

CSC 347 - Concepts of Programming Languages

Scala Introduction

Instructor: James Riely



Learning Objectives

- ❓ How to combine multiple programming paradigms in a single language?
 - Identify Scala basic syntax
 - Identify and describe tuples in Scala
 - Identify and describe 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 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)`

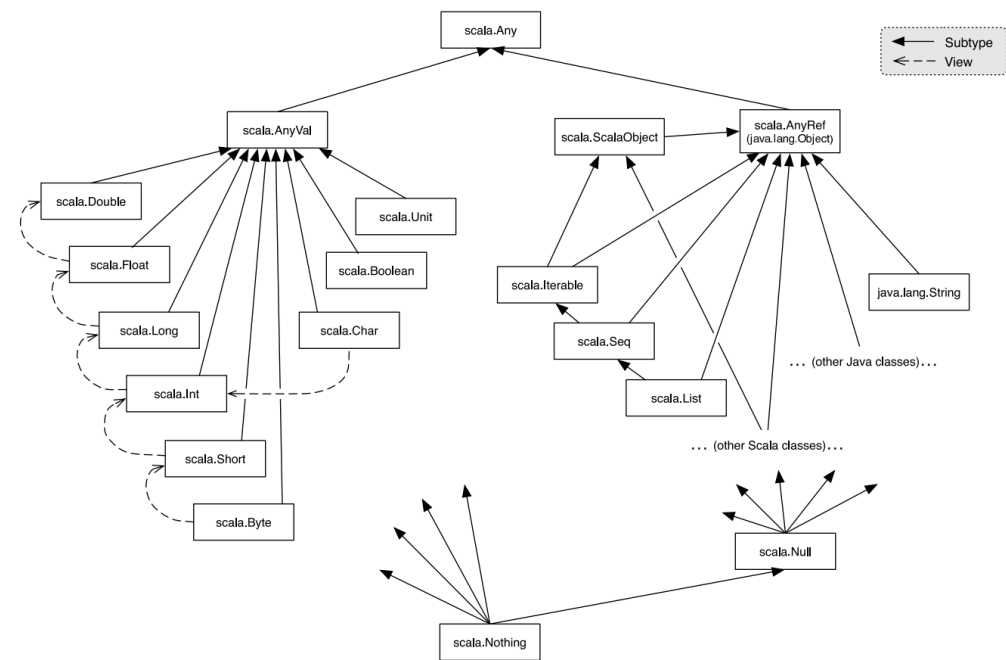


Scala Type Checking

- 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

- Scala

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

- ? Java
- ? C

Immutable Variables

- Scala

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

- ? Java
- ? C



Mutable and Immutable Variables

Mutable Variables

- Scala

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

- Java

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

- C

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

Immutable Variables

- Scala

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

- Java

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

- C

```
1 const int x = 10; // declare and initialize x
2 x = 11;           // assignment to x fails
3 // error: assignment of read-only variable 'x'
```



Expression Sequencing

C Statements

```
1 int f() {  
2     s_1;  
3     s_2;  
4     ...  
5     return e_n;  
6 }
```

Java Statements

```
1 int f() {  
2     s_1;  
3     s_2;  
4     ...  
5     return e_n;  
6 }
```

Scala Expressions

```
1 def f() : Int =  
2     e_1  
3     e_2  
4     ...  
5     e_n  
6 end f
```

Also allowed:

```
1 def f() : Int = {  
2     e_1;  
3     e_2;  
4     ...  
5     return e_n;  
6 }
```



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 (recursive)

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

- For-expressions (non-recursive)

Range generator and iterator

```
1 def fact(n: Int) : Int =  
2   var result = 1  
3   for i <- 1 to n /* by 1 */ do  
4     result *= i  
5   result  
6 end fact
```

Collection iterators

```
1 def sum(xs: List[Int]) : Int =  
2   var result = 0  
3   for n <- xs do  
4     result += n  
5   result  
6 end sum
```



Scala Collections

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



Tuples

💡 Tuples: fixed number of immutable heterogeneous data items

Scala Tuples

```
1 val p = (5, "hello")
2 // val p : (Int, String) = (5, "hello")
3 val x = p(0)
4 // val x : Int = p(0)
5 val q = 5 -> "hello" // alternate syntax
6 // val q : (Int, String) = (5, "hello")
```

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;
```

Expressed in C++

```
1 #include <tuple>
2 auto p = std::make_tuple(5, std::string("hello"));
3 // std::tuple<int, std::string> p = std::make_tuple(5, std::string("hello"))
4 auto x = std::get<0>(p)
5 // int x = std::get<0>(p)
```

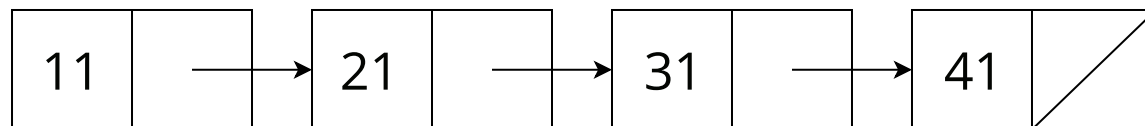


Linked Lists

💡 Lists: variable number of immutable homogeneous data items

- List constructors

```
1 List (11, 21, 30+1, 82/2)
2 11 :: 21 :: (30+1) :: (82/2) :: Nil
```



- Scala's `::` is an *infix* operator to construct lists (pronounced "cons")

```
1 val xs = 11 :: 21 :: 31 :: 41 :: Nil
2 // cons is right-associative - here it is with explicit parentheses
3 val ys = 11 :: (21 :: (31 :: (41 :: Nil)))
4 // method-call style, not encouraged!
5 val zs = Nil.:::(41).:::(31).:::(21).:::(11)
```



List Projections

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

```
1 val xs = List(11, 21, 31, 41)
2 val x  = xs.head // x == 11
3 val ys = xs.tail // ys == List(21, 31, 41)
```



Summary

- Scala combines functional and object-oriented programming

- **Everything is an expression** with a result value (even loops)
- **Type inference:** `val x = 10` is `val x: Int = 10`
- **Immutable vs mutable** variables: `val x = 10` vs. `var x = 10`

- **Methods**
 - require argument types
 - can infer return types
 - last expression in body determines return value
- **Tuples** `(1, "hello")` or `1 -> "hello"`
- **Immutable lists** `List(1,2,3)` or `1 :: 2 :: 3 :: Nil`



Execution of Factorial (From Lecture)

```
1 def fact(n: Int) : Int = if n <= 1 then 1 else n * fact (n - 1)
2
3 fact(3)
4 --> if n <= 1 then 1 else n * fact (n - 1) // where n=3
5 --> if 3 <= 1 then 1 else 3 * fact (3 - 1)
6 --> if false then 1 else 3 * fact (3 - 1)
7 --> 3 * fact (3 - 1)
8 --> 3 * fact (2)
9 --> 3 * (if n <= 1 then 1 else n * fact (n - 1)) // where n=2
10 --> 3 * (if 2 <= 1 then 1 else 2 * fact (2 - 1))
11 --> 3 * (if false then 1 else 2 * fact (2 - 1))
12 --> 3 * (2 * fact (2 - 1))
13 --> 3 * (2 * fact(1))
14 --> 3 * (2 * (if n <= 1 then 1 else n * fact (n - 1))) // where n=1
15 --> 3 * (2 * (if 1 <= 1 then 1 else 1 * fact (1 - 1)))
16 --> 3 * (2 * (if true then 1 else 1 * fact (1 - 1)))
17 --> 3 * (2 * 1)
18 --> 3 * 2
19 --> 6
```