

# Starting Scala

## Using SBT and the REPL, Language Basics

# Agenda

1. The Scala REPL
2. SBT and the REPL - `sbt console`
3. Variables and Values
4. Types
5. Function Definitions
6. If expressions
7. Try..Catch..Finally expressions
8. Simple Loops

# The Scala REPL

- If you downloaded Scala already, you can start it by just typing `scala` at the command line

```
$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.
```

- So let's try `:help`

```
scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>          edit history
:help [command]              print this summary or command-specific help
:history [num]               show the history (optional num is commands to show)
:h? <string>                search the history
...
```

- The REPL tries to evaluate anything you type in

```
scala> 1 + 2
res0: Int = 3
```

# SBT

- SBT, (build tool for Scala) also lets you start the REPL
- In addition, it lets you switch Scala versions very easily (no other install needed)
- Using SBT to run Scala 2.12:
  1. Create a new folder on your computer: `mkdir MyDemoProject`
  2. Create a `build.sbt` file in the new folder with the contents  
`scalaVersion := "2.12.4"` (or some other scala version)
  3. Run `sbt console` at the command line
  4. Possibly wait a little bit as things download

```
$ sbt console
[info] Loading global plugins from /home/dwall/.sbt/0.13/plugins
[info] Set current project to testproject (in build file: ~/TestProject/)
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.

scala>
```

# First Time in the REPL

- To quit, either use Ctrl-D or :quit
- REPL: Read, Evaluate, Print, Loop
- Attempts to evaluate each line as you type it

```
scala> val x = 1 + 2
x: Int = 3
```

- If it can't evaluate the current line, it gives you a continuation

```
scala> val x =
| 1 + 2
x: Int = 3
```

- If you get stuck in continuation lines, hit enter twice in a row

```
scala> val x =
|
|
You typed two blank lines. Starting a new command.
```

# vals and vars

- A **val** is a final variable definition, it cannot be re-assigned with a different value

```
scala> val x = 10
x: Int = 10

scala> x = 11
<console>:12: error: reassignment to val
      x = 11
```

- A **var** is a mutable variable definition, it can be reassigned with another value of the **same type**

```
scala> var y = 10
y: Int = 10

scala> y = 11
y: Int = 11

scala> y = "eleven"
<console>:12: error: type mismatch;
 found   : String("eleven")
 required: Int
      y = "eleven"
```

# Hiding a val with another val

- What about?

```
scala> val x = 10
x: Int = 10

scala> println(x)
10

scala> val x = 11
x: Int = 11

scala> println(x)
11
```

- Didn't this just re-assign a val?

# Scopes in the REPL

- In fact the second `val x` hides the first, it's like this:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val x = 10
println(x)

{  // start a new scope
  val x = 11 // hides x in the outer scope
  println(x)
}

// Exiting paste mode, now interpreting.

10
11
```

- Every new REPL prompt is like a new scope
- Also note `:paste` mode, which allows you to decide when the REPL will evaluate your code

# Scala and types

- Remember:

```
scala> var x = 10
x: Int = 10

scala> x = "ten"
<console>:12: error: type mismatch;
 found   : String("ten")
 required: Int
      x = "ten"
```

- Scala won't let us re-assign an integer var to a String
- This is because x has a **type** of `Int` when we create it
- We could have initialized it like this:

```
var x: Int = 10
```

- The type goes after the name of the variable, separated by a :
- Scala will infer the best type it can if you don't specify it

# Method/Function Definitions

- In addition to `val` and `var`, Scala has `def`
- `defs` can take parameters and act on them

```
scala> def add(x: Int, y: Int): Int = x + y
add: (x: Int, y: Int)Int
```

- Note the `=` before the function body
- The return type can be inferred, but the parameter types cannot:

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> def add(x, y) = x + y
<console>:1: error: ':' expected but ',' found.
def add(x, y) = x + y
```

- Scala has nothing to infer the parameter types from

# If expressions

- In Scala, if..else expressions can return values (unlike many more imperative languages):

```
scala> val a = 10
scala> val b = 12
scala> val m = if (a > b) a else b
m: Int = 12
```

- You can ignore the returned value, then it looks just like other languages:

```
scala> var m = 0
scala> if (a > b) {
|   m = a
| } else {
|   m = b
| }
scala> m
res6: Int = 12
```

# Functional Style

- Ignoring the return value means you will need side effects to do anything useful, e.g. in this example we mutate a variable as our side effect
- Purely functional code avoids side-effects (and does not need vars)
- Hence Scala places a greater emphasis on expressions (that return values) rather than statements (that do not)
- In a Scala expression consisting of more than one nested expression, the final inner expression becomes the value of the overall expression, e.g.

```
scala> val maxSquaredDoubled = if (a > b) {  
    |   val aSquared = a * a  
    |   aSquared * 2  
    | } else {  
    |   val bSquared = b * b  
    |   bSquared * 2  
    | }  
maxSquaredDoubled: Int = 288
```

- Scala does have a `return` keyword, but there is rarely any need to use it

# Try..Catch..Finally expressions

- Scala also opts for expressions over statements in exception handling

```
scala> val divided = try {  
    |   10 / 0  
    | } catch {  
    |   case ae: ArithmeticException => 0  
    | } finally {  
    |   println("This always runs, but does not affect the result")  
    |   42  
    | }  
This always runs, but does not affect the result  
divided: Int = 0
```

- The `finally` block will always run, but will not return a value (hence its only use is side-effecting)
- If an exception is caught, a value may be returned to *recover*
- Uncaught exceptions are automatically re-thrown
- You may also choose to re-throw a caught exception (or throw another):

```
case ae: ArithmeticException => throw new RuntimeException("Can't divide by 0")
```

# Simple Loops

- Scala has only one true looping construct: `while` (and the associated `do..while`)
- `while` is a statement, and has no useful return type of its own
- `while` is non-functional and is often replaced by `foreach` or `map` functions over collections, or by `for` and `for..yield` blocks (we will cover these soon)
- `while` is still used for various reasons, including performance

```
var x = 0

while (x < 10) {
    println(s"the square of $x is ${x * x}")
    x += 1
}
```

- `while` must have a side-effect to do anything useful
- In Scala, everything has a return type, there is no `void`
- `Unit` is provided as a return type for statements, it has one instance: ()

# do..while

- While checks the *predicate* first before running the body of the loop
- Possibly the body will never be executed
- Do while executes the body at least once, and checks the predicate to see if it should repeat:

```
var x = 0

do {
    println(s"the square of $x is ${x * x}")
    x += 1
} while (x < 10)
```

- Note the use of string interpolation: `s"the square of $x is ${x * x}"`

# Running and Loading Scala Scripts

- You can also run Scala scripts (or load them into the repl)
- Create a file with some Scala code in it, e.g. `squares.sc` with:

```
var x = 1

while (x <= 10) {
  println(s"The square of $x is ${x * x}")
  x += 1
}
```

- To run it as a script, use `scala squares.sc`
- To load it, start `scala` or `sbt console`, then `:load squares.sc`. *note* we use `.sc` since otherwise `sbt` will try and compile it (which won't work for reasons we will see later)
- `Ctrl-C` will break out of a running script if things go wrong

# Module 1 Exercises

1. Create a Scala script file called `timestable.sc` in your working directory (make a directory if you need to)
2. Write two while loops from 1 to 5, one inside the other. Call one loop variable `x` and the other `y`
3. `println` a message in the inner loop that says `s"$x times $y is ${x * y}`
4. Remember to increment both `x` and `y` in their respective loops
5. Either run your script using `scala timestable.sc` or `sbt console` then `:load timestable.sc` to check that it works. You should get 25 lines of output.

# Module 1 Exercises - Extra Credit

- `toString` is a method that can be called on anything in Scala, for example:

```
scala> val x = 123
scala> x.toString // results in a string of "123"
```

- Furthermore, Strings have a `.contains` method that checks to see if a Char is contained anywhere in the String:

```
scala> val s = "123"
s: String = 123

scala> s.contains('3')
res2: Boolean = true

scala> s.contains('4')
res3: Boolean = false
```

- Alter your previous `timestable.sc` to only print out lines if the result of the multiplication contains either a '4' or a '6' digit in the number produced. `||` is the logical or operator in Scala

# Next Steps with Scala

More Scala Basics, and Using Worksheets

# Agenda

1. IntelliJ and Scala Worksheets
2. Method Parameters and Return Types
3. Expressions vs Statements
4. Tuples
5. Re-writing Rules
6. Collections
7. Extension Methods (Intro Only)
8. Functional vs Imperative Style

# Scala Projects in IntelliJ

- Group Activity, load a project into IntelliJ IDEA
- Unzip the `exercises.zip` file
- Open IntelliJ IDEA
- Import Project
- Find `build.sbt` in the exercises, highlight and click OK
- Accept all of the defaults, wait...

# Creating (or Opening) a Worksheet

- Open the Projects tab, then `scripts` (`scripts` is just a convention I use, it can be anywhere)
- Double-click on a worksheet (or right click on `scripts` to create a new one)
- Anything you type will be evaluated, like the REPL (but with full IDE features)
- Examples in this course are included as worksheets under `scripts`

# Method Parameters and Return Types

- Let's take a look at a method definition:

```
def max(x: Int, y: Int): Int = if (x > y) x else y
```

- This is *fully typed*, in that the parameter, and return, types are specified. We can drop the return type, since Scala can infer it

```
def min(x: Int, y: Int) = if (x < y) x else y
```

- As you will see in the worksheet, the method has the same exact type, the Int return type is inferred by the compiler
- However you cannot leave the type annotations off of the parameters, since Scala has no context to infer those from

# Methods with No Return Types

- Java (and some other languages) have a `void` keyword, which denotes "no return type"
- In Scala, **every** method and variable has a type, there is no `void`
- The rough equivalent is `Unit`, of which there is only one instance: `()`

```
def sayHi(name: String): Unit = println(s"hello $name")
```

- Methods resulting in `Unit` must have side effects in order to be useful (IO is one such side effect)
- Scala still has *procedural syntax*, which has the same effect **but is deprecated**

```
def sayHello(name: String) {  
    println(s"hello $name")  
}
```

- You will get a warning, and IntelliJ will try to correct you, **always** use `:Unit =` instead of procedural syntax

# Expressions vs Statements

- An expression returns its payload as a return argument with a type, e.g.:

```
val min = if (x < y) x else y
```

- A statement returns `Unit` and has to have some side effect to be useful:

```
if (x > y) println(s"max is $x") else println(s"max is $y")
```

- Functional programming style prefers expressions over statements
- Remember that `if`, `try...catch`, `for`, and other common constructs in Scala are *expressions*
- `while` and `do...while` are the only built in control flow constructs that only return `Unit`:

```
var doIt: Boolean = true
val result = while (doIt) {
    println("Hello")
    doIt = false
}
```

# Statements and Expressions

- `val` and `var` also produce `Unit` return types, this is surprising at first:

```
var x = 5
val y = x = 10
println(x) // 10
println(y) // ()
```

- A common mistake when first learning Scala is ending a code block with a `val`:

```
def add(a: Int, b: Int) = {
    val result = a + b
}

val sum = add(5, 6) // sum will be (): Unit!
```

- Can be avoided by adding the expected return type `Int` which is considered good practice

# Tuples

- So far we have looked at simple types like `Int`, `String`, and `Unit`, also our methods have returned just one of these
- What if we want to return more than one thing from a method? Enter *tuples*

```
def sumAndDifference(a: Int, b: Int): (Int, Int) = {  
    val sum = a + b  
    val difference = a - b  
    (sum, difference)  
}
```

- Getting the result parts:

```
val results = sumAndDifference(10, 5)  
  
results._1 // 15: Int  
results._2 // 5: Int
```

- The types are carried through, `_1` and `_2` can be thought of as item 1 and item 2

# Tuples

- There's a nicer way to get the parts:

```
val (sm, df) = sumAndDifference(10, 5)
```

- And the tuple can have more than 2 items, and mixed types:

```
val (a,b,c,d,e) = (0, 'u', 8, 1, "too")
a    // 0: Int
b    // 'u': Char
c    // 8: Int
d    // 1: Int
e    // "too": String
```

- Tuples can have arity up to 22, because it had to stop somewhere
- Future versions of Scala may (probably will) create tuple arities on the fly

# Re-writing Rules, infix

- Scala has no operators (as such), although it appears to:

```
val x = 1 + 2
```

- So what's + if it's not an operator? A method! The above can be re-written:

```
val y = 1.+ (2)
```

- This is known as infix notation, it works for all methods on an instance with one parameter, e.g.

```
val s = "hello"  
s.charAt(1)  
s charAt 1 // same result as above
```

- It does not work without an instance before the method though:

```
println "hello" // will not compile, needs parens
```

# Re-writing Rules, apply

- Let's create an array:

```
val arr = Array("scooby", "dooby", "doo")
```

- Getting items out of an array can be achieved with the `apply` method:

```
println(arr.apply(1)) // prints "dooby"
```

- Scala has a shortcut for `apply`, any item (other than a method) followed by parens calls `apply` with the contents of the parens (if any):

```
println(arr(0)) // prints "scooby", same as arr.apply(0) would
```

- In fact, the `Array` creation line above also uses this rule:

```
Array("scooby", "dooby", "doo")
// is re-written to
Array.apply("scooby", "dooby", "doo")
```

which calls the `apply` method on the *companion* object using *varargs* (we will learn about both of these soon)

# Re-writing Rules, update

- What if we update the value in an Array (arrays are mutable so they can be updated):

```
arr(0) = "scrappy"
```

- This is re-written to a call to `update` with the value in parens as the first argument, and the value after the equals as the second, so:

```
arr(1) = "dappy"  
// is re-written to  
arr.update(1, "dappy")
```

- The result of `update` is defined as `Unit` so in order to do anything useful, it must have a side effect

# Re-writing Rules General Notes

- We will see other Scala re-writing rules as we go through material
- Re-writing is **only** done if the code won't typecheck without a re-write
- If an item doesn't have an `apply` or `update` method, the re-write will be attempted but will fail to compile:

```
val z = 10
z(2) // "Application does not take parameters" compile error
```

```
val xs = List(1,2,3)  // could be written List.apply(1,2,3)
xs(1)   // works, gives back 2: Int
xs(1) = 10 // compile error, since no update method on immutable List
```

# Quick Collections Intro

- So far we have seen `Array` which is mutable, and just now `List` which is immutable
- Both collection types have a type parameter specifying what they hold:

```
val array1: Array[Int] = Array(1,2,3)
val list1: List[String] = List("scooby", "dooby", "doo")
```

- The type parameter is not optional, but can be inferred from the initialization contents

```
val array2 = Array(1,2,3) // Array[Int] is inferred
val list2 = List("scooby", "dooby", "doo") // List[String] is inferred
```

- When specifying a collection type in a method parameter (or return parameter), the type parameter must be provided!

```
def squareRootsOf(xs: List[Int]): List[Double] =
  for (x <- xs) yield math.sqrt(x)
```

# List Initialization

- As seen you can initialize a list using the `List.apply` method (or `List(contents...)` using re-writing)

```
val lista = List(1,2,3)
```

- For lists only, you can also use the *cons* form of initialization, using `::`:

```
val listb = 4 :: 5 :: 6 :: Nil
```

- `::` is *right associative*, that is, it applies the parameter on the *left* side to the item on the right, e.g.

```
val listb = ((Nil :: (6)) :: (5)) :: (4)
```

- Any operator *ending* in `:` is right associative in Scala
- Another list-only operator is concatenate, `:::` which joins two lists (again right associative):

```
val listc = lista ::: listb
```

# Sequences

- List and Array are both sequences in Scala, subtypes of Seq
- There are others, notably Vector:

```
val v = Vector(1,2,3,4)
```

- All can be passed in to a method requiring a Seq of the right type:

```
def squareRootOfAll(xs: Seq[Int]): Seq[Double] =  
  xs.map(x => math.sqrt(x))
```

- Now, List[Int], Array[Int] and Vector[Int] can all be passed in:

```
squareRootOfAll(v)  
squareRootOfAll(listc)  
squareRootOfAll(array2)
```

- Don't worry about the `x => math.sqrt(x)` notation just yet, we will deal with function literals soon

# Sets

- A `Seq` (sequence) is an ordered collection of homogenous values that may be repeated
- By contrast, a `Set` is an unordered collection of homogenous values that are unique

```
val set1 = Set(1,2,3,1,2,4,5) // Produces a Set(5,1,2,3,4)
```

- A `Set` cannot be passed to a function expecting a `Seq`, it is not a sub-type of `Seq`:

```
squareRootOfAll(set1) // will not compile
```

# (Im)mutability of Collections

- `Array` is mutable, may be grown, values may be updated, etc.
- `List` and `Vector` are immutable, once created the only way to change the size or update them is to transform them into another reference (or use a `var` to reassign the reference)
- `Set` has both mutable and immutable implementations:

```
import scala.collection._  
  
val s1 = mutable.Set(1,2,3)  
var s2 = immutable.Set(1,2,3)
```

- Now if we use `+=` on both of these:

```
s1 += 4 // works because s1 has a += operator  
s2 += 4 // works because s2 is a var
```

- For `s2`, Scala uses a *re-writing* rule to the expression to `s2 = s2 + 4`
- It is **not** required (nor recommended) to use a `var` and a `mutable` collection together

# Maps

- A Map can be thought of as an associative sequence of tuples, the first item of the tuple can be used to look up the second item
- Like Sets, Maps have both mutable and immutable implementations

```
val m1 = mutable.Map('a' -> 1, 'b' -> 2, 'c' -> 3)
var m2 = immutable.Map('d' -> 4, 'e' -> 5, 'f' -> 6)
```

- Updating the maps

```
m1 += m2 // calls += on the mutable map
m2 += 'g' -> 7 // re-writes to m2 = m2 + 'g' -> 7
```

- What's this 'g' -> 7 syntax about?
- It's not syntax, it's an extension method

# The -> extension method

- -> can be called on an instance of any type with one parameter of any other type
- The result is a `tuple2[FirstType, SecondType]` with the values of both instances
- It's mainly syntactic sugar for creating maps, but it's not a keyword. Here's how it works:

```
1 -> "one"
// is re-written to
1.->("one")
// is expanded to
ArrowAssoc(1).->("one")
```

- No such -> method exists on `Int`, but an implicit called `ArrowAssoc` provides it just in time
- Implicits will be covered in-depth later in the course

# Simple Map Iteration

- All that effort for `->` is to make maps easy and pretty to initialize

```
val mapToRiches = Map(  
  1 -> "steal underpants",  
  2 -> "???",  
  3 -> "profit"  
)
```

- They are also easy (and pretty) to iterate over with a `for` expression

```
for ((step, instruction) <- mapToRiches) {  
  println(s"Step $step - $instruction")  
}
```

- The `(step, instruction)` unpacks the `tuple2` from the sequence in the map
- However, remember that the order may vary in some map implementations

# Mutability vs Functional Style

- Statements, side-effects, vars, and mutability are not functional programming style
- Instead, aim for expressions, vals and immutability whenever possible
- Use vars or mutability when dictated by performance or other factors
- You don't need a `var` with a mutable collection, instead choose one or the other
- Don't let mutability escape into the API
- Don't optimize for performance prematurely
- Also keep methods short and uncomplicated, separate early and often

# Opening and Reading a File

```
import scala.io.Source  
  
for (line <- Source.fromFile("somefile.txt").getLines()) {  
    println(line)  
}
```

- Source is not often used in production code, but it is useful for demos and learning Scala

# Module 2 Exercises

- Find Module02 test class in exercises
- Run class in Scalatest
- Follow instructions in class to complete exercises and get all tests passing
- There may be a surprise or two in the exercises, ask questions...

# Classes, Objects, Apps and More

# Agenda

1. Class Definitions
2. Class Constructor
3. Parameters, Fields and Parametric Fields
4. A Rational Class
5. Checking Preconditions
6. Referencing Self
7. Infix Notation and Symbolic Method Names (Operators)
8. Auxiliary Constructors
9. Companion Objects and Factory Methods
10. Private Constructors
11. Overloading Methods
12. Implicit Conversions

# A Scala Class Definition

- On the JVM, all methods/fields must go inside classes (unlike the REPL)
- In Scala, this includes objects, traits, and package objects (more later)

```
class DemoWithFieldsAndMethods {  
    val x: Int = 10  
    val y: Int = x * 2  
  
    def timesY(a: Int): Int = a * y  
}
```

```
class DemoWithParams(name: String) {  
    println(s"Constructing for $name")  
  
    def sayHi(times: Int): Unit = {  
        var time = 0  
  
        while (time < times) {  
            println(s"Hi, $name")  
            time += 1  
        }  
    }  
}
```

# Constructor

```
class DemoWithParams(name: String) {  
    println(s"Constructing for $name")  
  
    // rest of class...  
}
```

- Parameters on the class definition become *primary constructor* parameters
- Code in the class (not in defs) becomes the *primary constructor* code, runs when a new instance is constructed
- Can't access the constructor parameters from outside (private)

```
val demo = new DemoWithParams("Jill")  
demo.name  
// Error:(33, 83) value name is not a member of DemoWithParams
```

# Parameters, Fields and Parametric Fields

- Constructor parameters are `private` (actually `private[this]`), also `vals`
- `private` and `protected` are keywords, there is no `public` keyword, that's the default for `vals` and `defs` (but not for constructor parameters)
- Adding a `val` keyword before the parameter definition makes it a `public parametric field`:

```
class DemoWithParams(val name: String) {
    println(s"Constructing for $name")
}

val demo = new DemoWithParams("Jill")
demo.name // Jill
```

- Parametric fields are idiomatic Scala (remember they are `vals`)

```
// how Scala re-writes the above:
class DemoWithParams(_name: String) { // parameter still private[this]
    val name: String = _name // the public field definition
    println(s"constructing for $name")
}
```

# A Rational Class

- As in, let's make something to represent a Rational number from what we know so far:

```
class Rational(val n: Int, val d: Int) // look ma, no body!  
  
val half = new Rational(1, 2)  
// half: Rational = Rational@6c643605
```

- Every class has a `toString` method that can be overridden:

```
class Rational(val n: Int, val d: Int) {  
  override def toString: String = s"R($n/$d)"  
}  
  
val half = new Rational(1, 2)  
// half: Rational = R(1/2)  
  
val divByZero = new Rational(1, 0)  
// divByZero: Rational = R(1/0) -- probably should prevent this
```

# Checking Preconditions in the Constructor

```
class Rational(val n: Int, val d: Int) {  
    require(d != 0, "Zero denominator!") // precondition  
  
    override def toString: String = s"R($n/$d)"  
}  
  
val half = new Rational(1, 2)  
// half: Rational = R(1/2)  
  
val divByZero = new Rational(1, 0)  
// java.lang.IllegalArgumentException: requirement failed: Zero denominator!
```

- If you use `require` and the predicate fails, you will get an `IllegalArgumentException` thrown
- The `String` field in `require` is optional but recommended

# Referencing Self

- Could also write `require(this.d != 0, "Zero denominator!")`
- `this` is a reference to the current instance. It is inferred by Scala when possible

```
class Rational(val n: Int, val d: Int) {
    require(d != 0, "Zero denominator!") // precondition

    override def toString: String = s"R($n/$d)"

    def min(other: Rational): Rational =
        if ((n.toDouble / d) < (other.n.toDouble / other.d))
            this else other // have to use this to return
}

val half = new Rational(1, 2)
val fifth = new Rational(1, 5)

val smaller = fifth min half
// smaller: Rational = R(1/5)
```

- Could have used `this` for the `n` and `d` references in `min`
- Note also infix use of `min` method, equivalent to `fifth.min(half)`

# Infix and Symbolic Methods

```
class Rational(val n: Int, val d: Int) {  
    require(d != 0, "Zero denominator!")  
  
    override def toString: String = s"R(${n}/${d})"  
  
    // rational addition  
    def add(other: Rational): Rational =  
        new Rational(  
            this.n * other.d + this.d * other.n,  
            this.d * other.d  
        )  
    }  
  
    val half = new Rational(1, 2)  
    val fifth = new Rational(1, 5)  
  
    val sum = half add fifth  
    // sum: Rational = R(7/10)
```

- Scala doesn't have operator overloading, per-se
- But it does have symbolic method names, (and operator precedence rules for first character)

<http://scala-lang.org/files/archive/spec/2.11/06-expressions.html#infix-operations>

# Infix and Symbolic Methods

```
class Rational(val n: Int, val d: Int) {  
    require(d != 0, "Zero denominator!")  
  
    override def toString: String = s"R($n/$d)"  
  
    // symbolic rational addition  
    def +(other: Rational): Rational =  
        new Rational(  
            this.n * other.d + this.d * other.n,  
            this.d * other.d  
        )  
    }  
  
    val half = new Rational(1, 2)  
    val fifth = new Rational(1, 5)  
  
    val sum = half + fifth  
    // sum: Rational = R(7/10)
```

- Change `add` to `+` and infix does the rest

# Adding an Int to a Rational

- Strategy one: construct a Rational from an Int
- Can use an *auxiliary constructor* for this:

```
class Rational(val n: Int, val d: Int) {  
    ...  
    def this(i: Int) = this(i, 1)  
    ...  
}  
  
val fifth = new Rational(1, 5)  
val five = new Rational(5)  
  
val sum = five + fifth
```

- Auxiliary constructors are quite limited, they can **only** call another constructor
- Better alternative is to use *factory methods*

# Introducing: Companion Objects

- An object in the same source file with the same name as the class (or trait)
- Shares private state and behavior with the class (and vice versa)
- Scala's alternative to `static`
- Good place for a factory method (or two):

```
// in same source file as Rational class
object Rational {
  def apply(n: Int, d: Int): Rational =
    new Rational(n, d)

  def apply(i: Int): Rational =
    new Rational(i, 1)
}

val fifth = Rational(1, 5) // can drop the new
val five = Rational(5)

val sum = five + fifth
```

# Because It's a Companion

- It can access private behavior on the class
- We can make the constructor private and use the factory methods only:

```
class Rational private (val n: Int, val d: Int) { // note position of private
  ...
}

object Rational {
  def apply(n: Int, d: Int): Rational =
    new Rational(n, d) // companion can still call new

  def apply(i: Int): Rational =
    new Rational(i, 1)
}

val fifth = Rational(1, 5) // R(1/5)
val five = Rational(5)   // R(5/1)
val half = new Rational(1, 2) // not allowed!
```

- Factory methods and private constructors are idiomatic

# Adding an Int to a Rational

- Strategy two, overloading:

```
class Rational private (val n: Int, val d: Int) {  
    require(d != 0, "Zero denominator!")  
  
    override def toString: String = s"R(${n}/${d})"  
  
    def +(other: Rational): Rational =  
        new Rational(  
            this.n * other.d + this.d * other.n,  
            this.d * other.d  
        )  
  
    def +(i: Int): Rational =  
        this + Rational(i) // from companion  
    }  
  
object Rational {  
    def apply(n: Int, d: Int): Rational =  
        new Rational(n, d)  
  
    def apply(i: Int): Rational =  
        new Rational(i, 1)  
    }  
  
Rational(1, 2) + 5 // R(11/2)
```

# Adding a Rational to an Int

- Can do `half + 5` and `Rational(5) + half` but not `5 + half`... `implicit!`

```
class Rational private (val n: Int, val d: Int) {  
    require(d != 0, "Zero denominator!")  
  
    override def toString: String = s"R($n/$d)"  
  
    def +(other: Rational): Rational =  
        new Rational(  
            this.n * other.d + this.d * other.n,  
            this.d * other.d  
        )  
    }  
  
object Rational {  
    def apply(n: Int, d: Int): Rational =  
        new Rational(n, d)  
  
    implicit def apply(i: Int): Rational =  
        new Rational(i, 1)  
}  
  
val half = Rational(1, 2)  
half + 5  
Rational(5) + half  
5 + half
```

# Implicits

- For implicit conversion, must `import language.implicitConversions` to avoid warning
- No longer need the overloaded `+` method for `Int`
- Implicits used by Scala to solve type problems
- Implicit conversion has single "in" type and single "out" type, e.g.

```
implicit def apply(i: Int): Rational = new Rational(i, 1)
```

- Companion objects **for types involved** in type problem are **one** of the places Scala looks for implicits
- Implicits in a companion object are hard to "un-invite"
- Must be marked with `implicit` keyword
- Name does not matter to Scala, only types matter
- Implicits can also be used for `val`, `object` and `class` (more on implicits later)

# Exercises for Module 3

- Find the Module03 class and run it with ScalaTest
- This time it will be all green, but read the instructions and you will see that you need to uncomment the tests, write code, and make it compile and pass again
- As you get each section working, there may be a section following that also needs to be uncommented

# Control Structures in Scala

Built in control structures, statements vs expressions

# Agenda

1. Expressions vs Statements
2. Unit
3. Scala's if expression
4. val and var
5. try..catch..finally
6. While loops
7. For expressions
8. The Four Gs of For
9. For, not just for loops
10. Match expressions
11. Guards
12. String interpolation

# Expressions vs Statements

- An expression has a return value, a statement does not (at least not a useful one)
- In Functional Programming, a *pure* expression has no effects other than those seen in the return value
- Scala has no `void` keyword for expressions, `void` is the absense of a return type, but in Scala **everything** has a return type
- But for statements, that return type is `Unit`, conversely, a return type of `Unit` denotes an expression

```
// an expression
val x = 1 + 2  // x: Int = 3

// a statement
println(x)  // prints 3

// since everything has a return type
val un = println(x)  // un: Unit = ()

un == ()  // () is the only instance of Unit
// Warning: comparing values of types Unit and Unit using `=='
// will always yield true
```

# Unit

- A `Unit` return type implies that a method must have a side effect to do something useful
- E.g. I/O, set or update a variable
- A non-`Unit` return type does not imply that there are no side effects, however
- Many built in constructs in Scala are expressions rather than statements, few return only `Unit`
- Even if you have side effects, there may be something more useful than `Unit` that you can return, you can always ignore it if you don't want to use it
- There is only one instance of `Unit`, it is `()` (sometimes referred to as empty tuple)
- `Unit` is descended from `AnyVal` like the "primitive" types

# Returning Something Other Than Unit

- E.g. a simple file writer:

```
import java.io._

class WriterOutput(writer: PrintWriter) {
  def write(s: String): Unit = writer.println(s)
}

val ex1 = new PrintWriter(new File("ex1.txt"))

val out1 = new WriterOutput(ex1)

out1.write("Hello")
out1.write("to")
out1.write("you")

ex1.close()
```

- `write` method returns `Unit`, to write out multiple things, invoke it on the same unit multiple times
- `close()` method also returns `Unit` (and has a side effect). In Scala, the convention is to always put parens on zero parameter methods that have side effects

# Returning this Instead Of Unit

- If you return `this` instead in the above example, you get a fluent API cheaply

```
class WriterOutput2(writer: PrintWriter) {  
    def write(s: String): WriterOutput2 = {  
        writer.println(s)  
        this  
    }  
}  
  
val ex2 = new PrintWriter(new File("ex2.txt"))  
  
val out2 = new WriterOutput2(ex2)  
  
out2.write("Hello").write("to").write("you")  
  
ex2.close()
```

# Scala's if Expression

- In many languages, e.g. Java, `if` is a statement (has no return value)

```
// this is Java
String fileName = "default.txt"; // defines a variable

if (args.length > 0) {
    fileName = args[0]; // side effect
}
```

- Java also has a ternary operator which is an expression:

```
String fileName = (args.length) > 0 ? args[0] : "default.txt";
```

- Scala combines these two things into one (`if` is an expression)

```
val fileName = if (args.length > 0) args(0) else "default.txt" // can now be val
```

- The return type is the combination of the types on both sides of the `else`, e.g.

```
val res = if (x > 0) x else false // type of res will be AnyVal
```

# val and var

- As we saw in module 1, `var` can change but `val` cannot
- Prefer `val` when you can use it, it makes code easier to reason about, and easier to refactor
- IntelliJ will give code hints when something can be a `val`
- You can also make it highlight `vars`, for example color them red
- When you have more expressions, you can have more `vals`

# try ... catch ... finally

- Like `if`, Scala's `try ... catch ... finally` is an expression
- The result (and type) is decided by the `try` and `catch` blocks
- If present, the `finally` block always runs, but doesn't affect the result or type

```
val args = Array.empty[String]

val fileName2 =
  try {
    args.head // throws exception on empty array
  }
  catch {
    case _: NoSuchElementException => "default.txt"
  }
  finally {
    println("Wheeeee")
    "the finally block"
  }

// fileName2: String = default.txt
```

# While Loop

- A statement (returns Unit)

```
def greet(n: Int): Unit = {  
    var i = 0  
    while (i < n) {  
        i += 1  
        println("hello")  
    }  
}  
  
greet(5)
```

You can avoid the `while` (and the `var`):

```
@tailrec  
def greet(n: Int, curr: Int = 0): Unit = {  
    if (curr < n) {  
        println("hello")  
        greet(n, curr + 1)  
    }  
}
```

There is also a `do..while` loop where the body is always called at least once

# For

```
for (i <- 1 to 10) println i * i
```

```
(1 to 10).foreach(i => println(i * i))
```

```
for (i <- 1 to 3; j <- 1 to 3) println(i * j)
```

```
(1 to 3).foreach(i => (1 to 3).foreach(j => println(i * j)))
```

```
for {  
    i <- 1 to 3 // {}s turn on semi-colon inference  
    j <- 1 to 3  
} {  
    println(i * j)  
}
```

Without `yield` block, `foreach` is used and `Unit` is the result type

# For ... Yield

More idiomatic in Scala (does not need to have side effect)

```
for (i <- 1 to 10) yield i * i  
  
(1 to 10).map(i => i * i)  
  
for (i <- 1 to 3; j <- 1 to 3) yield i * j  
  
(1 to 3).flatMap(i => (1 to 3).map(j => i * j))  
  
for {  
    i <- 1 to 3  
    j <- 1 to 3  
    k <- 1 to 3  
} yield {  
    i * j * k  
}  
  
(1 to 3).flatMap(i => (1 to 3).flatMap(j => (1 to 3).map(k => i * j * k)))
```

- For yield blocks, all generators are flatMap except the last which is map

# The Four Gs of For

```
val forLineLengths =  
  for {  
    file <- filesHere  
    if file.getName.endsWith(".sc")  
    line <- fileLines(file)  
    trimmed = line.trim  
    if trimmed.matches(".*for.*")  
  } yield trimmed.length
```

- **Generator:** `file <- filesHere`, denoted by the `<-`, all generators in the same for block must be of the same type (e.g. a collection, or a Try).
- **Guard:** `if file.getName.endsWith(".sc")`, guards short-circuit the for expression, causing filtering behavior.
- **inline assiGnment:** `trimmed = line.trim`, a simple val expression that may be used in both the remainder of the for block, and in the yield block.
- **Give (or the yield),** after the `yield` keyword comes the payoff of the for block setup.

```
filesHere.filter(_.getName.endsWith(".sc")).flatMap { file =>  
  fileLines(file).filter(_.trim.matches(".*for.*")).map { line =>  
    line.trim.length  
  }  
}
```

# For is More Than Just Loops

```
import scala.concurrent._  
import duration._  
import ExecutionContext.Implicits.global  
  
val f1 = Future(1.0)  
val f2 = Future(2.0)  
val f3 = Future(3.0)  
  
val f4 = for {  
    v1 <- f1  
    v2 <- f2  
    v3 <- f3  
} yield v1 + v2 + v3  
  
Await.result(f4, 10.seconds)
```

- Easy asynchronous programming
- Also Try, Option, Either, \*your type here\*
- All you need is a type with `foreach`, `map`, `flatMap`, `withFilter` with the correct type signatures

# Match Expressions

Like Java's `switch` but more powerful:

```
val x = 1

x match {
  case 1 => println("it's one")
  case 2 => println("it's two")
  case _ => println("it's something else")
}
```

It's an expression:

```
val res = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "something else"
}

// res: String = "one"
```

# Match Expression Guards

```
val n = -1

n match {
  case 0 => "It's zero"
  case v if v > 0 => s"It's positive $v"
  case v => s"It's negative ${v.abs}"
}

// res0: String = It's negative 1
```

The progression of tests only continues until the first match of everything to the left of the =>

# Match and More Match

Can match Strings, Lists, more...

```
def matchIt(x: Any): String = x match {  
    case "Hello" => "Well, hello back"  
    case 1 :: rest => s"A list beginning with 1, rest is $rest"  
    case Nil => "The empty list"  
    case 5 => "The number 5"  
    case _: List[_] => "Some kind of list, not empty and not starting with 1"  
}  
  
matchIt(5)           // The number 5  
matchIt(List(1,2,3)) // A list beginning with 1, rest is List(2, 3)  
matchIt(List(1))     // A list beginning with 1, rest is List()  
matchIt(List(2,3))   // Some kind of list, not empty and not starting with 1  
matchIt(Nil)         // The empty list  
matchIt(2.0)         // Exception!: MatchError
```

More on match later in the course

# String Interpolation

```
val x = 10
val y = 2.12
val name = "Fred"

s"$name $x $y"          // Fred 10 2.12
s"$name is ${x * y}"    // Fred is 21.200000000000003
f"$name is ${x * y}%08.4f" // Fred is 021.2000
s"$names"                // won't compile!
s"${name}s"               // Freds
"\t\n"
raw"\t\n"                 // \t\n
"""\t\n"""" // \t\n
```

- f interpolation follows the printf notation
- raw does not escape literals in the string

# Exercises for Module 4

- Find the Module04 class and run it with ScalaTest
- Follow the instructions and fix the tests until everything is green

# Functions and Closures

# Agenda

1. Private and Nested Methods
2. Function Literals
3. How Function Literals Work
4. Higher Order Functions
5. Placeholder Syntax
6. Partial Application of Functions
7. Closures
8. Partial Functions
9. Var Args in Methods
10. Argument Expansion
11. Named and Default Parameters

# Private Methods

```
object FactSeq {  
  
    def factSeq(n: Int): List[Long] = {  
        factSeqInner(n, List(1L), 2)  
    }  
  
    @tailrec  
    private def factSeqInner(n: Int, acc: List[Long], ct: Int): List[Long] = {  
        if (ct > n) acc else  
            factSeqInner(n, ct * acc.head :: acc, ct + 1)  
    }  
  
    FactSeq.factSeq(8)  
    // List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)  
}
```

- In, say, Java, implementation methods are hidden by making them private

# Nested Methods

- In Scala, we have another alternative: methods may be nested in other methods

```
object FactSeqNested {  
  
    def factSeq(n: Int): List[Long] = {  
        @tailrec  
        def factSeqInner(n: Int, acc: List[Long], ct: Int): List[Long] = {  
            if (ct > n) acc else  
                factSeqInner(n, ct * acc.head :: acc, ct + 1)  
        }  
  
        factSeqInner(n, List(1L), 2)  
    }  
}  
  
FactSeqNested.factSeq(8)  
// List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)
```

- Note that the `n: Int` in the `factSeqInner` hides the `n` from the outer `factSeq` method

# Nested Method Scoping

- Because `n: Int` is in scope throughout the `factSeq` method, we can drop it from the `factSeqInner` definition:

```
object FactSeqScoped {  
  
    def factSeq(n: Int): List[Long] = {  
        @tailrec  
        def factSeqInner(acc: List[Long], ct: Int): List[Long] = {  
            if (ct > n) acc else  
                factSeqInner(ct * acc.head :: acc, ct + 1)  
        }  
  
        factSeqInner(List(1L), 2)  
    }  
}  
  
FactSeqScoped.factSeq(8)  
// List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)
```

# Function Literals

- Nested methods are handy, but they still need to be named
- A function literal (or lambda) is just a function (like a method) that may not have a name
- From the point of view of the caller, the syntax is interchangeable, e.g.

```
def multiplyMethod(a: Int, b: Int): Int = a * b
// multiplyMethod[](val a: Int, val b: Int) => Int

val multiplyFunction: (Int, Int) => Int = (a, b) => a * b
// (Int, Int) => Int = $Lambda$1281/148080390@6a84a9dd

multiplyMethod(2, 3)
// res0: Int = 6

multiplyFunction(2, 3)
// res1: Int = 6
```

- Note how the method has a name `multiplyMethod` but the Lambda just has a type. We assign it to a value so that we do have a name for it, but that is not required to use a function literal, only to identify it.

# Using an Anonymous Function Literal

- E.g. if you call the `map` method on a list, there is no need to name the function passed:

```
val nums = (1 to 5).toList

nums.map(x => x * x)
// List[Int] = List(1, 4, 9, 16, 25)

nums.map(x => x * 3)
// List[Int] = List(1, 4, 9, 16, 25)

nums.map(x => x % 2 == 0)
// List[Boolean] = List(false, true, false, true, false)
```

- We use the `map` method for all three different functions. They are never assigned a name.
- Notice also that the third usage converts an `Int` to a `Boolean`, and the result of the `map` is then a `List[Boolean]`

# How Function Literals Work

- Although they use Java 8 Lambdas now, behind the scenes the details are the same as they have always been for Scala function literals

```
val fn1: (Int, Int) => Int = (a, b) => a + b

val fn2 = new Function2[Int, Int, Int] {
  override def apply(a: Int, b: Int) = a + b
}

fn1(2, 3) // 5
fn1.apply(2, 3) // 5
fn2(2, 3) // 5
fn2.apply(2, 3) // 5
```

- Scala calls the `apply` method on any object or instance followed immediately by parens
- Therefore if we make a class or instance that overrides `apply`, that will be invoked by a *function call*
- When you create a new instance of a function, that is called a *function value*

# Other Methods on Function

- In addition to an auto-generated `apply` method, when you define a function you also get:
- `.curried` (we'll look at currying a little later in the course)

```
val fn1curried = fn1.curried
fn1curried(2)(3) // 5
```

- `.tupled`

```
val fn1tupled = fn1.tupled
val tup = (2, 3)

// fn1(tup) // won't compile

fn1tupled(tup) // 5
```

# Higher Order Functions

e.g. map, filter, span, partition and more:

```
val nums = (1 to 10).toList

nums.map(x => x * x)
// List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

nums.filter(x => x < 4)
// List(1, 2, 3)

nums.span(x => x % 4 != 0)
// (List(1, 2, 3),List(4, 5, 6, 7, 8, 9, 10))

nums.partition(x => x % 4 != 0)
// (List(1, 2, 3, 5, 6, 7, 9, 10),List(4, 8))
```

- Higher order functions are just functions (or methods) that take or return other functions
- If a method or function does not take or return another function, it is called a *first order function*

# Writing a Higher Order Function

E.g. compare neighbors within a list using a function:

```
def compareNeighbors(xs: List[Int], compare: (Int, Int) => Int): List[Int] = {  
    for (pair <- xs.sliding(2)) yield {  
        compare(pair(0), pair(1))  
    }  
}.toList  
  
compareNeighbors(nums, (a, b) => a + b)  
// List(3, 5, 7, 9, 11, 13, 15, 17, 19)  
  
compareNeighbors(List(4, 1, 7, 3, 4, 8), (a, b) => (a - b).abs)  
// List(3, 6, 4, 1, 4)
```

- The `compare: (Int, Int) => Int` is syntactic sugar for `Function2[Int, Int, Int]` and is the idiomatic Scala way to write a function literal type

# Placeholder Syntax

- So far we have seen function definitions like `(a, b) => a + b` and `x => x * 2` but for very simple definitions, there is a syntactic shortcut

```
val nums = (1 to 10).toList

nums.filter(_ < 4)           // placeholder style for x => x < 4
nums.span(_ % 4 != 0)        // placeholder style for x => x % 4 != 0
nums.partition(_ % 4 != 0)

compareNeighbors(nums, _ + _) // placeholder style for (a, b) => a + b
```

- Placeholder can only be used where each parameter is used **exactly once in order**
- E.g. `_ * _` cannot be used instead of `x => x * x` as `x` is used twice
- The `_s` cannot be inside parens either (that means something different), so

```
_ - _ // can be substituted for (a, b) => a - b
// but
(_ - _).abs // cannot be substituted for (a, b) => (a - b).abs
```

# Placeholders With Types

- If you have a function definition like:

```
def compareNeighbors(xs: List[Int], compare: (Int, Int) => Int)
```

- When you call it, only `Int` params will compile, so Scala will infer those types, though we could also type the params explicitly:

```
compareNeighbors(nums, _ + _) // can infer types  
compareNeighbors(nums, (_: Int) + (_: Int)) // explicit types
```

- If you define a function literal where Scala has nothing to infer from, the types are mandatory:

```
val addPair = (_: Int) + (_: Int)  
compareNeighbors(nums, addPair)  
  
val addPair2 = (a: Int, b: Int) => a + b  
compareNeighbors(nums, addPair2)
```

- `val addPair = _ + _` and `val addPair2 = (a, b) => a + b` will not compile, since Scala does not have enough information to infer the types

# Partial Application of Functions

- Not to be confused with **Partial Functions** coming up shortly

```
val add3Nums = (a: Int, b: Int, c: Int) => a + b + c
// (Int, Int, Int) => Int = $Lambda$1701/51817638@2ecbf5a8

val add6and3 = add3Nums(6, _: Int, 3)
// Int => Int = $Lambda$1702/435985129@4db04cd7

add6and3(5) // 14
```

- The type is not optional on the placeholder in this case
- This also works with methods:

```
def add3Method(a: Int, b: Int, c: Int) = a + b + c
// add3Method[ ](val a: Int, val b: Int, val c: Int) => Int

val add4and7 = add3Method(4, _: Int, 7)
// Int => Int = $Lambda$1718/831478568@22e3cec7

add4and7(2) // 13
```

# You Can Partially Apply All the Parameters

```
def add3Method(a: Int, b: Int, c: Int) = a + b + c

val add3Functionv1 = add3Method(_, _, _)
add3Functionv1(1,2,3) // 6

val add3Functionv2 = add3Method _ // alternative when replacing all params
add3Functionv2(1,2,3) // 6
```

```
def compareTriplets(xs: List[Int], compare: (Int, Int, Int) => Int): List[Int] = {
  for (triplet <- xs.sliding(3)) yield {
    compare(triplet(0), triplet(1), triplet(2))
  }
}.toList

val nums = (1 to 10).toList

compareTriplets(nums, add3Functionv1)
// List(6, 9, 12, 15, 18, 21, 24, 27)
compareTriplets(nums, add3Functionv2)
// List(6, 9, 12, 15, 18, 21, 24, 27)
compareTriplets(nums, add3Method) // eta expansion
// List(6, 9, 12, 15, 18, 21, 24, 27)
```

# Closures

- All closures are function literals, but not all function literals are closures
- A closure is so-called because it encloses around some other state than that passed in to the function as parameters

```
val incBy1 = (x: Int) => x + 1           // not a closure
val more = 10
val incByMore = (x: Int) => x + more    // a closure

incBy1(10) // 11
incByMore(10) // 20
```

- In Scala, closures can be made over vars!

```
var more = 10
val incByMore = (x: Int) => x + more

incByMore(12) // 22
more = 100
incByMore(12) // 122
```

- This is confusing, and usually unintentional. Don't do that! Take a defensive `val` copy of any state before using it

# Partial Functions

- Not to be confused with *partially applied functions*
- A `PartialFunction[T, R]` extends `Function1[T, R]` (which is idiomatically written `T => R`)
- It can therefore be used in place of any `Function1[T, R]`
- Any block of code with `case` inside of {}s is a Partial Function:

```
val pf1: PartialFunction[Int, Int] = {  
    case x: Int if x > 0 => x + x  
    case x => x * -1  
}  
  
val fn1: Int => Int = pf1 // upcast  
  
val nums = (-5 to 5).toList  
  
nums.map(pf1)  
// List(5, 4, 3, 2, 1, 0, 2, 4, 6, 8, 10)
```

# Partial Functions

- In the previous example, the function was complete for all inputs, but it needn't be:

```
val pf2: PartialFunction[Int, Int] = {  
  case x: Int if x > 0 => x + x  
}  
  
nums.map(pf2) // MatchError!
```

- You get a `MatchError` thrown if there is no case to handle the input
- You can also ask `PartialFunctions` if they are defined for an input:

```
pf2.isDefinedAt(5) // true  
pf2.isDefinedAt(-5) // false
```

# Partial Functions

- `map` may not be safe with a `PartialFunction`, but `collect` is:

```
nums.map(pf1)

val pf2: PartialFunction[Int, Int] = {
  case x: Int if x > 0 => x + x
}

nums.map(pf2)      // MatchError!
nums.collect(pf2) // List(2, 4, 6, 8, 10)
```

- `match` and `catch` use `PartialFunctions`

```
a match {
  case 4 => "It's four!"
}

try (1 / 0)
catch {
  case ae: ArithmeticException => 0
}
```

# Var Args in Methods

- Ever wonder how:

```
val xs = List(1,2,3)
val ys = List(1,2,3,4,5)
```

works for different numbers of params?

- We know Scala re-writes to:

```
val xs = List.apply(1,2,3)
val ys = List.apply(1,2,3,4,5)
```

so are there just lost of overridden apply methods?

# Var Args

```
def sayHello(names: String*): Unit = {  
    for (name <- names) println(s"Hello, $name")  
}  
  
sayHello()  
sayHello("Fred")  
sayHello("Fred", "Julie", "Kim")  
  
def greet(greeting: String, names: String*): Seq[String] = {  
    for (name <- names) yield s"$greeting $name"  
}  
  
greet("Hi", "Fred", "Julie", "Kim")
```

- The var args parameter has a \* after it
- It must always be the last parameter
- The parameter comes in as `Array[T]` for a parameter defined `item: T*`

# Calling var args with a Collection

- What if we want to greet an existing collection of names?

```
// greet a seq of names:  
val names = List("Fred", "Julie", "Kim")  
  
greet("Hi", names) // does not compile
```

- For this, we use the **expansion operator**:

```
greet("Hi", names: _) // expansion operator  
// List(Hi Fred, Hi Julie, Hi Kim)
```

- Note that if using expansion operator, the original collection type is retained (in this case, List) instead of converting to Array
- The expansion operator is occasionally useful, particularly for recursion over var-args methods

# Named and Default Parameters

- All method parameters have names:

```
def greet(name: String): String =  
  "hello" + name // here we use the parameter name  
  
greet("Fred")
```

- We can also use that `name` outside of the method when we call it:

```
greet(name = "Fred")
```

- This is considered best practice for Boolean parameters in particular:

```
def thingy(isCold: Boolean, isBroken: Boolean): Unit = {}  
  
thingy(true, false) // doesn't tell us much  
thingy(isCold = true, isBroken = false) // is much more readable
```

- And if you use the names, you can choose any order:

```
thingy(isBroken = false, isCold = true) // exactly the same meaning as above
```

# Couple With Default Parameters

E.g.

```
def gravity = 9.81 // meters/sec

def force(mass: Double = 1, acceleration: Double = gravity) =
  mass * acceleration

force()      // 9.81
force(12)    // 117.72

force(acceleration = 2 * gravity) // 19.62
force(acceleration = gravity / 13.0, mass = 100) // 75.46153846153847
```

or for recursion:

```
def factSeq(n: Int, acc: List[Long] = List(1L), ct: Int = 2): List[Long] = {
  if (ct > n) acc else factSeq(n, acc = ct * acc.head :: acc, ct = ct + 1)
}
```

# Custom Control Structures

## Abstracting Control Structures With First Class Functions

# Agenda

1. Simplifying Code Using Higher Order Functions
2. Loans and Resource Management
3. Currying and Multiple Parameter Lists
4. Higher Order Functions
5. Function Arity
6. By-name Functions
7. Custom Looping

# Using the Contents of a File

```
import java.io.File
import scala.io.Source

def fileContainsQuestion(file: File): Boolean = {
    val source = Source.fromFile(file)

    try {
        source.getLines().toSeq.headOption.map { line =>
            line.trim.endsWith("?")
        }.getOrElse(false)
    } finally source.close()
}
```

```
def emphasizeFileContents(file: File): String = {
    val source = Source.fromFile(file)

    try {
        source.getLines().toSeq.headOption.map { line =>
            line.trim.toUpperCase
        }.getOrElse("")
    } finally source.close()
}
```

# Using Generics and HoFs

```
def withFileContents[A](file: File, fn: String => A, default: A): A = {  
    val source = Source.fromFile(file)  
  
    try {  
        source.getLines().toSeq.headOption.map { line =>  
            fn(line)  
        }.getOrElse(default)  
    } finally source.close()  
}
```

- **A** is a *type parameter* and can often be inferred
- We supply a function literal from `String => A` to the function
- We also supply a fallback default value of type `A`

# Calling the Generic Method

```
val hamlet = new File(fileLoc, "hamlet.shkspr")  
  
withFileContents(hamlet, _.trim.endsWith("?"), false)  
// false  
  
withFileContents(hamlet, _.trim.toUpperCase, "")  
// THE LADY DOTH PROTEST TOO MUCH, METHINKS.  
  
// something more complex?  
// find most common letter  
withFileContents(hamlet, { line =>  
    val letters = line.toLowerCase.filterNot(_ == ' ').toSeq  
    val grouped = letters.groupBy(identity)  
    grouped.maxBy { case (char, seq) => seq.length }._1  
}, 'e')  
// 't'
```

- It works, but that syntax is awkward to both write and read

# Currying Revisited

- From previous example

```
val add3: (Int, Int, Int) => Int = (a, b, c) => a + b + c
// (Int, Int, Int) => Int

val add3curried = add3.curried
// Int => (Int => (Int => Int))
```

- We could write the function this way ourselves:

```
val add3c: Int => Int => Int = a => b => c => a + b + c
// Int => (Int => (Int => Int))
```

- The parens are not required, but can clarify what's happening
- To call these curried functions:

```
add3(1,2,3)      // 6
add3curried(1)(2)(3) // 6
add3c(1)(2)(3)    // 6

add3c.apply(1).apply(2).apply(3) // 6
```

# Currying in Methods

- Scala methods can likewise be curried:

```
def add3method(a: Int)(b: Int)(c: Int) = a + b + c
add3method(1)(2)(3) // 6
```

- When a parameter list has one parameter, can swap {}s for ()s

```
add3method { 1 } { 2 } { 3 } // 6
```

- In parens, you can have commas, in curly braces, you get semi-colon inference
- This syntax trick is useful for cleaning up our generic implementation

# Curried Generic Loan

```

def withFileContents[A](file: File, default: A)(fn: String => A): A = {
  val source = Source.fromFile(file)

  try {
    source.getLines().toSeq.headOption.map { line =>
      fn(line)
    }.getOrElse(default)
  } finally source.close()
}

withFileContents(hamlet, false)(_.trim.endsWith("?")) // curried with parens
withFileContents(hamlet, "")(_.trim.toUpperCase)

// find most common letter
withFileContents(hamlet, 'e') { line => // curried with curlies
  val letters = line.toLowerCase.filterNot(_ == ' ').toSeq
  val grouped = letters.groupBy(identity)
  grouped.maxBy { case (char, seq) => seq.length }._1
}

```

- Often, function parameters are curried in a separate parameter list at the end of the method definition

# Function Arity

- Functions have an Arity, which means the number of input parameters

```
val sq: Int => Int = x => x * x // Function1[Int, Int]
val add: (Int, Int) => Int = (a, b) => a + b // Function2[Int, Int, Int]
val mult3: (Int, Int, Int) => Int = _ * _ * _ // Function3[Int, Int, Int, Int]
```

- There is also a Function0:

```
import scala.util.Random
val makeARandom: () => Double = () => Random.nextDouble()

makeARandom() // some double value
makeARandom() // some different double value
```

- The function takes no parameters, but is not evaluated until () is applied, and is evaluated each time an apply happens

# Writing Our Own Loop

```
import scala.annotation.tailrec

@tailrec
def fruitLoop(pred: () => Boolean)(body: () => Unit): Unit = {
  if (pred()) {
    body()
    fruitLoop(pred)(body)
  }
}

var x = 0

fruitLoop(() => x < 5) { () =>
  println(x * x)
  x += 1
}
```

- This looks kind of like a while loop, except for those `() =>` bits when we call it

# By-name Functions

- To provide nicer syntax at the call site, Scala has *by-name* functions as an alternative to Function0:

```
@tailrec
def fruityLoop(pred: => Boolean)(body: => Unit): Unit = {
  if (pred) {
    body
    fruityLoop(pred)(body)
  }
}

var y = 0
fruityLoop(y < 5) {
  println(y * y)
  y += 1
}
```

- We have now constructed a loop syntactically identical to while
- The by-name function is evaluated without ()s, "by-name" only (except if you call another method expecting a by-name)
- By-names are easy to get wrong, beware! Convert to Function0 ASAP

# Composition and Inheritance

Classes, Abstract Classes, extends, super, Overriding  
rules, final

# Agenda

1. Classes and Abstract Classes
2. Uniform Access, `val`, `lazy val` and `def`
3. Inheriting and `extends`
4. Invoking super-class methods and constructors
5. The `override` key word
6. `final` members and classes
7. `case` classes (mention)
8. Domain models

# Classes and Abstract Classes

- A class definition can have new instances created for it

```
class Person(name: String, age: Int) {  
    def isAdult: Boolean = age >= 21  
}  
  
val p1 = new Person("Dave", 18) // Person@b19efe7  
val p2 = new Person("Jill", 25) // Person@8bb2a08  
p1.isAdult // false  
p2.isAdult // true
```

- Because it is not marked `abstract`, you are able to create a new instance
- Also because it is not marked `abstract`, all fields and methods must have definitions
- When you call `new` in Scala, you **always** get a new instance

```
"hello".eq("hello")          // true  
new String("hello").eq(new String("hello")) // false
```

- `eq` is instance equality in Scala, while `==` always calls `.equals`

# Abstract Classes

- By contrast, you cannot call `new` on a class marked `abstract`

```
abstract class Car(make: String, model: String, year: Int) {  
    def isVintage: Boolean = LocalDate.now.getYear - year > 20  
}  
  
val mustang = new Car("Ford", "Mustang", 1965)  
// Error: class Car is abstract; cannot be instantiated  
// however  
val mustang = new Car("Ford", "Mustang", 1965) {} // mustang: Car = $anon$1@7...
```

- When you include an empty body, a new anonymous concrete class is created
- `abstract` classes can also have field and method definitions omitted:

```
abstract class Car(make: String, model: String, year: Int) {  
    def isVintage: Boolean  
}
```

# Anonymous Classes and Overrides

```
abstract class Car(make: String, model: String, year: Int) {
    def isVintage: Boolean
}

val mustang = new Car("Ford", "Mustang", 1965) {
    def isVintage = LocalDate.now.getYear - year > 20
} // Error: not found: value year
```

- The `year` field referenced in the anonymous class is `private[this]`
- We can make it parametric to get around that:

```
abstract class Car(
    val make: String,
    val model: String,
    val year: Int
) {
    def isVintage: Boolean
}

val mustang = new Car("Ford", "Mustang", 1965) {
    def isVintage = LocalDate.now.getYear - year > 20
}
```

# Uniform Access

- In this example, given that `year` is constant, `isVintage` is likely to be constant too

```
abstract class Car(  
    val make: String,  
    val model: String,  
    val year: Int  
) {  
    val isVintage: Boolean  
}  
  
val mustang = new Car("Ford", "Mustang", 1965) {  
    val isVintage = LocalDate.now.getYear - year > 20  
}
```

- A `val` may override a `def`, but not the other way around
- What happens as the date changes?
- May also use

```
lazy val isVintage = LocalDate.now.getYear - year > 20
```

# val, def, lazy val

```

class Demo {
    val a: Int = {
        println("evaluating a")
        10
    }
    def b: Int = {
        println("evaluating b")
        20
    }
    lazy val c: Int = {
        println("evaluating c")
        30
    }
}

val demo = new Demo // "evaluating a"
demo.a           // res0: Int = 10
demo.b           // "evaluating b" res1: Int = 20
demo.b           // "evaluating b" res2: Int = 20
demo.c           // "evaluating c" res3: Int = 30
demo.c           // res3: Int = 30

```

- `lazy val` calculates if/when first used, then memoizes

# Inheriting and Extends

- Classes extend other classes using the `extends` keyword:

```
abstract class Food {  
    def name: String  
}  
  
abstract class Fruit extends Food  
  
class Orange(val name: String) extends Fruit  
  
val jaffa = new Orange("Jaffa")
```

- `Fruit` must either be `abstract` or provide `name` definition
- `val name: String` parametric field in `Orange` provides `name` override
- New instances of `Orange` can be made, providing the `name` to the constructor

# Invoking Super-class Methods/Constructors

```

abstract class Vehicle(val name: String, val age: Int) {
    override def toString: String =
        s"$name, $age years old"
}

class Car(
    override val name: String,
    val make: String,
    val model: String,
    override val age: Int
) extends Vehicle(name, age) {

    override def toString: String =
        s"a $make $model, named ${super.toString}"
}

val mustang = new Car("Sally", "Ford", "Mustang", 50)
// mustang: Car = a Ford Mustang, named Sally, 50 years old

```

- Must `override` the `vals` from the super-class with the same name
- Constructor parameters are passed on through the `extends`
- `super` calls in methods call into the super-class

# An Alternative Way to Define Car

```
abstract class Vehicle(val name: String, val age: Int) {
  override def toString: String =
    s"$name, $age years old"
}

class Car(
  name: String,
  val make: String,
  val model: String,
  age: Int
) extends Vehicle(name, age) {

  override def toString: String =
    s"a $make $model, named ${super.toString}"
}

val mustang = new Car("Sally", "Ford", "Mustang", 50)
// mustang: Car = a Ford Mustang, named Sally, 50 years old
```

- If the `override val` feels weird, you can just make those field `private[this]`
- They will still be public because of the super-class definition
- But you can't make them `vals` in `Car` without an `override`

# override keyword

- If a `val` or `def` defines a field or method with the same parameter types **over** another of the same name, it must be marked with `override`
- If a `val` or `def` defines a field or method that does not override a superclass field or method with the same parameter types, it must **not** be marked `override`
- If a `val` or `def` defines a field or method with the same parameter types implementing a previously `abstract` field or method, it may or may not be marked `override`

# override keyword

```
abstract class Superclass {  
    def blip: String  
    val bloop: String = "bloop"  
    def op(x: Int, y: Int): Int  
}  
  
class Subclass extends Superclass {  
    override def blip: String = "blip" // override optional  
    override val bloop: String = "bloop" // must be override *and* val  
    override def op(x: Int, y: Int): Int = x + y // override optional  
    def op(x: Double, y: Double): Double = x + y // does not override anything  
}
```

- Have a play with the worksheet to familiarize yourself better with the rules

# final keyword

- In Scala, == is always aliased to call .equals
- It's tempting to try and override it to do something else:

```
class BadClass {  
    override def ==(other: Any): Boolean = {  
        println(s"Comparing $this to $other")  
        false  
    }  
}
```

- But if we try:

```
// Error: overriding method == in class Object of type (x$1: Any)Boolean;  
// method == cannot override final member  
//  override def ==(other: Any): Boolean = {  
//      ^
```

- Redefining the meaning of == would be a very bad idea, so it is marked **final** in AnyRef

# final keyword

- We can do the same with our own classes

```
class Authority {  
    final def theWord: String =  
        "This is the final word on the matter!"  
}  
  
class Argumentative extends Authority {  
    override def theWord: String =  
        "No, it's not!"  
}  
  
// Error: overriding method theWord in class Authority of type => String;  
// method theWord cannot override final member  
//   override def theWord: String =  
//           ^
```

# final classes

- A whole class can be marked final as well, e.g. Java's `String` class:

```
class BadString extends String

// Error: illegal inheritance from final class String
// class BadString extends String
//           ^
```

- And again, we can do this ourselves:

```
final class Infinity

class Beyond extends Infinity

// Error: illegal inheritance from final class Infinity
// class Beyond extends Infinity
//           ^
```

- Sorry Buzz Lightyear...

# case Classes (mention)

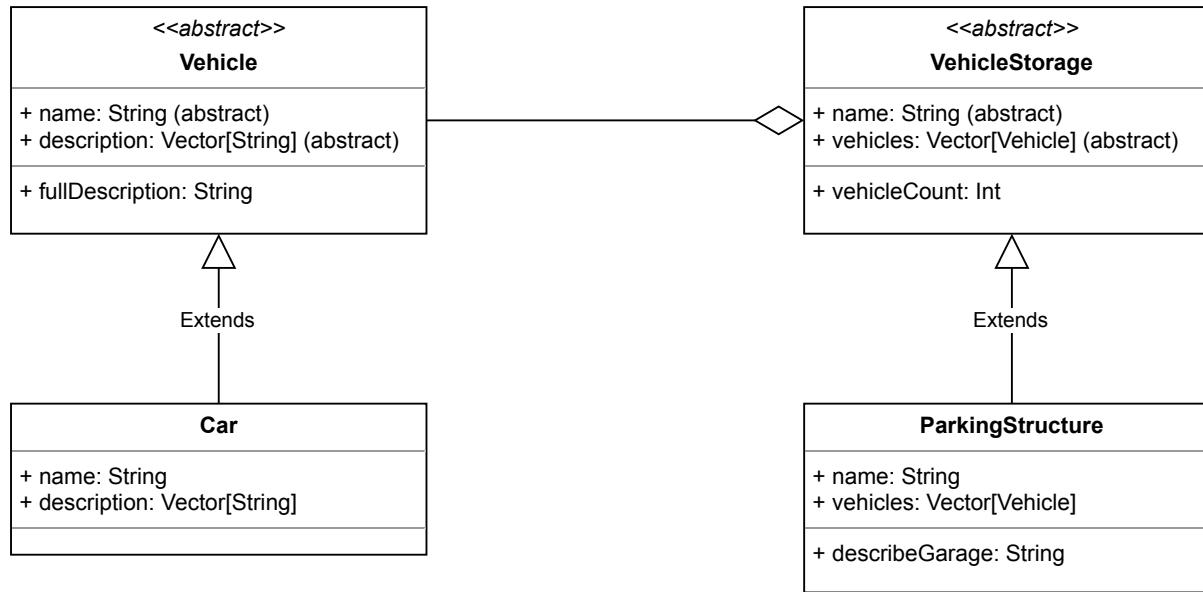
- We'll dig into case classes later in the course, but giving a quick look now:

```
case class Car(make: String, model: String, year: Int) {  
    lazy val isVintage: Boolean =  
        LocalDate.now.getYear - year > 20  
}  
  
val mustang = Car("Ford", "Mustang", 1965) // Car(Ford,Mustang,1965)  
  
mustang.make      // Ford  
mustang.model    // Mustang  
mustang.year     // 1965  
mustang.isVintage // true  
  
mustang == Car("Ford", "Mustang", 1965) // true  
mustang == Car("Ford", "Mustang", 1964) // false
```

- With the `case` class you get
  - Parametric immutable fields by default (no `val` needed)
  - A nice `toString` method
  - working `equals` and `hashCode`
  - Factory (`apply`) method (no `new` necessary)
  - More (to be seen later)

# Domain Models

- These are pure data abstraction modelling class definitions for a domain
- In Scala they can, and often do, have multiple classes in the same file
- Idiomatically they contain only value state and "pure" behavior directly related to the abstract model
- E.g. let's make a Domain Model for Cars and Parking Garages:



# Cars and Vehicles

```
abstract class Vehicle {  
    def name: String  
    def description: Vector[String]  
    override def toString: String = s"Vehicle($name)"  
  
    def fullDescription: String = {  
        (name +: description).mkString("\n")  
    }  
}  
  
case class Car(  
    name: String,  
    description: Vector[String] = Vector.empty  
) extends Vehicle  
  
val mustang = Car("Ford Mustang", Vector(  
    "1965 Mustang", "Metallic Blue", "302 ci V8"  
)  
) // Vehicle(Ford Mustang)  
  
val datsun = Car("Datsun 280Z", Vector(  
    "1982 Datsun 280Z", "Candy Apple Red", "2.8 Liter I6"  
)  
) // Vehicle(Datsun 280Z)  
  
mustang.fullDescription  
// Ford Mustang\n1965 Mustang\nMetallic Blue\n302 ci V8
```

# Parking Structure

```
abstract class VehicleStorage {
  def name: String
  def vehicles: Vector[Vehicle]

  def vehicleCount: Int = vehicles.size

  override def toString: String =
    s"$name with $vehicleCount vehicles"
}

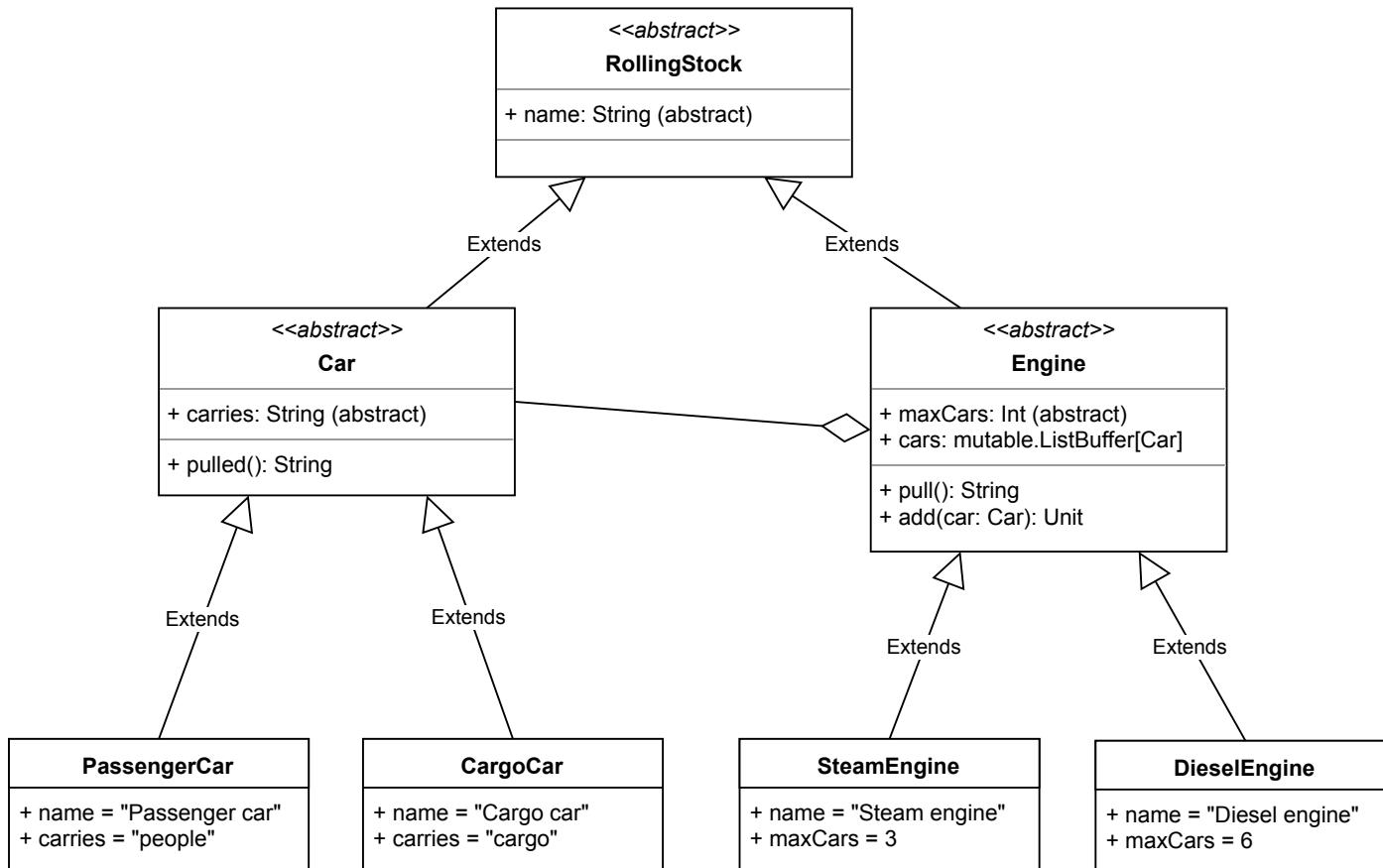
case class ParkingStructure(name: String,
  vehicles: Vector[Vehicle])
) extends VehicleStorage {
  def describeGarage: String = {
    val vehicleString = vehicles.mkString(", ")
    s"$name containing $vehicleString"
  }

  override def toString = describeGarage
}

val lot = ParkingStructure(
  "Parking garage",
  Vector(mustang, datsun)
)

lot.vehicleCount // Int = 2
```

# Now It's Your Turn:



# Hierarchy, Types and Options

Scala's Core Hierarchy, Type Calculus, Optionals and More.

# Agenda

1. Top Classes
2. Organization From the Top
3. Bottom Classes
4. Basic Type Calculus
5. Scala Primitives and Implicit Conversions
6. Value Classes and Extension Methods
7. Nil, Null, Nothing, None
8. Option Type
9. Equals and Hashcode
10. Product Types

# Top Classes

- OO Languages typically have an object at the top of the type hierarchy
  - e.g. Object in Java
- Scala has 3 such Top Classes:
  - Any is the absolute top of the hierarchy, everything in Scala is an Any (including instances)
  - AnyVal is under Any, but above all of the "primitive" types (+ Unit)
  - AnyRef is equivalent to Java's Object, it is above all user defined class types
- AnyRefs can be newed to make instances
- objects are also AnyRefs
- Any has methods: `toString`, `equals`, `hashCode`, `==`, `##`, `!=`
- AnyRef has methods: `isInstanceOf[T]`, `asInstanceOf[T]`, `eq`, `ne`, `synchronized`, `wait`, `notify`, `notifyAll`

# Top Types Example

```
val s: String = "hello"

val sa: Any = s      // OK
val sar: AnyRef = s // OK
val sav: AnyVal = s // Compile Error

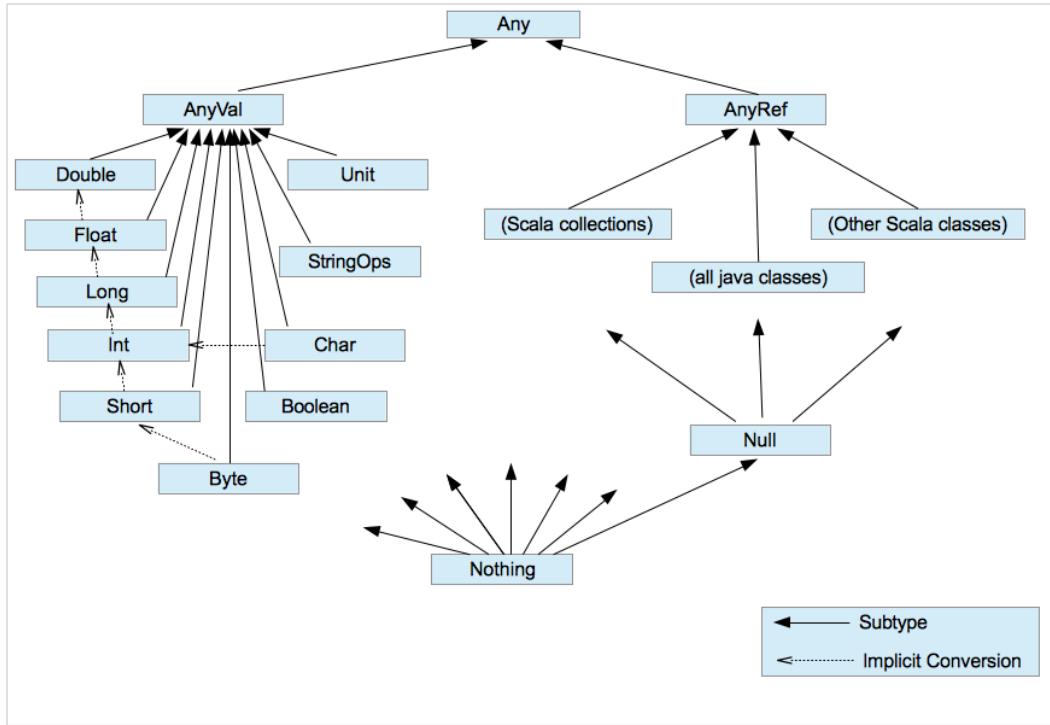
val i: Int = 10

val ia: Any = i      // OK
val iav: AnyVal = i // OK
val iar: AnyRef = i // Does not compile

ia.isInstanceOf[Int] // true
ia.asInstanceOf[Int] // Int: 10
ia.asInstanceOf[String]// Class cast exception

sa.isInstanceOf[String]// true
sa.asInstanceOf[String]// String: hello
sa.asInstanceOf[Int] // Class cast exception
```

# Organization From the Top



- The "primitives" and classes fan out from the top Any types
  - They also fan back in to Null and Nothing types, we'll look at those next

# Bottom Classes

- `null` is a concept familiar in many other languages
- It can be assigned to any reference type in those languages to denote the absence of a reference
- "It's as though there is a `Null` type that is a sub-class of all other types, with a single `null` instance" -- Paraphrased from the Java Language Spec
- In Scala this is literally the case, `Null` is a type, it is a sub-class of the whole set of `AnyRefs`, and there is a single instance `null` of that type.
- There is also a `Nothing` type which is the sub-type of everything in the Scala type system. There is **no instance** of `Nothing`, nor can there be one. It exists solely to complete the type system.

# Null and Nothing

```
val s1: String = "hello"  
s1.charAt(1) // Char: e  
  
val s2: String = null  
s2.charAt(1) // Null pointer exception!  
  
s1.isInstanceOf[String] // true  
s2.isInstanceOf[String] // false
```

- For `isInstanceOf` checks, `null` will always return `false`
- How can `Nothing` be useful?

```
val emptyList = List.empty // List[Nothing]  
  
1 :: emptyList           // List[Int] = List(1)  
"hello" :: emptyList     // List[String] = List(hello)
```

# Even More Nothing

- Is `Nothing` used for anything other than bottom type parameters?

```
def fail(msg: String): Nothing =  
    throw new IllegalStateException(msg)
```

- A method with a `Nothing` return type must throw an exception
- Why is `Nothing` useful?
  - It completes the type system...

# Scala Type Calculus (Simplified)

- if type A is a subtype of AnyRef, A + Null is A (because Null is a sub-type of all AnyRefs, so A becomes the LUB (Least Upper Bounds))
- For **any** type A in Scala, A + Nothing is A (because Nothing is a sub-type of everything so the same reasoning as above applies for the LUB)
- If B and C both sub-class A then B + C is A, that being the LUB (the first place the type-hierarchy converges for B and C as you go up through the super-classes)
- E.g. practical examples (using an if expression with two return types):

```

val flag = true // could be false...
if (flag) 1.0 else () // Double + Unit = AnyVal
if (flag) 1.0 // implicit Unit, Double + Unit = AnyVal
if (flag) "hi" // implicit Unit, String + Unit = Any
if (flag) "Hello" else null // String + Null = String
if (flag) 2.0 else null // Double + Null = Any
def fail(msg: String): Nothing =
  throw new IllegalStateException(msg)
if (flag) 2.0 else fail("not 2.0") // Double + Nothing = Double
if (flag) "yo" else fail("no yo") // String + Nothing = String

```

# Scala Type Inference Tricks

- Left to its own devices, Scala will often infer more than you need:

```
trait Fruit
case class Apple(name: String) extends Fruit
case class Orange(name: String) extends Fruit

if (true) Apple("Fiji") else Orange("Jaffa")
// Product with Serializable with Fruit = Apple(Fiji)

List(Apple("fiji"), Orange("Jaffa"))
// List[Product with Serializable with Fruit] = List(Apple(fiji), Orange(Jaffa))
```

- Since `case` classes extend both `Product` and `Serializable`
- You can explicitly type the result:

```
val result: Fruit = if (true) Apple("Fiji") else Orange("Jaffa")
```

- Or you can add `Product` and `Serializable` to the superclass:

```
trait Fruit extends Product with Serializable
```

# Primitives and Implicit Conversions

- Most languages have a form of Implicit Conversion, often referred to as *type coercion*
- E.g. in Java you can call a method expecting a `long` with an `int`
- Type widening of primitives in Scala is achieved with `implicits`
- And you can use them (carefully) for your own purposes as well
- Unlike Java, Scala makes no distinction in written code between primitives and boxed types
- Also while methods can be invoked like `1.+(2)` behind the scenes Scala implements these efficiently on primitive types
- In addition to implicit widenings, and implicit adapters for Java types, there are also "Rich" wrappers for primitive types and other common classes (like Strings)

# Rich Wrappers

```
val d1 = -1.5 // Double = -1.5
val d2 = 1.5 // Double = 1.5

d1.abs          // Double = 1.5
d1 min d2      // Double = -1.5
d1 max d2      // Double = 1.5

d1.floor        // Double = -2.0
d2.ceil         // Double = 2.0
d2.round        // Long = 2

val str = "hello" // String = hello
str.reverse      // String = olleh
str.toSeq         // Seq[Char] = hello
str.slice(2, 4)  // String = ll
```

- Double methods above come from `RichDouble` brought in by `Predef`
- String methods come from `SeqLike` and `WrappedString` also via `Predef`

# @specialized

- Autoboxing in Generics can introduce unwanted overhead
- @specialized can be used to generate templated specific versions for primitives:

```
def sumOf[@specialized(Int, Double, Long) T: Numeric](items: T*): T = {  
    val numeric = implicitly[Numeric[T]]  
    items.foldLeft(numeric.zero)(numeric.plus)  
}  
sumOf(1,2,3)
```

- In this case there will be one version of this method for AnyRefs + one each of Int, Double, and Long
- If you overuse this, you can end up with a type matrix explosion:

```
def pair[@specialized T, @specialized U](t: T, u: U): (T, U) = (t, u)
```

- Will produce 100 implementations of pair (9 primitives + AnyRef) \* (9 primitives + AnyRef)

# @specialized generation

```
$ javap Demo$.class
public final class Demo$ {
    public static Demo$ MODULE$;
    public static {};
    public <T, U> scala.Tuple2<T, U> pair(T, U);
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZZc$sp(boolean, bo
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZBc$sp(boolean, by
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZCc$sp(boolean, ch
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZDc$sp(boolean, do
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZFc$sp(boolean, fl
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZIc$sp(boolean, in
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZJc$sp(boolean, lo
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZSc$sp(boolean, sh
    public scala.Tuple2<java.lang.Object, scala.runtime.BoxedUnit> pair$mZVc$sp(bool
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mBZc$sp(byte, boole
    public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mBBc$sp(byte, byte)
...

```

- And it didn't even work - look at that return type

# Extension Methods and Implicit Classes

- RichDouble and WrappedString introduce extension methods on existing classes
- We can do that too:

```
implicit class TimesDo(i: Int) {  
    def times(fn: => Unit): Unit = {  
        for (_ <- 1 to i) fn  
    }  
}  
  
5 times {  
    println("hello")  
}
```

- `implicit classes` combine the class definition and an implicit conversion from the single type parameter to the class
- As such, because there is an `implicit def` included, implicit classes may only go inside another class, object or package object.

# Value class (extends AnyVal)

- The implicit def also means that the `Int` is actually wrapped in a new instance to run `times`
- adding `extends AnyVal` avoids this wrapping when possible (and also makes the definition work *only* outside of an enclosing class):

```
implicit class TimesDo(val i: Int) extends AnyVal {  
    def times(fn: => Unit): Unit = {  
        for (_ <- 1 to i) fn  
    }  
}  
  
5 times {  
    println("hello")  
}
```

- To extend `AnyVal` you must:
  - Have a single public parametric field
  - Have no other state in the class (because there may be no instance to hold the state)
- Behind the scenes, static methods are used

# Nil, Null, Nothing, None

- Scala has several "negative" types:
- `Nil` is the empty List, and is also the terminator of every `List`
- `Null` and its single instance `null` is the absence of an `AnyRef` instance
- `Nothing` is a type without an instance, and implies an exception being thrown
- `None` is the absence of a `Some` in the `Option` type, and is a safer alternative to `null`

# Option

- `null` is the absence of an instance. The compiler cannot help you:

```
val s1: String = "hello"    // compiles
val s2: String = null      // likewise

s1.length                  // compiles, gives 5
s2.length                  // compiles, NullPointerException
```

- Making an `Option` type means the compiler has your back:

```
val os1: Option[String] = Some("hello")
val os2: Option[String] = None

os1.length    // will not compile

os1.map(_.length) // Some(5)
os2.map(_.length) // None
```

- This takes a common runtime error, and moves it into a compile time concern

# Working with Option

- Option is supported throughout the core libraries and third party APIs

```
val numWords = Map(1 -> "one", 2 -> "two", 3 -> "three")
numWords(1) // one
numWords(4) // NoSuchElementException
val word1 = numWords.get(1) // Some("one")
val word2 = numWords.get(4) // None
```

- You can pattern match:

```
word1 match {
  case Some(word) => word
  case None => "unknown"
} // one
```

- Use `getOrElse`:

```
word2.getOrElse("unknown") // "unknown"
```

# Working with Option

- Or compose options with `for` expressions:

```
def fourthLetter(i: Int): Option[Char] = for {
    word <- numWords.get(i)
    char <- word.drop(4).headOption
} yield char
fourthLetter(1) // None
fourthLetter(3) // Some(e)
```

- Mixing Options and Collections:

```
def fourthLetters(nums: Seq[Int]): Seq[Char] = for {
    i <- nums
    word <- numWords.get(i).toSeq
    char <- word.drop(4).headOption.toSeq
} yield char
fourthLetters(List(1, 2, 3)) // List('e')
```

- In this case `toSeq` on the options is not required, but is recommended
- Collection following an option in a `for` expression needs `toSeq`

# equals and hashCode

- In Scala, == is aliased to call .equals and is marked final
- The equals and hashCode contract are hard to get right and defaults are not useful

```
class Person(val first: String, val last: String, val age: Int)

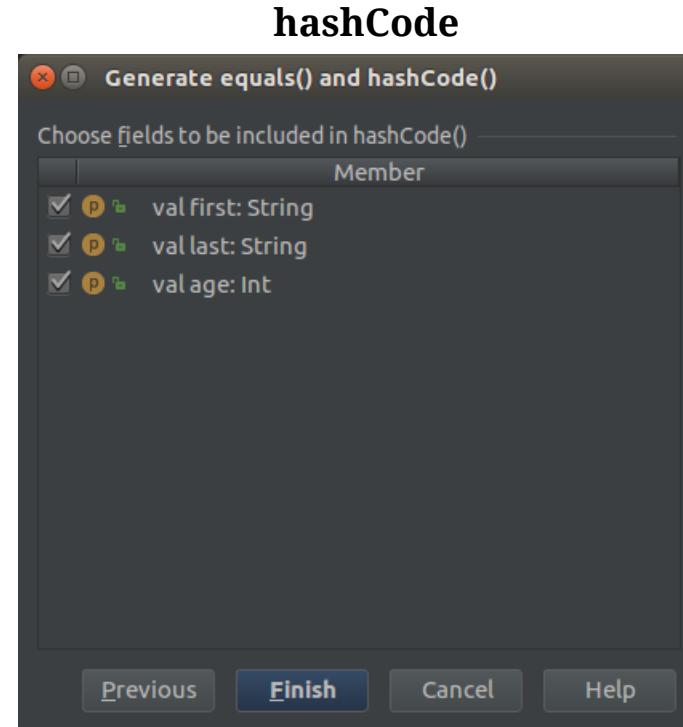
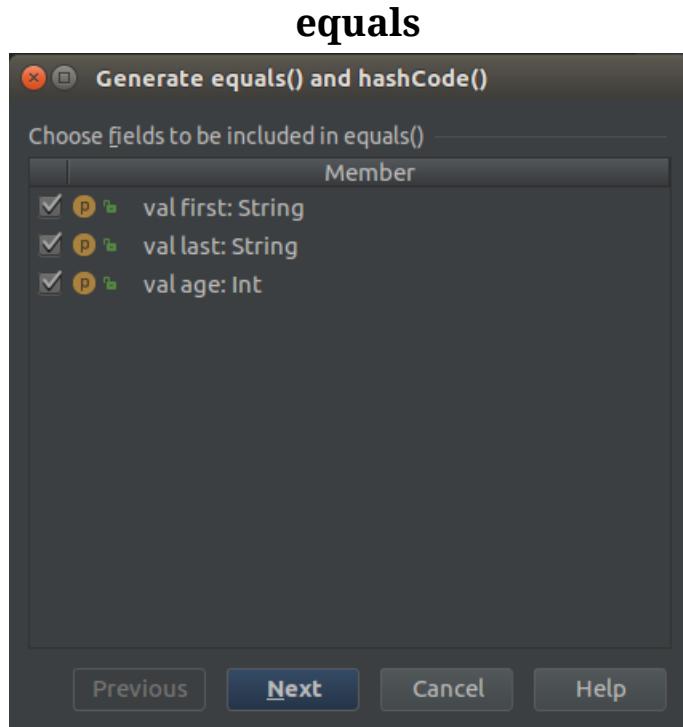
val p1 = new Person("Wavey", "Davey", 25)
val p2 = new Person("Wavey", "Davey", 25)

p1 == p2    // false
p1.##      // 1466295063
p2.##      // 405671158
```

- So we will usually need to provide better equals/hashCode if we want to use these in collections, etc.

# Option 1, Generate with IDEA

- Alt+insert, equals and hashCode, Select fields and Done



# Option 2, follow this formula:

```
class Fruit(val name: String) {  
  
    def canEqual(other: Any): Boolean = other.isInstanceOf[Fruit]  
  
    override def equals(other: Any): Boolean = other match {  
        case that: Fruit =>  
            (that canEqual this) &&  
            name == that.name  
        case _ => false  
    }  
  
    override def hashCode(): Int = name.hashCode  
}
```

- Simple case, no inheritance, only one field for hashCode

# Continued... sub-classes

```
class Apple(val brand: String, val color: String) extends Fruit("apple") {  
  
    override def canEqual(other: Any): Boolean = other.isInstanceOf[Apple]  
  
    override def equals(other: Any): Boolean = other match {  
        case that: Apple =>  
            super.equals(that) &&  
            (that canEqual this) &&  
            brand == that.brand &&  
            color == that.color  
        case _ => false  
    }  
  
    override def hashCode(): Int = {  
        41 * (  
            41 * (  
                41 + super.hashCode  
            ) + brand.hashCode  
        ) + color.hashCode  
    }  
}
```

- Remember `super` for both `equals` and `hashCode`

# Option 3, just use case classes

```
case class Banana(brand: String, ripe: Boolean) extends Fruit("banana")

val b1 = Banana("Ffyfes", true)
val b2 = Banana("Ffyfes", true)

b1 == b2    // true
b1.##      // -1396355227
b2.##      // -1396355227
```

- But!
  - Superclass `equals` must still be correct if present (will not be used if omitted)
  - `case classes` cannot extend other `case classes`
  - Technically all `case classes` should be marked `final`

# Product Types

- case classes and tuples are Product types
- Their equality is determined by their public state and (for case classes) their type

```

1, '2', "three") == (1, '2', "three")           // true
(1, 'a', "three") == (1, '2', "three")          // false

case class Triplet1(i: Int, ch: Char, str: String)

Triplet1(1, '2', "three") == Triplet1(1, '2', "three") // true
Triplet1(1, '2', "three") == Triplet1(2, '2', "three") // false
Triplet1(1, '2', "three") == (1, '2', "three")        // false

case class Triplet2(i: Int, ch: Char, str: String)

Triplet2(1, '2', "three") == Triplet2(1, '2', "three") // true
Triplet2(1, '2', "three") == Triplet2(2, '2', "three") // false
Triplet1(1, '2', "three") == Triplet2(1, '2', "three") // false
Triplet2(1, '2', "three") == (1, '2', "three")        // false

```

# Product Type Features

```
val triplet2 = Triplet2(1, '2', "three")

triplet2.productIterator.toList    // List[Any] = List(1, 2, three)
triplet2.productArity            // Int = 3
triplet2.productElement(2)        // Any = three
triplet2.productPrefix           // String = Triplet2

val tup3 = (1, '2', "three")

tup3.productIterator.toList      // List[Any] = List(1, 2, three)
tup3.productArity              // Int = 3
tup3.productElement(2)          // Any = three
tup3.productPrefix              // String = Tuple3
```

- Much more on `case classes` in a later module

# Exercises for Module 8

- Find the `Module08` class and follow the instructions to make the tests pass

# Traits

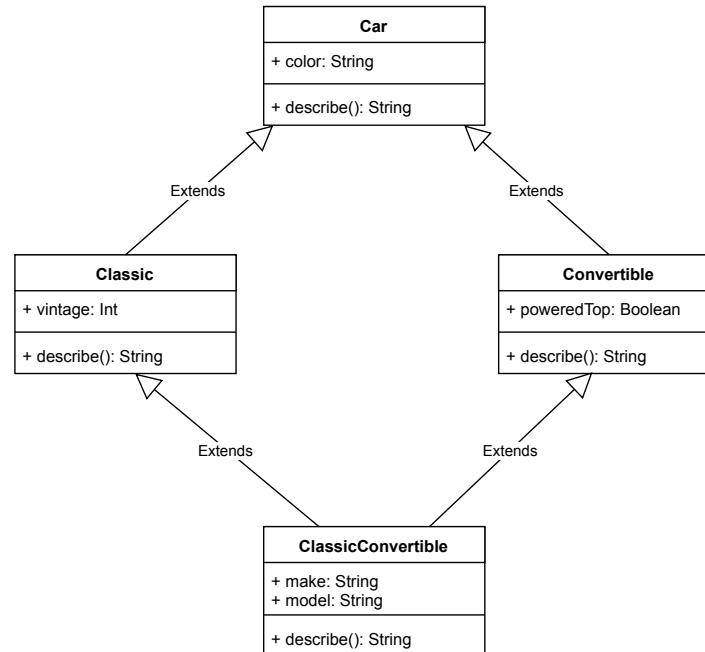
## Multiple Inheritance Without the Problems

# Agenda

1. Traits Compared To Interfaces
2. Creating a Trait
3. Multiple Traits
4. Linearization
5. Stacking Traits
6. Traits vs Classes
7. Trait Initialization
8. Traits with Type Parameters
9. Selfless Traits

# Multiple Inheritance

- The Diamond Inheritance Problem - how does `describe()` visit all instances?



# Traits Compared to Interfaces

- Java (and others) sidestep this problem using interfaces
- A class has a single superclass, and multiple interfaces
- Interfaces cannot have state or behavior, only abstract method definitions
- Therefore you cannot accidentally skip behavior, but you also are more limited in where behavior can be inherited from
- In Scala, traits are like interfaces (indeed pure abstract traits **are** Java interfaces)
- But they have been extended to include state and behavior
- A class still has a single super-class, but may have multiple traits mixed in as well
- The diamond inheritance problem is tackled in a new, clever way

# Creating a Trait

```
trait Car {  
    def color: String  
    def describe: String = s"$color car"  
}
```

- Like a `class` definition but using `trait` keyword instead
- Cannot take constructor parameters
- But can have abstract vals and defs
- Can also have real behavior and state (e.g. `describe` could be a `def` or `val`)
- Like an abstract class, you cannot make a new instance unless you supply a body

```
val mustang = new Car {  
    val color = "red"  
} // Car{val color: String} = $anon$1@5baf4194  
  
mustang.describe // red car
```

# Using a Trait in a Class

```
class ActualCar(val color: String, val name: String) extends Car  
  
val modelT = new ActualCar("black", "Model T")  
  
modelT.describe // black car
```

- You can extend a trait like a superclass, for syntactic convenience
- In fact, all traits have a single superclass as well, by default AnyRef
- When you use `extends` for a trait you are really extending the trait superclass and mixing in the trait. E.g. the above is really:

```
class ActualCar(val color: String, val name: String) extends AnyRef with Car
```

- Only a trait can go after the `with` keyword, not a class

# Polymorphism and Rich Interfaces

- Can still use a trait like an interface to give us polymorphism:

```
val car: Car = modelT  
car.describe // black car
```

- We care, because we get free stuff - implement a little, get a lot
- E.g. Function1

```
class Demo extends Car with Function1[String, String] {  
    override def color = "red"  
    override def apply(v1: String): String = s"$v1 $color"  
}  
  
val demo = new Demo  
demo("cherry") // cherry red  
  
val descriptionLength = demo.andThen(_.length)  
descriptionLength("cherry") // 10
```

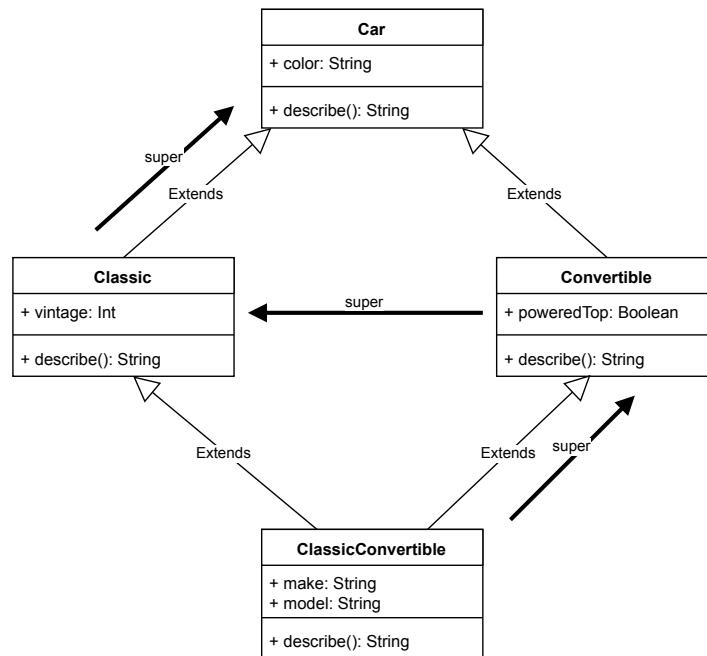
- andThen is a method we get for free from Function1
- <https://www.scala-lang.org/api/2.12.4/scala/collection/Traversable.html>

# Multiple Traits

```
abstract class Car {  
    def color: String  
    def describe: String = s"$color"  
    override def toString = s"$describe car"  
}  
  
trait Classic extends Car {  
    def vintage: Int  
    override def describe: String =  
        s"vintage $vintage ${super.describe}"  
}  
  
trait Convertible extends Car {  
    def poweredTop: Boolean  
    override def describe: String = {  
        val top = if (poweredTop)  
            "powered convertible" else "convertible"  
        s"$top ${super.describe}"  
    }  
}  
  
class ClassicConvertible(  
    val color: String, val vintage: Int, val poweredTop: Boolean  
) extends Car with Classic with Convertible  
  
val mustang = new ClassicConvertible("red", 1965, false)  
// mustang: ClassicConvertible = convertible vintage 1965 red car
```

# How'd it do that?

- The `super` is not decided until the `trait` is mixed in to a concrete class
- This is called *linearization*



# Stacking Traits

```
abstract class Car {  
    def color: String  
    def describe: String = s"$color"  
    override def toString = s"$describe car"  
}  
  
trait Classic extends Car {  
    def vintage: Int  
    override def describe: String =  
        s"vintage $vintage ${super.describe}"  
}  
  
trait Convertible extends Car {  
    override def describe: String =  
        s"convertible ${super.describe}"  
}  
  
trait PoweredConvertible extends Convertible {  
    override def describe: String =  
        s"powered ${super.describe}"  
}  
  
trait HardtopConvertible extends Convertible {  
    override def describe: String =  
        s"hard-top ${super.describe}"  
}
```

# Stacking Traits - Quiz

- What do the following `toStrings` output?

```
class ClassicConvertible1(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with Classic with HardtopConvertible

new ClassicConvertible1("red", 1965)
```

```
class ClassicConvertible2(val color: String, val vintage: Int)
  extends Car with Classic with PoweredConvertible with HardtopConvertible

new ClassicConvertible2("red", 1965)
```

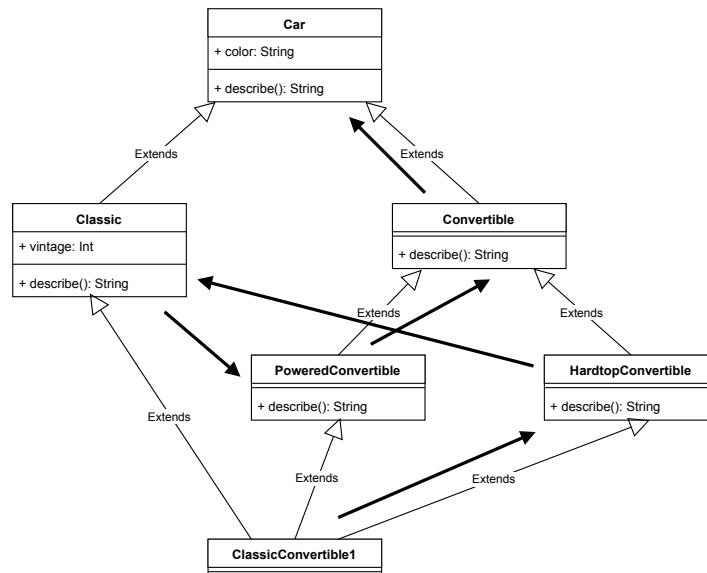
```
class ClassicConvertible3(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with HardtopConvertible with Classic

new ClassicConvertible3("red", 1965)
```

# Stacking Traits - 1

```
class ClassicConvertible1(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with Classic with HardtopConvertible

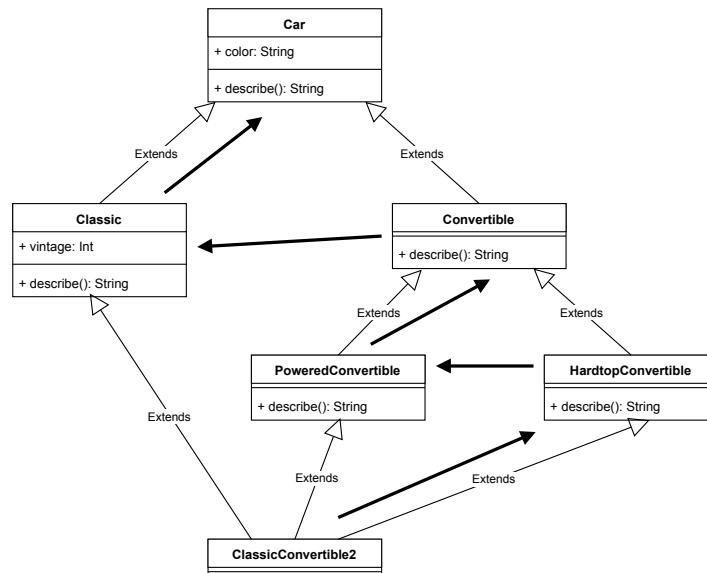
new ClassicConvertible1("red", 1965)
// hard-top vintage 1965 powered convertible red car
```



# Stacking Traits - 2

```
class ClassicConvertible2(val color: String, val vintage: Int)
  extends Car with Classic with PoweredConvertible with HardtopConvertible

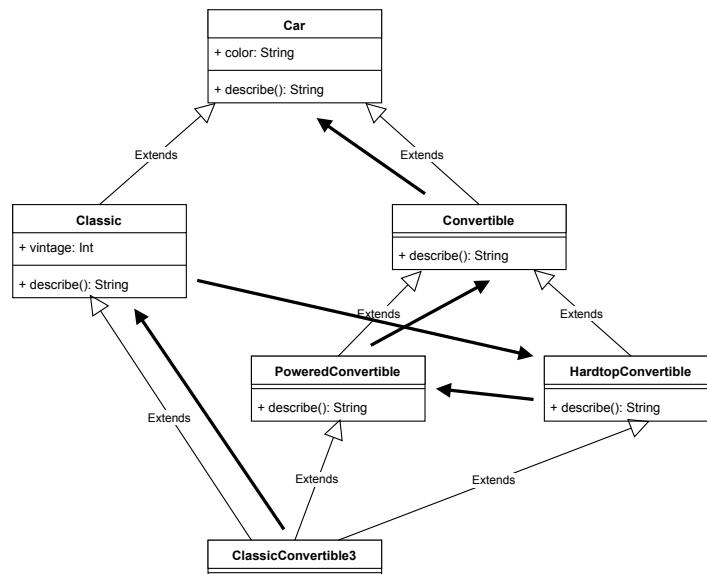
new ClassicConvertible2("red", 1965)
// hard-top powered convertible vintage 1965 red car
```



# Stacking Traits - 3

```
class ClassicConvertible3(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with HardtopConvertible with Classic

new ClassicConvertible3("red", 1965)
// vintage 1965 hard-top powered convertible red car
```



# Construction Composition

- You can also include traits as you are creating a new instance of a class (just in time)
- This has the effect of introducing a new anonymous sub-class and creating one instance of it immediately, which is given back to you

```
class ClassicCar(val color: String, val vintage: Int) extends Car with Classic
val ccc =
  new ClassicCar("red", 1965) with PoweredConvertible with HardtopConvertible
ccc.describe
// res3: String = hard-top powered convertible vintage 1965 red
```

# Traits vs Classes

- Classes (including abstract classes) can have constructor parameters, traits cannot
- This also means traits cannot have implicit parameters or context bounds
- This may change in the future

```
class CoordsC(val x: Double, val y: Double) {
  override def toString: String = s"($x, $y)"
  val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}

val c1 = new CoordsC(3.0, 4.0)  // CoordsC = (3.0, 4.0)
c1.distToOrigin                // Double = 5.0
```

```
// will not compile, traits can't have constructor params
//trait CoordsT(x: Double, y: Double)
// can use abstract vals instead
trait CoordsT {
  val x: Double
  val y: Double
  override def toString: String = s"($x, $y)"
  val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}
```

# Trait Initialization

```
case class Coords(x: Double, y: Double) extends CoordsT

val c2 = Coords(3.0, 4.0)    // Coords = (3.0, 4.0)
c2.distToOrigin             // Double = 5.0
```

- So far so good, but

```
val c3 = new CoordsT {
  val x: Double = 3.0
  val y: Double = 4.0
} // CoordsT = (3.0, 4.0)

c3.distToOrigin // Double = 0.0
```

- Huh!?
- x and y are not set to values until **after** distToOrigin has been calculated in the second code snippet

# Trait Initialization

- Fixing the problem: Option 1 - early initializers

```
val c4 = new {
    val x: Double = 3.0
    val y: Double = 4.0
} with CoordsT // CoordsT = (3.0, 4.0)
c4.distToOrigin // Double = 5.0
```

- Option 2 - use `lazy val` in the trait (recommended)

```
trait CoordsT {
    val x: Double
    val y: Double
    override def toString: String = s"($x, $y)"
    lazy val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}

val c3 = new CoordsT {
    val x: Double = 3.0
    val y: Double = 4.0
} // CoordsT = (3.0, 4.0)
c3.distToOrigin // Double = 5.0
```

- When defining a trait make `any val` computed from others `lazy`

# abstract override

- You can override a method in a trait that may be abstract in the superclass, using `abstract override`
- Some other trait must supply a non-abstract implementation in a concrete definition

```
abstract class Vehicle {  
    def describe: String // abstract describe  
    override def toString = s"$describe"  
}  
  
trait Classic extends Vehicle {  
    def vintage: Int  
    abstract override def describe: String =  
        s"vintage $vintage ${super.describe}"  
}  
  
trait Convertible extends Vehicle {  
    def poweredTop: Boolean  
    abstract override def describe: String = {  
        val top = if (poweredTop)  
            "powered convertible" else "convertible"  
        s"$top ${super.describe}"  
    }  
}
```

# Implementing the Abstract

```
trait Car extends Vehicle {  
    def color: String  
    def describe: String = s"$color car" // the actual implementation  
}  
  
class ClassicConvertible(  
    val color: String, val vintage: Int, val poweredTop: Boolean  
) extends Car with Classic with Convertible  
  
val mustang = new ClassicConvertible("red", 1965, false)  
// mustang: ClassicConvertible = convertible vintage 1965 red car
```

- Scala will tell you when you get it wrong:

```
Error:(17, 21) method describe in class Vehicle is accessed from super. It may not  
be abstract unless it is overridden by a member declared `abstract' and `override'  
  s"$top ${super.describe}"  
  ^
```

- There's no magic, someone has to fill in the implementation eventually

# Traits with Type Parameters

- Traits can have type parameters:

```
trait CompareAge[T] {  
    def older(item: T): T  
}  
  
def getOlder[T <: CompareAge[T]](item1: T, item2: T): T = {  
    item1 older item2  
}
```

```
case class VintageCar(make: String, model: String, year: Int)  
extends CompareAge[VintageCar] {  
  
    def older(other: VintageCar): VintageCar =  
        if (this.year < other.year) this else other  
}  
  
getOlder(  
    VintageCar("Ford", "Mustang", 1965),  
    VintageCar("Ford", "Model T", 1922))  
// VintageCar(Ford,Model T,1922)
```

# Another CompareAge class

```
case class Person(name: String, age: Int) extends CompareAge[Person] {
  override def older(other: Person) =
    if (other.age > this.age) other else this
}

getOlder(Person("Fred", 25), Person("Jill", 28))
/// Person(Jill,28)
```

- This is used in the Scala core libraries, e.g. Ordering

```
val people = List(Person("Fred", 25), Person("Jill", 28), Person("Sally", 22))

people.sorted // Error: No implicit Ordering defined for Person

implicit object PersonOrdering extends Ordering[Person] {
  override def compare(x: Person, y: Person) = x.age - y.age
}

people.sorted
// List(Person(Sally,22), Person(Fred,25), Person(Jill,28))
```

- A trait with a single type parameter is often referred to as a *type class*. A widely used pattern in Scala.

# Selfless Traits

- Choose trait mixin or import

```
trait Logging {  
    def error(msg: String): Unit = println(s"Error: $msg")  
    def info(msg: String): Unit = println(s"Info: $msg")  
}  
object Logging extends Logging  
  
class Process1 extends Logging {  
    def doIt(): Unit = {  
        info("Checking the cell structure")  
        error("It's all gone pear shaped")  
    }  
}  
val p1 = new Process1  
p1.doIt()  
  
class Process2 {  
    import Logging._  
  
    def doIt(): Unit = {  
        info("Checking the cell structure")  
        error("It's all gone pear shaped")  
    }  
}  
val p2 = new Process2  
p2.doIt()
```

# Exercises for Module 9

- Find the `Module09` class and follow the instructions to make the tests pass

# Packages, Imports and Scope

## Controlling Visibility and Access in Scala

# Agenda

1. Public, Protected and Private
2. Packages
3. Package Visibility
4. Imports from Packages
5. Package Objects
6. Imports from Objects and Instances
7. Importing Fu
8. Companion Objects

# Public, Protected and Private

- Scala has three visibility scopes, `private`, `protected` and `public`.
- There is no `public` keyword: classes, traits, objects, vals and defs not marked `private` or `protected` are public by default.
- `private` means only available to the current class, companion object, or inner classes/objects in the current class.
- `protected` means `private` access plus availability to any sub-class of the current class.
- `public` means anyone can see and access the item. This is the default when no other visibility is specified.
- Java's *package protected* default is not the default in Scala, and package access is handled through an orthogonal mechanism.
- Scala also has a `private[this]` visibility which means private to this specific instance only, companions, nested classes and even other instances of the same class cannot access a `private[this]`.
- `private[this]` is the default visibility for constructor parameters that are not parametric-fields (i.e. not in a case class or marked with a `val`).

# Packages

- Packages are a way of organizing classes and objects into different logical access units
- The standard way packages are used in Scala is the same way as Java, a package declaration at the start of a source file (and everything in that source file is then in that package):

```
package demo.food.domain.api

trait Dessert

case class IceCream(flavor: String) extends Dessert
case class Jello(color: String) extends Dessert
```

# Package Structure Alternatives

- Scala also supports C# namespace style

```
package demo {  
    package food {  
        package domain {  
            package api {  
                trait Dessert  
                case class IceCream(flavor: String) extends Dessert  
                case class Jello(color: String) extends Dessert  
            }  
        }  
    }  
}
```

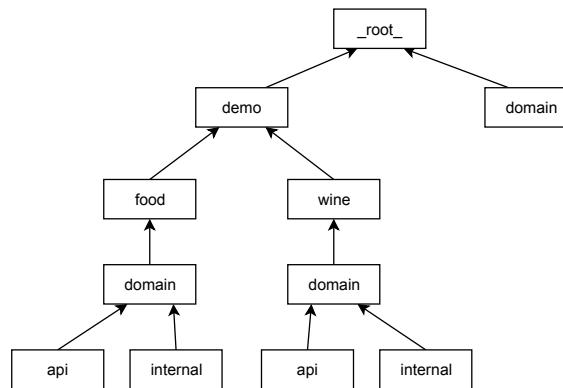
- or in namespace shorthand:

```
package demo.food.domain.api {  
    trait Dessert  
    case class IceCream(flavor: String) extends Dessert  
    case class Jello(color: String) extends Dessert  
}
```

- In practice, very few developers use this form, but you may occasionally see it.

# Namespace Notation

- Although not widely used, namespace notation does illustrate better the scoping
- It also looks like another object or class top level definition, and it is close, though only traits, classes and objects can go directly under a package
- Namespace notation also allows multiple different packages in a single source file (again, uncommon in Scala)
- Seeing the structures above will help explain some of the remainder of this module



# More Parts of the Model

```
package demo.food.domain.api

import demo.food.domain.allFoods // more on this in a moment

object FindFood {
  def lookupFood(name: String): Option[Dessert] = allFoods.lookupDessert(name)
}
```

```
package demo.food.domain.internal

import demo.food.domain.api.Dessert

import scala.collection.mutable

private[domain] class FoodDB {
  private[this] val desserts = mutable.Map.empty[String, Dessert]

  private[domain] def addDessert(name: String, dessert: Dessert): Unit =
    desserts.put(name, dessert)

  def lookupDessert(name: String): Option[Dessert] = desserts.get(name)
}
```

- `private[domain]` makes this private but accessible up to `demo.food.domain`
- `private[this]` on `desserts` means only this specific instance can access it.

# Package Visibility

- `private` and `protected` cover the visibility within the instance and subclasses
- Package scoping is handled with `[super-package]` after one of those keywords
- The super-package must be one of the parent packages or you will get a compile error
- This allows you to control visibility but still have freedom in organizing sub-packages
- Note that this is a Scala language specific feature, until compatibility with Java super-packages is implemented, package-scoped items are public when accessed from Java!
- `private[this]` is private to the specific instance. It cannot be accessed even from the companion object or another instance of this class. It is the only visibility modifier in Scala that does not have accessor methods created.

# Logger in the top level domain package:

```
package domain

object Logger {
    def log(msg: String): Unit = println(msg)
}
```

- Normally we would attempt to not have three different domain packages located at different parts of the hierarchy because it can make things confusing, but that is the point in this case - to demonstrate disambiguation.

# Now for the wine package

```
package demo.wine.domain.api

case class Wine(name: String, vintage: Int, varietal: String)
```

```
package demo.wine.domain.internal

import demo.food.domain.api.{Dessert, IceCream}
import demo.wine.domain.api.Wine

private[domain] object PairingDB {
  private[this] val winePairs = Map[Dessert, Wine](
    IceCream("Vanilla") -> Wine("Solis Old Vine Zin", 2014, "Zinfandel"))

  def pairWineWithDessert(dessert: Dessert): Option[Wine] = winePairs.get(dessert)
}
```

- Note the `demo.food.domain.api.{Dessert, IceCream}` to bring in both classes

# The PairWine object

```
package demo.wine.domain.api

import demo.food._
import domain.api.Dessert
import demo.wine.domain.internal.PairingDB
import _root_.domain.Logger

object PairWine {
  def pairWineWithDessert(dessert: Dessert): Option[Wine] = {
    Logger.log(s"Attempting to pair $dessert")
    PairingDB.pairWineWithDessert(dessert)
  }
}
```

- `import demo.food._` brings in the `domain` package from there
- `import domain.api.Dessert` references that `domain` from `demo.food`
- `import domain.internal.PairingDB` will not work, since that thinks you now mean starting from `domain` in `demo.food`. We have to use `import demo.wine.domain.internal.PairingDB`
- To import `domain.Logger`, we need to use `import _root_.` to force Scala to go back to the root of the package hierarchy.

# Package Objects

- Remember `import demo.food.domain.allFoods` ?
- In the JVM, methods and fields must be inside classes
- Unlike the REPL, defs and vals can't just be defined without an enclosing `class`, `trait` or `object`, unless you use a `package object`

```
package demo.food

import demo.food.domain.api.IceCream
import demo.food.domain.internal.FoodDB

package object domain {
    lazy val allFoods: FoodDB = {
        val foodDB = new FoodDB
        foodDB.addDessert("vanilla ice cream", IceCream("Vanilla"))
        foodDB
    }
}
```

- Can now `import demo.food.domain.allFoods` or `import demo.food.domain._` will bring the `allFoods` instance in as well
- package objects can be a useful place to put `implicits` as well.

# Importing from an object

- Scala can import anything, from anywhere, at any point in your code
- Importing from a package is the most common, but import from objects and instances is also useful
- E.g. to import the `log` method from `Logger` directly:

```
import demo.food._  
import domain.api.Dessert  
import demo.wine.domain.internal.PairingDB  
  
object PairWine {  
    import _root_.domain.Logger.log  
    def pairWineWithDessert(dessert: Dessert): Option[Wine] = {  
        log(s"Attempting to pair $dessert")  
        PairingDB.pairWineWithDessert(dessert)  
    }  
}
```

- Note that the `import` is within the `PairWine` object body, `log` is only in scope inside `PairWine`
- You can import and scope to a single method, or even an arbitrary code block in `{}`s if desired

# Importing from an instance

```
import demo.food.domain.api._

val iceCream = IceCream("Vanilla")

def thirdLetterOfDessert(dessert: IceCream): Char = {
    import dessert._
    import flavor._

    charAt(3)
}

thirdLetterOfDessert(iceCream) // Char = i
```

- We import the members of `dessert`, then the members of the `flavor` string
- `charAt(3)` then refers to the "Vanilla" string, resulting in `i`
- These imports are only in scope within the `thirdLetterOfDessert` method

# Importing Fu: Renaming

```
// import the demo.food.domain package and rename it fooodomain
import demo.food.{domain => fooodomain}

// import the demo.wine.domain package and rename it winedomain
import demo.wine.{domain => winedomain}

// Now there is no confusion referring to the top domain for Logger
import domain.Logger

Logger.log("happy")

import fooodomain.api.{Jello => Jelly, _}
import winedomain.api._
IceCream("Vanilla")
Jelly("green")
Jello("red") // compile error - not found: value Jello
Wine("Foo", 1987, "Cab Sav")
```

- `Jello` is renamed to `Jelly` on the way in, everything else imported from food domain with its regular name

# Selective Importing

- Import just `HashMap` and `HashSet` from `immutable`

```
import scala.collection.immutable.{HashMap, HashSet}

HashSet(1,2,3)
HashMap(1 -> "one", 2 -> "two")
```

- Import all of `java.util` except from `Date` and `Deque`, import `ArrayDeque` as `Deque` instead:

```
import java.util.{Date => _, Deque => _, ArrayDeque => Deque, _}

new ArrayList[Int](10) // java.util.ArrayList[Int] = []
new Deque[Int](10) // java.util.ArrayDeque[Int] = []
new Date // compile error
```

- Standard imports for all source files:

```
import java.lang._    // everything in the java.lang package
import scala._        // everything in the scala package
import Predef._       // everything in the Predef object
```

# Companion Objects

- Companions can share private state, but not private[this]:

```
class ShippingContainer[T] private (val items: Seq[T]) {  
    private[this] val maxCount = 10  
    private def isFull: Boolean = items.length >= maxCount  
    override def toString: String = s"${ShippingContainer.containerColor} Container"  
}  
  
object ShippingContainer {  
    def apply[T](items: T*) = new ShippingContainer[T](items)  
    private def containerColor: String = "Green"  
  
    def maxItems(container: ShippingContainer[_]): Int =  
        container.maxCount // compile error, maxCount is private[this]  
  
    def containerFull(container: ShippingContainer[_]): Boolean =  
        container.isFull // fine  
}  
  
val sc = ShippingContainer("a", "b", "c") // Green Container  
sc.items // Seq(a,b,c)  
sc.maxCount // compile error  
sc.isFull // compile error  
ShippingContainer.containerFull(sc) // false
```

# Exercises for Module 10

- Find the `Module10` class and follow the instructions to make the tests pass
- `Module10` is under `module10/src/test/scala/koans`
- The `Laser` scala file is there for your implementation of the "Shoot a LASER beam" test since packages cannot go in your test class.

# Testing in Scala

Pre and Post Conditions, Unit and Integration Tests,  
Mocking, Stubbing, Property Driven Testing

# Agenda

1. Pre and Post Conditions
2. Scalatest
3. Tests vs Specs
4. Matchers
5. Exception Handling
6. Mocking and Stubbing
7. Property Driven Testing
8. Testing With Futures

# Pre and Post Conditions

- Idea from Bertrand Meyer, pre-conditions, post-conditions and invariants

```
assert(2 > 3)  // java.lang.AssertionError: assertion failed
assume(2 > 3)  // java.lang.AssertionError: assumption failed
(1 + 1) ensuring (_ > 3) // java.lang.AssertionError: assertion failed
```

- These can (and should) include String explanations

```
assert(x > 3, "x must be larger than 3")
// java.lang.AssertionError: assertion failed: x must be larger than 3

assume(x > 3, "x must be larger than 3")
//java.lang.AssertionError: assumption failed: x must be larger than 3

(x - 1) ensuring (_ > 3, "x is not large enough")
// java.lang.AssertionError: assertion failed: x is not large enough

def square(x: Int): Int = {
  x * x
} ensuring (_ >= 0, "squares cannot be negative")
```

# assert and assume Can Be Elided

- But ensuring still throws...

```
14:28 $ scala -Xdisable-assertions
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.

scala> val x = 2
x: Int = 2

scala> assert(x > 3, "x must be larger than 3")

scala> assume(x > 3, "x must be larger than 3")

scala> (x - 1) ensuring (_ > 3, "x is not large enough")
java.lang.AssertionError: assertion failed: x is not large enough
  at scala.Predef$Ensuring$.ensuring$extension3(Predef.scala:219)
  ... 29 elided
```

# Requirements

- Since `assert` and `assume` can be turned off, we have `require` (and `requireState`) which always throw on failure

```
val x = 2

require(x < 0, "x must be negative")
// java.lang.IllegalArgumentException: requirement failed: x must be negative

import org.scalactic.Requirements._
requireState(x < 0, "x must be negative")
// java.lang.IllegalStateException: 2 was not less than 0 x must be negative
```

- `require`, and `requireState` from Scalactic are idiomatic, `assert` and `assume` less so

# Testing

- Testing took over from pre-conditions, post-conditions and invariants in most development
- Scala has various testing frameworks:
  - JUnit
  - TestNG
  - Specs2
  - Scalatest
- In the course so far, you have been using Scalatest
- Scalatest supports many styles of testing and offers many features

# Scalatest

- Scalatest is a free, open source testing framework that is well documented and widely used
- It supports many different styles of tests:  
[http://www.scalatest.org/user\\_guide/selecting\\_a\\_style](http://www.scalatest.org/user_guide/selecting_a_style)
- We will concentrate on a couple of styles:
  - FunSuite - very similar to the koans style you have been using throughout the course
  - FunSpec - a more BDD influenced suite for writing specifications
- We will also look at the Matchers DSL and support for mocking and property testing
- Keep the [http://www.scalatest.org/user\\_guide](http://www.scalatest.org/user_guide) handy to write your own (there is lots here)

# FunSuite

```
import org.scalatest._

class TestingSuiteDemo extends FunSuite with Matchers {
    val nums: List[Int] = (1 to 20).toList

    test("Filtering a list") {
        val filtered = nums.filter(_ > 15)
        assert(filtered === Seq(16, 17, 18, 19, 20))
    }

    test("Summing a list") {
        nums.sum should be(210)
    }

    test("Try something else")(pending)
}
```

- tests all have a string name, names must be unique in that suite
- First test uses `assert` with `==` (this is not the elidable pre-condition assert)
- Second test uses `Matchers` for more readability
- The `pending` test at the end will put out a warning until defined

# FunSpec

- More BDD (behavioral driven design): requirements and specs

```
class TestingSpecDemo extends FunSpec with Matchers {

  describe ("Retrieving the weather from the weather service") {
    it ("should call getWeather only if operational is true")(pending)
    it ("should not call getWeather if service not operational")(pending)
    it ("should not call getWeather if operational throws an exception")(pending)
    it ("should call getWeather with different codes when necessary")(pending)
  }

  describe ("Various matchers") {
    describe("on a list of numbers") {
      they("should allow a wide and varied language for matching")(pending)
    }
    describe("on a case class example") {
      they("should allow easy field checking")(pending)
    }
  }

  describe ("Handling exceptions") {
    it ("should expect and intercept exceptions")(pending)
  }
}
```

# Matchers

- Matchers is a large DSL that makes tests easier to read (though possibly harder to write)

```
describe ("Various matchers") {
  describe("on a list of numbers") {

    val nums = (1 to 20).toList
    val threeMults = nums.filter(_ % 3 == 0)

    they("should allow a wide and varied language for matching") {
      val x = 10 * 2

      x should be (20)

      threeMults should have size (6)
      threeMults should contain allOf (3, 6, 12, 15)
      threeMults should not contain (10)
      threeMults should be (Vector(3, 6, 9, 12, 15, 18))
      threeMults should be (sorted)
      all(threeMults) should be > (0)
      atLeast(3, threeMults) should be > (10)
    }
  }
}
```

# Matchers

- Also support for fields in case classes, e.g.

```
describe("on a case class example") {
  they("should allow easy field checking") {
    case class Person(first: String, last: String, age: Int)

    val p1 = Person("Harry", "Potter", 34)

    p1 should have(
      'first ("Harry"),
      'last ("Potter"),
      'age (34)
    )
  }
}
```

# Matchers

- And for checking exceptions:

```
describe ("Handling exceptions") {  
    it ("should expect and intercept exceptions") {  
        an [IllegalArgumentException] should be thrownBy {  
            require(1 == 2, "One equals two?")  
        }  
  
        val ae = intercept [ArithmetricException] (1 / 0)  
        ae.getMessage should be ("/ by zero")  
    }  
}
```

- Also checking floating point values within tolerance

```
describe ("Floating point values") {  
    they ("should be matched within a tolerance") {  
        val sqrt = math.sqrt(2.0)  
        sqrt should be (1.41421356 +- 1e-6)  
        sqrt should (be > 1.41421355 and be < 1.41421357)  
    }  
}
```

# Unit vs Integration Testing

- Unit tests are small, self contained and fast, requiring no external resources
- Integration tests may use external resources, and are useful for checking a complete system
- Neither type of testing is enough without the other
- To Unit test, one must often isolate from external resources
- This is where fakes, mocks and stubs come in useful

# Scalamock

- Scalatest has support for several mocking frameworks, I tend to use *Scalamock*
- Imagine a service to look up external weather details from a web API

```
case class Weather(temp: Double, precip: Double, pressure: Double)

trait WeatherService {
  def getWeather(icaoCode: String): Weather
  def operational(): Boolean
}

def lookupWeather(service: WeatherService, icaoCode: String): Option[Weather] = {
  if (!service.operational()) None
  else Some(service.getWeather(icaoCode))
}
```

# Unit Testing with Mocks

- For Unit testing, we don't want to use the real service (slow/unreliable), we also might want to simulate failures to test how they are handled:

```
class TestingSpecDemo extends FunSpec with Matchers with MockFactory

describe ("Retrieving the weather from the weather service") {
  it ("should call getWeather if operational is true") {
    val mockWS = mock[WeatherService]

    (mockWS.operational _: () => Boolean).expects().returning(true)
    (mockWS.getWeather _).expects("PDX").returning(Weather(55.0, 0.0, 1012.0))

    val weather = lookupWeather(mockWS, "PDX")

    val tolerance = 1e-6

    weather should be (defined)
    weather.get.precip should be (0.0 +- tolerance)
    weather.get.pressure should be (1012.0 +- tolerance)
    weather.get.temp should be (55.0 +- tolerance)
  }
}
```

# Advantages of Mocks

- Mocks allow full scripting of responses (including failures)

```
it ("should not call the getWeather if operational throws an exception") {  
    val mockWS = mock[WeatherService]  
  
    (mockWS.operational _: () => Boolean).expects().  
        throwing(new IllegalStateException("network failure"))  
  
    val ex = intercept[IllegalStateException] {  
        lookupWeather(mockWS, "PDX")  
    }  
  
    ex.getMessage should be ("network failure")  
}
```

- They also verify calls are made, in the correct quantity (e.g. fail if too many or few)

# Mocks vs Stubs

- With Mocks, you script up what you expect
- With Stubs, the emphasis is on verifying what you got:

```
it ("should call the lookup weather with different codes when necessary") {  
    val stubWS = stub[WeatherService]  
  
    (stubWS.operational _: () => Boolean).when().returning(true)  
    (stubWS.getWeather _).when("PDX").returning(Weather(55.0, 0.0, 1012.0))  
    (stubWS.getWeather _).when("SFO").returning(Weather(65.0, 0.3, 1008.0))  
  
    val results1 = lookupWeather(stubWS, "SFO") // check results1 here  
    val results2 = lookupWeather(stubWS, "PDX") // check results2 here  
  
    (stubWS.operational _: () => Boolean).verify().twice()  
    (stubWS.getWeather _).verify("PDX")  
    (stubWS.getWeather _).verify("SFO")  
}
```

# Fakes

```
trait DBAccess {
  def save[T](item: T): String
  def load[T](id: String): Option[T]
}

class FakeDBAccess extends DBAccess {
  private[this] var itemMap = Map.empty[String, Any]

  def save[T](item: T): String = {
    val uuid = UUID.randomUUID().toString
    itemMap = itemMap + (uuid -> item)
    uuid
  }

  def load[T](id: String): Option[T] =
    Try(itemMap(id).asInstanceOf[T]).toOption
}

case class Person(name: String, age: Int)
val fake = new FakeDBAccess
val uuid = fake.save(Person("Sally", 23)) // 1a889c78-ea0a-45ae-b8ce-9e6305d6ec24
fake.load(uuid) // Some(Person(Sally,23))
```

# Property Driven Testing

- Scalatest has support for Scalacheck, which can generate random testing data:

```
import org.scalatest._  
import org.scalatest.prop.PropertyChecks  
  
class TestingSpecDemo extends FunSpec with Matchers with PropertyChecks {  
  describe ("Property checks") {  
    they ("should ensure abs on all Ints returns a positive number") {  
      forAll { (i: Int) =>  
        whenever(i != Int.MinValue) {  
          i.abs should be >= 0  
        }  
      }  
    }  
  }  
}
```

- Without that whenever:

```
// TestFailedException was thrown during property evaluation.  
//   Message: -2147483648 was not greater than or equal to 0
```

# Custom Property Generators

```
import org.scalatest.Gen

val validChars = ('a' to 'z') ++ ('A' to 'Z') ++ ('0' to '9')
val char = Gen.oneOf(validChars)
val strGen = for {
    n <- Gen.choose(0, 30) // 0 to 30 length strings
    seqChars <- Gen.listOfN(n, char)
} yield seqChars.mkString

describe ("Property checks and generators") {
    they ("should test .reverse.reverse on any string gives you back original") {
        forAll(strGen) { str =>
            println(str)
            str.reverse.reverse should be (str)
        }
    }
}
```

- The strGen will generate random length, random character strings and run several of them through the test each time it is run.

# Testing with Futures

- This should fail:

```
// using...
def calc(n: Int, iterations: Int): Future[Double]

test("Calculating PI Asynchronously") {
    import ExecutionContext.Implicits.global
    val calcPi = new CalcPi

    val resultF = calcPi.calc(500, 1000000)

    for (piBy4 <- resultF) yield {
        println(piBy4*4)
        piBy4*4 should be (10.0 +- 0.001)
    }
}
```

- But this passes, and never prints anything, the test finishes before the future is ready...

# Waiting for the Future

- One solution is simply to Await:

```
test("Calculating PI Asynchronously") {
    import ExecutionContext.Implicits.global
    val calcPi = new CalcPi

    val resultF = calcPi.calc(500, 1000000)
    val piBy4 = Await.result(resultF, 1.minute)

    println(piBy4*4)
    piBy4*4 should be (10.0 +- 0.001)
}
```

- Now it fails: 3.1414943040000063 was not 10.0 plus or minus 0.001
- Blocking is OK in main methods and tests, but there are still better options for tests...

# whenReady

```
import org.scalatest._  
import org.scalatest.concurrent._  
import PatienceConfiguration._  
  
class TestingSuiteDemo extends FunSuite with Matchers with ScalaFutures {  
    test("PI when ready") {  
        val calcPi = new CalcPi  
  
        val resultF = calcPi.calc(500, 1000000)  
  
        whenReady(resultF, Timeout(1.minute)) { result =>  
            result * 4 should be (3.141 +- 0.001)  
        }  
    }  
}
```

# Or... If Everything Being Tested is a Future

```
import org.scalatest._

class AsyncTestingSuiteDemo extends AsyncFunSuite with Matchers {
  test ("Calculating PI") {
    val calcPi = new CalcPi

    val resultF = calcPi.calc(500, 1000000)

    for (result <- resultF) yield {
      println(result * 4)
      result * 4 should be (3.141 +- 0.001)
    }
  }
}
```

- Async tests must result in a Future (so use `for...yield` usually)
- While writing the tests you can end with `succeed` to stop everything turning red

# Exercises for Module 11

- Find the `Module11` class and follow the instructions to make the tests pass
- `Module11` is under `module11/src/test/scalatest/koans`
- Unlike previous exercises where you had to get the tests to pass, this time you have to write the tests

# Pattern Matching

Simple and Compound Pattern Matches, Case Classes,  
Custom Extractors

# Agenda

1. Simple Constant Patterns
2. Variable Loading and Binding
3. Guards
4. Options, Collections, Tuples, Try
5. Case Classes
6. Constructor Patterns
7. Type Patterns
8. Pattern matching in `vals` and `fors`
9. Partial functions reprise
10. Custom Extractors

# Simple Constant Patterns

- In its simplest usage, Scala's `match` is like Java's `switch`:

```
def matchIt(x: Any): Unit = x match {  
    case 10          => println("The number 10")  
    case true        => println("This is the truth")  
    case 2.0         => println("Double precision 2.0")  
    case "hello"     => println("Well, hi there")  
    case ()          => println("Unit")  
    case _           => println("It's something else")  
}  
  
matchIt(10)      // The number 10  
matchIt(2.0)      // Double precision 2.0  
matchIt("hello") // Well, hi there  
matchIt()         // Unit  
matchIt(3)        // It's something else
```

- `case _` is the equivalent of `default` in Java
- There is no automatic fall-through, no `break` keyword, and no need for `{}`s (next `case` keyword is the demarcation between handlers)

# match is an expression

- The cases are checked in order, and the first successful match consumes the match event
- Unlike Java, Scala's `match` is also an expression

```
def pair(s: String): String = s match {  
    case "fish"      => "chips"  
    case "bacon"     => "eggs"  
    case "tea"        => "scones"  
    case "horse"     => "carraige"  
}  
  
pair("fish")      // chips  
pair("tea")       // scones  
  
pair("universe") // MatchError!
```

- Also this example has no default `case _` which means a non-matching result will throw a `MatchError` exception

# Variable Loads

- As well as simply matching constants, Scala can load a value for use in the code block:

```
def opposite(s: String): String = s match {  
    case "hot"          => "cold"  
    case "full"         => "empty"  
    case "cool"         => "square"  
    case "happy"        => "sad"  
    case anythingElse   => s"not $anythingElse"  
}  
  
opposite("cool")    // square  
opposite("happy")   // sad  
opposite("sane")    // not sane
```

- While the constants will be matched first, anything that doesn't match those constants will be put into the value `anythingElse` and the `s"not $anythingElse"` will be returned (this also makes the pattern match complete for all inputs)

# Binding vs Loading

- A variable identifier in a pattern match is loaded with the value
- But there is an alternative, useful for multiple-matches and in other situations, binding:

```
def opposite2(s: String): String = s match {
    case "hot"          => "cold"
    case "full"         => "empty"
    case "cool"         => "square"
    case "happy"        => "sad"
    case inWord @ ("sane" | "edible" | "secure") => s"in$inWord"
    case anythingElse => s"not $anythingElse"
}
opposite2("happy")    // sad
opposite2("sane")     // insane
opposite2("edible")   // inedible
opposite("fish")      // not fish
```

- `("sane" | "edible" | "secure")` matches any of those words, and the `@` binds the result into `inWord`
- More generally, `@` binds the pattern match on the right of it to the variable on the left of it

# Case Matters!

- Identifiers starting with lower case are treated as **variables**
- Identifiers starting with upper case are treated as **constants**

```
val MaxLimit = 10 // constants start with upper case
val minLimit = 1

def isALimit(x: Int) = x match {
  case MaxLimit => true // constant match works as expected
  case _ => false
}

isALimit(10) // true
isALimit(3) // false
```

# But!

```
def isALimit(x: Int) = x match {  
    case MaxLimit => true // constant match works as expected  
    case minLimit => true // this is treated as a load and will always match!  
    case _ => false  
}  
  
isALimit(10) // true  
isALimit(1) // true  
isALimit(3) // true!
```

- If you must use lower case constants, put them in backticks in the match:

```
def isALimit(x: Int) = x match {  
    case MaxLimit => true  
    case `minLimit` => true // backticks make this work as constant match  
    case _ => false  
}  
  
isALimit(10) // true  
isALimit(1) // true  
isALimit(3) // false
```

# Guards

- Anything on the left of the `=>` is part of the pattern match, anything on the right is what to do
- `if` expressions can be used on the left of the `=>`:

```
def describeNumber(x: Int): String = x match {  
    case 0                      => "zero"  
    case n if n > 0 && n < 100  => "smallish positive"  
    case n if n > 0              => "large positive"  
    case n if n < 0 && n > -100 => "smallish negative"  
    case _                      => "large negative"  
}  
  
describeNumber(-99) // smallish negative  
describeNumber(99)  // smallish positive  
describeNumber(0)   // zero  
describeNumber(101) // large positive  
describeNumber(-101) // large negative
```

- Remember that the first full match stops the attempt going any further

# The Wrong Way to Guard

- It's easy to forget that the if goes before the =>

```
def badDescribeNumber(x: Int) = x match {
  case 0 => "zero"
  case n => if (n > 0 && n < 100) "smallish positive"
  case n => if (n > 0) "large positive"
  case n => if (n < 0 && n > -100) "smallish negative"
  case _ => "large negative"
} // badDescribeNumber[](val x: Int) => Any!

badDescribeNumber(-99) // ()
badDescribeNumber(99) // smallish positive
badDescribeNumber(0) // zero
badDescribeNumber(101) // ()
badDescribeNumber(-101) // ()
```

- Remember, guards go on the left of the =>

# Matching Options

```
def matchOption(o: Option[Int]) = o match {  
    case Some(n) if n > 10 => "It's a number above 10"  
    case Some(_)          => "It's a number 10 or less"  
    case None             => "No number given"  
}  
  
matchOption(Some(50)) // It's a number above 10  
matchOption(Some(5))  // It's a number 10 or less  
matchOption(None)     // No number given
```

- There are only two states for Option, Some(x) and None
- Can unpack variables from inside the option (and use them)
- This is common usage, but there are often more idiomatic ways of dealing with Option (e.g. `map`, `getOrElse`)

# Matching Tuples

```
def matchTuple3(tup: (Int, Boolean, String)): String = tup match {  
    case (1, flag, string) => s"a 1 followed by $flag and $string"  
    case (i, true, "Fred") => s"a true Fred with int $i"  
    case (a, b, c)         => s"Some other tuple int $a, flag $b, string $c"  
}  
  
matchTuple3((1, false, "Sally"))  
// a 1 followed by false and Sally  
matchTuple3((1, true, "Harry"))  
// a 1 followed by true and Harry  
matchTuple3((2, true, "Fred"))  
// a true Fred with int 2  
matchTuple3((2, false, "Fred"))  
// Some other tuple int 2, flag false, string Fred
```

- `true` is a keyword, so there is no confusion about load or constant match there, it's a constant

# Matching Lists

- For Lists specifically, you can use :: (cons) notation for matches:

```
def matchList(xs: List[Int]): String = xs match {  
    case 1 :: 2 :: rest => s"A 1, 2 list followed by $rest"  
    case a :: b :: _ => s"A list of at least 2 items, starting with $a, $b"  
    case a :: Nil => s"A single element list of $a"  
    case Nil => "The empty list"  
}  
  
matchList(List(1,2,3))  
// A 1, 2 list followed by List(3)  
matchList(List(1,2))  
// A 1, 2 list followed by List()  
matchList(List(1,3,4))  
// A list of at least 2 items, starting with 1, 3  
matchList(List(4))  
// A single element list of 4  
matchList(Nil)  
// The empty list
```

- very common to see `case head :: tail =>` in recursive functions

# Other Collections

- You can do similar matches for other collections (but not with cons notation):

```
def matchSeq(xs: Vector[Int]): String = xs match {  
    case 1 :: 2 :: rest => s"A 1, 2 vector followed by $rest"  
    case Vector(a, b, _) => s"A vector of at least 2 items, starting with $a, $b"  
    case Vector(a) => s"A single element vector of $a"  
    case Vector() => "The empty vector"  
}
```

- `::` stands in for `:_`
- Can also use expansion operator `_*` to match remainder in "constructor" style
- And bindings, so `Vector(1, 2, rest @ _*) =>` is equivalent to `1 :: 2 :: rest =>`
- This syntax also works for Lists (but with `List` replacing `Vector` of course)

# Matching Try

```
import scala.util._

def matchTry(t: Try[_]): String = t match {
  case Success(x) => s"It worked, result is $x"
  case Failure(e) => s"It failed with $e"
}

matchTry(Try(4/2)) // It worked, result is 2
matchTry(Try(4/0)) // It failed with java.lang.ArithmetricException: / by zero
```

- Other core libraries often have pattern match support too, like Future, Either, etc.

# Case Classes

- When you define a `case class` you get a bunch of things, including pattern matching

```
case class Address(street: String, city: String, postCode: Option[String])
case class Person(name: String, phone: Option[String], address: Option[Address])
```

- Factory methods for easy construction

```
val harry = Person("Harry", None, Some(Address(
  "123 Little Whinging way", "Purley", Some("PN22 6RT"))
))

val sally = Person("Sally", Some("321-222-3344"), None)
```

- Built in useful default `toString`

```
harry
// Person(Harry,None,Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT)))))

sally
// Person(Sally,Some(321-222-3344),None)
```

# Case Classes

- You also get... `equals` and `hashCode` that work

```
sally == harry           // false
sally == sally           // true
sally == Person("Sally", Some("321-222-3344"), None) // true
sally == Person("Sally", Some("321-234-3344"), None) // false

sally.hashCode           // -171467737
Person("Sally", Some("321-222-3344"), None).hashCode // -171467737
harry.hashCode           // 1544670842
```

- Public parametric fields

```
harry.name           // Harry
harry.address.map(_.city) // Some(Purley)
harry.phone          // None
sally.phone          // Some(321-222-3344)
```

# Case Classes

- And, a `copy` method

```
val sally2 = sally.copy(address = harry.address, phone = Some("321-333-2211"))
// Person(Sally,Some(321-333-2211),
//   Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT))))
val harry2 = harry.copy(phone = sally2.phone)
// Person(Harry,Some(321-333-2211),
//   Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT))))
```

- case classes are immutable by default, but `copy` makes them easy to work with in a functional way
- And, you get pattern matching...

# Compound Pattern Matches

```
def postCodeForHarry(person: Person) = person match {
  case Person("Harry", _, Some(Address(street, city, Some(postcode)))) =>
    println("Harry found with postcode")
    println(s"City $city")
    println(s"Street $street")
    postcode
  case _ => ""
}

postCodeForHarry(harry) // PN22 6RT
postCodeForHarry(harry2) // PN22 6RT
postCodeForHarry(sally) // ""
postCodeForHarry(sally2) // ""
```

- Mix and match constants, `case` patterns, `Options` and anything matchable
- Could also get the harry match as a whole in the above with:

```
case harry @ Person("Harry", _, Some(Address(street, city, Some(postcode)))) =>
```

- Because they look like the constructors for case classes, these are called *constructor patterns*

# Typed Pattern Matches

```
def describeType(x: Any) = x match {
  case i: Int if i > 0 => s"Int ${i * i}"
  case d: Double => s"Double $d"
  case s: String => s"String ${s.reverse}"
  case p: Person => s"Person, name = ${p.name}"
  case _ => "Some other type"
}

describeType(3)      // Int 9
describeType(3.4)    // Double 3.4
describeType("Hello") // String olleH
describeType(harry)  // Person Harry
describeType(true)   // Some other type
```

- Once matched, the variable is typed on both the left and right of the =>
- This is idiomatic and favored over the form:

```
val s: Any = "Hello"
if (s.isInstanceOf[String]) {
  s.asInstanceOf[String].reverse
}
```

# Beware Type Erasure!

```

def withIntStringMap(x: Any): Int = x match {
  case m: Map[Int, String] => m.head._1 * m.head._1
  case _ => 0
}

// Warning: non-variable type argument Int in type pattern
// scala.collection.immutable.Map[Int, String] (the underlying of Map[Int, String])
// is unchecked since it is eliminated by erasure
// case m: Map[Int, String] => m.head._1 * m.head._1
//           ^

```

- Scala will match the `Map` vs not, but will **believe** you on the inner erased types, so you can get:

```

withIntStringMap(Map(2 -> "two")) // 4 - as expected
withIntStringMap(List(2))          // 0 - not a match
withIntStringMap(Map("One" -> 1)) // ClassCastException!

```

- The safe way to match erased type parameters is `case m: Map[_, _]`
- Alternatively, type-tags can be used - see advanced course

# val and pattern matching

- val is a pattern-match

```
val Person(name, phone, Some(Address(_, _, postCode))) = harry
// name: String = Harry
// phone: Option[String] = None
// postCode: Option[String] = Some(PN22 6RT)
```

- Which means it can fail...

```
val Person(name2, phone2, Some(Address(_, _, postCode2))) = sally
// scala.MatchError: Person(Sally,Some(321-222-3344),None)
```

- This fails because sally has no Address recorded
- Be aware of this if you use a val with a pattern match - you may get a match error

# for and pattern matching

- Generators in a `for` block are also pattern matches

```
val numbersMap = Map(1 -> "one", 2 -> "two", 3 -> "three")  
  
for ((k, v) <- numbersMap) { // unpack the key -> value tuples  
    println(s"$k is $v")  
}
```

- A non-match will just short-circuit the `for` so there's no exception if no match

```
val people = List(harry, harry2, sally, sally2)  
  
for {  
    Person(name, phone, _) <- people  
    if phone.isDefined  
} yield name -> phone.get  
// List((Harry,321-333-2211), (Sally,321-222-3344), (Sally,321-333-2211))
```

# Partial functions and pattern matches

- Remember a `PartialFunction[T, R]` extends `Function1[T, R]`
- This means that a partial function (which is a pattern match) can substitute for any `Function1`

```
numbersMap.map {  
    case (1, w) => s"It's 1 and the word is $w"  
    case (k, v) => s"Not 1 but ($k, $v)"  
}  
// List(It's 1 and the word is one, Not 1 but (2, two), Not 1 but (3, three))
```

- If you use a partial function that is incomplete in a function expecting a `Function1`, you may end up with a `MatchError`

# Sealed Class Hierarchies

- Sometimes you want to control what different types may be in a hierarchy
- The `sealed` keyword gives you this, and pattern matches can then give warnings about incomplete matches

```
sealed class AccountType
case object Checking extends AccountType
case object Savings extends AccountType

def checking(at: AccountType): Boolean = at match {
  case Checking => true
}
// Warning:(6, 43) match may not be exhaustive.
// It would fail on the following inputs: AccountType(), Savings
// def checking(at: AccountType): Boolean = at match {
```

- `sealed` means that the only sub-types of the sealed class or trait must be defined in the same source file

# Extractors and Unapply

- How does it all work?

```
case class Person(first: String, last: String, age: Int)

val p1 = Person("Fred", "Frederickson", 28)

Person.unapply(p1)
// res1: Option[(String, String, Int)] = Some((Fred,Frederickson,28))
```

- unapply is auto-generated for case classes in the companion object

```
val xs = List(1,2,3,4)

List.unapplySeq(xs)
// res3: Some[List[Int]] = Some(List(1, 2, 3, 4))
```

- unapplySeq allows for matching repeated, var-arg matches in collections
- And we can write our own

# Custom Extractors

```
val coordsStr = "-121.432, 34.002"

object Coords {
  def unapply(coordsStr: String): Option[(Double, Double)] = Try {
    val fields = coordsStr.split(",").map(_.trim.toDouble)
    (fields(0), fields(1))
  }.toOption
}

coordsStr match {
  case Coords(x, y) =>
    println(s"x = $x")
    println(s"y = $y")
}
// x = -121.432
// y = 34.002
```

# Custom Seq Extractors

```
object CoordSeq {  
    def unapplySeq(coordsStr: String): Option[Seq[Double]] = Try {  
        coordsStr.split(",").toList.map(_.trim.toDouble)  
    }.toOption  
}  
  
coordsStr match {  
    case CoordSeq(c @ _*) =>  
        c foreach println  
}  
// -121.432  
// 34.002  
  
coordsStr match {  
    case CoordSeq(x, y, _) =>  
        println(x)  
        println(y)  
}  
// -121.432  
// 34.002
```

# Exercises for Module 12

- Find the `Module12` class and follow the instructions to make the tests pass
- `Module12` is under `module12/src/test/scala/koans`, but there are other classes in that source file as well (part of the testing)

# Lists

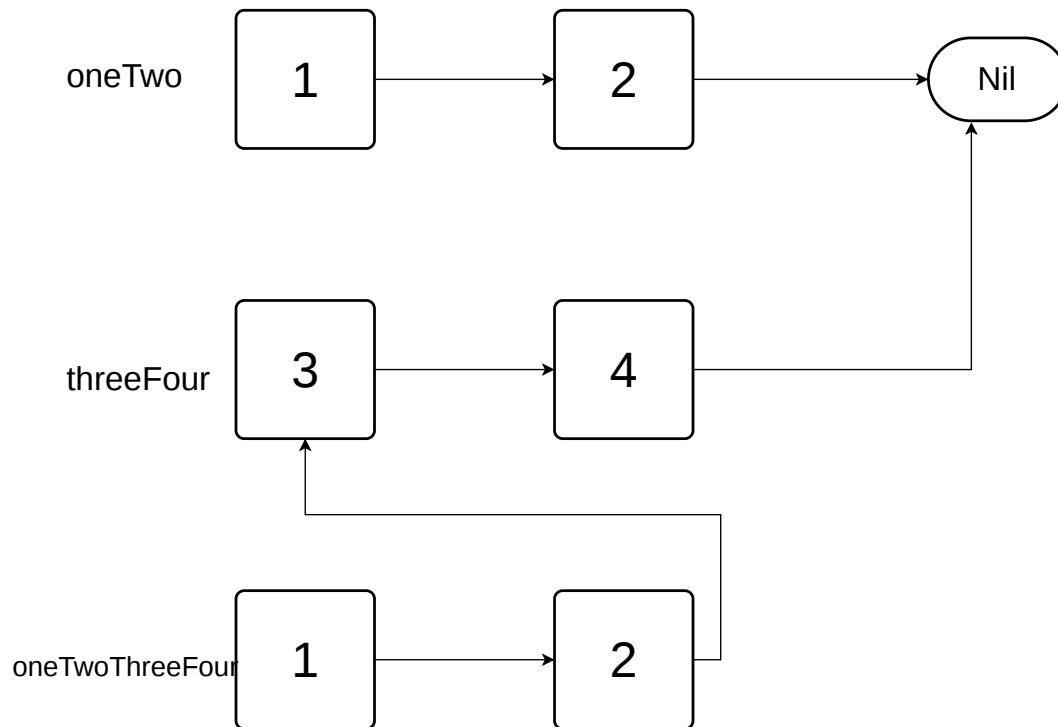
Exploring Scala's simplest functional immutable data structure

# Agenda

1. The Immutable Linked List
2. Initializing / Converting to List
3. Constant Time Operations
4. Linear Time Operations
5. Higher Order Functions
6. Predicate Functions
7. Folds
8. Sorting
9. Even More Functions

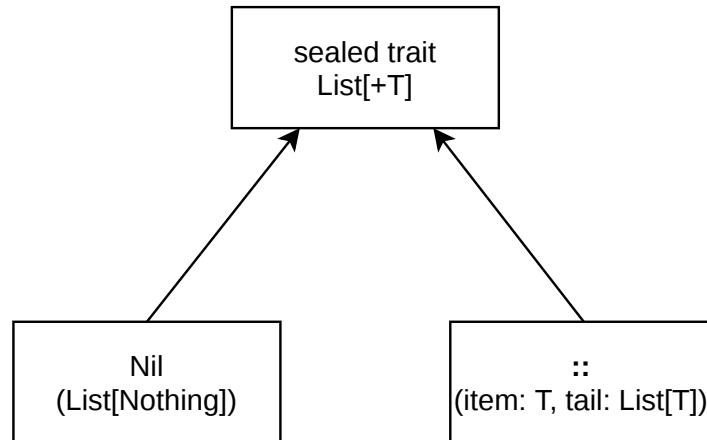
# The Immutable Linked List

```
val oneTwo = List(1,2)          // List(1,2)
val threeFour = 3 :: 4 :: Nil    // List(3,4)
val oneTwoThreeFour = oneTwo :::: threeFour // List(1,2,3,4)
```



# List Properties

- Immutable
- Performance and memory efficient for head operations
- Covariant
- Always terminated by the singleton Nil
- Simple implementation



# Initializing Lists

- Lists have a factory method on the `List` companion object

```
val oneTwo = List(1,2)
```

- Or you can use the *cons* notation

```
val threeFour = 3 :: 4 :: Nil
```

- `::` is *right associative*, so the final item must be a List (i.e. `Nil`)
- The `List` companion object also has a number of factory methods for initialization

```
List.fill(10)(0)          // List(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
List.tabulate(10)(x => x * x) // List(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
List.range(0, 10)           // List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

# Converting to Lists

- Other collection types can be converted to List easily:

```
Vector('a', 'b', 'c').toList  
// List[Char] = List(a, b, c)

Set(1.0, 2.0, 3.0).toList  
// List[Double] = List(1.0, 2.0, 3.0)

Map(1 -> "one", 2 -> "two").toList  
// List[(Int, String)] = List((1,one), (2,two))

"hello world".toList  
// List[Char] = List(h, e, l, l, o, , w, o, r, l, d)
```

- Implicits allow a String to be treated as a List of Char in most circumstances

# List is Covariant

- If A extends B then List[A] is a subtype of List[B]
- List also widens as needed on cons and concatenation

```
val xs1 = 1 :: Nil      // List[Int] = List(1)
val xs2 = true :: xs1  // List[AnyVal] = List(true, 1)
val xs3 = "hi" :: xs2  // List[Any] = List(hi, true, 1)

def sizeOfList(xs: List[Any]): Int = xs.size

sizeOfList(xs1)  // 1
sizeOfList(xs2)  // 2
sizeOfList(xs3)  // 3
```

# Constant Time Operations

- Operations at the head of a List are constant time, e.g. `.head`, `.tail`, `.isEmpty`

```
val nums = (1 to 10).toList

nums.head // 1
nums.tail // List(2,3,4,5,6,7,8,9,10)
nums.isEmpty // false
nums.nonEmpty // true

0 :: nums // List(0,1,2,3,4,5,6,7,8,9,10)
nums.:::(0) // List(0,1,2,3,4,5,6,7,8,9,10)
```

- `:::` is also constant time, it re-uses the existing list and only creates one new node

# Linear Time Operations

- Operations that are more expensive on List include

```
val nums1 = (1 to 5).toList

nums1.last    // 5
nums1.init    // List(1,2,3,4)
nums1.length  // 5

nums1.reverse // List(4,3,2,1)
```

- These are all linear time, each must traverse the entire list
- `init` must make a copy of the entire List minus the final element
- Lists are ideal if you work at the head exclusively, but sub-optimal for other uses

# Operations that Depend on Position

- Also there are functions that depend on their parameters for their order

```
nums1(3)          // 4 (aka nums.apply(3))

nums1.drop(3)    // List(4,5)
nums1.take(3)    // List(1,2,3)

val nums2 = (6 to 10).toList

val allNums = nums1 ::: nums2 // List(1,2,3,4,5,6,7,8,9,10)

allNums.drop(8).headOption   // Some(9)
allNums.drop(20).headOption   // None
allNums.updated(4, 100)      // List(1,2,3,4,100,6,7,8,9,10)
```

- ::: (concat) must duplicate the first List but re-uses the second
- drop with headOption will not throw an exception, even if it exhausts the list
- updated must make a new List up to the specified position, but re-uses the rest

# Higher Order Functions

- Higher Order Functions are simply functions that take other functions

```
val words = List("four", "four", "char", "word")
// List(four, four, char, word)

words.map(_.reverse)
// List(ruof, ruof, rahc, drow)

words.reverse.map(_.reverse)
// List(drow, rahc, ruof, ruof)

words.map { word => word.toList }
// List(List(f, o, u, r), List(f, o, u, r), List(c, h, a, r), List(w, o, r, d))

words.flatMap { word => word.toList }
// List(f, o, u, r, f, o, u, r, c, h, a, r, w, o, r, d)

words foreach println
// four
// four
// char
// word
```

# Predicate Based Functions

- A predicate is just a function returning Boolean, as such predicate based functions are higher order functions

```
words.filter(_.contains("a"))      // List(char)
words.filter(_.contains("f"))      // List(four, four)

words.find(_.contains("a"))        // Some(char)
words.find(_.contains("z"))        // None
words.indexWhere(_.contains("a"))  // 2
words.indexWhere(_.contains("z"))  // -1
words.indexWhere(_.contains("r"))  // 0
words.lastIndexWhere(_.contains("r")) // 3

words.filterNot(_.contains("a"))   // List(four, four, word)
words.partition(_.contains("a"))    // (List(char),List(four, four, word))

words.takeWhile(_.contains("f"))   // List(four, four)
words.dropWhile(_.contains("f"))   // List(char, word)
```

# Folds

```
val words = List("four", "four", "char", "word")
val nums = List(2,3,5,8,13,21)

val sumNums = nums.foldLeft(0)((a, b) => a + b) // 52
val prodNums = nums.foldLeft(1)(_ * _)           // 65520

val asString = words.foldLeft("")(_ + ", " + _) // , four, four, char, word
```

- Can also use `foldRight` or just `fold`, but `foldLeft` works best for List traversal
- There is also `reduceLeft` etc

```
val sum2 = nums.reduceLeft(_ + _) // 52

// but!
List.empty[Int].foldLeft(0)(_ + _) // 0
List.empty[Int].reduceLeft(_ + _) // UnsupportedOperationException
```

# Fold Alternatives

- For many common fold operations, there are ready-made alternatives.  
e.g. for Lists of Numerics

```
nums.sum    // 52  
nums.product // 65520
```

- and for any kind of List where you want to create a string representation:

```
words.toString          // List(four, four, char, word)  
words.mkString          // fourfourcharword  
words.mkString(",")    // four,four,char,word  
words.mkString("[", ",","\n", "]") // [four,four,char,word]
```

# Sorting

```

case class Person(name: String, age: Int)
val xs = List(Person("Harry", 25), Person("Sally", 23), Person("Fred", 31))

xs.sortWith((p1, p2) => p1.age < p2.age)
// List(Person(Sally,23), Person(Harry,25), Person(Fred,31))

xs.sortBy(_.name)
// List(Person(Fred,31), Person(Harry,25), Person(Sally,23))

List(5, 2, 3, 4, 8, 1, 7).sorted
// List(1, 2, 3, 4, 5, 7, 8)

```

- `sorted` requires definition of an `Ordering[T]` for `List[T]`

```

implicit object PersonOrdering extends Ordering[Person] {
  override def compare(x: Person, y: Person) = {
    if (x.name == y.name) x.age - y.age
    else if (x.name > y.name) 1 else -1
  }
}
xs.sorted
// List(Person(Fred,31), Person(Harry,25), Person(Sally,23))

```

# Even More Functions

- Need to transpose a matrix?

```
val matrix = List(List(1,2,3), List(4,5,6), List(7,8,9))
// List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
val transpose = matrix.transpose
// List(List(1, 4, 7), List(2, 5, 8), List(3, 6, 9))
```

- Sum up all the numbers in that matrix:

```
matrix.flatten.sum // 45
```

- Group by first letters of a word:

```
val words = List("four", "four", "char", "word")
words.groupBy(_.head)
// Map(w -> List(word), c -> List(char), f -> List(four, four))
```

# Even More Functions

- Filter by a type in a List, and return just a List of that type

```
trait Fruit
case class Apple(name: String) extends Fruit
case class Orange(name: String) extends Fruit

val fruits = List(Apple("Fiji"), Orange("Jaffa"), Apple("Cox's")) // List[Fruit]

fruits.collect {
  case a: Apple => a
} // List[Apple] = List(Apple(Fiji), Apple("Cox's"))
```

- `collect` is like a `filter` and `map` combined into one, takes a `PartialFunction`, and will narrow the resulting List type if possible

# Permutations and Combinations

```
val nums = List.range(0, 10)
// List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

nums.grouped(3).take(5).toList
// List(List(0, 1, 2), List(3, 4, 5), List(6, 7, 8), List(9))

nums.sliding(3).take(5).toList
// List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4), List(3, 4, 5), List(4, 5, 6))

nums.combinations(3).take(5).toList
// List(List(0, 1, 2), List(0, 1, 3), List(0, 1, 4), List(0, 1, 5), List(0, 1, 6))

nums.permutations.take(5).toList
// List(List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), List(0, 1, 2, 3, 4, 5, 6, 7, 9, 8),
//      List(0, 1, 2, 3, 4, 5, 6, 8, 7, 9), List(0, 1, 2, 3, 4, 5, 6, 8, 9, 7),
//      List(0, 1, 2, 3, 4, 5, 6, 9, 7, 8))

val numsPlusOne = nums.map(_ + 1)

nums.corresponds(numsPlusOne)((a, b) => a + 1 == b) // true
```

# Indices, zip, unzip

```
val chars = List.range('a', 'h')
// List[Char] = List(a, b, c, d, e, f, g)

val idx = chars.indices
// scala.collection.immutable.Range = Range 0 until 7

chars.zip(idx)
// List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4), (f,5), (g,6))

val zipped = chars.zipWithIndex
// List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4), (f,5), (g,6))

zipped.unzip
// (List[Char], List[Int]) = (List(a, b, c, d, e, f, g), List(0, 1, 2, 3, 4, 5, 6))
```

- And many, many more...
- <https://www.scala-lang.org/api/current/scala/collection/immutable>List.html>

# Exercises for Module 13

- Find the `Module13` class and follow the instructions to make the tests pass
- These exercises are based on simplified versions of a real problem I needed to solve using the collections API.
- These do use `Sets` in a simple way, to make a `Set` from a `c: Char`, just use `Set(c)`
- If you add something to a `Set` that is already in the `Set`, it will not be added again so that you can add the same thing any number of times and it will only be in there once.
- We'll learn more about `Set` in the next module, but this will be enough to complete these exercises.

# Collections

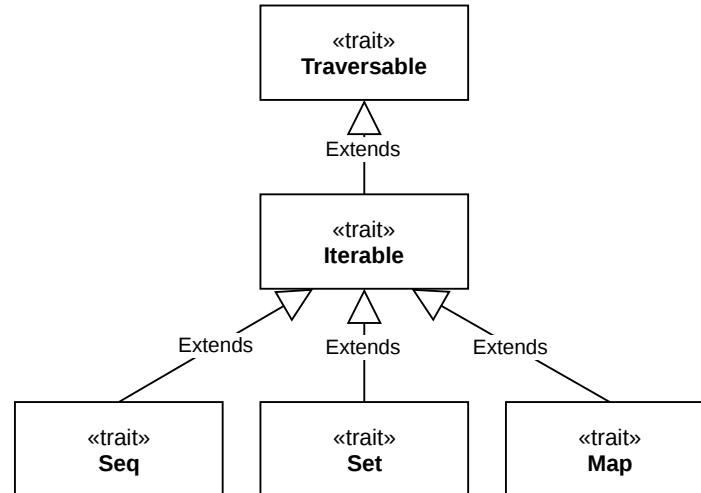
A look at Scala's other common collections, and their performance trade-offs

# Agenda

1. Other Collections
2. Mutable vs Immutable
3. Consistent API for Collections
4. Other Sequences
5. The Mighty Vector
6. Sets
7. Maps
8. Concrete Implementations
9. Iterators, Views and Streams

# Other Collections

- Scala has a rich hierarchy of collections in addition to List



- Scala has three broad categories of collection:
  - Seq** maintains order of insertion
  - Set** maintains uniqueness but not order (though may be sorted)
  - Map** is key -> value association, also unique by key

# Sequences (Performance)

**immutable head tail apply update prepend append insert**

List	C	C	L	L	C	L	-
Stream	C	C	L	L	C	L	-
Vector	eC	eC	eC	eC	eC	eC	-
Stack	C	C	L	L	C	L	L
Queue	aC	aC	L	L	C	C	-
Range	C	C	C	-	-	-	-
String	C	L	C	L	L	L	-

**mutable**

ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	-	-	-
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C	-	-	-

# Sets and Maps (Performance)

## **immutable**   **lookup** **add** **remove** **min**

HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC
ListMap	L	L	L	L

## **mutable**

HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC
TreeSet	Log	Log	Log	Log

## Key

### **Code**               **Description**

**C**      Constant (fast)

**eC**     Effectively Constant

**aC**    Ammortized Constant

**Log**   Proportional to the log of the size

**L**      Proportional to the size

-      The operation is not supported.

- <https://docs.scala-lang.org/overviews/collections/performance-characteristics.html>

# LinearSeq vs IndexedSeq

- Seq is further divided into two broad categories: LinearSeq and IndexedSeq
- LinearSeq is optimized for head-first, forward linear access
  - Default implementation in Scala is List
- IndexedSeq is optimized for random access
  - Default implementation in Scala is Vector
- Both List and Vector are immutable. These two form the most common choice
  - If you know you can work exclusively at the head (e.g. recursion) use List
  - For anything else, typically use Vector
- For sheer performance, particularly with primitives, you sometimes will use Array
  - But remember, Array is mutable and has no thread safety
  - Profile and prove a performance problem before using Array

# mutable vs immutable

- Along with the big 3 (List, Vector and Array) there are many other more specialized collections
- Many of these, e.g. Set, Queue, Stack have both mutable and immutable versions
- These are under `scala.collection.immutable` and `scala.collection.mutable` packages
- Best Practice, don't import directly from these, import the packages instead

```
import scala.collection.mutable
import scala.collection.immutable

def popImmutableQueue(q: immutable.Queue[Int]): (Int, immutable.Queue[Int]) = {
    q.dequeue
}

def popMutableQueue(q: mutable.Queue[Int]): Int = {
    q.dequeue()
}
```

# Consistent API

```

import scala.collection.immutable

val xs = List(1,2,3,4)          // List(1, 2, 3, 4)
val exs = List.empty[Int]        // List()

val v = Vector(1,2,3,4)          // Vector(1, 2, 3, 4)
val ev = Vector.empty[Int]       // Vector()

val q = immutable.Stack(1,2,3,4) // Stack(1, 2, 3, 4)
val eq = immutable.Stack.empty[Int] // Stack()

val s = Set(1,2,3,4)
val es = Set.empty[Int]

q == xs    // true
q == v     // true
xs == v    // true
ev == es   // false

```

- Equality between Seq s works based on contents (**except Array** - use `.deep`)
- Consistent construction, empty, toString, etc.

# Easy Conversions

```
val arr = Array(1,2,3,4)
xs == arr
xs == arr.deep

xs.toVector          // Vector(1, 2, 3, 4)
xs.toArray           // Array(1, 2, 3, 4)
xs.toSet              // Set(1, 2, 3, 4)
xs.zipWithIndex.toMap // Map(1 -> 0, 2 -> 1, 3 -> 2, 4 -> 3)
xs.toList             // List(1, 2, 3, 4)

xs.toQueue // compile error - no built in toQueue method

xs.to[immutable.Queue] // Queue(1, 2, 3, 4)
```

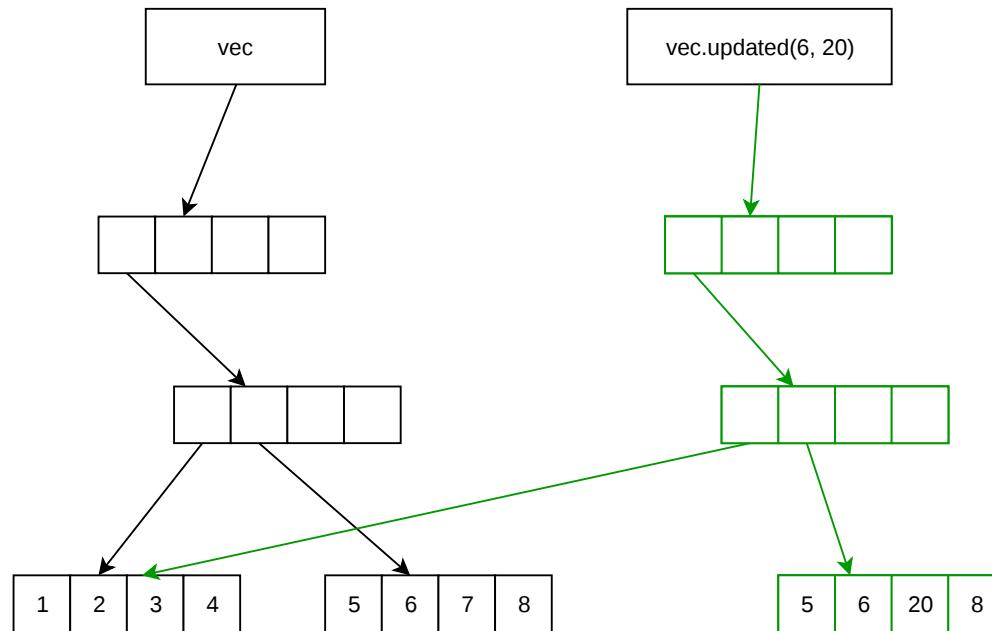
- `toList` on a `List` is a no-op (so you can call these with little-to-no overhead)

# Other Sequences (overview)

- **Queue** - FIFO - implemented as pair of Lists for immutable (one forward, one reverse)
- **Stack** - LIFO - can just use a List instead (exists for backwards compatibility)
- **Array** - Mutable and Random Access. Direct alias to Java Array. Supports primitives
- **Range** - An arithmetic progression, implemented lazily
- **Iterator, View and Stream** - Lazy sequences, Stream may be infinite. More on these later
- **Vector** - Random Access, effectively constant performance for all supported operations, clever memory re-use. We'll look at this next

# The Mighty Vector

- Example is Arity 4, the real Vector is Arity 32 making all operations Log32 (or effectively constant)



# The Mighty Vector

- Vector is really Arity 32
- In Int addressable space, any size vector cell can be navigated to in at most 7 hops
- And even the largest vector will only need  $7 * 32$  words duplicated for a single cell update operation
- If you only work at the head of a collection, List is still a marginally better choice
- For anything else, you are best served going straight to Vector for an immutable, ordered sequence
- Also, the 32 word organization happens to be a size that the JVM manages very well

# Sets (immutable)

- Sets maintain unique identity, but not order

```

val vowels = Set('a', 'e', 'i', 'o', 'u')

vowels.contains('a')      // true
vowels.contains('t')      // false
vowels('a')                // true
vowels('t')                // false

vowels + 'y'              // Set(e, y, u, a, i, o)
vowels + 'e'              // Set(e, u, a, i, o)

val commonLetters = Set('e','t','a','o','i','n','s','r','h')

commonLetters intersect vowels // Set(e, a, i, o)
commonLetters diff vowels     // Set(s, n, t, h, r)
vowels diff commonLetters     // Set(u)
commonLetters union vowels    // Set(e, s, n, t, u, a, i, h, r, o)

"hello to me".count(vowels) // 4

```

- `Set.apply` and `Set.contains` are equivalent
- `Set[T]` extends `T => Boolean` and can be used as a predicate

# Sorted and Mutable Sets

- Sets don't maintain insertion order, but can be sorted, e.g. `TreeSet`

```
import scala.collection.immutable  
immutable.TreeSet('u', 'o', 'i', 'e', 'a') // TreeSet(a, e, i, o, u)
```

- Mutable sets can be added to and removed from (of course)

```
import scala.collection.mutable  
  
val vowelsMut = mutable.Set('a', 'e', 'i', 'o', 'u')  
  
vowelsMut += 'y'           // Set(u, y, e, o, i, a)  
vowelsMut('y')            // true  
  
vowelsMut -= 'y'           // Set(u, e, o, i, a)  
vowelsMut('y')            // false  
  
vowelsMut('y') = true     // Set(u, y, e, o, i, a)  
vowelsMut('y')            // true  
  
vowelsMut('y') = false    // Set(u, e, o, i, a)  
vowelsMut('y')            // false
```

# Maps

- Maps are Key -> Value associations where the keys are a Set
- Like Sets, Maps have both immutable and mutable implementations
- Map[K, V] extends function K => V

```
val numWords = Map(1 -> "one", 2 -> "two", 3 -> "three", 4 -> "four", 5 -> "five")

numWords(1)      // one -- don't use this
numWords.get(1) // Some(one) -- use this
numWords.getOrElse(1, "?") // one -- or this

val nums = List(1,2,3,2,5)
nums.map(numWords) // List(one, two, three, two, five)

for ((num, word) <- numWords) {
  println(s"$num -> $word")
}
// 5 -> five
// 1 -> one
// 2 -> two
// 3 -> three
// 4 -> four
```

# Sorted and Mutable Maps

- Like Set, there are Maps that maintain a sort order

```
val tm = immutable.TreeMap.empty[Int, String] ++ numWords
// Map(1 -> one, 2 -> two, 3 -> three, 4 -> four, 5 -> five)
```

- There is also a ListMap that does maintain insertion order, but performance is dismal
- Maps can be mutable too

```
val mm = mutable.Map.empty[Int, String] ++ numWords

mm -= 2          // Map(5 -> five, 4 -> four, 1 -> one, 3 -> three)
mm += 2 -> "two" // Map(2 -> two, 5 -> five, 4 -> four, 1 -> one, 3 -> three)
```

# Maps - Key and Value Operations

```

numWords.keys      // Iterable[Int] = Set(5, 1, 2, 3, 4)
numWords.keySet    // Set[Int] = Set(5, 1, 2, 3, 4)
numWords.values    // MapLike.DefaultValuesIterable(five, one, two, three, four)

numWords.filterKeys(_ % 2 == 0)  // Map(2 -> two, 4 -> four)
numWords.mapValues(_.reverse)
// Map(5 -> evif, 1 -> eno, 2 -> owt, 3 -> eerht, 4 -> ruof)

numWords.transform { case (k, v) => s"$v($k)" }
// Map(5 -> five(5), 1 -> one(1), 2 -> two(2), 3 -> three(3), 4 -> four(4))

```

- You can also swap keys and values, but beware non-unique values

```

numWords.map(_.swap)
// Map(four -> 4, three -> 3, two -> 2, five -> 5, one -> 1)

val evens = (for (i <- 1 to 5) yield i -> (i % 2 == 0)).toMap
// Map(5 -> false, 1 -> false, 2 -> true, 3 -> false, 4 -> true)

evens.map(_.swap)
// Map(false -> 3, true -> 4)  -- oops

```

# Concrete Implementations, immutable

- `List`
- `Stream` - potentially infinite
- `Vector` - persistent immutable data structure with constant access time
- `Stack`
- `Queue`
- `Range`
- `String`
- Hash tries (`HashSet`, `HashMap`, `Set1..4`, `Map1..4`)
- `TreeSet`/`TreeMap`
- `BitSet`
- `ListMap`

# Concrete Implementations, mutable

- ArrayBuffer
- ListBuffer
- StringBuilder
- Queue
- ArraySeq
- Stack
- ArrayStack
- Array
- HashSet and HashMap
- WeakHashMap
- BitSet

# Iterators

- Lazy collection that returns a potentially different value on each `.next` call

```
val nums = List.range(1, 21)

val numsIter = nums.iterator // get an iterator from any collection

if (numsIter.length > 0) numsIter.next() // No such element exception!
```

- In this example, the call to `.length` exhausts the iterator
- Watch out for surprising outcomes like this
- Either stick to `.hasNext` and `.next()` or convert to another collection

# Views

- Lazy collection that stores up functions to run later on demand

```
val vec = Vector.range(0, 20)

val vecView = vec.view

def calcSquare(x: Int): Int = {
  println(s"Calculating for $x")
  x * x
}

val squaresView = vecView.map(calcSquare) // does nothing, yet

squaresView(2) // calls calcSquare(2)
squaresView(4) // calls calcSquare(4)
squaresView(2) // calls calcSquare(2)

val squares = squaresView.force // forces eval of new eager collection

squares
// Vector(0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121,
//         144, 169, 196, 225, 256, 289, 324, 361)
```

# Stream

- Lazy, potentially infinite collection allowing custom implementations

```
val numsFromOne = Stream.from(1) // infinite
// Stream[Int] = Stream(1, ?)

val firstTenNums = numsFromOne.take(10) // stops after 10
// Stream[Int] = Stream(1, ?)

firstTenNums.toList
// List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val factorial: Stream[BigInt] = 1 #:: factorial.zip(Stream.from(2)).
  map { case(a, b) => a * b }
// Stream[Int] = Stream(1, ?)

val firstTenFacs = factorial.take(10)
// Stream[Int] = Stream(1, ?)

firstTenFacs.toList
// List(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)
```

- Note that `Stream` is tricky, it memoizes. You must `drop` or `tail` to release earlier references and free up memory

# Exercises for Module 14

- Find the `Module14` class and follow the instructions to make the tests pass
- These exercises are a continuation of the problem started in Module 13

# Building Scala

Building Scala, Using Scala with Java

# Agenda

1. Scala Build Tools
2. SBT Intro
3. SBT Commands
4. Customizing SBT
5. Mixing Java and Scala
6. Java 8 Support (Lambdas, etc.)
7. Java Collections, Primitives and Nulls

# Maven

- Uses Zinc for incremental compilation
- Strong polyglot languages support
- Fast to create a new Scala project:

```
mvn archetype:generate  
mvn clean test
```

- Many archetypes, including Scala, Akka, Play, etc.

# Gradle

- Scala plugin: `apply plugin: 'scala'`
- DSL in Groovy
- Support for Zinc:

```
tasks.withType(ScalaCompile) {  
    scalaCompileOptions.useAnt = false  
}
```

- Dependency management:

```
dependencies {  
    testCompile "org.scala-lang:scala-library:2.12.1"  
}
```

# Other Options

- Pants: Twitter's Open Source Build Tool
  - Python DSL
  - <http://pantsbuild.github.io/>
- Apache Buildr
  - Ruby DSL
  - <http://buildr.apache.org/>
- Ant
  - XML DSL :-)
  - <http://tutorials.jenkov.com/scala/compiling-with-ant.html>
- sbt

# SBT

- Written in Scala, includes Scala like DSL
- Officially, name means nothing
- Fast Compile/Test, also Continuous
- <https://scala-sbt.org>

# Using SBT

- Interactive mode
  - help & tasks
- Common commands:
  - clean
  - compile
  - project (for multiple project builds)
  - test & test:compile
  - publish, publish-local & publish-signed
  - console & test:console
- ~ commands
- Create a new project, e.g.: sbt new scala/scala-seed.g8

# SBT Project Source Layout

- Same as Maven defaults

```
src/
  main/
    resources/
      <files to include in main jar here>
    scala/
      <main Scala sources>
    java/
      <main Java sources>
  test/
    resources
      <files to include in test jar here>
    scala/
      <test Scala sources>
    java/
      <test Java sources>
```

- Play projects do things a little differently

# build.sbt

- Easiest way in to sbt
- Scala like DSL, simplified dialect
- Now only 3 main operators to learn:

<code>:=</code>	- set a value
<code>+=</code>	- add a value to existing
<code>++=</code>	- add a sequence of values to existing

- Blank lines between expressions are now optional
- Can embed any standard Scala code in {}s

# Example build.sbt

```
name := """scala-library-seed"""

organization := "com.example"

licenses += ("MIT", url("http://opensource.org/licenses/MIT"))

javacOptions ++= Seq("-source", "1.6", "-target", "1.6")

scalaVersion := "2.10.4"

crossScalaVersions := Seq("2.10.4", "2.11.2", "2.12.1")

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "2.2.1" % "test"
)

bintraySettings

com.typesafe.sbt.SbtGit.versionWithGit
```

# Example plugins.sbt

```
resolvers += Resolver.url(
  "bintray-sbt-plugin-releases",
  url("http://dl.bintray.com/content/sbt/sbt-plugin-releases"))(
  Resolver.ivyStylePatterns)

addSbtPlugin("me.lessis" % "bintray-sbt" % "0.1.2")

resolvers += "jgit-repo" at "http://download.eclipse.org/jgit/maven"

addSbtPlugin("com.typesafe.sbt" % "sbt-git" % "0.6.4")
```

# Making a Custom Setting

In `build.sbt`:

```
val isAwesome = settingKey[Boolean]("Some boolean setting")
isAwesome := true

val totally = settingKey[String]("rating of totalness of the statement")
totally := "100% totally"

val totallyAwesome = settingKey[String]("How awesome is this project")
totallyAwesome := totally.value + {
  println("Checking project awesomeness")
  if (isAwesome.value) " awesome." else " not awesome."
}
```

# And a Custom Task

```
val checkAwesome = taskKey[Unit]("Check project awesomeness")

checkAwesome := {
  val _ = (compile in Test).value // force the test:compile task
  println("The project is " + totallyAwesome.value)
}
```

- Settings are evaluated once per sbt run (like a val)
- Tasks are evaluated once per build operation (like a def)

# Multiple Project Support

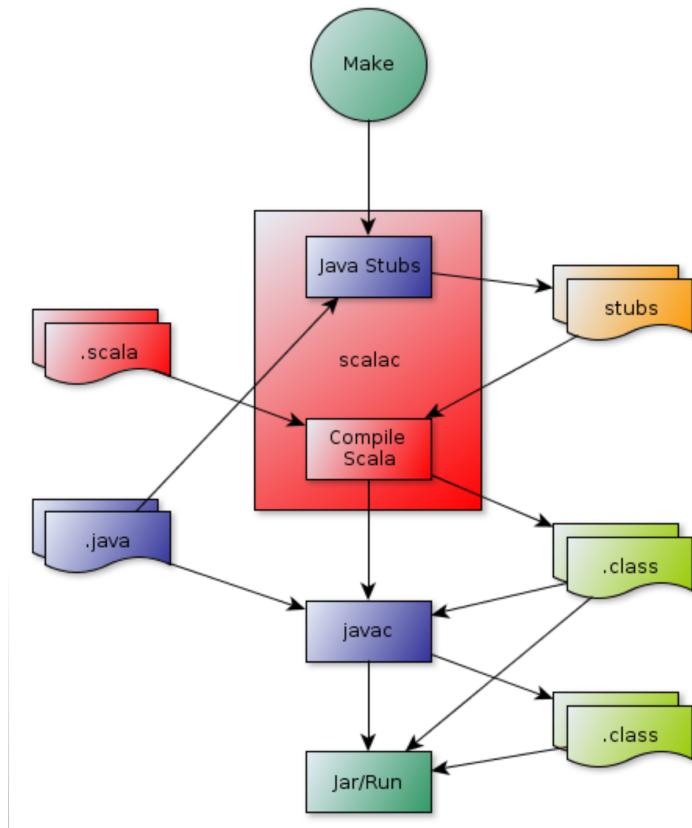
- Multiple projects can be defined in the same build.sbt file
- Assign `project` definitions to vals that represent the projects
- By default the name of the project val will define the folder used, this may be overridden

```
lazy val util = project  
  
lazy val extras = project  
  
lazy val prod = project.dependsOn(util, extras)  
  
lazy val root = project.in(file("."))  
  .aggregate(util, extras, prod)  
  .settings(aggregate in update := false)
```

# Mixing Scala and Java

- And using Java libraries
- Scala can use any Java library
- But some make more sense than others
- This includes persistence layers, web frameworks, numerical and scientific libraries, etc.
- On the other hand, many of these have Scala wrappers already if they are popular

# Scala/Java Compile Cycle



# Calling Java from Scala

- Import any Java library
- Call Java methods just like Scala
- Can leave off ()s for empty params
- Can call using infix notation
- Can extend or "with" Java interfaces
- Can instantiate Java classes
- Scala handles conversion to/from primitives
- In Scala 2.12, SAMs can be satisfied with Scala functions

# Scala 2.12 and Java 8

- Scala 2.12 requires Java 8
- Scala function literals compile to method handles
- Scala supports SAMs - Single Abstract Methods
- FunctionN are Java FunctionInterfaces
- @interface to guarantee trait can be used from Java
- Java 8 Streams API also available to use

# Java 8 / Scala 2.12 Function Compatibility

```
case class Person(name: String, age: Int)

import java.util.{Arrays, Comparator}

val javaArray = Array(Person("Harry", 25), Person("Sally", 22), Person("Tim", 33))

// the old SAM way
val comp1 = new Comparator[Person] {
  override def compare(o1: Person, o2: Person) = o1.age - o2.age
}

Arrays.sort(javaArray, comp1)

javaArray
// Array(Person(Sally,22), Person(Harry,25), Person(Tim,33))

// the new Java Lambda way
Arrays.sort(javaArray, (p1: Person, p2: Person) => p2.age - p1.age)

javaArray
// Array(Person(Tim,33), Person(Harry,25), Person(Sally,22))
```

# Handling Nulls

- Nulls are discouraged in Scala
- But they are a fact of life in Java libraries

```
val a = javaObj.methodCanReturnNull(x)  
  
a.toString          // oops  
// java.lang.NullPointerException  
  
val b = Option(javaObj.methodCanReturnNull(x))  
  
b.map(_.toString)    // safe  
None
```

- Wrapping an object from Java in `Option` will convert a reference to `Some` and a `null` to `None`

# Options to Nulls

- Going back the other way is also easy, imagine a Java method that accepts nulls

```
val s1: String = "hello"
val s2: String = null // no no no no no

val os1 = Option(s1)
val os2 = Option(s2)

os1.orNull // "hello"
os2.orNull // null
```

- `orNull` on `Option` returns either the contents or a null reference (if `None`)

# Java -> Scala Collections

```
val jl = new java.util.ArrayList[Int]
jl.add(1); jl.add(2); jl.add(3)

jl.map(_ * 2)
// error: value map is not a member of java.util.ArrayList[Int]
//       jl.map(_ * 2)

scala> import scala.collection.JavaConverters._

scala> jl.asScala.map(_ * 2)
res1: scala.collection.mutable.Buffer[Int] = ArrayBuffer(2, 4, 6)
```

- **Note:** `scala.collection.JavaConverters._` should be used, **not** `scala.collection.JavaConversions._`

# Boxed Types Trouble

- Sometimes JavaConverters on their own is not enough:

```
// Java method signature:  
public List<Integer> someJavaFunc(List<Integer> list) { ... }  
  
val sl = List(1, 2, 3)  
  
// Int is not an Integer!  
val r1 = someJavaFunc(sl.asJava)  
// Error:(28, 27) type mismatch;  
// found   : java.util.List[Int]  
// required: java.util.List[Integer]  
  
val jl2 = sl.map( new java.lang.Integer(_) ) // explicitly box first  
// jl2: List[Integer] = List(1, 2, 3)  
  
val r2 = someJavaFunc(jl2.asJava)  
// r2: java.util.List[Integer] = [1, 2, 3]
```

# Scala Traits and Java Interfaces

- Java interfaces can be used as pure abstract Scala traits
- Conversely pure abstract Scala traits can be used as Java interfaces:

```
// Scala

trait DoSomethingToString {
  def doIt(s: String): String
}

// Java
class Shout implements DoSomethingToString {
  public String doIt(String s) {
    return s.toUpperCase();
  }
}
```

- Pure Scala traits may be the best solution for Java calling Scala

# General Advice

- Java calling Scala
  - Provide empty trait based API around Scala implementation
  - Avoid function literals
  - Convert between nullable and Option
- Scala calling Java
  - Remember `scala.collection.JavaConverters`
  - Use implicit conversions (respectfully)
  - Remember the REPL

# Exercises for Module 15

- Find the `Module15` class and follow the instructions to make the tests pass
- Some of the examples for this exercise are under `src/test/java` and are Java code

# Futures

## Asynchronous Programming in Scala

# Agenda

1. Futures Intro
2. Future States
3. Composing Futures
4. Other Future Operations
5. Promises
6. Adapting Java Futures (and Other Async)
7. Common Future Patterns
8. Alternatives to Futures

# Futures

- Futures are a lightweight parallelism API provided in the Scala core API
- While not without some niggles, they are ubiquitous and reliable, and hence widely used

Common Imports:

```
import scala.concurrent._
```

this brings in the main Futures API, including Future itself

```
import scala.concurrent.duration._
```

This brings a DSL for specifying times and timeout

```
import scala.concurrent.ExecutionContext.Implicits.global
```

When working with futures, an ExecutionContext is required (usually implicit) to provide thread pool configuration. Implicits.global is handy when you don't have another source.

# Creating a Future

This is quite easy using the Future factory method

```
val futureInt = Future(1 + 1)
```

Can also use curlies for multi-line

```
val futureResult = Future {  
    val a = someLongCalculation()  
    val b = someOtherCalculation(a)  
    b.refineResult()  
}
```

If you already have results (or a failure) you can create success or failure immediately with

```
Future.successful(result)  
// or  
Future.failed(someException)  
// also  
Future.fromTry(Try(1 / 0))
```

# Some Initial Rules

- Never block or Await in a future (this includes `Thread.sleep()`) - for testing or at the very top level you sometimes have to ignore this advice, but nowhere else!
- Try not to mix `ExecutionContexts` unless you know what you are doing
- Futures start executing immediately after they are created (do if you don't want this, make it lazy with a function)
- Scala futures are not cancellable (but if you run in the context of an actor system, you can terminate that)

# Future States

(for demo purposes also I am allowed to use Await and Thread.sleep())

```
val f1 = Future { Thread.sleep(1000); 10}

f1.value      // Option[scala.util.Try[Int]] = None
f1.isCompleted // Boolean = false
// While None is in the value, the Future is not yet resolved either way

Thread.sleep(1000) // force a pause to wait for the future to complete

f1.value      // Option[scala.util.Try[Int]] = Some(Success(10))
f1.isCompleted // Boolean = true
// Completed successfully, type is whatever the Future block evaluated to

val f2 = Future { 1 / 0 }
Thread.sleep(10) // micro pause
f2.value // Option[scala.util.Try[Int]] =
// Some(Failure(java.lang.ArithmetricException: / by zero))
```

# Composing Futures

Probably the single most useful thing about Futures is that they compose with `map` and `flatMap` while remaining asynchronous:

```
val f1 = Future { Thread.sleep(1000); 10}
val f2 = f1.map(_ * 10)
```

```
f1.value      // None
f1.isCompleted // false
f2.value      // None
f2.isCompleted // false
```

```
Thread.sleep(1000)
```

```
f1.value      // Some(Success(10))
f1.isCompleted // true
f2.value      // Some(Success(100))
f2.isCompleted // true
```

`flatMap` also works asynchronously. Combining with `for` expressions is even cooler

# Futures with for

Let's say we have a simple expression like this

```
val a = 1
val b = 2
val c = 3
val s = "The answer is"

val sum = a + b + c
s"$s $sum" // res11: String = The answer is 6
```

We can apply a simple re-writing pattern to make this fully async (and this trick always works)

```
val fRes = for {
  a <- fa // where these are future results of a, b, c and s
  b <- fb
  c <- fc
  s <- fd
} yield {
  val sum = a + b + c // the exact same code as before moved into yield
  s"$s $sum"
}
```

# Async Evaluation

```
val fa = Future(1)
val fb = Future { Thread.sleep(1000); 2 }
val fc = Future(3)
val fd = Future { Thread.sleep(500); "The answer is" }

val fRes = for {
    a <- fa
    b <- fb
    c <- fc
    s <- fd
} yield {
    val sum = a + b + c
    s"$s $sum"
}

fRes.isCompleted      // false
fRes.value            // None

Thread.sleep(1000)
fRes.isCompleted      // true
fRes.value            // Some(Success(The answer is 6))
```

# Forcing a Result

Using `for` and other combinators you can avoid blocking in almost all circumstances, but eventually you will need to get the answer. Use `Await.result` or `Await.ready` for this.

```
val success = Future( 2 / 1 )
val failure = Future( 1 / 0 )

Await.ready(failure, 1.second)

failure.value

failure.failed
```

`Await.ready` waits for the Future to be resolved one way or the other before continuing, but does not evaluate the result. If you used `Await.result` in this example, an Exception would be thrown at that point.

# Other Future Operations

In addition to `map` and `flatMap` there are a number of other future operations:

```
val fa: Future[Any] = Future(10)
val fi = fa.collect {
  case i: Int => i
}

val ffi = fi.filter(_ > 11)

Await.ready(fi, 1.second)
Await.ready(ffi, 1.second)
```

```
ffi.transform(i => i * 5, {ex =>
  println(ex.getMessage)
  throw new RuntimeException("it failed to filter", ex)
})
```

# More Operations

```
val f6 = Future(2)
val f7 = f6.andThen { // for event side effects
  case Success(i) if i % 2 == 0 => println(s"it's even")
}
Await.result(f7, 1.second) // It's even, Int: 2
```

```
f7.onComplete {
  case Success(i) => println(s"It worked, and the answer is $i")
  case Failure(ex) => println(s"It failed: ${ex.getMessage}")
}
```

```
f7.foreach(i => println(s"Got an $i")) // simpler side effect for success
f7.failed.foreach(ex => println(ex.getMessage)) // and for failure
```

# Recovering from Failures

```
val failedFuture = Future.failed(new RuntimeException("nah"))
failedFuturefallbackTo(Future.successful(0))
```

or

```
val ff = Future.failed(new IllegalArgumentException("nope!"))

val fr = ff.recover {
  case _: IllegalArgumentException => 22
}
```

or

```
val ff2 = Future.failed(new IllegalStateException("Again, nope!"))
val fr2 = ff2.recoverWith {
  case _: IllegalArgumentException => Future.successful(22)
}
```

# Dealing with Multiple Futures

```
val nums = List(1,2,3,4,5)
def square(i: Int): Future[Int] = Future(i * i)

val futs: List[Future[Int]] = nums.map(square)
val futList = Future.sequence(futs)
Await.ready(futList, 1.second)
// Future[List[Int]] = Future(Success(List(1, 4, 9, 16, 25)))

val futList2 = Future.traverse(nums)(square)
Await.ready(futList2, 1.second)
// Future[List[Int]] = Future(Success(List(1, 4, 9, 16, 25)))
```

A map over a Seq, followed by a `Future.sequence` can be replaced by a `Future.traverse`

`Future.sequence` takes any `Seq[Future[T]]` and turns it into a `Future[Seq[T]]` without blocking (also works for Sets)

# Other Future Sequence Operations

```
val ft1 = Future { Thread.sleep(10); 10 }
val ft2 = Future { Thread.sleep(5); 5 }
val ft3 = Future { Thread.sleep(20); 20 }
val sft = List(ft1, ft2, ft3)
Await.ready(Future.firstCompletedOf(sft), 1.second)    // Future(Success(5))

Await.ready(Future.foldLeft(sft)(0)(_ + _), 1.second) // Future(Success(35))

Await.ready(Future.reduceLeft(sft)(_ + _), 1.second)   // Future(Success(35))
```

# Promises

A promise is the "server" to the "client's" Future

By creating a Promise you create a socket into which you can send something

You can also obtain the Future from the Promise and hand that to someone else

When you put the value (or failure) into the Promise, the Future is resolved with that outcome

```
val promise = Promise[Int]
val future = promise.future

future.isCompleted // false
future.value      // None

promise.success(10) // fulfill the promise

future.isCompleted // true
future.value       // Some(Success(10))
```

# A Broken Promise

```
val promise2 = Promise[Int]
val future2 = promise2.future

promise2.failure(new IllegalStateException("oops"))

future2.isCompleted // true
future2.value       // Some(Failed(IllegalStateException: oops))
```

A Promise allows you to easily adapt another asynchronous event into a Scala Future

# Working with Java's Futures

For Java's `CompletableFuture`, there is already an adapter

```
import java.util.concurrent.CompletableFuture
import java.util.function.Supplier

val supp = new Supplier[Int] {
  def get: Int = {
    Thread.sleep(500)
    10
  }
}
val cf = CompletableFuture.supplyAsync(supp)
```

```
import scala.compat.java8.FutureConverters._

val sf2 = cf.toScala

Await.result(sf2, 1.second) // Int: 10
```

# Future Patterns - Batching

```
def calc(i: Int): Future[Int] = Future {
    println(s"Calculating for $i")
    Thread.sleep(500)
    i * i
}

def processSeq(xs: Vector[Int]): Future[Vector[Int]] = {
    val allFutures: Vector[Future[Int]] = xs.map(calc)
    Future.sequence(allFutures)
}
```

- Scala's Execution Context will prevent too many threads running at once
- But you can still overwhelm other resources, or run out of memory, etc.
- Need a way to batch up Futures into groups

# Future Batching - foldLeft and flatMap

Combining `foldLeft` and `flatMap` makes this fairly easy

```
def processSeqBatch(xs: Vector[Int], batchSize: Int): Future[Vector[Int]] = {  
    val batches = xs.grouped(batchSize)  
    val start = Future.successful(Vector.empty[Int])  
    batches.foldLeft(start) { (accF, batch) =>  
        for {  
            acc <- accF  
            batchRes <- processSeq(batch)  
        } yield acc ++ batchRes  
    }  
}  
  
val nums = (1 to 20).toVector  
  
val f = processSeq(nums)  
Await.result(f, 20.seconds) // starts all 20 at once  
  
val f2 = processSeqBatch(nums, 2)  
Await.result(f2, 20.seconds) // waits for each 2 to finish before starting next
```

# Future Patterns - Retrying

An example failing call that eventually succeeds:

```
import scala.util.Try

var time = 0
def resetTries(): Unit = time = 0

def calc(): Int = {
    if (time > 3) 10 else {
        time += 1
        throw new IllegalStateException("not yet")
    }
}

Try(calc())    // fail
Try(calc())    // fail
Try(calc())    // fail
Try(calc())    // fail
Try(calc())    // success(10)

resetTries() // back to not working
```

# Retrying Futures (naive)

```
def fCalc(): Future[Int] = Future(calc())

val f2 = fCalc().
    fallbackTo(fCalc()).
    fallbackTo(fCalc()).
    fallbackTo(fCalc()).
    fallbackTo(fCalc())

Await.ready(f2, 10.seconds) // success(10)
```

But this is clunky, we can do better

# Retrying Futures (loop)

```
def retry[T](op: => T, retries: Int): Future[T] =  
  Future(op) recoverWith { case _ if retries > 0 => retry(op, retries - 1) }  
  
resetTries()  
  
val f3 = retry(calc(), 3)  
Await.ready(f3, 10.seconds) // failure  
  
resetTries()  
  
val f4 = retry(calc(), 5)  
Await.ready(f4, 10.seconds) // success
```

# Retrying with Back-off

```
val timer = new Timer("retrying", true)

def after[T](duration: FiniteDuration)(op: () => Future[T])(  
    implicit ec: ExecutionContext): Future[T] = {  
    val promise = Promise[Future[T]]  
    val futureHandle = promise.future  
    val task = new TimerTask {  
        override def run(): Unit = {  
            ec.execute(() => promise.success(op()))}  
    }  
    timer.schedule(task, duration.toMillis)  
    futureHandle.flatten  
}

def retryBackoff[T](op: => T, backoffs: Seq[FiniteDuration])(  
    implicit ec: ExecutionContext): Future[T] =  
    Future(op) recoverWith {  
        case _ if backoffs.nonEmpty =>  
            after(backoffs.head)((() => retryBackoff(op, backoffs.tail)))  
    }
```

# Retrying with Back-off part 2

```
var time = 0

def calc(): Int = {
    if (time > 3) 10 else {
        time += 1
        println("Not yet!")
        throw new IllegalStateException("not yet")
    }
}

val f5 = retryBackoff(calc(),
    Seq(500.millis, 500.millis, 1.second, 1.second, 2.seconds))
println(Await.ready(f5, 10.seconds))
```

# Alternatives to Futures

- The Scala Futures API is well tested, ubiquitous and performant
- If you do need more, you may want to check out a `Task` implementation like those provided by [Monix](#) or [FS2](#)
- If you need even more control, see the later section on Functors, Monads and Applicative Functors
- But at the end of the day, `Future` does 99% of what you need and is always there, and usually the underlying implementation of all these others.