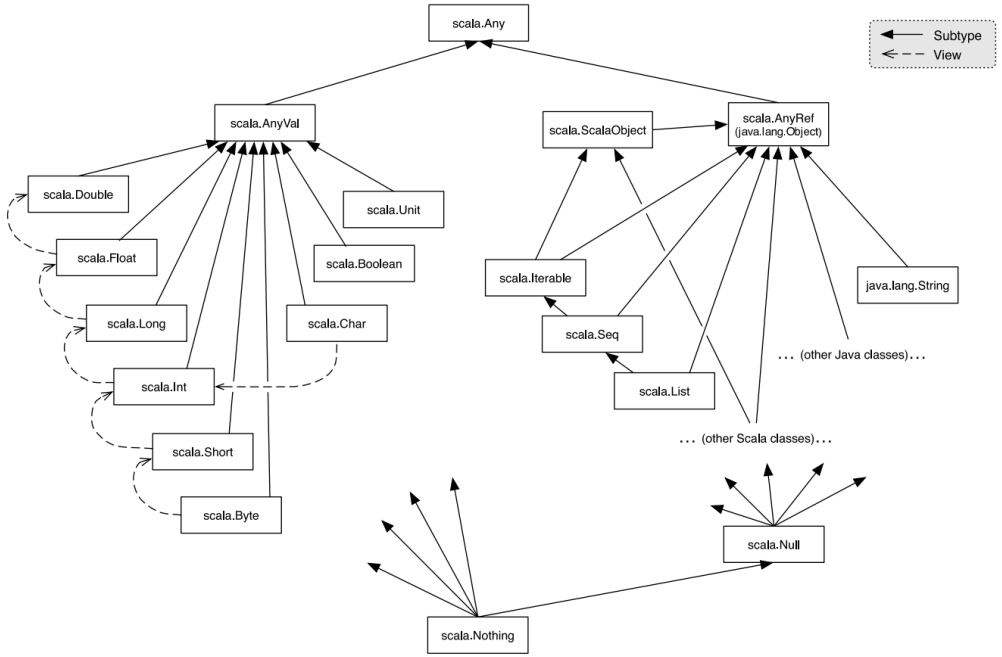
- `5:Int` is an object of type `Int` with methods: `5.toDouble`
- Methods can have symbolic names (see `scala.Int`): `5.+ (6)`
- `scala.runtime.RichInt` adds more methods: `5.max (6)`
- Any unary function `e1.f(e2)` can be written as `e1 f e2`
  - `5 + 6` is `5.+ (6)`
  - `5 max 6` is `5.max (6)`

Scala Type Checking

- Scala performs static type checking

```
1 def f () = 5 - "hello"   // rejected by type checker
```

- REPL prints types of expressions
- Java type hierarchy is embedded in Scala
  - Java primitive types are Scala value types
  - Java reference types are Scala reference types
  - `java.lang.Object` is `scala.AnyRef`



Mutable and Immutable Variables

Mutable Variables

- Java

```java
int x = 10;          // declare and initialize x
x = 11;              // assignment to x OK
```

- C

```c
int x = 10;          // declare and initialize x
x = 11;              // assignment to x OK
```

- Scala

```scala
var x = 10           // declare and initialize x
x = 11               // assignment to x OK
```

Immutable Variables

- Java

```java
final int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails
// error: cannot assign a value to final variable x
```

- C

```c
const int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails
// error: assignment of read-only variable ‘x’
```

- Scala

```scala
val x = 10         // declare and initialize x
x = 11             // assignment to x fails
// error: reassignment to val
```

Expression Sequencing

C Statements

```c
int f() {
  s_1;
  s_2;
  ...
  return ...;
}
```

Java Statements

```java
int f() {
  s_1;
  s_2;
  ...
  return ...;
}
```

Scala Expressions

```scala
def f() = {e_1; e_2; ...; return e_n}
```

Semicolons and `return` optional

```scala
def f() : Int = {
  e_1
  e_2
  ...
  e_n
}
```

Methods

- Parameters require type annotations

```
1 def plus (x:Int, y:Int) : Int = x + y
2 def times (x: Int, y:Int)     = x * y
```

- Return types
  - can often be inferred
  - but are required for recursive methods
- Body of a method is an expression; its value is returned

Methods

- Conditional expressions

```
1 def fact (n:Int) : Int = if n <= 1 then 1 else n * fact (n - 1)
```

- Compound expressions for side-effects

```
1 def fact(n:Int) : Int =
2   println("called with n=%d".format(n))
3   if n <= 1 then
4     println("no recursive call")
5     1
6   else
7     println("making recursive call")
8     n * fact(n - 1)
```

- Syntax like C statements, but are expressions!

Methods vs. Fields

- `def` can be used non-parameterized: `def x = 5`; non-strict, executed every time
- `val` declares a variable: `val x = 5`; strict, initialized once
- `lazy val`: memoized `def`, initialized on demand

Scala

```
1 class C:
2   val x = 1
3   lazy val y = 1 + 2
4   def z = 1
```

Expressed in Java

```
 1 public class C {
 2   private final int x = 1;
 3   private Integer y = null;
 4   public int x() { return x; }
 5   public int y() {
 6     if (y == null) y = 1 + 2;
 7     return y;
 8   }
 9   public int z() { return 1; }
10 }
```

Mutable Fields

Scala

```scala
class C:
  val x = 1
  var z = 1
```

Expressed in Java

```java
public class C {
  private final int x = 1;
  private int z = 1;
  public int x() { return x; }
  public int z() { return z; }
  public void z_$eq(int z) { this.z = z; }
}
```

Structured Data

- Tuples: fixed number of heterogeneous items `(1, "hello")`
- Lists: variable number of homogeneous items
  `List(1, 2, 3)` or `1 :: 2 :: 3 :: Nil`
- Immutable and mutable variants
- Pattern matching to decompose structured data into its components

Scala Collections

- Scala collections guide
- `scala.collection`
- `scala.collection.immutable`
- `scala.collection.mutable`
- `java.util` is available
- Scala has arrays `Array[Int]`

Mutability: Fields vs. Data

- Field mutability is different from data mutability
- Java *mutable linked list* by default

```
1 List<Integer> xs = new List<> ();
2 final List<Integer> ys = xs; // aliasing
3 xs.add (4); ys.add (5); // list is mutable through both references
4 xs = new List<> ();     // reference is mutable
5 ys = new List<> ();     // fails; reference is immutable
```

- Scala *immutable linked list* by default

```
1 var xs = List (4, 5, 6)
2 val ys = xs
3 xs (1) = 7; ys (1) = 3  // fails; list is immutable
4 xs = List (0)           // reference is mutable
5 ys = List ()            // fails; reference is immutable
```

Tuples

💡 Tuples are immutable heterogeneous complex data items

Scala Tuples

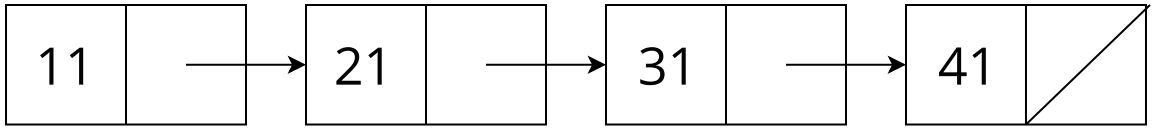
```
1 val p : (Int, String) = (5, "hello")
2 val x : Int = p(0)
```

Expressed in Java

```
1 public class Pair<X,Y> {
2   final X x;
3   final Y y;
4   public Pair (X x, Y y) { this.x = x; this.y = y; }
5 }
6
7 Pair<Integer, String> p = new Pair<> (5, "hello");
8 int x = p.x;
```

Linked Lists

- Scala's `::` is an *infix* operator to construct lists




```scala
val xs = 11 :: (21 :: (31 :: (41 :: Nil))) // List(11, 21, 31, 41)
val xs = 11 :: 21 :: 31 :: 41 :: Nil       // right associative
// method-call style, not encouraged!
val xs = Nil.::(41).::(31).::(21).::(11)
```

- List constructors

```scala
List (1, 2, 1 + 2)
1 :: 2 :: (1+2) :: Nil
```

List Projections

- Projections extract components of a list: often called `head` and `tail`

```
1 xs.head
2 xs.tail
```

Summary

- Scala combines functional and object-oriented programming
- Builtin support for tuples