

CompSci 220

Programming Methodology

01: Administration and Scala

Today's Goals

Administration

Introduction to Scala



Administration

Let us talk a little about how we manage this course...

Course Description

Development of individual skills necessary for designing, implementing, testing and modifying larger programs, including: use of integrated design environments, design strategies and patterns, testing, working with large code bases and libraries, code refactoring, and use of debuggers and tools for version control. There will be significant programming and a mid-term and final examination.

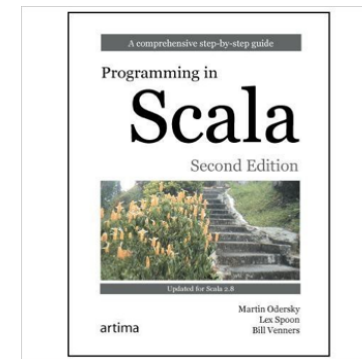
Prerequisite: CMPSCI 187 or ECE 242. 4 credits.

Course Objective

The objective of this course is to elevate your understanding of programming in general and in object-oriented and functional programming techniques in particular. We accomplish this through coverage of common design techniques, exposure to useful tools, exploration of typical patterns found in software engineering, and dissection of these techniques using a new programming language and concepts that will make you a better programmer and software engineer.

Books

- Programming in Scala 2nd Edition
Martin Odersky, Lex Spoon, Bill Venners
- Scala for the Impatient
Cay Horstmann
- Functional Programming in Scala
Paul Chiusano, Runar Bjarnason

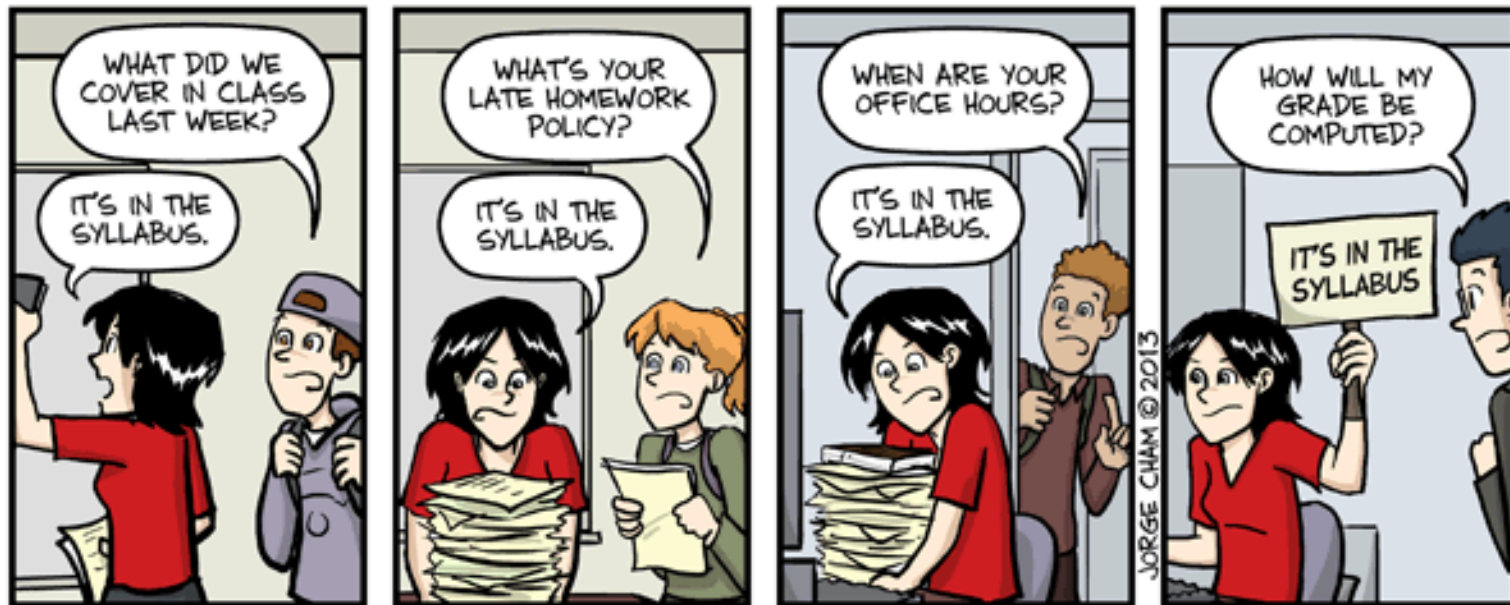


Course Website

<http://umass-cs-220.github.io>



Syllabus



IT'S IN THE SYLLABUS

This message brought to you by every instructor that ever lived.

WWW.PHDCOMICS.COM

Read the syllabus 😊



Reading the syllabus



Ignoring the syllabus



Schedule

- Part 1: Language and Environment
 - Scala Programming Language
 - Git Version Control
 - Comments and Documentation
 - Testing
- Part 2: Object-Oriented Design Patterns
 - Inheritance and Strategy Pattern
 - Observer Pattern
 - Factory and Singleton Pattern
 - Adaptor Pattern
 - Template Method and State Pattern

Schedule

- Part 3: Functional Programming
 - Function Expressions, Closure, and Evaluation
 - Immutability and Functional Data Structures
 - Handling Errors without Exceptions
 - Strictness and Laziness
 - Purely Functional State and Parallelism
 - Functional APIs
- Part 4: Technique
 - Regular Expressions
 - Relational Databases, SQL, and Database Programming

Software

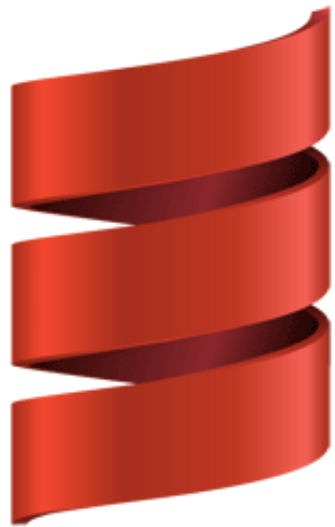
piazza

Software

The Moodle logo features the word "moodle" in a bold, lowercase, orange sans-serif font. A black graduation cap with a red tassel is positioned on top of the first letter 'm'.

moodle

Software



Scala

Software



IntelliJIDEA

Software

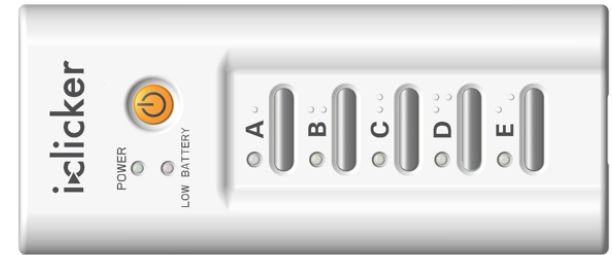


git

Software



Hardware



iclicker®

Assignments



- Project Assignments
- Exercise Assignments
- Unit Exams
- Final Exam
- i-Clicker

Assignments

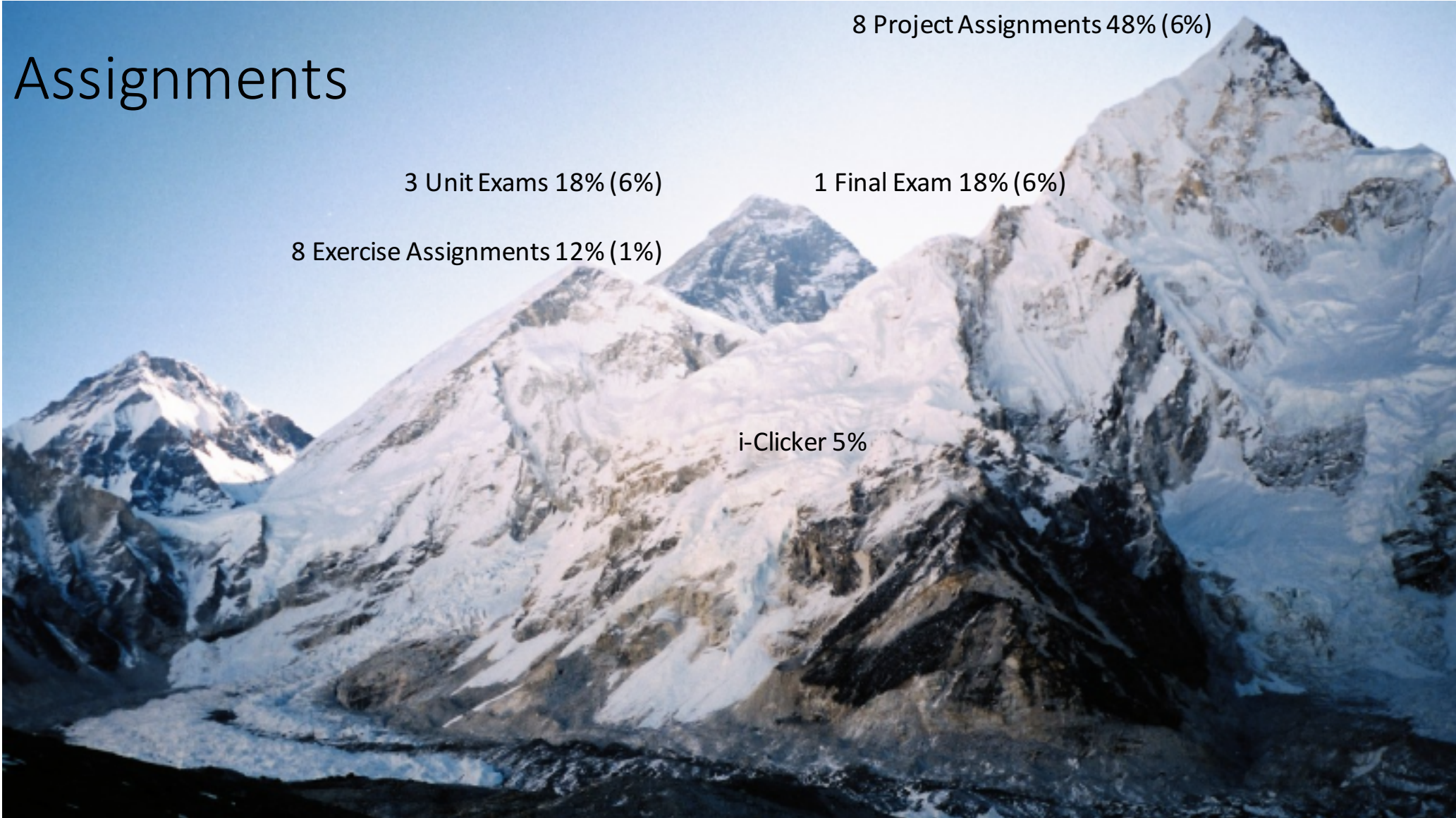
8 Project Assignments 48% (6%)

3 Unit Exams 18% (6%)

1 Final Exam 18% (6%)

8 Exercise Assignments 12% (1%)

i-Clicker 5%



Grading

- Points-based System

- 8 Project Assignments @ 100 pts each (800 pts total)
- 8 Exercise Assignments @ 25 pts each (200 pts total)
- 3 Unit Exams @ 100 pts each (300 pts total)
- 1 Final Exam @ 300 pts
- i-Clicker (80 pts total)

Total Points in Course: **1680**

No opportunity for extra-credit

1520-1680 = A- to A

1350-1519 = B- to B+

1176-1349 = C- to C+

1010-1175 = D to D+

0-1009 = F

Project Assignments



- Unit Tests
- Automatic Grading
- Grading Assistant
- Git Version Control
- Submission to Moodle

Exercise Assignments



- Unit Tests
- Automatic Grading
- Grading Assistant
- Git Version Control
- Submission to Moodle

Unit Exams



- On Moodle
- There are 3 of these
- 2 hours total
- Time Window

Final Exam



- On Moodle
- There is 1 of these
- 3 hours total
- During final exam slot

i-Clicker



- Active participation in class
- There are many of these

Policies

Late Submissions

Policies

Grade Issues

Policies

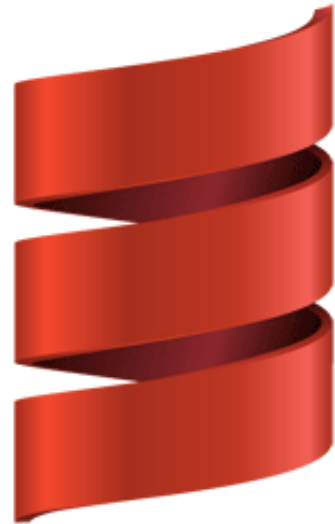
Academic Honesty

Policies

Disability Services

Policies

Incompletes



Scala

Introduction to Scala

Now on to some technical content...

Typesafe Activator

- Typesafe is a company that promotes the Scala programming language.
- Founded by Martin Odersky
- Activator is a tool for running, compiling, and testing Scala programs.



The Scala Interpreter (REPL)

```
scala>
```

Declaring Values and Variables

- Instead of using the names `res0`, `res1`, and so on, you can define your own names:

```
scala> val answer = 8 * 5 + 2
answer: Int = 42
scala> 0.5 * answer
res3: Double = 21.0
```

Declaring Values and Variables

- We use variables instead of values if you really need to change the contents.
- Perhaps surprisingly for Java and C++ programmers, most programs do not need many var variables.

```
var counter = 0
counter = 1      // OK, can change a var
```

Declaring Values and Variables

- We use variables instead of values if you really need to change the contents.
- Perhaps surprisingly for Java and C++ programmers, most programs do not need many var variables.

```
var counter = 0  
counter = 1 // OK, can change a var
```



NOTE: You need not specify the type of a value or variable. It is inferred from the type of the expression with which you initialize it.

Specifying the Type

- However, you can specify the type if necessary.

```
val greeting: String = null  
val greeting: Any = "Hello"
```



NOTE: in Scala, the type of a variable or function is always written after the name of the variable or function.

No Semicolons ;

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
```

```
val greeting, message: String = null  
// greeting and message are both strings,  
// initialized with null
```



NOTE: You may have also noticed that there were no semicolons after variable declarations or assignments. In Scala, semicolons are only required if you have multiple statements on the same line.

Commonly Used Types

- Include the following:
 - Byte
 - Char
 - Short
 - Int
 - Long
 - Float
 - Double
 - Boolean

However, unlike Java, these types are **classes**. There is no distinction between primitive types and class types in Scala.

Commonly Used Types

- Include the following:
 - Byte
 - Char
 - Short
 - Int
 - Long
 - Float
 - Double
 - Boolean

However, unlike Java, these types are **classes**. There is no distinction between primitive types and class types in Scala.

```
1.toString()
```

Or, more excitingly,

```
1.to(10)
```

Wrapper Types

- In Scala, there is no need for wrapper types. It is the job of the Scala compiler to convert between primitive types and wrappers to make it more efficient for execution on the Java Virtual Machine.
- For example:
 - Int becomes int
 - Char becomes char
 - Float becomes float

Rich Classes

- As you noticed, the Scala language relies on the underlying `java.lang.String` class for strings.
- However, it augments that class with well over a hundred operations in the `StringOps` class.

```
"Hello".intersect("World")
```

The `intersect` method is a method of the `StringOps` class not the `java.lang.String` class.

Rich Classes

- As you noticed, the Scala language relies on the underlying `java.lang.String` class for strings.
- However, it augments that class with well over a hundred operations in the `StringOps` class.

```
"Hello".intersect("World")
```



NOTE: Scala implicitly converts `String` objects to `StringOps` objects to apply the **intersect** method. Similarly, there are classes such as `RichInt`, `RichDouble`, `RichChar`, and so on, that enrich the available methods on `Int`, `Double`, `Char` respectively.

Arithmetic and Operator Overloading

- Arithmetic operators in Scala work just as you would expect in Java or C++:

```
val answer = 8 * 5 + 2
```

- The + - * / % operators do their usual job, as do the bit operators & | ^ >> <<.
- However, in Scala, these operators are actually methods!

Arithmetic and Operator Overloading

`a + b`



NOTE: Here, + is the name of the method. Scala allows alphanumeric characters in method names.

is a *shorthand* for

`a .+ (b)`

Arithmetic and Operator Overloading

In general, you can write:

a method b

as a *shorthand* for

a.method(b)

Calling Functions and Methods

- Scala has functions in addition to methods.

```
import scala.math._  
sqrt(2)  
pow(2, 4)  
min(3, Pi)
```


Calling Functions and Methods

- Scala does not have static methods.
- Rather, it has **singleton objects** and more specifically **companion objects** whose methods act like static methods in Java.
More on this next week...

```
BigInt.probablePrime(100, scala.util.Random)
```



Methods Without Parameters

- Scala methods without parameters often don't use parenthesis.
- A call to a parameterless method without parenthesis typically indicates that the method does not modify the object.

```
"Hello".distinct
```

This returns a new String containing the distinct letters in the original String. It does not modify the original String.

The apply Method

- In Scala, it is common to use a syntax that looks like a function call. For example, if s is a string, then $s(i)$ is the i^{th} character of the string.
- In C++, you would write $s[i]$
- In Java, it would be $s.charAt(i)$

In Scala, you can think of this as an *overloaded* form of the $()$ operator. It is implemented as a method with the name *apply*.

StringOps.apply

- The documentation for the StringOps class you will find a method:

```
def apply(n: Int) : Char
```

- That is, “Hello”(4) is a shortcut for

```
“Hello”.apply(4)
```

BigInt.apply

- In the documentation for the BigInt companion object you will find apply methods that let you convert strings or numbers to BigInt objects:

```
BigInt("1234567890")
```

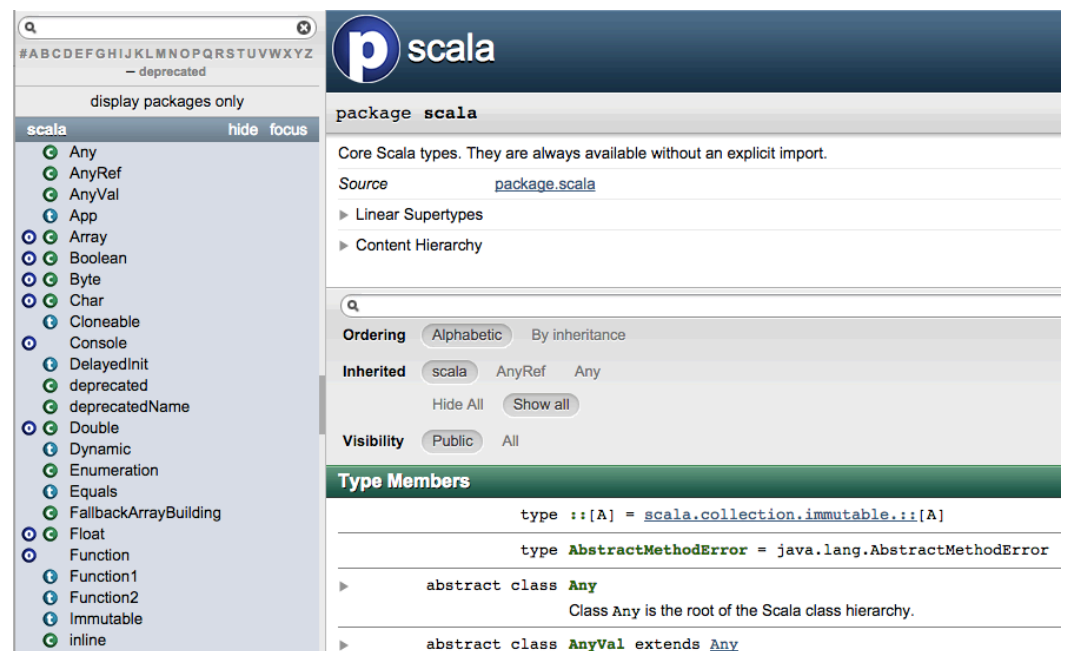
- is a shortcut for

```
BigInt.apply("1234567890")
```

Scaladoc

- API documentation for Scala (similar to Javadoc).
- Navigating Scaladoc is a bit more challenging than Javadoc. Scala classes often have many more convenience methods than Java classes (rich interfaces).
- Some methods use features that you haven't learned yet

<http://www.scala-lang.org/api/current>



The screenshot displays the Scala API documentation for the `scala` package. The left sidebar shows a navigation menu with a search bar and a list of classes and packages, including `Any`, `AnyRef`, `AnyVal`, `App`, `Array`, `Boolean`, `Byte`, `Char`, `Cloneable`, `Console`, `DelayedInit`, `deprecated`, `deprecatedName`, `Double`, `Dynamic`, `Enumeration`, `Equals`, `FallbackArrayBuilding`, `Float`, `Function`, `Function1`, `Function2`, `Immutable`, and `inline`. The main content area shows the `scala` package page, which includes a search bar, a header with the Scala logo, and a description: "Core Scala types. They are always available without an explicit import." Below this, there are links for "Source" (pointing to `package scala`), "Linear Supertypes", and "Content Hierarchy". The "Type Members" section lists several types: `type ::[A] = scala.collection.immutable.::[A]`, `type AbstractMethodError = java.lang.AbstractMethodError`, `abstract class Any` (with a note: "Class Any is the root of the Scala class hierarchy."), and `abstract class AnyVal extends Any`.

Conditional Expressions

- Scala has an if/else construct with the same syntax as in Java or C++.
- However, in Scala, an if/else **has a value**.

```
if (x > 0) 1 else -1
```

Has a value of 1 or -1, depending on the value of x.

This makes it more clear,

```
val s =  
  if (x > 0) 1 else -1
```



NOTE: You could also declare a var s and assign the value of s in the corresponding branch. However, the form on the left is better because it can be used to initialize a val – making it easier to reason about.

More on this later in the course...

Conditional Expressions

- What about this?

```
if (x > 0) "positive" else -1
```

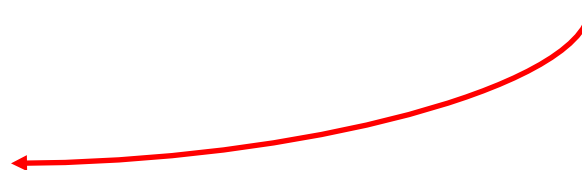
- And, what about this one?

```
if (x > 0) 1
```

This is a special value of type `Unit`

- One more:

```
if (x > 0) 1 else ()
```



Statement Termination

- In Java and C++, every statement ends with a semicolon.
- In Scala (also JavaScript), a semicolon is never required if it falls just before the end of the line.
- However, if you want to have more than one statement on a single line, you need to separate them with semicolons.

```
if (n > 0) { r = r * n; n -= 1 }
```

- If you want to continue a long statement over two lines make sure the line ends in a symbol that cannot be the end of a statement.

```
s = s0 + (v - v0) * t +  
    0.5 * (a - a0) * t * t
```