Week 1: Basics

What is Scala?

- Developed by the EPFL in Switzerland
- Object Oriented
- Functional
- Java Virtual Machine Runtime
- Combines improved Java OO with Haskell functional concepts, along with other features.

What's it look like?

```java
// Java
package com.xorinc.scalatour;

public class Main {

   public static void main(String args...) {

      System.out.println("Hello, world!");
   }
}
```

```scala
// Scala
package com.xorinc.scalatour

object Main extends App {

   println("Hello world!")
}
```

What are the cons?

- Runtime is slower than Java, because many Scala features don't work directly on the JVM.
- Compiler is orders of magnitude slower than Java's, because Scala does a lot of inference.
- Some of the syntax can be abused to create hard-to-modify programs.

If it's got so many issues, why do people use it?

- Scala's functional features make writing a large class of programs easy and natural.
- Scala removes large amounts of the "boilerplate" Java developers know and love.
- The language is incredibly flexible and extensible, allowing libraries to abstract complex logic away.

Why learn Scala?

- Learning Scala teaches functional techniques, which are common to many modern functional languages.
- Functional programming is becoming more popular in industry because it is powerful yet maintainable.
- Algorithms *can* be simpler to understand in functional form. Not all algorithms are like this, however.

Basic Syntax

```
// I am a comment
/* I am a
   multiline comment */
package com.xorinc.scalatour // package declaration *

object Main extends App { // singleton declaration and inheritance *

    val anInt = 1; // immutable `val`ue definition
                   // the semicolon is allowed but not required
    var anDouble: Double = 0.5 // mutable `var`iable definiton
    anDouble = 1.2 // var reassignment

    def plusTwo(n: Int) = n + 2 // function `def`inition

    println(plusTwo(anInt)) // function call/application
}
// * we'll talk about these later
```

Identifiers

```
// In scala, pretty much anything is a legal identifier

val alpha1234 = 1
val 0isanumber = 0 // identifiers can't start with numbers

// symbols can be used but can't mix with letters/numbers
val =@= = "?"

def foo_=(i: Int) // alphanum_symbol is allowed only in that order

// if wrapped in backticks, a string becomes an identifier
var `Look Im an Identifier!` = -3
```

Identifiers

```
// none of these can be used as identifiers

abstract    case        catch       class       def
do          else        extends     false       final
finally     for         forSome     if          implicit
import      lazy        match       new         null
object      override    package     private     protected
return      sealed      super       this        throw
trait       try         true        type        val
var         while       with        yield
_     :     =     =>     <-     <:     <%     >:     #     @
⇒     ←

// note that wrapping any of these in backticks makes them legal
val `null` = null
```

Literals

```
// Numbers
0 -42 78213 5 // Int literals
-53L 82l 0L // Long literals
1.0 0D -0.0 12.34d // Double literals
3.3f -54F // Float literals

// Stringlike
'a' '"' '\'' '\0' // Char literals
"Hi I'm a String" "" """I am a
multiline string""" // String literals
'this 'symbol // Symbol literals

// Other
true false // Boolean literals
null // Null literal (more on this later)
() // Unit literal
```

`val` and `var`

```
// equivalent, compiler can guess the type
val anInt = 1
val anInt: Int = 1

// compiler complains, anInt is
// immutable and can't be changed
anInt = 2

var anDouble = 0.2
var anAny: Any = 0.2

// compiler complains, `anDouble` is of type Double
anDouble = "foo"

// anything can be assigned to an Any var
anAny = "foo"
```

def

```
// a basic function definiton
def doSomething(n: Int) = n + 1

// equivalent definition
def doSomething(n: Int): Int = {
    // return keyword is optional
    return n + 1
}

doSomething(3) // function application; evaluates to `4`

foo.bar(baz) // method application (more on what this means later)

// infix notation
1 + 2 /*and*/ (1).+(2)
// are equivalent

// beware of symbol vs alphanum indents
==#>foo`def` /*and*/ `==#>`.`foo`(`def`)
// are equivalent (!)
```

`Unit`

- `Unit` is the analogue to Java's `void`.
- Unlike `void`, `Unit` has exactly one value: `()`.
- Functions declared to return `Unit` (sometimes called procedures) insert a `()` literal at the end of the definition; the last expression is evaluated but ignored.

Arithmetic

```
// arithmetic ops are the same as Java

// addition, subtraction, product, quotient, modulo (remainder)
+ - * / %

// bitwise and, or, xor, not; bit shifts
& | ^ ~ << >> >>>

// boolean and, or, not
&& || !

// equals, not equals, less, greater, less/eq, greater/eq
== != < > <= >=

// No ++ or -- (!)
// Remember a.==(b) is a legal way to write a == b
```

Conditionals

```
// Note that conditionals (if-else statements) are expressions
val aThing =
    if(condition) {
        ...
    } else if(condition2) {
        ...
    } else {
        ...
    }

// also, note that
val anotherThing = if(condition) { ... }
// is the same as
val anotherThing =
    if(condition) { ... }
    else { () /*literal Unit*/ }

// note that these is no `a ? b : c` ternary
// operation in scala; if-else replaces it.
```

`while` Loops

```
// exactly like Java's while loop
while (cond) {
    ...
}

// exactly like java's do-while
do {
    ...
} while (cond)

// Note that these are valid:
while (cond) foo()
do foo() while (cond)

// there is no "break"
// or "continue" in scala
```