# Scala Programming

QA

## Chapter Overview

### Objectives

### Exercise

## Outline

- Course Introduction
- Scala Overview
- Fundamentals:
  - The Language
  - Basic Types
- Flow
- Functions
- Object Orientation
- Inheritance
- Functional Programming
- Generics
- Collections
- Combinators
- Pattern Matching
- Testing
- Application Design in Scala
- Review

## Course Delivery

- The course will be made up from:
  - lecture material coupled with the course workbook
  - informal questions and exercises, and structured practical sessions
- The course notebooks contain all the relevant information on the syntax and features of scala we will cover.
- In the practical exercise sessions, you will be given the opportunity to experiment and consolidate what has been taught during the lecture sessions.

# Pre-Requisites

- previous experience with java and the JVM is expected
- or, *at least* equivalent programming knowledge
- this course does not teach programming!

# Reminders

## # Introductions

title: Scala Programming subtitle: Introducing Scala author: QA

geometry: margin=2.7cm geometry: a4paper

next: 02x-exercise-introduction next-title: Exercise – Introduction prev: 01-course-introduction

—

# Chapter Overview

## Objectives

- unknown
- unknown

## Exercise

- unknown

# Why Scala?

- scala is a powerful programming language with several killer libraries
  - especially relevant in BigData applications
  - increasingly used as a modernizing replacement for java replacement
- scala is a statically typed, "object-functional" language
- major object-oriented programming features
  - additions that make object-oriented design easier
  - simplified java-style OO syntax
  - *reasonably powerful* type inference
  - *many* features from typed functional languages

- scala interoperates with java
  - has access to all java libraries
  - java programs can use scala classes (with some considerations)
  - scala files compile to .class files and jar packages as expected

## Object Oriented Programming

- scala programs are phrased using a duality of paradigms, both object-oriented and functional
- in imperative programming the basic phrase is a statement: a command
  - the most frequent type of command is a state modification (a change to the memory)
  - object-oriented and procedural paradigms are both imperative
- object orientation entails phrasing program in terms of objects, ie., orienting the design around objects
  - in procedural programs, functions are asked to modify data
  - objects however combine both state (remembered data) and the behaviour (methods) which modify their own state
- moving from procedural programming to object-orientation is mostly a matter of rephrasing the major imperatives: the state-changing commands
  - we ask objects to control their own state, in the hopes that they know how it should change

## Functional Programming

- in declarative programming the basic phrase is an expression: a calculation of value
  - in order to generate *new* data, expressions recombine existing data without modifying it
  - we *declare* what we wish to see, rather than spell out the *commands* of its calculation
  - functional programming is declarative
- functional programming entails phrasing a program in terms of functions
  - but *not* procedures (functions which *command*)!
  - mathematical functions: those which take an input value and *merely* map it to an output
- functional programming emphasizes referential transparency
  - function calls can be replaced with the values they calculate
  - so that *reasoning* about how a program behaves is much easier
  - and so every component of a program may be straight-forwardly put-together without exponentially increasingly complexity

## A First Look at Scala *for General Programmers*

- here we define an object called `MyApp`
  - that is, we associate a name with some value
  - the name is `MyApp` and the value is an object

```
object MyApp extends App {
    println("Hello World")
}
```

```
$ scalac myapp.scala && scala MyApp

Hello World
```

## A Second Look

```
object MyApp extends App {
    println("Hello World")
}
```

- `MyApp` is an object
  - a *value* in the program, something you can pass around or assign
  - with a dual aspect: behaviour and state (or "remembered data")
  - the behavioural aspects of objects, ie. what they do, are known as methods
- the code between `{ }` is known as a "code block"
  - it contains a sequence of instructions (expressions or statements)
  - here the block of instructions are taken to be part of `MyApp`'s behaviour
  - more specifically, this code block is taken to be MyApp's main method
  - a method called 'main' which scala executes when it runs the program

## A First Look at Scala *for Java Programmers*

```
object HelloWorld {
        def main(a: Array[String]) {
            println("Hello World!")
        }
}
```

```
$ scalac main.scala
$ tree
.
|--- HelloWorld.class
|--- HelloWorld$.class
`--- main.scala

0 directories, 3 files

$ javap HelloWorld.class

Compiled from "main.scala"
public final class HelloWorld {
    public static void main(java.lang.String[]);
}
```

## A Second Look *for Java Programmers*

- Scala provides a (limitedly)-useful feature called delayed initialization:

```
object MyApp extends App {
    println("Hello World")
}
```

- this should seem strange to you
  - the body of an object definition is "real code"
  - rather than just more definitions
- in scala the definition body is just any sort of initialization code
  - here scala delay's the initialization of the `MyApp` object
  - and *stuffs its contents inside its main function*
  - this process is not predictable and not good practice
  - however it is useful pedagogically – and *this is why they're keeping it*

## Seeing Java from Scala

- here's javap's analysis of the MyApp version
  - it shows scala's power in understand a lot about a small amount of code

```
$ scalac myapp.scala
$ tree
.
|--- HelloWorld.class
|--- HelloWorld$.class
|--- main.scala
|--- MyApp.class
|--- MyApp$.class
|--- MyApp$delayedInit$body.class
`--- myapp.scala

0 directories, 7 files

$ javap MyApp.class

Compiled from "myapp.scala"
public final class MyApp {
    public static void main(java.lang.String[]);
    public static void delayedInit(scala.Function0<scala.runtime.BoxedUnit>);
    public static java.lang.String[] args();
    public static void scala$App$_setter_$executionStart_$eq(long);
    public static long executionStart();
    public static void delayedEndpoint$MyApp$1();
}
```

## The REPL

- scala code may be entered at it's Reading, Evaluating and Printing Loop (ie., shell)

*The Scala REPL (via cmd.exe)*

- many dynamic and declarative languages benefit from REPL-first development
  - since functional programs are built up by composing and combining expressions

  - each of these expressions may be tried out in the REPL first *before* being placed within a much larger, more complex, context
- type `:quit` to exit the REPL

## Exercise: Try the REPL

- open the command prompt or terminal
- type `scala` (assuming scala is installed!)
- try the following commands:

```
val name = "Michael"
var age = 27

println(name + " is " + age)

if(age >= 18) {
    println("You're allowed in!")
} else {
    println("You're not allowed in!")
}

def describe(name: String, age: Int) = s"${name} (${age})"

println(describe(name, age))

println(
    describe(name, age)  + " is " + (
        if(age >= 18) { "allowed in" } else { "not allowed in" }
    )
)
```

- do you notice anything interesting about scala's syntax?

## Why Scala? *Language History*

- The Numbers:
  - ~100, 000 developers
  - ~200, 000 students @ coursera
  - ~15th in Language Rankings
- Major Versions:
  - 1.0 released in 2004 : first stable version
  - 2.0 released in 2006 : first version with .NET support
  - 2.11 released in 2011 : first version to target Java 8+
  - 2.12 released in 2016 : much expanded use of Java 8 features
- The author is Martin Odersky, an academic with focus on industry problems

## Scala Versioning

- each major version is "a new language"
  - there is less focus on backwards compatibility
  - features will first be deprecated *then removed*
  - code compiled with one major version is utable on another
  - however older versions will be maintained
  - (… so, if you want to use a previous version, just use it?)

  "scala is not a monothlic company language" – Odersky
- you cannot just lean it and work *outside the community*
  - scala as a community project is much more fluid than a typical corporate product
  - therefore there are some experimental areas of the language

## Documentation

- scala-lang.org
  - home of official reference documentation
  - API docs can be quite terse and unhelpful
  - however there are tutorials and alternative styles of help
- a good reference book is recommended
  - Scala Cookbook —for a practical reference
  - Scala by Example (free from Odersky, 2014) —for a language feature reference
- there are many online videos (eg. via youtube) and a coursesa course

## Simple Build Tools

- sbt is commonly used with scala to manage compilation and general "build tasks"
  - fetching other libraries
  - running tests
  - analog of maven (, ant..)
- it's a *programatic* build manager
  - defines own DSL (domain specific language) in scala
  - you then program your build system with scala
  - (…rather than xml)
  - you can use a minimal syntax to specify build basics or write an entire scala program
- sbt can run continuously, i.e., watch for file changes and run build tasks
  - this is especially useful for running files you're editing
- there's also scaladoc, the equivalent of javadoc (with more features)
  - it generates api documentation from your code

## The IDEs

- Integrated development environments make coding easier:

  - provide code-inspect hints, help & check correct usage
  - make it easier to compile and run code you're writing
  - +lot's of plugins and tools for typical development
- pedagogically it's sometimes useful to "do the work of the IDE" ourselves
  - so that we are learning about the tools themselves, not some secondary one
- however you're free to use what you'd like, and some recommended IDEs are:
  - Intellij
  - eclipse
  - NetBeans

## The Libraries

- Core Frameworks
  - scalaz is a key library which provides extensions to core language
    * especially in a functional direction
  - Akka is heavily used for concurrent applications
- Testing frameworks
  - ScalaTest (more OO)
  - ScalaCheck (more functional)
- Web Frameworks
  - play (more OO)
  - lift (more functional)
  - scalatra (ruby-like)

```
mkdir project-dir
cd project-dir
sbt
show scalaVersion
set scalaVersion := "2.11.8"
set organization := "qa"
set version := "1.0-DEV"
session save
exit
```

```
C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example>sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
> show scalaVersion
[info] 2.10.6
> set scalaVersion := "2.11.8"
[info] Defining *:scalaVersion
[info] The new value will be used by *:allDependencies, *:crossScalaVersions and 12 others.
[info]  Run `last` for details.
[info] Reapplying settings...
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
> set organization := "qa"
[info] Defining *:organization
[info] The new value will be used by *:organizationName, *:projectId and 1 others.
[info]  Run `last` for details.
```

```
[info] Reapplying settings...
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
> set version = "1.0-DEV"
<set>:1: error: reassignment to val
version = "1.0-DEV"
        ^
[error] Type error in expression
> set version := "1.0-DEV"
[info] Defining *:version
[info] The new value will be used by *:isSnapshot, *:projectId and 3 others.
[info]  Run `last` for details.
[info] Reapplying settings...
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
> session save
[info] Reapplying settings...
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
>

>
> exit

C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example>type build.sbt
scalaVersion := "2.11.8"

organization := "qa"

version := "1.0-DEV"

C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example>sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed
[error] [C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example\build.sbt]:1
Project loading failed: (r)etry, (q)uit, (l)ast, or (i)gnore? q

C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example>sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed
[info] Set current project to sbt-example (in build file:/C:/Users/Michael/Dropbox/QA/courseware-
> diary_app/compile
[info] Compiling 1 Scala source to C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-project
[error] C:\Users\Michael\Dropbox\QA\courseware-dev\scala\scala-projects\sbt-example\diary-app\src
[error]         def default() = ArticleRenderer.render(Article("Michael", "Hello?"))
[error]                                 ^
[error] one error found
[error] (diary_app/compile:compileIncremental) Compilation failed
[error] Total time: 1 s, completed 19-Sep-2016 12:08:14
```

## Exercise 1: Using Scala

### Aim

To ensure that scala is setup correctly and that we can use it in three different ways.

### Objectives

0. Use the scala REPL
1. Create an IntelliJ worksheet
2. Setup an SBT projet
3. Write your first Hello World statement

### Use the REPL

Open a terminal window. * If you're using a mac, press the apple key and space to open spotlight and typing 'terminal' then enter. * In windows, press the windows key to open the start menu, type 'cmd' and then enter. * In ubuntu, press CTRL ALT and t.

In the console window open the REPL by typing `scala`.

REPL stands for 'Read Evaluate Print Loop'. It will evaluate commands it is given and provide the output they produce and then wait for some more input. We can test scala commands using this and view the output straight away. This is different from some programming languages where we would need to compile our code before running it. The REPL evaluates every statement and retains previous statements in memory allowing us to reuse them.

Enter some expressions, such as 2+2, and see how they are evaluated. The answer should be returned to you straight away.

```
Welcome to Scala version 2.11.6 (OpenJDK 64-Bit Server VM, Java 1.7.0_91).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 2 + 2
res0: Int = 4
```

The result of the statement has been printed to the screen. When types are not explicitly defined they are inferred by the scala interpreter.

In this case we can see that the result is 4, which has the type Int. Scala has generated an identifier for the result, 'res0' We can reuse in future commands, for example:

```
scala> res0 + 1
res1: Int = 5
```

Next, enter a function definition

```
scala> def Hello() = "hello"
```

The REPL should reply with `Hello: ()String`

This shows that it infers Hello to be a function that has no parameters and returns a string. It also shows you that the return value from a function is the last value calculated.

To execute the function we use the name.

```
scala> Hello
res2: String = hello
```

This has evaluated the value "Hello", and the REPL tells you that it has evaluated it to the string

To print the actual string, use println()

```
scala> println(Hello)
hello
```

Note two things about this line: first, that most of the time you don't have to use semicolons unless you have more than one statement on a line. And second, you don't have to use empty parentheses if a function doesn't have any arguments.

To exit the REPL type `:quit`

To see the full list of available commands in the REPL type `:help`

### Create an IntelliJ Worksheet

#### IntelliJ Setup

The scala plugin is not included in the community edition of IntelliJ by default. To include the plugin use the following steps. If you already have the plugin then just straight to 'IntelliJ Worksheet'.

To install the scala plugin: 0. Open IntelliJ 0. Click the Configure option at the bottom of the window and select "plugins" 0. Click Browse Repositories at the bottom 0. In the search bar at the top type "scala" 0. The plugins to install are the "SBT" plugin and the "Scala" plugin, select these and click "Install plugin" on the right 0. Once they have installed click close, then ok. 0. Restart IntelliJ when prompted

Note: To install and run IntelliJ on a mac you need to have the legacy Java 1.6 runtime installed.

#### IntelliJ Worksheet

Open IntelliJ and create a new project. Down the left hand side you should see a 'scala' option. Select this and create a scala (not sbt) project. The worksheets do not function correctly with SBT projects due to how they are compiled and run.

A new window will open. Give your project a name. We then need to include a JDK and Scala SDK before continuing. If there is a JDK in the drop down box, select this, we want Java 1.7+. If there is not one then click the button to the right, `New…`

Select the JDK option and then navigate to where your Java installation is located. It should pick up on the standard installation directory. Click okay and ensure that a JDK is now included.

Next select a Scala SDK, if there is no option available click the button to the side, "Create …". We want to be using at least version 2.11, so if there is not a 2.11 option then click the Download

button and select the most recent version to download. Click okay to have this installed by IntelliJ.

The final setup should look something like this:

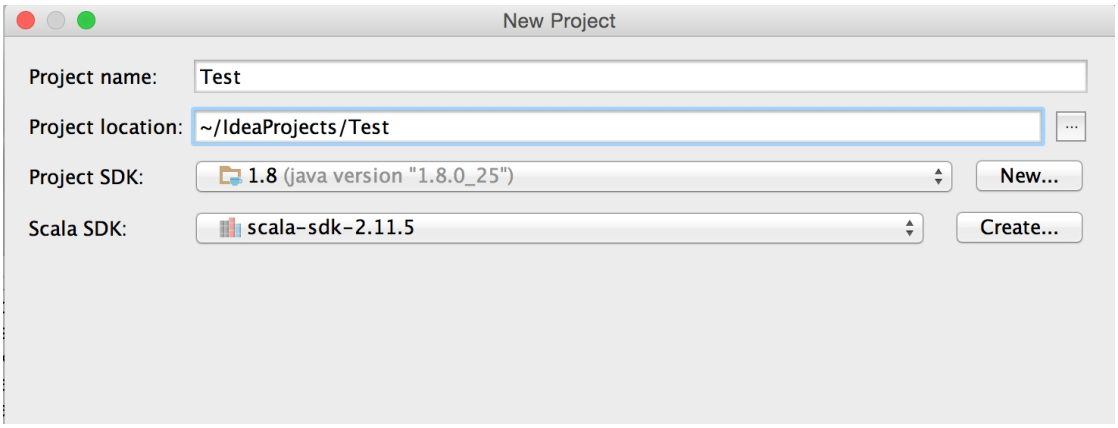


Figure 1: IntelliJ Project Screen

Once these steps have all been completed then you should be able to create the new project. This setup is only something you need to do once, when creating a new project the JDK and SDK will be available for use.

To create a Scala worksheet expand the project on the left hand side of the screen and right click the 'src' folder. Select new and Scala Worksheet. Give the file a name.

Now type your scala commands in the window. When you save the file (or whenever you change if it auto-updates are enabled) then file will be evaluated and the output displayed on the right hand side of the screen.

Create your hello world function and call it using the worksheet.

### Using SBT – Simple Build Tool

The simple build tool is similar to dependency and build managers such as maven or gradle. It also has the ability to pick up on changes in the source code and automatically rebuild a project, allowing for continuous development and integration.

SBT can be downloaded from: http://www.scala-sbt.org/

To start an sbt project create a new directory on your machine by typing into a terminal window

```
mkdir sbttest
cd sbttest
```

SBT projects require an object to be created rather than individual statements as we have been using in the REPL and worksheets. We will go into how these work later in the course. For now just create a new file in the sbttest directory called Hello.scala which contains the following

```
object hello extends App {
    println("Hello World")
}
```

Then in the terminal window type sbt. This will start up the sbt command line. From here we have another set of commands we can use such as compile, or run. Type run and your project should build itself and run outputting "hello world" to the screen.

```
[info] Set current project to sbttest (in build file:/home/sbttest/)
> run
[info] Updating {file:/home/sbttest/}sbttest...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Compiling 1 Scala source to /home/sbttest/target/scala-2.10/classes...
[info] Running MyApplication
Hello World!
[success] Total time: 3 s, completed 11-Dec-2015 20:40:31
```

The first time you run sbt it will download a set of dependencies required for it, so the initial run of any sbt project may take a while, but after this point it will be faster.

Write any scala commands inside the { … } brackets in the file, save and re-run the file from sbt. You will see that it recompiles each time before running. You will need to enclose the commands in a println() statement to output them to the screen.

For standard uses, SBT does not give you the direct feedback that the worksheets or repl does.

To exit sbt type `exit`

**If you have time...**

## The hello world tutorial (http://www.scala-sbt.org/0.13/tutorial/Hello.html) guides you through more advanced setups using the SBT build files. If you have time take a look at this and see if you can create a full SBT project.

title: Scala Programming subtitle: Core Syntax I – The Language author: QA

geometry: margin=2.7cm geometry: a4paper

next: 03.2-core-syntax-basictypes next-title: Core Syntax II – Basic Types prev: 03.1-core-syntax-language —

# Chapter Overview

## Objectives

## Exercise

# The Language

- scala programs are composed of expressions which connect *objects* by *operators*
- there are two principal types of code:
  - expressions: code that evaluates to an object (naming or producing one)
  - definitions: code that defines what objects look-like

```
3 + 2
```

- addition an expression of two objects (`3`, `2`) which calculates a third, `5`
  - `+` is a behaviour (method) of the `3` object !
- methods with non-alphanumeric names are often called operators

# Expressions over Statements

- statements are commands which have no value, ie. produce no object
  - eg., "go make me a coffee" is a command
- expressions have values, ie. produce objects
  - eg., "the height of this building added to its width" is an expression
- everything in scala, where possible, is an expression:

```
scala> println(  if(27 > 18) { "Allowed" } else { "Not Allowed" }  );

Allowed
```

- you may find this an unusual property of the language
  - however it is essential for functional programming:
  - where the passing-around of values is the central design consideration
  - so nearly all code should evaluate so that it may aid "passing around"

# Objects

- the word "object" however is very general, it could refer to:
  - values (immutables)
  - variables (mutables)
  - methods
  - types
- every object has both:
  - behaviour (methods)
  - state (remembered data, properties)

- every value is an object:
  - an identifier must refer to an object
  - ie., the contents of any identifier must be an object

## Calling Methods

- the previous expression is actually a method call:

```
3.+(2)
```

- this applies the `+` method on `3` to the argument `2`
  - or, "calls the `+` method with the input `2`"
  - the term *application* is more often used by functional programmers
    - especially when the kind of function in question *only calculates a value*

## Methods on the REPL

- the methods available on objects can be found by opening the scala REPL
  - or, "read, evalulate and print" loop – a shell
  - the REPL which provides method inspection with a tab
  - typing true `<TAB>`…

```
scala> true.
!=   #   &   &&   <init>   ==   ^   asInstanceOf   equals
getClass   hashCode   isInstanceOf   toString   unary_!   |   ||
```

*we can then type the method we'd like*

```
true.&&(true)
```

*we may omit the parentheses on infix method calls*

```
true && true
```

## Operator-Method Associativity

- which object an operator method belongs to depends on its associativity
  - operator method names that end with colons are right-associative
  - every other is left-associative
  - therefore `+` is left-associative
    - ie. "is a method of the left-side object"

```
2 + 3
2.+(3)
```

- if it existed, `+:` would be right-associated
  - ie. "is a method of the right-side object"

```
2 +: 3
3.+:(2)
```

- however `+:` isn't defined on ints

## Right-Associative Operators

*we can peak-ahead to lists to provide a real example:*

```
5 +: List(1, 2, 3)
```

```
scala> 5 +: List(3,4)
res3: List[Int] = List(5, 3, 4)
```

- the List-type objects have a `+:` method which will append an object to them
  - since `+:` is right associative, `List.+:` is called with the argument 5

*compare with +*

```
scala> 5 + List(3,4)
<console>:8: error: overloaded method value + with alternatives:
```

- this is an error – a list cannot be added to 5
  - the `5.+` method takes only integer arguments

## Deprecation

- some language features or methods will come with warnings
  - eg., `readLine()` which reads from the stdin (usually the keyboard)

```
scala> readLine("Prompt? ")
warning: there was one deprecation warning; re-run with -deprecation for details
Prompt? res47: String = Hello
```

*we can rerun scala with* ***`-deprecation`*** *to get the specific message*

```
$ scala -deprecation

scala> readLine("Prompt? ")
<console>:8: warning: method readLine in trait DeprecatedPredef is
deprecated: Use the method in `scala.io.StdIn`
```

- these warnings appear on features likely to be removed in future versions
  - here we are told to use the method on the `StdIn` object directly,
  - `scala.io` defines `StdIn` and `StdOut` for reading and writing to stdin and stdout

```
scala> scala.io.StdIn.readLine("Prompt? ")
Prompt? res1: String = Hello
```

## Values and Variables

```
val name = "Michael"
val height = 1.8
var age  = 26
```

- values (`val`) are immutable – once assigned values cannot change
- variables (`var`) are mutable – they may change after assignment

```
scala> height += 1
<console>:9: error: value += is not a member of Double
```

- methods which change the state of an object are called mutators
  - mutator methods are not defined on an immutable `Double`s

```
scala> age += 1

scala> age
res9: Int = 27
```

## The Container not the Contents

- `val` and `var` are modifiers on *identifers* not on what they contain
  - a `val` will always refer to the same object
  - if that object itself contains mutable state, it may be modified
- scala will use the type of the identifier to know what it may do:

```
scala> val builder = new StringBuilder("Hello")
builder: StringBuilder = Hello

scala> builder.append("World")
res2: StringBuilder = HelloWorld
```

- values, variables and methods share a namespace
  - any valid name could be either a value or a variable

## Changing a `var`

- when a variable *varies* it changes the object to which it refers
  - *`var`-iability* has nothing to do with how mutable *that object* is
- when incrementing an integer `var` the number to which the `var` refers changes
  - first referring to `18`, then to referring to `19`
  - the object `18` doesn't itself change!

```
scala> var age = 18
age: Int = 18

scala> age += 1
```

```
scala> age
res5: Int = 19
```

## Types

- every object belongs to a single, *principal* type
  - examples thus far have defined `val`s and `var`s without specifying their type
  - they have one however, and it is *statically* inferred
    - ie., it can be inferred from the code *as-written*
  - scala has quite general type inference compared to java so often type information can be omitted
- objects belong to both a *principal* type and many others
  - abstractly, a type *is just* the set of all objects having a common property (eg. being a Cat)
    - eg., the type Int is the set of objects $\{2^{31} - 1.., 0, .. - 2^{31}\}$
    - eg., the type Boolean is the set $\{true, false\}$
  - to be an Int is just to be in that set (math's $\in$ is scala's : )
- particular objects, eg. your Cat, will belong to many types
  - eg., it will also be an Animal, a Mammal:
  - $Mammals \equiv \{YourCat, YourDog, ...\}$
  - $Animals \equiv \{MyBird, ..., YourCat, YourDog, ...\}$
    - $YourCat : Animal$
    - $YourCat : Mammal$

## The Type of Expressions

- expressions evaluate to objects
  - therefore every *expression* has a (principal) type
- what is the type of an if-expression?

```
scala> val isChild = if(age < 18) 1 else "Nope"
isChild: Any = Nope
```

- it is the most general type which can describe both branches
  - `1` is an `Int`
  - `"Nope"` is a `String`
  - the only type `"Nope"` and `1` share is `Any`
    - the type of everything (aka. the bottom type)

## Aside: Reflection on Types

- scala has a quite powerful reflection library

*we can get hold of a type itself, and ask questions about it…*

```
scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> val intType = typeOf[Int]
intType: reflect.runtime.universe.Type = Int

scala> intType.baseClasses map { _.fullName }
res23: List[String] = List(scala.Int, scala.AnyVal, scala.Any)
```

*strings have more complex membership…*

```
scala> typeOf[java.lang.String].baseClasses map { _.fullName }
res27: List[String] = List(java.lang.String, java.lang.CharSequence,
java.lang.Comparable, java.io.Serializable, java.lang.Object, scala.Any)
```

- the lists for `Int` and `String` only have `Any` in common
  - any object belonging to `Int` is also a `scala.Int`, `scala.AnyVal` and `scala.Any`
  - scala's String type is just an alias for java's string type
  - any container (or expression) which can describe both must be of type `Any`

## Aside: The Type of Exceptions

- type checking is *static*
  - types must be inferable by *reading the code* rather than running it
  - every expression *must* have a single principal type
- question: what is the type of a *throw* expression?
  - *thrown* exceptions take any type (*not* the `Any` type…)

```
scala> val x = if(true) 1 else throw new Exception("Hello")
x: Int = 1
```

- the type of 'throw …' is whatever you define it to be
  - this is so that exceptions do not muck-up the typing system
  - exceptions are **heavily discouraged** in scala
  - they are there only to support java's style of error handling

## Aside: Type Aliases

- types may be aliased, that is, referred to by another name
  - aliasing a type does not introduce an new one
  - just another identifier which refers to the same class

```
type Age = Int
type Name = String

def person(n: Name, a: Age) = {
    println(s"Name: $n, Age: $a")
}
```

```
scala> person("Michael", 20)
Name: Michael, Age: 20
```

- aliasing *can* make complex types clearer
  - however it isn't idiomatic scala
  - can make code harder to understand

## Practical Exercise

- Q. use the REPL to find out what methods a java.util.Date object has
- Q. in the following, if chosen date is the sherlock's, increase his age by 1

```
val chosen_day = 01
val chosen_month = 10
val chosen_year = 1875

val name = "Sherlock Holmes"
var age = 27
val data = "1875-10-01,purple,hobbies:photography-portraiture;programming-functional"
```

- Q. complete the following – ie., define categories

```
println(s"My Name is $name" )
println(s"My Age is $age")
println("My Hobbies are " + categories)
```

## Solution

"'{.scala} val name = "Sherlock Holmes" var age = 27 val data = "1875-10-01,purple,hobbies:photography-portraiture;programming-functional"

val chosen_day = 01 val chosen_month = 10 val chosen_year = 1875

//Q. if chosen date is the sherlock's, increase his age by 1 val csvParts = data.split(",") val dateParts = csvParts(0).split("-")

if ( dateParts(0).toInt == chosen_year && dateParts(1).toInt == chosen_month && dateParts(2).toInt == chosen_day ) { age += 1 }

//Q. complete the following – ie., define categories val hobbies = csvParts(2) val categories = hobbies.split(":")(1)

println(s"My Name is $name" ) println(s"My Age is $age") println("My Hobbies are" + categories) "'— title: Scala Programming subtitle: Core Syntax II – Basic Types author: QA

geometry: margin=2.7cm geometry: a4paper

next: 03.3-core-syntax-flow next-title: Core Syntax III – Flow Control prev: 03.1-core-syntax-language —

# Chapter Overview

## Objectives

## Exercise

# Numeric Types

- Integers are circular (wrapping around from $2^{31}$ to $-2^{31}$), as usual

```
scala> var population = 2147483647
population: Int = 2147483647

scala> population += 2

scala> population
res10: Int = -2147483647
```

- Larger types (eg. `Long`) exist, but vars won't be promoted to them
- `Byte` ( ś127 ) and `Char` types $(0 - 255)$ have narrower ranges

```
scala> val nlByte : Byte  = 10
nlByte: Byte = 10

scala> val nl: Char = '\n'
nl: Char =

scala> nl == nlByte
res11: Boolean = true
```

- Characters are single-byte, initialized with single quotes

# Unit

- `Unit` is a special type with only one value given by `()`
- it is primarily used as a return type from functions which do not return values (ie. actions)

```
scala> def say(message: String): Unit = println(message)
say: (message: String)Unit

scala>  say("hello")  * 2
<console>:14: error: value * is not a member of Unit
        say("hello")  * 2
```

- since `Unit` has very few operations defined on it, it cannot be used in many expressions – and therefore as close a value can be to being no value at all

```
scala> val empty = ()
empty: Unit = ()
```

```
scala>  say("hello") == empty

hello
res21: Boolean = true
```

## Symbols

- symbols are *interned* strings where each instance refers to an existing object in memory
- created with a leading single-quote

```
scala> val option = 'Off
option: Symbol = 'Off

scala> val choiceA = 'Off
choiceA: Symbol = 'Off

scala> val choiceB = 'On
choiceB: Symbol = 'On

scala> option == choiceA
res12: Boolean = true

scala> option == choiceB
res13: Boolean = false
```

- a little more efficient than using the equivalent string
- most helpful in programmer-initalized data structures, eg. as keys in a `Map`

## Strings

- scala can evaluate strings differently depending on how they're defined
- double quotes defines a standard string which replaces typical escape characters:

```
scala> println("\tHello\n\tWorld")
    Hello
    World
```

- we may define other kinds of strings by prepending them with modal characters

## String Interpolation (aka Substitution)

- interpolated strings are defined by prepending an `s`, as in `s"Hello"`
- and should contain scala-expressions which will be evaluated in-place

```
val myAge = s"I am ${18 + 8} years old!"
```

- the `$` indicates you wish to take the string value (`.toString`) of the expression in `{ }`

```
scala> println(myAge)
I am 26 years old
```

- this is particularly useful for including variables in strings,

```
val name     = "Michael"
val location = "The United Kingdom"
val message  = s"$name is in $location"

scala> println(message)
Michael is in The United Kingdom
```

## String Formatting

- scala will also perform print-f style formatting on interpolated strings using the `f` modifier

```
val height = 1.8

val message = f"Height: $height%.2f"
```

```
scala> message
res65: String = Height: 1.80
```

## Raw Strings

- by default scala will interpolate escape characters (eg. `\n`) – raw strings prevent escaping – particularly useful when `\` has a special meaning eg., in windows paths or in regex
- to define a raw string use the modifier `raw` or use triple-`"`

```
val path  = raw"C:\Windows\Users\Public\Documents"
val regex = raw"\b[\w|£]+\b"
val eg    = raw"a\nb"

val anotherPath = """C:\Windows\system32\Drivers\etc"""
```

```
scala> println(path); println(regex); println(eg); println(anotherPath)
C:\Windows\Users\Public\Documents
\b[\w|£]+\b
a\nb
C:\Windows\system32\Drivers\etc
```

- and note if we didnt use `raw`...

```
scala> val stdpath  = "C:\Windows\Users\Public\Documents"
<console>:1: error: invalid escape character
```

## Regex

- raw strings are particularly useful for regular expressions
- strings define a `.r` method which converts them into a `Regex` object

```
scala> """\w+""".r
res45: scala.util.matching.Regex = \w+

scala> raw"\w+".r
res46: scala.util.matching.Regex = \w+
```

```
val numbers = """\d+""".r
val service = "ftps              990/tcp   #FTP control, over TLS/SSL"

scala> numbers.findFirstIn(service)
res24: Option[String] = Some(990)

val validate = """[^@]+@[^.]+(\.\w+)+""".r

scala> validate.findFirstIn("invalid_email")
res25: Option[String] = None
```

## Spliting, Joining and Slicing Strings

- a common operation is to divide a string by a separator into an `Array[String]`, ie. split it
  - or conversly to join together string pieces into a single string, i.e., to mkString

```
scala> "Michael John Burgess".split(' ')
res5: Array[String] = Array(Michael, John, Burgess)

scala> res5.mkString(",")
res7: String = Michael,John,Burgess
```

- arrays will be covered later however `slice` is a useful method when splitting

```
scala> "Michael John Burgess" split(' ') slice(1,2)
res11: Array[String] = Array(John)

scala> "Michael John Burgess" split(' ') slice(-1,2)
res12: Array[String] = Array(Michael, John)

scala> "be the change you want to see".split(" ").drop(2).takeRight(4)
res85: Array[String] = Array(you, want, to, see)
```

## Other Interesting String Methods

```
scala> val quote = "be the change you wish to see in the world"
quote: String = be the change you wish to see in the world

scala> quote.contains("world")
res13: Boolean = true

scala> quote.indexOf("change")
res14: Int = 7

scala> quote.indexOf("leader")
res15: Int = -1

scala> quote.toLowerCase
res16: String = be the change you wish to see in the world

scala> quote.toUpperCase
res17: String = BE THE CHANGE YOU WISH TO SEE IN THE WORLD

scala> quote.substring(7)
res19: String = "change you wish to see in the world"

scala> quote.substring(7, 7 + 6)
res21: String = change

scala> quote.reverse
res22: String = dlrow eht ni ees ot hsiw uoy egnahc eht eb
```

## Introduction to Collections

- a collection is an object which represents several pieces of data, eg. a List
  - collections are zero index, and are defined with the type of the elements they hold

```
val names = List[String]("Sherlock Holmes", "Mycroft Holmes")

scala> println(names(0))
Sherlock Holmes

scala> println(names(1))
Mycroft Holmes

scala> println(names)
List(Sherlock Holmes, Mycroft Holmes)
```

- the inner type can often be inferred

```
scala> val ages = List(10, 20, 30)
ages: List[Int] = List(10, 20, 30)
```

- note also, there is no use of `new` here which you might expect – this is related to features yet to be introduced
  - the collections chapter will cover collections in more detail

# Chapter Overview

## Objectives

## Exercise

# Conditionals

- recall that every expression evaluates to a value
  - including conditional expressions
  - its value is that of whichever branch was selected
  - its type is the supertype of both branches

```
val isAdult = if(age > 18) "Allowed" else  "Not Allowed"
scala> isAdult
res17: String = Allowed
```

- less concisely,

```
if(age >= 21) {
    println("Allowed")
} else if (age == 18) {
    println("Try Again Later!")
} else {
    println("Not Allowed")
}
```

- the type of this entire expression is `Unit`

# Introduction to Pattern matching

- we will cover pattern matching in much more detail later on, but it is useful to see it now
  - many features scala provides make much more sense in terms of this fundamental feature

```
val name = "Sherlock"
val location = "London"

val message = name match {
    case "Sherlock" => "Good Evening, Holmes!"
    case "Watson"   => "Hello, Dr. Watson"
    case _ => "Bonjour!"
}
```

```
scala> println(message)
Good Evening, Holmes!

scala> println(location match {
         case "UK" => "Welcome to the UK"
         case _    => "I'm afraid I do not speak that language!"
       })

I'm afraid I do not speak that language!
```

- the `match` expression evaluates to which ever branch is chosen
  - here we are only comparing one string to several others
  - however `matching` is much more powerful and can be used with complex data structures

## for Comprehnsions

- for comprehensions are a very general syntax to operate on a variety of types
  - in particular those which define `map` and `flatMap` (ie. monads)
  - the type which defines the interface for this is `FilterMonadic`
  - for-comprehensions are desugared into calls to these methods
- collections (eg. Strings, Lists, Arrays, etc.) define these methods
  - which enable for-comprehensions to iterate over their data

```
for(c <- "Hello") println(c)
```

- which reads: for each value extracted from "Hello", println

```
scala> for(c <- "Hello") println(c)
H
e
l
l
o
```

- comprehensions are based on set-builder notation from mathematics

## Multiple Extractables

- for-comprehensions can extract from multiple sources

```
val letters = "ABCDEF";
val numbers = 0 until 6;

for(x <- letters; y <- numbers) yield x + y.toString
```

```
scala> for(x <- "ABCDEF"; y <- 0 until 6) yield x + y.toString
res23: scala.collection.immutable.IndexedSeq[String] = Vector(A0, A1, A2,
```

```
A3, A4, A5, B0, B1, B2, B3, B4, B5, C0, C1, C2, C3, C4, C5, D0, D1,
D2, D3, D4, D5, E0, E1, E2, E3, E4, E5, F0, F1, F2, F3, F4, F5)
```

- note that *for every* $x$ the y extraction runs to completion
- `until` also returns a range object, however it is not inclusive of its argument
  - ie. (0 to 10) includes 0 and 10; (0 until 10) runs till 9

## Guards

- filtering conditions may be added to the extraction source, called guards

```
val myInts = for(
  x <- 1 to 100

  if x % 2 == 0
  if x < 50
) yield x
```

```
myInts: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8,
10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44,
46, 48)
```

- this may look similar to SQL
  - SQL is itself a *declarative* language, a property of functional languages
- it says: select an element (field) from a collection (table) if (where) a condition is true

## Ranges

- the `to` method defined for some objects returns a range
- a range is an object which can be iterated over, returning a new value on each iteration

```
1 to 5
```

- you can also define a step using the `by` method on a range

```
scala> 1 to 5 by 2
res32: scala.collection.immutable.Range = Range(1, 3, 5)
```

- ie.,

```
1.to(5).by(2)
```

## Ranges in Comprehensions

- ranges can be iterated over in a for comprehension

```
for(x <- 1 to 10 by 2) println(x * 2)
```

- recall that every expression has a value

  - the value of a for-comprehension is specified with the yield keyword

```
val ints = for(x <- 1 to 10) yield x * 2
```

- this defines a sequence of values (`ints`)
  - on each iteration the value of the `yield` expression is collected as an element of this sequence
- this can considered a sequence *defintion*
  - i.e. specifying a sequence by how it is made
- the sequences we get back from for-comphrensions are therefore typically immutable
  - that is, scala does not create an mutable collection an adds elements to it
  - but takes the for-comprehsnion as an initialization/defintion of an immutable sequence

## let-expressions

- let-expressions introduce variables local to for-comprehensions

```
val people = List("Michael Burgess", "John Doe", "Jane Doe")

val doeNames = for(
  person <- people;                          //semi-colon required...
  lastName = person.split(' ')(1);

  if lastName == "Doe"
) yield person
```

- above the assignment makes it eaier to use the lastName in the guard
  - otherwise .split would have to be called twice

## Other Loops

- the traditional while and do-while looping constructs exist in scala
  - however they are rarely used
  - the more powerful for-comprehsnion should be used wherever possible

```
scala> var population = 0
population: Int = 0

while(population < 10) {
    population += 1
}

scala> println(population)
10
```

## Aside: The Type of a For-Comprehnsion

- what is the type of this value (ie. what type of sequence is it)?
  - scala will intelligently infer "correct" sequence type via the definition of map/flatMap,
  - here scala choses an immutable ordered sequence, ie., a vector
  - this might lead to some surprising results

*compare,*

```
scala> val str = for (c <- "Hello World") yield c
str: String = Hello World
```

*and,*

```
scala> val chrs = for(c <- "Hello") yield c + ""
chrs: scala.collection.immutable.IndexedSeq[String] = Vector(H, e, l, l, o)
```

- the first a `String` and the second a `Vector`
  - in the first `yield`, `c` is a character; whereas `c + ""` is a `String`
  - the first case defines an immutable ordered sequence of characters (ie. a `String`)
  - the second case defines an immutable ordered sequence of strings (ie. a `Vector` of `String`s)

## Practice Exercise

```
val mixed = List(1, "2", false, 3.4)
val people = List("Michael 10", "John 20", "Watson 40")
```

- Q. print out every element of `mixed`
- Q. print out the names of the adults in `people`
  - HINT: .toInt, .split
- Q. use a for-comprehension to create a sequence of 5 `'x'`s called `row`
  - HINT: the input must be the same length as the output, in this case
- Q. print out `row` on one line
  - HINT: use print()
- EXTRA:
  - Q. use a for-comprehension to create a sequence of 5 rows called `grid`
  - Q. print out `grid` in a grid "'

## Solution

```
val mixed = List(1, "2", false, 3.4)
val people = List("Michael 10", "John 20", "Watson 40")

// Q. print out every element of mixed
for(item <- mixed) println(item)

// Q. print out the names of the adults
for(person <- people;
```

```
    if person.split(" ")(1).toInt >= 18) println(person.split(" ")(0))


// Q. use a for-comprehension to create a sequence of 5 'x's called row
val row = for( i <- 1 to 5) yield 'x'

// Q. print out row on one line
for(cell <- row) print(cell)

// EXTRA:
// Q. use a for-comprehension to create a sequence of 5 rows called grid
// Q. print out grid in a grid

val grid = for(i <- 1 to 5) yield row
for(row <- grid) { println(); for(cell <- row) print(cell) }
```

# Chapter Overview

## Objectives

## Exercise

# Code Blocks

- code blocks ({ }) group sequences of expressions
- the value of code block is value of its last expression
  - ie., a code block "returns" its last expression
- expressions can be sequenced with newlines or semi-colons
  - newlines are idiomatic
- since code blocks are expressions, they may be assigned

```
val amount = {
    val ratio = 2.5
    val distance = 12

    distance * ratio
}

println(amount)
```

- here the code block is evaluated immediately so that amount is 30

# Functions

- functions are named code-blocks
  - evaluated after definition, with free variables (ie. parameters)

```
def sum(a: Int, b: Int) = {
  a + b
}

scala> sum(5,10)
res33: Int = 15
```

- `sum` defined (`=`) to be a codeblock in the scope of parameters `a: Int` and `b: Int`
  - note the type annotations (`:`) following the identifiers

*or, with return type*

```
def sum(a: Int, b: Int): Int = a + b
```

- contrast functions with methods which are bound to an object
- note that braces just group expressions, one expression alone does require them

## Practice Exercise

- Q. write a function which determines if an int is divisible by 2 called `isEven`
- HINT: $x\%2$ is 0 when $x/2$ has no remainder

## Solution

- most succinctly this can be written as,

```
def isEven(x: Int) = x % 2 == 0
```

- note for single expressions the `{}` may be omitted
  - isEven is defined to be a single expression whereas a code block is a sequence of expressions
- most verbosely we could write,

```
def isEven(x: Int): Boolean = {
  if(x % 2 == 0) {
        true
  } else {
        false
  }
}
```

- the return type can often be omitted
  - since `x % 2 == 0` evaluate to `true` or `false`, `if` is not necessary

## Returning Unit

- if you omit an equals sign when defining a function the return type `Unit` is implied
  - `Unit` is the type of the value `()` which means "nothing"

  - the ability to omit the `=` may be removed from later versions and should be avoided anyway

```
scala> def sum(a: Int, b: Int) {
         a + b
       }
sum: (a: Int, b: Int)Unit
```

- note after the definition of `sum` in the REPL scala provide the type of the `sum` function
  - the return type is `Unit` (contra `Int` which is what we expect)
- a returned Unit displays nothing at the REPL
  - and printing shows `()` – the type `Unit`'s single value

```
scala> sum(50, 10)

scala> println(sum(50, 10))
()
```

## Default Arguments

- parameters can take defaults, supplied on function defintion

```
def repeat(str: String, N: Int = 10) = {
    var result = ""

    for(i <- 1 to N) result = result + str

    result
}

scala> repeat("Ho! ", 3)
res0: String = "Ho! Ho! Ho! Ho! "

scala> repeat("Ho! ")
res1: String = "Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! "
```

- parameters take their defaults when not supplied at calling

## Anonymous Functions (lambda)

- functions do not have to be name and may be used anonymously

```
val printer = { (msg: String) => println(msg) }
```

- here, printer is a assigned the expression `RHS => LHS`
  - it is the fat-arrow which establishes the LHS as a function of the RHS parameters

*without optional braces,*

```
scala> val printer = (msg: String) => println(msg)
printer: String => Unit = <function1>

scala> printer("Hello")
Hello
```

- note the *type* of printer, `String => Unit`
  - the arrow in *types* indicates a function

## Argument Passing Sytle

```
def appendMark(phrase: String, mark: String = "?") = phrase + mark
```

- parameters can be passed by position

```
scala> appendMark("what time is it", "!!?")
res50: String = what time is it!!!?
```

- parameters can by passed by default

```
scala> appendMark("what time is it")
res51: String = what time is it?
```

- parameters can be passed by name – and therefore out of order

```
def config(host: String, user: String, pass: String) =
    println(s"$user:$pass@$host")

scala> config(user = "Michael", pass = "Test", host = "UK")
Michael:Test@UK
```

## Variadic functions (the splat operator)

- variadic functions are those of a variable number of arguments
  - to defined a varaidic parameter append a star to the type

```
def sum(numbers: Int*) = {
  var counter: Int = 0
  for (n <- numbers) {
    counter += n
  }
  counter
}

scala> sum(1,2,3,5,6)
res54: Int = 17

scala> sum(1,2,3)
res56: Int = 6
```

- any number of arguments may be passed to `sum`
  - when we do so, scala collects these parameters and puts them into a *sequence* `numbers`

```
scala> def id(x: Double*) = { x }
id: (x: Double*)Seq[Double]
```

## Passing Sequences to Variadics

```
//another way of defining sum

scala> def sum(nums: Int*) = nums reduce { _ + _ }
sum: (nums: Int*)Int

scala> sum(1,2,3)
res2: Int = 6

scala> val ages = List(18, 20, 22)
ages: List[Int] = List(18, 20, 22)

scala> sum(ages : _*)
res3: Int = 60
```

- anotating with : `_*` *in a variadic context* passes the sequences "straight through"

## Recursion

- In functional programming recursion is preferred to looping
  - data structures are analyzed by passing them to a function which call itself
  - on each subsequent call a smaller piece of the data structure is passed until we get to a base element
- to write a recursive function you should first define the terminating (or base case) condition
  - this condition should return a single value
- the 'else' of this condition should be a call to the function you have just defined

## Practice Exercise

- Q. write the factorial function using recursion
- HINT:
- $fact(0) == 1$
- $fact(n) == (n-1) * fact(n)$
- $fact(n) == (n-1) * (n-2) * ... * 1$

## Solution

- a typical recursive problem
  - establish terminating condition
  - pass around an accumulator that represents a growing/partial solution to the problem
  - can use default parameters to initialize accumulator
  - here the accumulator is called `product`

```
def fact(a: Int, prd: Int = 1) = if(a == 0) prd else fact(a - 1, prd * a)
```

- note also that this *isn't* the factorial function
  - the factorial function takes *one* argument
  - here we *could* call `fact(5, 2)` which would be `2 * 5!` and not `5!`

```
def factorial(a: Int) = {
  def fact(a: Int, prd: Int) = if(a == 0) prd else fact(a - 1, prd * a)

  fact(a, 1)
}
```

## Functional Programmers prefer Internal Iteration

- functional programmers prefer *declarative* programming
  - where goals are expressed, not the mechanism of how they are achieved
  - (SQL is a good example)
  - this often means hiding iteration, or "handing it off" to some other function to perform for us

```
def factorial(n: Int) = (1 to n).product
```

- often there are functions like `product` which do the work of aggregating information
  - here a range `(1 to n)` is used for its `product` method
  - it's better to rely on this declarative style where possible
    - unless very inefficient
  - external iteration describes the result in terms of instructions to perform (ie. *imperatives*)
  - internal iteration describe the result in terms of what it ought to look like / be (ie. it *declares*)
- a declarative style is especially important to functional programming
  - if x **is** y then x is substituable for y
  - whereas with x **does** a, b, c its not really clear what it can be replaced with

## def vs val

- a def-declarations can be used to define, as with val with the same semantics for literals

```
def helloByDef = "hello"
val helloByVal = "hello"
```

```
scala> println(helloByDef)
hello

scala> println(helloByVal)
hello
```

*however, consider*

```
def nowByDef = new java.util.Date
val nowByVal = new java.util.Date

scala> nowByDef
res46: java.util.Date = Fri Nov 26 21:00:00 GMT 2016

scala> nowByVal
res47: java.util.Date = Fri Nov 26 21:00:01 GMT 2016

scala> nowByDef
res48: java.util.Date = Fri Nov 26 21:01:30 GMT 2016

scala> nowByVal
res49: java.util.Date = Fri Nov 26 21:00:00 GMT 2016
```

## def, val call-time and use-time

- `def` expressions assign their right-hand-side lazily
  - ie. the RHS is evaluated at call-time, not definition-time
- the RHS of a `val` expression is evaluated when it is assigned
  - remains forever that assigned value
  - ie. `val`s have strict initialization and `def`s lazy
- a function's `def` is therefore saying "evaluate this code block whenever it is *used* rather than *defined*"

```
scala> val age = {
          val start = 26
          val next = 1
          start + next
    }
age: Int = 27
```

## Lazy Vals

- recall the distinction between `val` and `def`

```
val strict = new java.util.Date //11:00:00 am

strict  //11:00:00 am
strict  //11:00:00 am
```

```
def deffered = new java.util.Data // ---

deffered //11:00:00 am
deffered //11:00:55 am
```

*consider a* ***lazy val****,*

```
lazy val deferOnce = new java.util.Data // ---

deferOnce //11:00:00 am
deferOnce //11:00:00 am
```

- `val`s take their initial value strictly, ie. their RHS is evaluated at define-time
- `def`s acquire their values at call-time, ie. their RHS is eval'd on use
- `lazy val`s acquire their value on *their first use* and remain that value

## The meaning of Lazy

- strictness vs lazyness
  - concerns when is value calculated
  - ie., when the expression is evaluated
  - lazy = late, strict = early
- lazyness implies expensive or real-time operations
  - lazy values particularly imply 'expensive but determinate' (eg. getting a web page)
  - lazy values esp. are "calculated on need" despite only having one final value
- strictness implies cheap or not time-dependent
  - eg. the value of 2*2 does not depend on time

## Aside: Referential Transparency

- one of the primary goals of pure functional programming is referential transparency
  - when an expression is substitutable for the value it evaluates to
  - ie. what an expression refers to (its value) is transparent
- transparent expressions are much easier to reason about and understand their effects

```
val heightM = 1.8
val height = heightM * 100
var age = 26
age += 1

println(s"I am $height cm and $age years old")
```

- values preserve referential transparency and variables break it
  - everywhere in our application `height` can be substituted for 1.8
  - however we cannot substitute `age` for `26` because at one point age increases in size to 27
- when expressions are not referentially transparent they are said to have 'side effects'

  - side effects are any effect that cannot be achieved by mere calculation
  - any IO (printing, writing to file, reading input, writing to a socket, etc.)
  - any *observable* modification to memory

## Aside: Higher-Order Functions

- since functions can be used as literal values, we may pass them as arguments
  - take the repeater before, now rather than just concatenating (+) on each go around, let the caller decide what operation is repeated

```
def repeat(str: String, N: Int, fn: (String, String) => String) = {
    var result = ""

    for(i <- 0 to N) result = fn(result, str)

    result
}
```

```
scala> repeat("Ho ", 10, { (l: String, r: String) => l + r } )
res19: String = "Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho "

scala> repeat("Ho ", 10, (l, r) => l + r )
res19: String = "Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho "

scala> repeat("Ho", 10, (l, r) => l + r + "! " )    // lambda type inferred from repeat()'s argum
res26: String = "Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! "
```

## Aside: Aliases and Defaults

- a function type, like any other, can be aliased

```
type Repeatable =  (String, String) => String

def repeat(str: String, numTimes: Int, fn: Repeatable) {
    var result = "";

    for(i <- 0 to numTimes) result = fn(result, str)

    result
}
```

- function-type arguments can take defaults too

```
def repeat(str: String, N: Int, fn: Repeatable = (l,r) => l + r)  = {
    var result = "";

    for(i <- 0 to N) result = fn(result, str)
```

```
    result
}

scala> repeat("ho! ", 10)
res1: String = "ho! ho! ho! ho! ho! ho! ho! ho! ho! ho! "
```

## Practice Exercise

- Q. define a function which takes a csv string and returns its parts
- Q. define a variadic getCred function
  - given several String paramters it will just return a List of the first two
  - HINT: String*
- Q. define a login function that takes an authenticating function and a List of Strings
  - the function passed should take List of Strings and return Bool
  - pass the List of Strings to the auth function
  - if it comes out true print "ALLOWED" otherwise "DENIED"
  - HINT: one of its parameters will have type: List[String] => Boolean
- Q. combine all these functions to get "ALLOWED"
  - HINT: start with the string "myuser,mypass,myemail@example.com"
  - HINT: the first argument of login is a function, yet to be defined – this may be a lambda

## Solution

```
// Q. define a function which takes a csv string and returns its parts
def parseCSV(csv: String) = csv.split(",")

// Q. define a variadic getCred function
// given several String paramters it will just return a List of the first two
// HINT:  String*

def getCred(args: String*) = List(args(0), args(1))

//ALSO:  def getCred(args: String*) = List(args.slice(0,2) : _*)

// Q. define a login function that takes an authenticating function and a List of Strings
//.. the function passed should take List of Strings and return Bool
//.. pass the List of Strings to the auth function
//.. if it comes out true print "ALLOWED" otherwise "DENIED"
//HINT: one of its parameters will have type:  List[String] => Boolean

def login(auth: List[String] => Boolean, cred: List[String]) =
    if(auth(cred)) {
```

```
        println("ALLOWED")
    } else {
        println("DENIED")
    }

// Q. combine all these functions to get "ALLOWED"
//HINT: start with the string "myuser,mypass,myemail@example.com"
//HINT: the first argument of login is a function, yet to be defined -- this may be a lambda

// ALSO:
login((s: List[String]) => s(1) == "pass", getCred(parseCSV("user,pass,email")))
```

# Exercise 2: Core Syntax

## Aim

To get some experience with basic language features in scala and experiment with what we have seen so far.

## Objectives

0. Use var and val syntax and infer their types.
1. Test conditions using if-expressions.
2. Solve some problems using for-comprehensions.

Open a new worksheet for these exercises. This will help you see the program develop and allow you to refer back to previous parts of the exercise. The worksheets are a 'work in progress' both for eclipse and IntelliJ. They will sometimes not recognise valid scala. If you are sure something should work then copy the relevant line to the REPL and see if it is a bug you've found. At this stage you shouldn't find many bugs, but as the course progresses, especially with objects, the work sheets can sometimes cause some issue.

## Quick Review

```
val name = "Sherlock Holmes"
var age = 27
val data = "1886-09-20,dark purple,hobbies:photography-portraiture;programming-functional"

val day = 20
val month = 9
val year = 1886
```

**1. use the REPL to find out what methods data has**

**2. if the given day, month and year is Sherlock's birthday, increase his age by 1**

HINTS: The 0th element of an Array parts, is parts(0) You'll need the methods .split and .toInt

**3. Print out the name, age, favourite colour and hobbies of Sherlock**

## Vars and Vals

Create a new val and var, observe the types returned from the expressions.

```
val imVariable = 5
var mVariable = 5.0
```

Reassign the values for n and m to be 10.0 and 20 respectively. Does it work?

`val`s are 'immutable' variables. Their values can't change.

We can change the value of `mVariable` to anything we like as long as the type doesn't change, but once a `val` is declared it will always hold the same value.

Create some more `val`s in the worksheet. Look at the type returned for each of these. The compiler will infer the type where it is able.

You should already have `val imVariable = 5; var mVariable = 5.0;`and a reassignment for `mVariable`.

```
val name = "Alice"
val c = 'a'
val f = 3.0f
val b = true
val m = mVariable
val n = imVariable
```

Some operations will take different types. Try running the following expressions and see what the results are.

```
name + c
c + n
name + n
name + b
b & false
b | false
f + m
f + m + n
n + f
```

When you add strings and characters together you get a different result from when you add a character and a number, or a string and a number. Characters can be interpreted as numeric references (it uses ASCII to determine what number the character is). Adding strings and numbers together will output another string. Always make sure of your type when calling methods on objects.

It is considered good practice to let the compiler infer type for scala expressions. Sometimes you will need to override this. We can do that when the val is declared. Declare these two vals and see what happens when you try change the value between 10, 10.0 and 10f.

```
val thisIsADouble : Double = 10;
val thisIsAFloat : Float = 10;
```

Code blocks are sequences of expressions using braces to group statements together { … }

An example of this is:

val result = {val a = 20; val b = 5; a/b}

Code blocks can be on more than one line, the semi-colons are only needed when more than one statement is on the same line. The last line in the sequence is taken as the value of the entire expression.

Create some code blocks to evaluate the following (you should create the `val`s required inside the code blocks):

- `y = m * x + c` where m is 1.0, x is 10 and c is 15
- `total = input + (input * 20%)` where the input is 10.00
- `fullname = first name <SPACE> surname`
- `a = 5, b = 10, print a + b`

What is the return type of the final block?

## Control Flow

An if statement in scala has the form:

“‘{.scala caption=“Exercise: if-expressions} if (c == ‘a’) println(”character is a“) if (c != ‘a’) println(”character is not a“)

if (true) “Yes” else “No” “‘

Write an if-expression that determines whether a given year is a leap year. The rules for a leap year are: * a year is divisible exactly by four and * not divisible exactly by 100 and * not divisible exactly by 400

Create a function, `isLeapYear`, which uses your if-expression.

## For Comprehensions

Write a for-comprehension that tests each of the years from 1960 to the present to see if they are leap years. Hint: Use a Range from 1960 and reuse your `isLeapYear` function.

Write a for-comphrensions that prints out every letter in hello with stars in between

For example: *hello world*

**Write a for comprehension with a guard that yields a list of all even numbers between 1 and 100.**

title: Scala Programming subtitle: Object Orientation author: QA

geometry: margin=2.7cm geometry: a4paper

next: 04.x-exercise-oo next-title: Exercise – Object Orientation prev: 03.x-exercise-core-syntax

—

# Object Orientation

- almost everything in scala is an expression concerning objects
- objects include:
  - functions
  - methods
  - values
  - literals
- being an object entails you are an instance of a type, which defines:
  - your methods (behaviour)
  - your properties (data, state)
- object oriented programs are phrased as series of requests between objects
  - these requests control the state change of the objects and ensure it changes in predictable ways

# Scala vs. Java

- each scala file may declare a package
  - that package may define as many types (classes, traits, ..) as it wishes
- there are no access modifiers on classes nor any enforced *file* naming convention
  - scalac will generate several .class files per .scala file
- for java, type defintions are usually comprised of 100+-lines and so several per file are forbidden
  - however many class definitions scala will be one line, and most under 50
- compared to java, interfaces, statics and constructors are all vary in syntax and semantics
  - but are close-enough in simple cases to map to java's implementation

# Classes

```
class Counter {
    private var count = 0
    def increment() = { count += 1 }
    def value() = count
}
```

- here we define a class which may be `new`'d to create an instance

- what is the type of `increment` ?
  - *Unit* – oops!
- assignment is a statement: it should be treated as having no value
  - but since scala is a functional language, we want everything to have a value
  - therefore it evaluates to `Unit`

```
scala> var c = 0
c: Int = 0

scala> (c += 1)

scala> (c += 1) *2

<console>:13: error: value * is not a member of Unit
```

## Java Inspection of a .class

```
class Person {
    val name : String = "Bob"
    var age : Int = 21

    def birthday = age += 1
}
```

- scalac, javap to insepect class...

```
mjburgess@mjb:classes$ scalac person.scala
mjburgess@mjb:classes$ tree
.
|--- Person.class
`--- person.scala

0 directories, 2 files

mjburgess@mjb:classes$ javap Person.class

Compiled from "person.scala"
public class Person {
  public java.lang.String name();
  public int age();
  public void age_$eq(int);
  public void birthday();
  public Person();
}
```

## Getters and Setters

- javap illustrates existence of getter and setter methods:
  - scala does not allow direct access to the value of attributes
  - every access is – in principle – mediated by a method
    - this may be optimized away
  - these getter/setter methods allow more fined grained control and access to the object's state

## Method Sugar

*consider the modification of age on an instance of Person*

```
val p = new Person
p.age = 10
```

- `=` is an infix operator which gets desugar'd to,

```
p.age_=(10)
```

- attributes are publically accessibly by default
  - ie. available to "the world outside the instance"
- since scala *strongly encourages* immutability, accessibility is a none issue
  - traditional OO uses encapusaltion to control *state change*
    - if there is no state change encapsulation is a significant hinderance
    - consider the difficulties that would arise from private data in tables for SELECT statements
  - note: there are no setters on `val`s! as they can't change…

## Custom Getters and Setters

- we cannot override the scala-provided getters and setters ourselves
  - we can provide a `def`-interface to a private `var`
  - if named with the correct infix operator behaves as expected

```
class Person {
      private val name : String = "Bob"
      private var _age : Int = 21

      def birthday = _age += 1

      def age = _age

      def age_=(a: Int) {
          _age = if(a < 18) 18 else a
      }
}
```

```
scala> val me = new Person
me: Person = Person@3eea3111

scala> me.age = 18
me.age: Int = 18
```

- note the addition of `private` to restrict access to `_age` to *within the instance itself*
- scala will still generate getters and setters for our private attributes (including `_age`)
  - though these may be optimized away

## Simple Constructors

- scala classes may define their primary constructor as paramters on the type itself
  - compared to java, the amount of boilerplate typing here can be reduced significantly

```
class Person(val name: String, var age: Int)
```

- (this includes the getters and setters for `name` and `age`)
- arbitrary initialization code may be within the code block of a class-definition expression
  - this forms part of the primary constructor

```
class Person(val name: String, var age: Int) {

  println("Hello World")

  def x(y: Int) = y
}

scala> new Person("Me", 100)
Hello World
res5: Person = Person@7cfcef1f
```

## Secondary Constructors

- classes may define additional constructors
  - their parameter list should be a subset of the primary constructor
  - useful if there are multiple ways of reasonable initialization
- secondary constructors are named `this`

```
class Database(val host: String, val user: String, val pass: String) {
    def this(user: String, pass: String) = this("localhost", user, pass)
}
```

*thus we may create an instance as,*

```
scala> val localDb = new Database("mjb", "pwd")
localDb: Database = Database@1361fefb
```

```
scala> val remoteDb = new Database("192.0.0.2", "mjb", "pwd")
remoteDb: Database = Database@2bfd5f0e
```

## Method Overloading (ad hoc polymophism)

- multiple methods of the same name may be defined if their parameter signature differ

```
class Room {
    val myKey = 12

    def open() = println("The Door Rattles!")

    def open(theirKey: Int) = println(
        if (theirKey == myKey)
            "The Door Opens!"
        else "The Key Breaks!"
    )
}

scala> shed.open()
The Door Rattles!

scala> shed.open(12)
The Door Opens!

scala> shed.open(14)
The Key Breaks!
```

- scala will select the correct `open` depending on the parameters you supply
  - this is essentially a form of polymophism: the behaviour of `open` is type-dependent

## Operator Methods

- operator methods may be overloaded

```
class Fraction(t: Int, b: Int) {
    val top: Int    = t / gcd(t, b)
    val bottom: Int = b / gcd(t, b)

    private def gcd(x: Int, y: Int): Int =
        if (x == 0)
            y
        else if (x < 0)
            gcd(-x, y)
        else if (y < 0)
            -gcd(x, -y)
        else
```

```
            gcd(y % x, x)

    def +(that: Fraction) = new Fraction(
        this.top * that.bottom + that.top * this.bottom,
        this.bottom * that.bottom
    )
}

var x = new Fraction(0, 1)

for (_ <- 1 to 3) x += new Fraction(3, 8)

scala> println(s"${x.top}/${x.bottom}")
15/8
```

## objects

- scala uses the keyword `object` to create singleton instances, meaning:
  - the `object` keyword defines an implied anonymous class (`Name$`)
  - and creates a single unique instance of that class (`Name`)

```
object UnitedKingdom {
    val capital = "London"
    val population = 6E7

    var eduImmigration = 1.9E5
    var wrkImmigration = 2.8E5
    var emmigration = 1E5

    def netImmigration() =
        UnitedKingdom.eduImmigration
    +   UnitedKingdom.wrkImmigration
    -   UnitedKingdom.emmigration

    def annualWorkMigration() =
        0.1 * scala.math.round (
            UnitedKingdom.wrkImmigration * 1E4 / UnitedKingdom.population
        )
}

scala> UnitedKingdom.annualWorkMigration()
res10: Double = 4.7
```

- the single instance is reffered to by its name, here `UnitedKingdom`

## Companions

- objects with the same name as classes are called 'companions' (to those classes)

```
class Car(private val colour: String = "Red")

object Car { }
```

- all instances of `class Car` access *the same* companion object `object Car`
  - companion objects can call any members
  - especially, for example, private constructors
  - cf. java: companion objects essentially provide static methods and variables to the class
- *however* the class cannot access the object's methods via `this`
  - must *always* specify the object name to access its methods or attributed, eg `UnitedKingdom.capital`

## Callable Companions

- apply methods on objects define their call behaviour
  - that is, what happens when the call operator `()` is used on them

```
object UnitedKingdom {
    val capital = "London"
    def apply() = println("The capital of the UK is " + UnitedKingdom.capital)
}

scala> UnitedKingdom()
The capital of the UK is London
```

- this syntax looks like "a class being called" (rather than new'd)
  - rather the companion object is providing the functionality

## Companions as Factories

- we can use the object's apply method to create a factory for our class
  - ie. we can hand-off instancing (newing) our class to our companion object
  - and scala often does idiomatically
- nb: a factory is any method returning a newly instanced object

```
object Car {
  def apply(make: String) = {
    new Car(make, true)
  }
}

class Car private (val make: String, val isSportsCar)
```

```
val c = Car()
```

- notice when using the object's apply, we do not need new keyword
  - we are *calling the object*
  - `class Car` defines it constructor as `private`
    - so it may only be instanced by its companion object!

## Case Classes

- the `case` keyword creates classes with a companion object automatically defined

```
case class Car(val make: String)

scala> case class Car(val make: String)
defined class Car

scala> Car("MyCar")
res1: Car = Car(MyCar)
```

- case classes provide many additional features and are very useful for data classes
  - ie. when you're class is just some representation of *typically immutable* data ("a model")
- in particular, case classes (via the companions they generate)...
  - can be pattern matched
  - define hash-code and equals
  - define a better toString
  - automatically define getter methods for the constructor arguments
  - may not be inherited from or inherit from anything else

## Pattern Matching with Case Classes

- one of the biggest motivations for case classes is pattern matching
  - patttern matching can extract the data within instances easily
  - for an instance to be used in a `match` context, a companion must provide special methods
  - case classes generate this methods automatically

```
case class Person(val name: String, val age: Int)
case class Pet(val name: String, val age: Int)

val me = Person("Sherlock Holmes", 27)
val dog = Pet("Winston", 5)

def greeting(thing: Any) = thing match {
    case Person(name, age) => s"Hello, $name!"
    case Pet(name, age) => s"Good $name!"
}
```

```
scala> println(greeting(me))
Hello, Sherlock Holmes!

scala> println(greeting(dog))
Good Winston!
```

## ASIDE: Case Objects

- case *objects* may be defined: case classes which have only one instance

```
sealed trait Colour     //`sealed` restricts inheritance to within the file alone
case object Red extends Colour
case object Green extends Colour
case object Amber extends Colour
```

- this can be read as defining three objects each of their own type (Red, Green, Blue)
  - all belonging to the supertype Colour

```
def action(signal: Colour) = signal match {
    case Red => "Stop!"
    case Green => "Go!"
    case _ => "Wait!"
}

scala> val theSignal = Amber
theSignal: Amber.type = Amber

scala> println(action(theSignal))
Wait!
```

- advanced for now: more on pattern matching later

## Exceptions

- exceptions are *heavily* discouraged in scala programming
  - they are mostly for compatibility with java
  - the syntax is broadly the same

```
def brokenFunction() = throw new Exception("Some Error!")
```

- to try/catch:

```
import java.io.IOException
try {
    brokenFunction()
} catch {
    case io: IOException => println("bad io")
```

```
    case e:    Exception => println("general error")
}
```

- which uses pattern matching on types (discussed later)
  - note the `e`, `io` labeling is arbitrary

## Packages

- defined using `package` keyword
  - provide namespace grouping, ie. identifier prefixes
  - classloading handled by compiler
  - the compiler will generate a folder structure which corresponds to the package structure
- packages cannot be defined at the REPL

```
package Outer {
  package Inner {
    case class Person(val name: String)
  }
}

import Outer.Inner

object Application {
    def main(a: Array[String]) = println(Outer.Inner.Person("Mycroft"))
}
```

```
$ scalac person.scala use.scala
$ scala Application
Person(Mycroft)
```

## Braces are Optional

- nesting is automatic without braces:
  - everything below the keyword is within the parent

```
package Outer
package Inner

case class Node(val weight: Int)

import Outer.Inner

object Application {
    def main(a: Array[String]) = println(Outer.Inner.Node(10))
}
```

## Importing

- to include features defined in a package (, module = file, …) use `import`
  - the import path specified may then be omitted from future uses of imported types

```
import java.util.Data
import java.util._
```

- the _ is a wild card in imports, meaning *import everything*
- or specify a list of particular items to import and aliases
  - especially if existing are likely to clash with names in our namespace

```
import outer.{InnerClass, InnerObject}
```

```
import outer.inner.{
    LongPointlessNameThatDescribesTooMuch => Pointless,
    NameThatClashes => OtrInrName
}
```

## Importing from a `package object`

- when `package object` is specified scala will permit the inclusion of object-level features:
  - such as type aliases, variables and functions

```
package Outer {
    package object Inner {
        type Age = Int
        type Name = String

        def log(msg: String) = println(msg)


        class Venue(val name: Name, val est: Age)
    }
}
```

- type aliases within objects can hide implementation details
  - allowing you more freedom to change representations in the future
- importing from package objects behaves just like package importing

## Practice Exercise

- Q. define a Item class with properties: name and rrp
- Q. define a ReducedItem with properties: name, rrp, discount
  - give ReducedItem a secondary constructor with a default discount of 0.1 * rrp
- Q. define a DiscountRates object
  - give it christmas, easter and summer vals with 0.1, 0.2, 0.3 respectively
- Q. create a List of ReducedItems (a basket) and caluclate the total
  - it should contain reduced items made with the default constructor,

– the secondary, and pass'd a val from DiscountRates

## Solution

```
// Q. define a Item class with properties: name and rrp
class Item(val name: String, val rrp: Double)

// Q. define a ReducedItem with properties: name, rrp, discount
class ReducedItem(val name: String, val rrp: Double, val discount: Double) {
    def this(name: String, rrp: Double) = this(name, rrp, 0.1 * rrp)
}

// Q. define a DiscountRates object
object DiscountRates {
    val christmas = 0.1
    val easter = 0.2
    val summer = 0.3
}

// Q. create a List of ReducedItems (a basket) and caluclate the total
val basket = List(
    new ReducedItem("A", 10, 5),
    new ReducedItem("B", 10),
    new ReducedItem("C", 20, DiscountRates.summer)
)

var total = 0.0
for(item <- basket) {
    total += item.rrp - item.discount
}

println(total)

// or just:  (covered later...)
println( basket map { b => b.rrp - b.discount } reduce { _ + _ } )
```

# Exercise 3 - Features for Object-Orientation

## Aims

The objective behind this exercise is to get some familiarity with objects and classes in scala and how we can interact with them. We also look at companion objects and their functionality.

## Objectives

0. Create a Person class with getters and setters.
1. Create a Bank Account class with a secondary constructor.
2. Create a companion object for your Bank Account class.
3. Tie everything together

## Create a Person class.

- start a new worksheet
- A person has a name, an age and an email address
- Write a constructor for the class
- Ensure that all the variables are private
- Write some getters and setters for the fields.
- The age field should not allow someone to decrease their age — you're not getting any younger!
- Override the toString method and provide some sensible implementation
- Create some person objects from the class and print out their contents.
- Change the various values and test if your class is behaving as it should

## Implement a bank account class

- A bank account should have two constructors
  - The primary constructor takes an account number and an opening balance
  - The secondary constructor takes just the account number and sets the balance to zero
- Add a deposit and withdraw method and a getter for the balance
- Override the toString method for the class
- Ensure that you can get the account number.
  - Hint — you can do this without a getter method!
- The withdraw method should check that the person cannot withdraw more money than they have in their account.
- Create some bank account objects

## Create a companion object for BankAccount

- Store the next bank account number to use in your companion.
- Remember that you need to place the companion object in the same file as the class is declared.
- You will need to create some apply methods which will return a new object of type bank account. One of these apply methods should require no parameters and set the account number to the next in the sequence (stored in the companion object) and set the balance to zero.

- You may get an error saying, "Wrong forward reference" in your sheet, this happens when there is a reference to something that appears in the worksheet later on. They are read from top to bottom, and so the Person class may not know about the BankAccount companion object at the time it is being created — so just change the order of defintion.

### Tying it together

- Now edit the person object to include a bank account created with the companion object
- — this bank only allows people to have a single account. When we cover collections later we will look at how to create lists of objects.
-

### How would you design this without any mutable state?

title: Scala Programming subtitle: Inheritance and Traits author: QA

geometry: margin=2.7cm geometry: a4paper

next: 05.x-exercise-inheritance next-title: Exercise – Inheritance prev: 04.x-exercise-oo —

## Inheritance and Abstract Types

- inheritance is a relationship between types
  - the child type (subtype) *inherits* from the parent type (supertype)
  - gaining any non-private methods or properties defined on the parent
- child *instances* therefore belong to (at least) two types: the parent *and* the child
  - calling a method on a child instance may look-up methods across both parent and child
- eg., suppose the type `Dog` `extends` from `Animal`
  - then instances of `Dog` are also instances of `Animal`
- inheritance is **not** idiomatic approach to design in scala
  - *traits* provide the main mechanism to reuse functionality across classes
  - scala programs are functional with "OO when *necessary*"
  - however we can use inheritance and there are *some* functional uses

## Defining Parents and Children

```
class Point(val x: Int, val y: Int) {
  def moveNewPoint(dx: Int, dy: Int) = new Point(x + dx, y + dy)

  override def toString = s"<$x, $y>"
}

class WeightedPoint(xw: Int, yw:Int, ww: Int) extends Point(xw * ww, yw * ww) {
```

```
    override def toString = super.toString
}
```

- here a a child type `WeightedPoint` inherits from a parent type `Point`
  - when extending from a parent class we must call its constructor
  - WeightedPoint introduces new variables xy, yc in its constructor
  - as these are not prefixed with `val` or `var` scala will optimize them away

```
scala> val weighted = new WeightedPoint(10, 20, 5)
weighted: WeightedPoint = <50, 100>

scala> weighted.moveNewPoint(1,1)
res4: Point = <51, 101>
```

## Overriding Methods

- when accessing a method of an object scala first looks to the instance's principal type
  - ie., its most specific; the type it was created from
  - then to supertypes as defined by the inheritance hierarchy
- child classes may override their parent's methods
  - the child type and a parent type may both contain the same method signature
  - if a child overrides a parent method, scala will "stop looking" when it finds it on the child
- if we wish to override the parents methods, the override keyword is required
  - (optional annotation in java)

*we may override the toString method on AnyRef*

```
class Person(val name: String, var age: Int) {
  override def toString = s"Person(name: $name, age: $age)"
}
```

```
scala> println(new Person("Bertrand Russell", 97))
Person(name: Bertrand Russell, age: 97)
```

## Checking Types: isInstanceOf

- the `isInstaceOf` method of all objects provides the ability to determine their type
  - polymorphic (i.e., the answer is *of the object* not the type of its container)
  - takes a "type argument" supplied in square brackets (more on this, via generics, later)

```
scala> val d = new Dog
d: Dog = Dog@273e53

scala> d.isInstanceOf[Dog]
res24: Boolean = true
```

```
scala> case class Cat()
defined class Cat
```

```
scala> val c: AnyRef = Cat()
c: AnyRef = Cat()

scala> c.isInstanceOf[Cat]
res25: Boolean = true
```

## Abstracts

- attributes (, defs, vars) can be abstract
  - ie., without a concrete implementation
  - so long as they appear in classes declared `abstract`
- as there is no concrete implementation it is impossible to `new` these classes
  - they may only form parents of other classes
  - so `abstract` typically means 'unspecified by the parent': the child needs to implement it
- anything uninitialized in an abstract parent is abstract

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

*children are required to provide an implementation for* `move`

```
class Cat(val name: String, val age: Int) extends Animal(name, age) {
  override def toString = s"Cat(name: $name, age: $age)"
  def move = "pounce"
}
```

## Type Members

- in addition to `val`s, `var`s and `def`s, `type`s may also belong to types
  - we can alias a type within, say, a class definition
  - if we leave a type abstract we are requiring our children to specify it

```
abstract class MyContainer {
    type Index = Int
    type Element

    def get(position: Index): Element
}

class MyCollection(val arr: Array[String]) extends MyContainer {
    type Element = String

    def get(position: Index): Element = arr(position)
}
```

```
scala> val col = new MyCollection(Array("To", "be", "or", "not"))
col: MyCollection = MyCollection@5b4deb2c

scala> col.get(0)
res4: col.Element = To
```

## Traits

- a trait is a set of features that multiple types may have in common
  - eg., all Animal types may also be `Mover` where the `Mover` trait defines the movement ability
  - they dynamically or statically expand the number of types an object belongs to
  - an object, in belonging to the trait type, then has access to the trait methods
- there are "somewhat like" interfaces in java (Java 8)
  - traits are always abstract, ie. require more information to instance
  - they provide default implementations of methods
  - often contain purely abstract definitions
- traits may be statically included in the class (as a kind of inheritance)
  - OR *particular instances* – an instance may be dynamically composed at `new`-time

```
trait Walkable {
  def walks: String
}

class Alice extends Walkable {
  def walks = "Alice Walks"
}

(new Alice).walks
```

```
scala> (new Alice).walks
res6: String = Alice Walks
```

## Trait Inheritance

- traits can extend one-another in the usual inheritance sense
  - it is possible to `override` a method in a trait child of a trait parent

```
trait Moveable {
    def move: String = "Moves..."
}

trait Walkable extends Moveable {
    override def move: String = "Walks..."
}

trait Talkable {
```

```
  def talks: String
}

class Bob extends Walkable with Talkable {
  def talks = "Bob Talks"
}
```

- traits can be multiply-inherited using "with"
  - the definition always reads `extends ... with ...` to emphasize linearization order
  - if Talkable overrides something in Walkable, talkable's implementation will be used
    * read right-to-left

## Composing Objects with Traits

- you can add traits to *instances*

```
trait Talker {
  def talks = "Someone talks!"
}

class Fred

val f = new Fred with Talker

//and consider
trait Talkable {
  def talk: String
}

val person =  new Talkable {
  def talk = "Hello!"
}

val cat = new Talkable {
  def talk = "Meow"
}
```

- for any *instances* who differ on their behaviour
  - that is, mostly for singleton instances
  - (making type-level distinctions into value-level distinctions)

## The Meaning of `super`

- explictly access the members defined on parent types with `super`
  - the order of lookup is said to be the class "linearization"
  - `super` follows this linearization
- orindary java-style inheritance has a familiar linearlization: child, parent1, parent2, …

```
class Parent {
    var money = 100

    def withdraw() = {
        money -= 10
        10
    }
}

class Child extends Parent {
    var piggyBank = 10
    def widthraw() = {
        val pocketMoney = super.withdraw()

        pocketMoney + piggyBank
    }
}


scala> println(s"Spending ${(new Child).widthraw()}")
Spending 20
```

## Layering Traits

- using `super`, traits may call methods on one another when "layered"

```
val myObject = new TypeX with TraitA, TraitB, TraitC
```

- `TraitC` may call `TraitB` (, `TraitA`)
  - this is defined *positionally* so `C` is able to call whatever is in the `B` (, `A`, ..) position
  - linearization is from *right to left*
  - traits call left-wise implementations using `super`
- `TraitC` is called first, and it deals with its own inheritance heirachy
  - then `TraitB` and its inheritance heirachy then `A` and so on

## Linearization

```
trait Mover {
  def move = println("Moving")
}

trait Walker extends Mover {
  override def move = { println("Walk"); super.move }
}

trait Flyer extends Mover {
  override def move = { println("Flying"); super.move }
```

```
}

trait Runner extends Mover {
  override def move = { println("Runs"); super.move }
}

trait Sprinter extends Runner {
  override def move = { println("Sprint"); super.move }
}

class FastBird extends Walker with Flyer with Sprinter

scala> (new FastBird).move
Sprint
Runs
Flying
Walk
Moving
```

## Traits as Mixins

- no state!

## The Meaning of `sealed`

- `sealed` means amenable to extension and overriding within the *module* (file)
  - a `sealed` type (trait, class, …) cannot be extended outside the file

## Nothingness

- `Null`: Trait
  - null: instance of Null
  - corresponds to java's null
- `Nil`: empty List
  - used to build lists
- `Nothing` is a Trait – subtype of every type.
  - No instances
  - Used in type signatures of empty objects (eg. empty lists)
- `None`: Object
  - Subtype of Option.
  - used when returning an Option
  - and when no senisble value can be returned
- `Unit`: Class
  - () – single instance of Unit.

  - used for action ("void") situations, eg. I/O

## ASIDE: Bottom Type Hierarchies

- scala has a triadic "bottom type" structure:
- `scala.AnyVal`
  - base of any typically unbox'd value in java
  - child type is `scala.Nothing`
  - includes `scala.Unit` type whose single value is `()`
  - `AnyVal`s of a single data property may be optimized away to just that property
- `scala.AnyRef`
  - everything else (incl. java.lang.Object)
  - child type is `scala.Null`
- `scala.Any` is parent (supertype) of both
- unrelated to this structure there is also `Nil`
  - it means "empty sequence"
  - it is `List[Nothing]`

## Practice Exercise

- Q. define a trait Talker that has a speak method that println's "Hello"
- Q. define a class Dog which extends from this trait and overrides speak to println "woof"
- Q. define a class Person
- Q. create a List of Talkers which contains Person and Dog objects
  - HINT: new … with …
- Q. define a *function* say() which takes a List of Talkers
  - it should iterate through this list and ask each to speak

## Solution

```
// 1. define a trait Talker that has a speak method that println's "Hello"
trait Talker {
    def speak = println("hello")
}

// 2. define a class Dog which extends from this trait and overrides speak to println
class Dog extends Talker {
    override def speak = println("Woof")
}

// 3. define a class Person
```

```
class Person

// 4. create a List of Talkers which contains Person and Dog objects
val talkers = List(
    new Person with Talker,
    new Dog
)

// 5. define a function say() which takes a List of Talkers
def say(ts: List[Talker]) = for(t <- ts) t.speak

say(talkers)
```

# Exercise 4: Inheritance and Abstraction

## Aim

To consider how to improve classes from the previous exercise using inheritance, traits, and typical methods of object-oriented-style abstraction.

## Objectives

## Create an Animal class.

- Start a new worksheet
- The class should be abstract
- The constructor should take in a name and an age
- Override the toString method
- Provide an abstract method called move that returns a string.

## Create three subclasses of the animal class

- `Cat`, `Dog` and `Rabbit`
- Your classes should call the parent constructor
- Implement the move class for each animal differently (return a different string. For example, a rabbit could return "hop") Override the toString method for the class

4. Create some cat, dog and rabbit classes to test your inheritance. Test the toString methods.

## Create a SavingsAccount

- Copy in your bank account class from the previous exercise (we don't need the companion object)
- Create a SavingsAccount which inherits from account

- A savings account has a percentage interest rate and an `addInterest` method which increase the balance by that percentage
  - For example: If the account had a balance of £100 and an interest rate of 10% the addInterest method would increase the balance by 10% to £110

7. Create a current account which has an overdraft facility Override the withdraw method to take into account the overdraft as well as the current balance in the account. Traits Traits can be applied to classes when the class is declared, or when a new object is created from that class. We will be looking at both of these situations.
8. Create a 'vocal' abstract trait that has a single method, makeNoise. This should return a string value
9. Add this trait to the animal class. All subclasses of animal will now need to implement the methods in the vocal trait. Add some implementation for these. Example: abstract class Animal(…) extends Vocal { … }
10. Call the makeNoise method on your cat, dog and rabbit objects.
11. Remove the vocal trait from animal and add it directly to the Dog class. We can now call makeNoise on dog objects via the trait, but the others have become standard methods. Comment out the method in the cat and rabbit classes
12. Create a new cat object and add the Vocal trait at declaration time (rather than in the class description) Does it work? Why not? Create a concrete Vocal trait (provide some implementation for the makeNoise method) and apply it to a cat object at declaration time. Call the makeNoise method on your new cat object
13. Implement the Logger trait The logger trait should have a single method, Log which takes in a string and does nothing with it (slide 82 has an example)
14. Create a ConsoleLogger trait that inherits from the Logger trait The console logger should print the messages to the screen
15. Add the Logger trait to your BankAccount class. Add some Log calls where you would like output to be.
16. Create a new BankAccount object and add the ConsoleLogger to it at declaration time. Test by calling your methods and seeing if the output is printed to the screen.
17. Now create a different logger, the TimeStampLogger This works out the current date and time and prints this before printing the message to the screen Hint: there are ways of getting the current date in scala, but you can also call classes and methods in Java as well!
18. Create a bank account that uses both the ConsoleLogger and TimeStampLogger def b1 = new BankAccount("123", 100) with ConsoleLogger with TimeStampLogger Do some deposit and withdraw calls and look at the output, then swap the order of traits and repeat this. When all the traits have the same parent trait the version of the method called is the one added to the class last, so in the example above, the TimeStampLogger method would be called, rather than the ConsoleLogger. We can use this to override functionality in classes and traits even at object creation time.
19. Finally, we're going to have a look at a built in trait called App. Create a new scala class (not a worksheet!) Change the class definition to be an object that extends App Write anything you like in the body of the object The App trait takes any commands in the body of the object and runs them as if they were in a methods, or being typed into the repl. Add as many scala commands as you like and then run it to see the output. Hint: To see output you will need println() statements Your object should look something like this: object Ex4 extends App{

println("Anything here is run as individual statements due to the app trait")

} # Functional Programming

- the primary motivation for scala

  is **not** "java without semi-colons* – Odersky

- rather to provide a realistic functional paradigm to JVM-focused programs
  - ie., compositions of functions, side-effect free code, provable code (cf. laws)

  Pure functional languages have this advantage: all flow of data is made explicit. And this disadvantage: sometimes it is painfully explicit. A program in a pure functional language is written as a set of equations.

  Explicit data flow ensures that the value of an expression depends only on its free variables. Hence substitution of equals for equals is always valid, making such programs especially easy to reason about. Explicit data flow also ensures that the order of computation is irrelevant, making such programs susceptible to lazy evaluation [and parallelization].

  – Philip Wadler

*from the paper in which he introduced monads in Haskell*

# Defining *function*

- a pure *function* **is a mapping** from a set of inputs to a set of output
  - such that for every input there is single output
  - (as in mathematics, logic, functional programming …)
- a *function* is therefore **equivalent to** a set of pairs

$(input0, output0), (input1, output1), ..., (inputN, outputN)$

*eg.,* `add` *is equivalent to the infinite set,*

$((0,0),0), ((0,1),1), ((1,0),1), ((1,1),2)...$

- in mathematics this equivalence is a much stronger claim:
  - functions are not primitives, ***sets*** *are*
  - functions are understood as sets of this kind
- functions are therefore referentially transparent (or 'pure')
  - meaning that f(input0) *is just* output0

# Defining pureity and infinite sets

- sets are often infinite
  - (eg. the function which repeats a string `n` times is defined for every n, $[0... + inf)$
  - so we write *expressions which define them*, rather than list every element
- to preserve the equivalence between "a set from which we chose one value" and "an expression which defines a set"
  - functions must be pure
  - ie. we must everywhere be able to substitute for f(input0), output0

        * even if $f$ is doing more work than a set-lookup
    - referential transparency is key: we wish to be able to substitute any expression for its evaluation
- purity ensures we able to reason clearly about our program
    - and allows us to *operate on our program as we operate on data*
    - since each function call is *equivalent to some data*

## The Function Type

- the function type `String => Int` *means*
    - for *every* particular string, provide a *single* integer
- a function is *partial* if it doesn't provide a single output for a particular input
    - often when dealing with user input (etc.) we cant provide something valid
    - getName(): Name cannot always provide a Name (say we enter "££!")
    - we can make this *total* by using an Option type
        * always return an option value, sometimes None sometimes Some("The Name")

```
def getName(): Option[Name]
```

*… Wadler continues…* > It is with regard to modularity that explicit data flow becomes both a blessing and a curse. On the one hand, it is the ultimate in modularity. All data in and all data out are rendered manifest and accessible, providing a maximum of flexibility. On the other hand, it is the nadir of modularity. The essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use.

*Odesky's approach for Scala is to introduce impure OO to pure functional programming in order to redress these problems (monads do the same work in Haskell – and indeed, somewhat, in scala)*

## Functions as data

- for functional programs, in some sense, we wish to "treat our program as data"
    - to combine elements of it together as we combine data together
- to combine numbers together, we can add them

```
1 + 2
```

- to combine functions together we can compose them

```
f compose g
```

- consider the power behind manipulating data (eg., with SQL):
    - transform data of one kind into another
    - summate, aggregate, join, extract, etc. information
- consider now how powerful these operations might be if our program was itself data

## Anonymous Functions (lambdas)

```
val square = (x: Int) = x * x

val printHello = () => println("Hello")

val add = (x: Int, y: Int) => x + y
val sub = (x: Int, y: Int) => x - y
```

- what are the types of these vals?
- we can use these types in parameter signatures to write higher-order functions
  - ie. functions which take other functions as arguments

```
def runOp(op: (Int, Int) => Int, x: Int, y: Int) = op(x, y)

runOp(add, 1, 2)
runOp(sub, 2, 4)
```

## Higher Order Functions

- the power of functional programming, in part, derives from its ability to structure programs as data
- higher-order functions are essential to this:
  - HOFs take other functions and operate on them
  - ie., HOFs are the functions which treat other functions as data

```
val flip = (f: (Char, Char) => String) => (x: Char, y: Char) => f(y, x)

val join = (a: Char, b: Char) => "" + a + b

val rjoin = flip(join)
```

```
scala> join('O', 'K')
res5: String = OK

scala> rjoin('O', 'K')
res6: String = KO

scala> flip(join)('O', 'K')
res7: String = KO
```

- `flip` transforms our *function*
  - taking the function `join` (ie. concatenate) and producing a new function `rjoin`
- we can think of this simply if we recall the mathematical definition of function:
  - an expression which *defines* a set of pairs

## ASIDE: HOFs as set transformations

*consider the join function, part of its definition is then* $..., (('O','K'),'OK'), ...$

- somewhere in the set which describes `join` is the pair

$(('O','K'),'OK')$

- which means input ('O', 'K') maps to 'OK'

***`rjoin`*** *is the function defined by*

$..., (('O','K'),'KO'), ...$

- all `flip` is doing is changing the pairs
  - in particular, it is swapping the order of the *first*, the arguments
- in fact, this is what all HOFs do
  - generate a new set from the old set
  - or a new "function" from the old "function"
- NB. thus `flip` *itself* is the set $..., (join, rjoin), ...$

$..., (..., (('O','K'),'OK'), ..., ..., (('O','K'),'KO'), ...)...$

## Currying

- funcitonal programs are then just a series of transformations of data:
  - either "simple" data, or comple (eg. functions)
- often pre-existing transformations will exist that have the wrong 'shape'
  - eg. a sum function exists, of two parameters, and we just want a function of one parameter
  - eg., a function that +1s its argument...

```
val sum = (x: Int, y: Int) => x + y

sum(1, 2)

val sumWith = sum.curried
val sumOne = sumWith(1)
valu sumTwo = sumWith(2)

sumOne(4)
sumTwo(4)
```

*or,*

```
scala> def sum(a: Int)(b: Int)(c: Int) = a + b + c
sum: (a: Int)(b: Int)(c: Int)Int

scala> val addOne = sum(1) _   // _ means "omitting every other argument"
addOne: Int => (Int => Int) = <function1>

scala> addOne(2)(3)
```

```
res28: Int = 6

scala> val addOne = Function.uncurried(sum(1) _)
addOne: (Int, Int) => Int = <function2>

scala> addOne(2,3)
res29: Int = 6
```

## Misshaped Functions

- often we have some input data `(in0, in1, in2)` and we want some output data `out1`
  - to connect our input to our output we have some function specialized to our problem
  - eg., `readFromSocket` ie., `readFromSocket(127.0.0.1, 80, 1024)`
- often however things dont match up that neatly
  - either there is a misshaped function, eg. `readFromSocket(ip, port)`
  - or our input data is of a different type
  - eg., `List('127.0.0.1:80', '127.0.0.2:80', '127.0.0.3:80')`
- in an imperative program we would have to rewrite our domain function (`readFromSocket`)
  - or write a wrapper
- in a functional program we can just use functions to "refit or reshape" other functions to the types and arguments we have

## Writing Curried Functions

- for example, consider an authentication library
  - it requires a callback to determine *how* its going to authenticate
  - the callback returns a Connection that the service uses
  - it calls the callback passing in user/password credentials from some provider-service
- to provide a Connection our connection-function requires three parameters
  - a location to connect to (IP host)
  - a username
  - a password

```
def configure(host: String, user: String, pass: String): Connection
```

- to pass in this function we need to remove the `host` part

```
val configureWith = configure.curried
val configureUK = configureWith("uk-host")
val configureFrance = configureWith("french-host")


val ukConn = authService.use(configureUK)
val frConn = authService.use(configureFrance)
```

- suppose `authService` required an ordinary function (ie. `auth(u, p)` vs `auth(u)(p)`), then

```
val normalConfigureUK = Function.uncurried(configureUK)
```

## Defining Functions in a Curried-Style

- scala allows us to define functions curry-ready
- recall also the an example from earlier, the `repeat` function…

```
def repeat(str: String, numTimes: Int)(fn: (String, String) => String) = {
    result = "";
    for(i <- 0 to numTimes) result = fn(result, str)
    result
}

scala>
repeat: (str: String, numTimes: Int)(fn: (String, String) => String)String
```

- here it is defined already curried
  - ie. if called without a second argument it returns a function
  - `repeat("!", 10)` returns a function of one argument

```
repeat("!", 10) {
    (l: String, r: String) => l + r
}
```

- scala will insert parentheses when a code block is used as an argument

## ASIDE: Eliminating Side-Effects

- all programs can be phrased as a series of transformations on data
- if the language itself is merely aware of *data*
  - not the VM's eventual understanding of this data (ie. operating-system calls, etc.)
- then then program itself is pure

*for example, "println" could be defined as the set*

$..., (somestringinput, PrintOperation(somestringinput)), ...$

- where the output is just some *token*, not actually any operation at all
  - the *VM* then reads this token sequence, at the end
  - without this final operation "performed outside of the program" it wouldn't *do* anything
- if IO functions are just defined this way (as returning tokens)
  - then we can transform *them* as we transform other kinds of functions/data/etc.
- in imperative programs we cannot link transformations together
  - they work by side-effects
  - they do not define relations between inputs and outputs
  - ie., they "munge" inputs until they become outputs
  - destroying the input
  - usually this munging process is so specialized to the input, it cannot itself be reused

## Reuse by Composition

```
def getWebPage(): String
```

```
f andThen g andThen h
h compose g compose f
```

```
val parseCsvFile = splitLines andThen removeEmpty andThen splitCommas andThen trim
val parseXmlFil = splitLines andThen removeEmpty andThen untagLine andThen trim


parseCsvFile(myFile)
```

- the same transformations can be resued for different applications
  - some functions are highly specialized to our domain (eg. `untagLine`)
  - many are not and can be reused
  - not possible with side-effect driven code

```
proxyWebPage = rewriteCssUrls andThen rewiteAnchorUrls andThen rewriteLinkUrls //...
```

```
val doPost = connectToWeb andThen getUrl and readPostFields andThen submitPostData
val readForm = connectToWeb andThen readPostFields
val doGet = connectToWeb andThen getUrl
//...
```

- compositional programming is all about reuse:
  - object orientation hides state "behind walls"
    - you need methods on objects to be able to read the (private, localized) state
  - eg., each kind of object has to have its own "connect" method
- transformations over immutable data treat all "right-shaped" data the same
  - you can use the same connect function over widely different kinds of objects
  - as long as you "reshape" these objects to fit
- we'll see this with the for-comprehension

## Tail-Calls and Recursion

- sometimes we need to loop (over data, forever, etc.)
- in functional programming this achieved through recursion (ie. a function calling itself)
  - however recursion, on the jvm, can lead to stack overflow (hence the website..)
  - java requires a stack frame for each call
  - given a large set of function calls this will overflow the stack
- if the last call a function makes is to itself, it can be transformed into an ordinary loop
  - the "tail call" is last thing a function does

```
def fn(x: Int): Int =
    if(f(x))
        g(x)
    else
        h(x) + 1
```

- g() is in tail call position
- h() not a tail call, since it has to +1
  - +1 is the last operation this function performs

## Tail-Call Recursive Functions

```
def fact(x: Int): Int = if(n == 0) 1 else n * fact(n - 1)
```

- last operation is a multiplication
  - not a call to itself
- if the caller has to perform some "last operation" then it needs your return value to do that
  - "needing the return value" means control has to be passed back to the caller
  - which means you need to remember who called, which means you need a stack frame

```
|----a------|
.  |----b----| ------> return ^a
.    |----c----| ---> return ^b

|----a------|
     b
     c ----> return ^a
.
```

- where `a` is the *code* of a
- and `|----a----|` is the calling of a
- a tail-call recursive version looks like:

```
def fac(n: Int): Int = {

    def go(i: Int, acc: Int) =
        if (i == 0)
            acc
        else
            go(i - 1, acc * i)

    go(n, 1)
}
```

- go is tail-call recursive, since in the second branch the last thing the function does is a call

*calls are optimized away as jumps to callers, which is what a while loop does…*

```
def fac(n: Int): Int = {
    var i = n
    var acc = 1

    while (i > 0) {
        acc *= i
        i   -= 1
```

```
    }

    acc
}
```

- the @tailrec annotation asks scala to check the function is tail-call recursive

## Lazy Evaluation of Parameters

- putting `<=` before the type of an argument allows you to pass it lazily (`lazy val`ly, ie. once on use)

```
def x(y: <= String) = y
```

*compare,*

```
def ifS(val cond: Boolean, val trueVal: Int, falseVal: Int) =
    if(cond) trueVal else falseVal

def ifL(val cond: Boolean, val trueVal: <= Int, falseVal: <= Int) =
    if(cond) trueVal else falseVal

ifS(true, fact(100), fact(50))

ifL(true, fact(100), fact(50))
```

- in first case, ifS evaluates its arguments at call-time so fact() is calculated twice
- in the second case, ifL evaluates its arguments only when used, ie. since only one gets used only one fact is calculated # Exercise: Functional — title: Generics and Other Complex Types —

## Tuples and Product Types

- tuples are *single value* representations of complex objects
  - a table, for example, is a single object but has "many parts" —a top, four legs, etc.
  - contrast this with a collection of objects
    - for example a pot of pens (a pen is not "part" of the pot)
- tuples are defined by writing each "part" followed by a comma all in parentheses

```
scala> val mytuple = (1, "Hello", false)

mytuple: (Int, String, Boolean) = (1,Hello,false)
```

- we can refer to each part of a tuple by accessing it with `._1` for the first part, `._2` for the second, etc.

```
scala> mytuple._3
res0: Boolean = false
```

## Tuples as Objects

- tuples can be thought of as objects with anonymous fields, ie.

```
scala> val person = ("Michael", 26)
person: (String, Int) = (Michael,26)
```

is much the same as,

```
val person = new Person("Michael", 26)
```

- note the type of the tuple "`(String, Int)`" – this is called a product type
  - since it has, as many legal values as String * as many legal values as Int
- since tuples are *only data* they lend themselves to being passed around during certain transformations
- note that when defining `function` earlier, the `pair` (input, output) was a tuple
  - pure functions "are just" a set of such pairs

## Arrow Syntax

- scala has a special syntax for creating pairs (tuples of two parts), the arrow `->`

```
scala> 1 -> 2
res1: (Int, Int) = (1,2)

scala> (0,0) -> 0
res2: ((Int, Int), Int) = ((0,0),0)

scala> Set(0 -> 0, 1 -> 2, 2 -> 3)
res3: scala.collection.immutable.Set[(Int, Int)] = Set((0,0), (1,2), (2,3))
```

- have product types `(Int, Int)` not simple types
  - eg. `List[Int]` is just one type (list of ints) not two types `(List[X], Int)`
  - `List[ ]` itself isn't a type, but a type constructor (it makes particular types, ie. `List[Int]`)
  - ..more on that later

## Type Aliasing Tuples

- we have just seen a product type, the tuple

```
(1, "Hello", false) : (Int, String, Boolean)
```

- here its type is *annotated* on definition

*we could define its type and use that,*

```
type WebSite = (String, String, String)

val eg : WebSite = ("Example Site", "A website used for examples", "http://www.example.co.uk")

scala> eg
res3: WebSite = (Example Site,A website used for examples,http://www.example.co.uk)
```

- here we declare a `WebSite` to be composed of three parts: a title, a description and a url

## Algebraic Data Types

- often in scala the return types of an operation may be more general than the principal type of the value returned

```
scala> for(char <- "Message") yield char + 1
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(78, 102, 116, 116, 98, 104, 102)
```

- `IndexedSeq` is a more general type than `Vector`
  - i.e., `Vector` is a child of `IndexedSeq`
  - what types can we assign to a variable of type `IndexSeq` ?

*or below, "what types can we assign to an `Animal` variable?*

```
class Animal
class Dog extends Animal
class Cat extends Animal
```

- in a variable (container) of type `Animal` there can be *either* an `Animal`, a `Cat` or a `Dog`
- the name in functional programming for a "type composed of other types" is an algebric data type

## Pseudo-code for ADTs

" *we could write the type of a tuple as,*

```
type WebSite = String AND String AND String
```

*we could write the type of* ***`Animal`*** *as,*

```
type Animal = Animal OR Cat OR Dog
```

- whenever there is a `WebSite` value, there is a `String AND String AND String`
- whenever there is an `Animal`, there is either an `Aniaml OR Cat OR Dog`
- in functional programming, `Animal` is known as a sum type
- if we have declared `Animal` to be itself abstract, then values of the type could only contain its children

```
abstract class Animal
class Dog extends Animal
class Cat extends Animal
```

```
//pseudo-code equivalent
type Animal = Cat OR Dog

// thus we may only have
val myAnimal : Animal = new Cat
val myAnimal : Animal = new Dog
```

## Defining an Algebra

- an "algebra" in the functional programming sense is a schematic of types in this fashion
  - often case classes which extend from a parent are used to provide a structure

```
abstract class LogicExpr

case class True() extends LogicExpr
case class False() extends LogicExpr

case class And(val l: LogicExpr, val r: LogicExpr) extends LogicExpr
case class Or(val l: LogicExpr, val r: LogicExpr) extends LogicExpr
```

```
scala> val myX : LogicExpr = And(True(), And(False(), Or(True(), False())))
myX: LogicExpr = And(True(),And(False(),Or(True(),False())))
```

```
Expr = True OR False OR And(Expr, Expr) OR Or(Expr, Expr)
```

- these data types *and the patterns they follow* are often said to define an "algebra"
- defined above is an algebra for simple boolean expressions
  - there is no *interpretation* to this algebra
  - it doesn't mean (, *do*) anything

## Semantics for Algebra

- giving meaning, or interpreting, an algebra usually just means implementing some methods or functions for those types

```
abstract class LogicExpr {
    def meaning: Boolean
}

case class True() extends LogicExpr {
    def meaning = true
}

case class False() extends LogicExpr {
    def meaning = false
}
```

```
case class And(val l: LogicExpr, val r: LogicExpr) extends LogicExpr {
    def meaning = l.meaning && r.meaning
}

case class Or(val l: LogicExpr, val r: LogicExpr) extends LogicExpr {
    def meaning = l.meaning || r.meaning
}
```

## Evaluating an Algebraic Structure

```
scala> val myA : LogicExpr = And(True(), And(False(), Or(True(), False())))
myA: LogicExpr = And(True(),And(False(),Or(True(),False())))

scala> val myO : LogicExpr = Or(True(), And(False(), Or(True(), False())))
myO: LogicExpr = Or(True(),And(False(),Or(True(),False())))

scala> myAndExpr.meaning
res4: Boolean = false

scala> myOrExpr.meaning
res5: Boolean = true
```

## Type Arguments

*here is a list,*

```
val drinks = List("Coke", "Pepsi")
```

- lists are not all the same:
- a list of drinks is not the same as a list of pens
- however a list of cans of coke *is* a like of drinks
- to distinguish between kinds of lists we can specify to the type *constructor*, an type-making type-function
  - `List[Int](1,2,3)`
  - this extra type is specified with the square brackets `[ ]`
- `List` itself is not a "complete type", `List[Int]` however is
  - its called higher-kinded where 'a kind' just means a complete type
  - *or* a generic type where 'generic' means it's more general (`List`) than a specific type (`List[Int]`)

## Defining Generic Types

- consider a type `Person` which comprises an id
  - in the case of a database this id might be an integer
  - in the case of an interactive (/GUI) session, it might be a string (eg., the persons name)

*we may therefore define,*

```
class Person[A](id: A)
```

- `Person` is a generic type
  - the type of "what's inside it" (`A`) can vary
  - when defining `Person` we effectively treat `A` as an argument to it
  - `A` is whatever specified when `new`-ing (constructing) a `Person` object

```
val mike = Person[String]("Michael")

val dbuser = Person[Int](1001)
```

## Variance

- these "inner types" matter for determining what an object *is*
  - for a pragmatic definition of "*is*" read: substitutable
  - a child *is a* parent because a child can be substituted when a parent is expected
  - eg. a `Cat` *is an* `Animal`
- when designing programs types should follow intuitions:
  - a `List[Person]` is not a `List[Dog]`
  - a `List[Pepsi]` is a `List[Cans]`
- if asked for a collection of cans, a collection of pepsis would be correct
- if asked for a list of dogs, a list of people would not be correct
- therefore the how a complex types (`List[A]`) varies given simple types `A` should be explicit

## Covarience

- a list of cats is a list animals if a `Cat` is an `Animal`

```
class Animal(val name: String)
class Cat(name: String, owner: String) extends Animal(name: String)

val cats: List[Animal] = List(new Cat("Fluffy", "Tim"), new Cat("Tabs", "Tim"))
```

- this assignment works due to the definition of List, as List[+A]
  - the `+` symbol on `A` means `List` follows `As is a` relationship
- when defining our own types to indicate that a container of one sort is the same as another, use a +

```
class Dog(name: String, owner: String) extends Animal(name: String)

class Person[+A](val pet: A) {
    override def toString = s"Person()"
}
```

- now a `Person[Dog]` is a `Person[Animal]`

## Covarient Combinations

- what happens when we add two lists together?

```
val myList = List(1, 2, 3) ++ List(1.0, 2.0, 3.0)
```

- ...becomes List[AnyVal]
  - useless because the only type Doubles and Ints "have in common" is AnyVal

```
val catOwners = List(new Person(new Cat("Fluffy", "Michael")))
val dogOwners = List(new Person(new Dog("Spot", "Tim")))

val animalOwners = catOwners ++ dogOwners

scala> dogOwners
dogOwners: List[Person[Dog]] = List(Person())

scala> catOwners
catOwners: List[Person[Cat]] = List(Person())

scala> animalOwners
animalOwners: List[Person[Animal]] = List(Person(), Person())
```

- notice the types changing

## Type Bounds

- `Person` had no useful `toString` method (nor any at all)
  - to add a useful `toString`, more information about `A` is needed,

```
class Person[+A](val pet: A) {
    override def toString = s"Person(owns = ${pet.name})"
}

<console>:8: error: value name is not a member of type parameter A
           override def toString = s"Person(owns = ${pet.name})"
```

- `A` is too generic for scala to know that `.name` exists on it
  - more information about `A` can be given however

```
class Person[+A  <: Animal ](val pet: A) {
    override def toString = s"Person(owns = ${pet.name})"
}
```

*and now…*

```
scala> val animalOwners = catOwners ++ dogOwners
animalOwners: List[Person[Animal]] = List(Person(owns = Fluffy), Person(owns = Spot))
```

- the "extra information" we supplied was to relate A to Animal
  - i.e., that A was a subtype (`<:`) of Animal
  - we can specify that A is a supertype (parent) of Animal with `>:`

## Special Generic Types: Option

- exceptions break referential transparency and are otherwise bad practice:
  - they introduce non-local flow
  - can easily be ignored
  - rather than throw exceptions, return an `Option` of that value
- here is a simple implementation of `Option` for an int (called `Maybe`)

```
abstract class Maybe
case class Just(val maybe: Int) extends Maybe
case class Empty() extends Maybe

val maybeAge: Maybe = Just(26)
val maybeDay: Maybe = Empty()

def division(dd: Int, divr: Int): Maybe =
    if(divr == 0)
        Empty()
    else
        Just(dd/divr)
```

## Scala's Option

- scala provides its own `Option[A]` type with methods on it that make pulling out values
  - useable within for comprehensions

```
def dv(dd: Int, divr: Int): Option[Int] =
    if(divr == 0)
        None
    else
        Some(dd/divr)
```

```
scala> dv(10, 0)
res0: Option[Int] = None
```

```
scala> dv(100, 10)
res1: Option[Int] = Some(10)
```

## Methods on Option

- there are helper methods on option (`.get`, `.isDefined`, etc.) or use `for`

```
scala> val (someX, someY, someZ) = (dv(100, 10), dv(20, 5), dv(1,0))
someX: Option[Int] = Some(10)
someY: Option[Int] = Some(4)
someZ: Option[Int] = None

scala> for(x <- someX; y <- someY) yield x + y
res3: Option[Int] = Some(14)

scala> for(x <- someX; y <- someY; z <- someZ) yield x + y - z
res5: Option[Int] = None
```

- `for` preserves the `Option` type
  - ie. the computation `x + y - z` happens "within the Option context"
  - its `None` when we expect
  - the `for` **exists** as soon as it finds a None and returns None!

## Special Generic Types: Try

- the Try type provides `Success` and `Failure` children
  - compare to `Option` providing `Some` and `None`
- in the `Try` case, `Failure` contains an Exception object

```
import scala.util.Try

def division(dd: Int, divisor: Int): Try[Int] = {
  Try {
    dd / divisor
  }
}
```

- here we use `Try`'s apply method

```
scala> division(10,0)
res7: scala.util.Try[Int] = Failure(java.lang.ArithmeticException: / by zero)

scala> division(10,1)
res8: scala.util.Try[Int] = Success(10)
```

- `Try`s are used to wrap java libraries mostly
  - so that Exceptions become value-level

## Collections and Other Special Types

- collections are by default immutable
  - import the `mutable` package to use mutable version
- all collections are `Traversable` and `Iterable`
  - `Traversable` uses `foreach`
  - `Iterable` uses `next` method (`next` implies "knowing current position" ie. state)
    - Every `Iterable`is a `Traversable`
- of these the basic types are:
  - `Seq`s (sequences: indexable ordered collections)
  - `Set`s (sets: unordered unique collections)
  - `Map`s (maps: unordered sets of key-value pairs)
- The most important kinds of collections are:
  - the `Seq` and `IndexSeq`, `Vector`
  - the `Seq` and `LinearSeq`, `List`
  - the `Map`, `HashMap`
  - the `Set`, `HashSet`

## Diagram

*Collections Diagram*

## Particulars

*considered here:*

- collections
  - `Range`
  - `Array` and `ArrayBuffer`
  - `List` and `ListBuffer`
  - `Vector`
  - `Set`s and `Map`s
- Iteration and Zipping
- Also:
  - Option
  - Try

## Range

- a Range is a *generator* (*not* a collection of values)
  - which generates values in an ordered series (Double or Int or Char)

```
val rangeInt = 1 to 10 by 2

val rangeDouble = 1.0 to 5.0 by 0.2

val rangeChr = 'a' to 'z'
```

```
scala> ('a' to 'z').toList
res2: List[Char] = List(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o,
p, q, r, s, t, u, v, w, x, y, z)
```

## Array

- arrays are: fixed in length, generic and monotonically indexed

```
val letters = Array('l', 'e', 't', 't', 'e', 'r')
val numbers = Array(1,2,3,4)
val decimal = Array[Double](2,34,56)

scala> numbers
res3: Array[Int] = Array(1, 2, 3, 4)

scala> decimal
res4: Array[Double] = Array(2.0, 34.0, 56.0)
```

- lookup elements using parentheses `( )`
  - ie. the `apply` method – or "calling" – means *lookup*

```
scala> "" + letters(0) + letters(1) + letters(2)
res6: String = let
```

- `mkString` is a join (concatenation) method

```
scala> letters.mkString
res7: String = letter
```

## ArrayBuffer

- array buffers are: mutable, variable length arrays
  - anything mutable has to be imported

```
import scala.collection.mutable.ArrayBuffer

val buffer = ArrayBuffer[Char]()
```

- `+=` is defined for adding a value of the buffer type or a tuple of values
  - and `++=` for adding another sequence

```
scala> buffer += 'h'
res12: buffer.type = ArrayBuffer(h)
```

```
scala> buffer += ('e', 'l', 'l', 'o')
res13: buffer.type = ArrayBuffer(h, e, l, l, o)

scala> buffer ++= " World!"
res14: buffer.type = ArrayBuffer(h, e, l, l, o,  , W, o, r, l, d, !)
```

## List

- lists are immutable and generic
  - especially suited to functional programming (cheap to recurse, etc.)
  - Nil-terminated (where `Nil` == `List[Nothing]()`)

```
val people = List("Winston Churchill", "Richard Feynman", "Mark Rothko")
val times = List(1800, 1930, 2155)
```

- the `cons` (list construction) operator is important and adds a head element
  - note right-associative due to `:`-prefix

```
val places = "UK" :: "France" :: "US" :: Nil
```

- each `cons` operation creates a new list(-pointer)
  - since structure is immutable we may cheaply point to any part of it
  - no remove because each reference take on an element is a reference to somewhere in the linked list
- some useful methods for functional programming include:
  - head and tail
  - take and drop
  - init and last
- some useful operations:
  - ++ `Sequence` concatenation
  - ::: `List`-specific concatenation

## List Buffer

- list buffers are mutable lists
  - `+:` to construct elements
  - `+=` to add

```
import scala.collection.mutable.ListBuffer

val people = ListBuffer[String]()

val peeps = "Winston" +: "Richard" +: "Mark" +: people

people += "Issac"
people += "Aristotle"
```

```
scala> people
res28: scala.collection.mutable.ListBuffer[String] = ListBuffer(Issac, Aristotle)
```

```
scala> peeps
res29: scala.collection.mutable.ListBuffer[String] = ListBuffer(Winston, Richard, Mark)
```

## Vector

- why do for-comprehensions typically return `Vector`s?
- `Vector`s are much more efficient than `List`s for typical random-access (non-sequential) programming
- `List`s are mostly specialized to specific applications in functional programming,
  - particularly for recursive algorithms defined with head/tail

```
scala> val vec = scala.collection.immutable.Vector.empty
vec: scala.collection.immutable.Vector[Nothing] = Vector()

scala> val vec2 = vec :+ 1 :+ 2
vec2: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> val vec3 = 100 +: vec2
vec3: scala.collection.immutable.Vector[Int] = Vector(100, 1, 2)

scala> vec3(0)
res1: Int = 100
```

## Sets

- unique, unindexed collections
- have set operations

```
val people = Set("Matt", "Mark", "Mike", "Tim", "Luke", "John")
val writers = Set("Matt", "Mark", "Luke", "John")
```

- in both sets

  ```
  scala> people & writers
  res35: scala.collection.immutable.Set[String] = Set(Luke, John, Mark, Matt)
  ```

- in people but not in writers

```
scala> people &~ writers
res37: scala.collection.immutable.Set[String] = Set(Mike, Tim)
```

- (re)adding “Mike”

```
scala> people + "Mike" + "Mike"
res41: scala.collection.immutable.Set[String] = Set(Luke, John, Mark, Mike, Matt, Tim)
```

## Map

- maps: key -> value associations (key-value pairs)
- pairs are unique, i.e. form a set

```
val preferences = Map(
    "Michael" -> List("Coke", "Pepsi"),
    "Tim" -> List("Apple Juice", "Water")
)
```

- access with ( ) as usual,
  - exception if the key doesn't exist
  - prefer to use the `get` method which returns an `Option` type (cf. below)

```
scala> preferences("Tim")
res62: List[String] = List(Apple Juice, Water)

scala> preferences("Michael")(0)
res63: String = Coke
```

## Modifying Maps

```
scala> preferences + ("Lucie" -> List("Beer", "Wine"))
res64: scala.collection.immutable.Map[String,List[String]] =
    Map(Michael -> List(Coke, Pepsi),
        Tim -> List(Apple Juice, Water),
        Lucie -> List(Beer, Wine))

scala> preferences - "Tim"
res65: scala.collection.immutable.Map[String,List[String]] =
    Map(Michael -> List(Coke, Pepsi))
```

- and mutate with = on mutables

```
preferences("Michael") = List("Apple", "Banana")
preferences += ("Lucie" -> List("Beer", "Wine"))
```

- and there are various interesting methods

```
preferences.keySet()

val value: Option[List[String]] = preferences.get("Susan")
```

## Collection Idioms: Iteration

- iterate over any collection using a for-comprehension

```
for(element <- collection) {
    println(element)
}
```

- where `element` is a pair in the case of a Map (ie. a `(key, value)`)

```
scala> for(person <- List("Matt", "Mark", "Luke")) println(person)
Matt
Mark
Luke

scala>

scala> for(pair <- Map("Cat" -> "Mammal", "Raven" -> "Bird")) println(pair)
(Cat,Mammal)
(Raven,Bird)

scala>

scala> for((animal, kind) <-
    Map("Dog" -> "Mammal", "Crow" -> "Bird")) println(s"$animal is a $kind")

Dog is a Mammal
Crow is a Bird
```

## Collection Idioms: For Comprehensions for New Sequences

- generate new sequences from old ones in the usual way,

```
scala> val simple = for(
    (animal, kind) <- Map("Dog" -> "Mammal", "Crow" -> "Bird")
) yield kind + ":" + animal

simple: scala.collection.immutable.Iterable[String] =
List(Mammal:Dog, Bird:Crow)
```

## Collection Idioms: Zipping

- zipping is the operation of making a map out of two other sequences
  - note: a map is merely an ordered set of pairs (ie., key-value pairs)

```
val names = List("Michael", "Mark", "Tim")
val cs = Array("Purple", "Green", "Blue")
```

```
scala> for((name, c) <- names.zip(cs)) println(s"$name is assigned $c")
Michael is assigned Purple
Mark is assigned Green
Tim is assigned Blue
```

*to get a real* ***Map*** *object*

```
scala> names.zip(cs).toMap
res77: scala.collection.immutable.Map[String,String] =
    Map(Michael -> Purple, Mark -> Green, Tim -> Blue)
```

*or just,*

```
names zip cs
```

## Collection Idioms: zipWithIndex

- for-comprehensions are designed to iterate over data structures
  - often do not have – or should not have – a loop counter variable
- to get an index, use zipWithIndex…

```
scala> for((letter, index) <- alphabet.zipWithIndex) println(s"$letter is the ${index + 1}th lett
A is the 1th letter of the english alphabet
B is the 2th letter of the english alphabet
C is the 3th letter of the english alphabet
D is the 4th letter of the english alphabet
E is the 5th letter of the english alphabet
```

- `.zipWithIndex` on a sequence returns a sequence of the same type where each element is a tuple `(element, index)`

## Collection Idioms: Calling `traits`

- some traits (such as `Seq`) have companion objects with apply methods which return "canonical" implementations
  - the canonical `Seq` in Scala is the `List` and the canonical `IndexedSeq` is a `Vector`

```
val orderedSupplies = Seq("Apertif", "Soup", "Starter", "Main Course", "Dessert", "Coffee", "Dige
val randomSupplies = IndexedSeq("Food", "Water", "Shelter")
```

```
scala> randomSupplies
res3: IndexedSeq[String] = Vector(Food, Water, Shelter)

scala> orderedSupplies
res4: Seq[String] = List(Apertif, Soup, Starter, Main Course, Dessert, Coffee, Digestif)
```

## Collection Idioms: Empty

- Sequences have a constant-valued .empty property

```
scala> Nil
res18: scala.collection.immutable.Nil.type = List()

scala> List.empty
```

```
res14: List[Nothing] = List()

scala> List.empty[Int]
res15: List[Int] = List()

scala> Seq.empty[Int]
res16: Seq[Int] = List()

scala> Vector.empty[Int]
res17: scala.collection.immutable.Vector[Int] = Vector()

scala> Map.empty[String,String]
res20: scala.collection.immutable.Map[String,String] = Map()
```

- `.empty` is consistent across various types and is not a method call (`.apply()`)
  - it is sometimes preferred to construct (eg. `::`) from `.empty` # Combinators and Comprehensions

*a combinator..*

> ” is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments” —wiki

- combinators are the heart of functional programming
  - **much more so than explicit recursion**

## map

- map is the simplest combinator,
  - it operates on the values “inside” a type
  - for a collection, that’s it’s elements

```
val appendLocation = name: String => name + " lives in the UK"

val ukPeople = List("Michael", "Dan", "Tim")

ukPeople.map(appendLocation)
```

*or,*

```
val ages = List(7, 12, 18, 20, 21, 22)
val isAdult = (age: Int) => age > 18

ages.map(isAdult)
```

## Mapping with Simplified Syntax

- when a function is expected (as a typed parameter) simpler lambda-defintion syntax can be used

- it is also possible to take the parentheses off map
  - and use braces to indicate a code-block

```
val ages = List(7, 12, 18, 20, 21, 22)

ages.map(x => x > 18)

ages map { x => x > 18 }
```

- blocks can use an underscore to anonymously refer to parameters in sequence

```
age map { _ + 1}
```

- which is idiomatic scala

## flatMap

- flatMap when applied to containers is straightforward…

```
val people = List("Michael", "Tim")

def workLocations(name: String) = {
  val officeDB = Map(
    "Michael" -> List("London", "Leeds", "Cheltenham"),
    "Tim"     -> List("Manchester", "London")
  )

  officeDB(name)
}

val listOfLocations = people.map(workLocations)
val locations = people.flatMap(workLocations)

val uniqueLocations = people.flatMap(workLocations).toSet
```

```
scala> listOfLocations
listOfLocations: List[List[String]] = List(List(London, Leeds, Cheltenham),
List(Manchester, London))

scala> people.flatMap(workLocations)
locations: List[String] = List(London, Leeds, Cheltenham, Manchester, London)
```

- here flatMap "removes a layer of List"
  - in general flatMap does something similar to map:
  - it for a generic type `F[A]` it applies a function "within" that type, ie. to F's A(s)
  - however it expects that when it applies this function it will get another F back (so we would have `F[F[A]]`)
  - flatMap then 'flattens' this structure by knowing how F's are related to one another
- in the case of List this 'flattening' in effect means concatenation…
  - ie. note that the two inner lists are concatenated

## folding

- folding means 'reduce down to one value' by applying one operation over-and-over
  - as in the folding of a piece of paper: reduce it down by applying the fold operation several times

```
val areAdults =  ages map { _ > 18 }

val areAllAdults = areAdults.foldLeft(true)( _ && _  )
val areAnyAdults = areAdults.foldLeft(false)( _ || _ )
```

- note that when "passing underscores" like this a code block is implied…
- left vs right won't make a difference if your operation is commutative,
  - ie., if fn(a, b) == fn(b, a)

```
val chars = List('H', 'e', 'l', 'l', 'o')

chars.foldLeft("") { (total, next) => total + next }
chars.foldRight("") { (total, next) => total + next }
```

- left-wise folds are tail-recursive

## Predicates

```
val isAdult = (age: Int) => age > 18
```

- isAdult is a predicate
  - 'predicate' just means property, so a function of this kind just tells us what properties things have
  - which is why they are usually phrased as isSomeProperty, as in isBlue, isTall, isNextTo

## exists and forall

- the methods `exists` and `forall` on Traversables "converts to bools" via a predicate function and folds for us
- the `exists` method tells us if there is something in a list which has a property
  - ie. "does something exist with this property"
  - ie. apply a predicate function to every element and fold by OR'ing them

```
val people = List(18, 19, 20, 12)

people.exists(isAdult)  // are there any adults, == true
```

- the `forall` method requires a predicate to be true of every element
  - ie. "does everything have this property"
  - ie. apply a predicate function to every element and fold by AND'ing

```
people forall { _ > 18 } //are they all adults? false
```

```
val locations = List("UK - London", "UK - Leeds")

locations exists {  _ contains "FR" } //any locations in france? - false
//ie. does a string exist in the list for which the predict returns true

locations forall { _ contains "UK" } // are all locations UK? - true
//ie. is the predicate function true for all the strings
```

## reduce

- often the initial element to fold from is contained in the list
  - reduce will pick the first element as the initial value of its accumulator

```
val places = List("This ", "is ", "a ", "sentence ")

places reduceLeft { _ + _ }
```

- "left" versions are often more efficient (over Lists, etc.), and are the standard

## reduce by composition

- reduce a list of functions by composition ( `_ andThen _` )
  - gives you a single function that then calls the others

```
val inputList = List("michael  ", "26", "leeds  ", "united kingdom")

def printList(list: List[String])    = { println( list mkString ";" ); list }
def trimList(strings: List[String])  = strings map { _.trim }
def upperList(strings: List[String]) =  strings map { _.capitalize }

val flow = List(printList _ , trimList _ , upperList _ , printList _ )
val myProgram =  flow reduceLeft { _ andThen _ }

myProgram(inputList)
```

*or even,*

```
 List(printList _ ,
      trimList _ ,
      upperList _ ,
      printList _ ) reduceLeft { _ andThen _ } apply(inputList)
```

## For Comprehensions

- the for-comprehension syntax is syntax sugar
  - it effectively desugars into `map`, `flatMap` and `foreach` calls
  - though, significantly, an extraction is a pattern match

- `foreach` iterates over a collection like `map` without collecting values
  - ie., the return type of `foreach` is `Unit`

## For Comprehensions: yield and map

```
for(id <- List(1, 2, 3)) yield id + 1000

List(1, 2, 3) map { id => id + 1000 }

res6: List[Int] = List(1001, 1002, 1003)
```

- a yield is a map over a collection
  - ie. return a value for every value

```
for(pet <- List("Fluffy", "Spot")) println(pet)

List("Fluffy", "Spot") foreach { pet => println(pet) }

scala> List("Fluffy", "Spot") foreach { pet => println(pet) }
Fluffy
Spot
```

- without a yield a comprehension is just `foreach`ing

## For Comprehension: extraction and flatMap

- extractions correspond to `flatMap`s and `yield`s to `map`s
  - except when there is only one extraction, there is no need to `flatMap` it

```
val ids  = List(1, 2, 3)
val pets = List("Fluffy", "Spot")

for(id <- ids; pet <- pets) yield s"${id + 1000}: $pet"

ids.flatMap { id =>
    pets.map { pet => s"${id + 1000}: $pet" }
}

scala> for(id <- ids; pet <- pets) yield s"${id + 1000}: $pet"

res3: List[String] = List(1001: Fluffy, 1001: Spot, 1002: Fluffy,
1002: Spot, 1003: Fluffy, 1003: Spot)

scala> ids.flatMap { id => pets.map { pet => s"${id + 1000}: $pet" } }

res4: List[String] = List(1001: Fluffy, 1001: Spot, 1002: Fluffy,
1002: Spot, 1003: Fluffy, 1003: Spot)
```

- note the yield is still a map
  - it now concerns two collections (`id` from `ids`, `pet` from `pets`)

- the `flatMap` over `ids` says "forevery id perform a map over pets, with this id"
  - it's a `flatMap` because this `map` returns a List
  - (and we wish to have one list for the total set of ids, not a list of lists)
  - note that since a function is passed to `flatMap`, if `flatMap` is empty (, None, etc.) this function will not execute

## For Comprehensions: combinators over Option, List

```
val pidsNil = Nil flatMap { id =>
     pets map { pet => s"$pet is $id" }
}

val pidsSome = Some(10) flatMap { id =>
    Some("Fluffy") map { pet => s"$id: $pet" }
}

val pidsNone = None flatMap { id =>
    Some("Fluffy") map { p => throw new Exception("OMG!") }
}
```

```
scala> pidsNil
res11: List[String] = List()

scala> pidsSome
res12: Option[String] = Some(10: Fluffy)

scala> pidsNone
res13: Option[String] = None
```

## Pattern Matching

- pattern matching is the deconstruction of objects into their constituent parts
  - compare with regex which extract information from strings
- destructuring tuples (or 'decomposing' contra compose)

```
scala> val (y, (_, z)) = ("File System", ("NTFS", 1024))
y: String = File System
z: Int = 1024
```

- this is just a specific case of a very general syntax
  - in general, a large number of patterns may be used on the LHS of an assignment
  - matched to the RHS
  - thus extracting values from the RHS

```
case class OperatingSystem(val name: (String, String), val addressSize: Int)

scala> val OperatingSystem((publisher, title), _) =
    OperatingSystem(("Microsoft", "Windows 10"), 64)
```

```
publisher: String = Microsoft
title: String = Windows 10
```

## Pattern Equality

- scala defines equality between patterns and values
  - to test to see if a pattern matches

```
scala> (("Microsoft", _), _) == (("Microsoft", "Windows 10"), 32)
<console>:8: error: missing parameter type for expanded function
...
```

- the underscore syntax here means 'discard'
- patterns more often go in the `case` clauses of `match` expressions

```
val os = (("Microsoft", "Windows 10"), 32)

val isMicrosoft = os match {
    case (("Microsoft", _), _) => true
    case _ => false
}
```

- os is compared to two cases (pattern expressions) if it matches the first it's true, and the second its false
  - note that _ in the case-context now becomes a wildcard (which discards information as before)

## Matching and Extracting

- values may be extracted whilst pattern matching

```
val os = (("Microsoft", "Windows 10"), 32)

val whichMicrosoft = os match {
    case (("Microsoft", os), bit) => s"MS $os - $bit bit"
    case _ => 'None'
}
```

## `case` Clauses

- java has a primitive form of basic case-matching with the `switch` statement

```
def day(name: String) = name slice (0, 2) match {
    case "Mo" => 1
    case "Tu" => 2
```

```
    case "We" => 3
    case "Th" => 4
    case "Fr" => 5
    case "Sa" => 6
    case "Su" => 7
    case _ => 0
}
```

- this compares the `case` expression the value being inspected `name slice(0, 2)`
  - (the first two letters of name)

```
scala> day("Monday")
res16: Int = 1

scala> day("Tue")
res17: Int = 2

scala> day("Nonesense")
res18: Int = 0
```

- `_` matches any input, i.e., it provides the default case
  - the method will throw an exception if there is no wildcard case (`_`)

## Options

- we can conveniently match an option

```
val maybeName = Some("Michael")
val maybeAnotherName = None

val message = maybeName match {
  case Some(name) => name + " likes Scala!"
  case None => ""
}
```

## Sequences

```
List("Michael", "John", "Burgess")
Vector("Michael", "John", "Burgess")
IndexedSeq("Michael", "John", "Burgess")
Map("type"-> "human", "height" -> 2, "weight" -> 100).toSeq
```

- scala can match against all of these kinds of sequence (noting .toSeq on Map)
  - recall that when matching against a tuple its structure is described to access its parts
  - the same holds for sequences

```
scala> 1 +: Seq(2,3)
res0: Seq[Int] = List(1, 2, 3)
```

*therefore,*

```
def breakUp[T](seq: Seq[T]): String = seq match {
  case head +: tail => s"$head +: " + breakUp(tail)
  case Nil => "END"
}
```

- `Nil` matches the end of any sequence
- `+:` is defined for every sequence ( in the `List` case this is just `::`)

```
scala> breakUp(Map("type"-> "human", "height" -> 2, "weight" -> 100).toSeq)
res26: String = (type,human) +: (height,2) +: (weight,100) +: END

scala> breakUp(IndexedSeq("Michael", "John", "Burgess"))
res27: String = Michael +: John +: Burgess +: END
```

## Tuples

```
val startsWith = ("First", "Second", "Third") match {
    case ("A", _, _) => "Starts with A"
    case ("First", _, _) => "Starts with First"
    case (_, _, _) => "Unknown!"
}

val tupleList = List(("A", "B", "C"), (1, 2, 3), (false, 1, "B"))

val eachStartsWith = for (triple <- tupleList)
    yield triple match {
        case ("A", _, _) =>  "Starts with A"
        case (1, _, _)   =>  "Starts with 1"
        case (_, _, _)   =>  "Unknown!"
    }
```

```
scala> startsWith
res32: String = Starts with First

scala> eachStartsWith
res33: List[String] = List(Starts with A, Starts with 1, Unknown!)
```

- for yields a `List` because we're for-ing over one (ie. calling `map` on a `List`)

## Types

- pattern matching on types is limited
  - generics have to be wild-carded `_`, due to type-erasure

```
type ElementFunction[R] = Seq[S] => R
```

```
def matchSequence[S, D, R](s: Seq[S], nil: D)(): R = seq match {
  case Nil => nil
  case head +: _ => {

  }
}

for {
  x <- Seq(List(5.5,5.6,5.7), List("a", "b"), Nil)
} yield {
  x match {
    case seq: Seq[_] => (s"seq ${doSeqMatch(seq)}", seq)
    case _           => ("unknown!", x)
  }
}
```

## Regex

- regex fits into the pattern matching system very naturally

```
val people = List("Richard Dawkins", "Richard Feynman", "Lewis Carol", "David Lewis", "Carol Rich

val Richards = raw"Richard (\w+)".r
val Lewises =  raw"(?:\w+ )?Lewis(?: \w+)?".r


for (person <- people) person match {
     case Richards(name) => println("a scientist named: " + name)
     case Lewises() => println("a logician")
     case _ => println("something else")
}
```

```
a scientist named: Dawkins
a scientist named: Feynman
a logician
a logician
something else
```

## Guards

- matches can also filter on the *value* of the data matched

```
type Person = (String, Int)

val people: Seq[Person] = Seq(("Michael", 26), ("Sarah", 15), ("John", 18))
```

```
for(person <- people) println(
    person match {
        case (name, age) if age >= 18 => s"$name is eligible for full-time work "
        case (name, _) => s"$name is not eligible for full-time work or needs a performance licen
    }
)
```

```
Michael is eligible for full-time work
Sarah is not eligible for full-time work or needs a performance licence
John is eligible for full-time work
```

## case classes

- how does pattern matching work?
  - the `unapply` method on a companion object handles the matching process
  - it should return an `Option` of a tuple

```
class Person(val name: String, val age: Int)

object Person {
  def unapply(p: Person) = Some((p.name, p.age))
}

val aliceSm = Person("Alice Smith", 8)
val bobSm   = Person("Bob Smith", 35)
val aliceDo = Person("Alice Doe", 21)

val message = aliceDo match {
  case Person("Alice Doe", _)
    => "hello alice doe!"

  case Person(name, _)
    if name.contains("Alice")
    => "hello alice!"

  case Person(_, age)
    if age > 18
    => "hello adult!"

  case Person(_, _)
    => "hello child"
}
```

- `unapply`'s tuple-order defines match order, and matching behaviour
  - return None from companion to prevent matching
  - or make matching "loose" (eg. name.contains(match))
  - this is a bad idea mostly, just let people use guards
- case classes automatically create a companion object with expected behaviour,

```
case class Person(val name: String, age: Int)
```

- sequence matching (eg. with `::`) is a little different
  - the `unapplySeq` performs the matching and should return an `Option` of a `List`

# Developing with Scala

# Test-Driven Development

- "ordinary development" often leads to fragile codebases that seem to work by accident rather than design
- due to the total complexity of larger applications no area can be modified with confidence — and there is always the expectation that modification will unpredicably break another area of the application
- codebases developed in this ad-hoc manner become difficult to reason about, understand and change
- Test-Driven Development is one proposed solution to this problem
- the name does not express its purpose very well: for "Test" we should read, "Executable Specification" — that is, when practicing TDD we first write a specification our code should follow, where this specification is itself executable
- we *then* develop code to match this specification and we continue until it executes without any failures (ie. missing features)
- we are then in a position to:
  - read the specification in order to understand what the code does
  - rely on the specification to check our code works **esp. under modification to the code**
  - use the specification to see how refactoring our code affects the rest of the code base
  - use the specification as documentation, ie. to pass on knowledge about the codebase

# Assertions

- we *could* write specifications very simply ourselves
- scala provides `assert` function which throws an `AssertionError` on failure
- suppose we wish to provide some functions to get data from a person…

```
type Person = (String, String) //gives us freedom to use classes, etc. later on
val person: Person = ("", "")

def getName(p: Person) {}
def getLocation(p: Person) {}
```

```
assert(getName(person) == "Michael")
assert(getLocation(person) == "UK")
```

- these will fail, we can make the UI at little neater

```
type Specification

def runSpec(fn: Specification) {


}

def assert(c: Boolean): Boolean {
    //...
}

runSpec {
    assert(getName(person) == "Michael")
    assert(getLocation(person) == "UK")
}
```

- rather than write a library of this kind however, we can use some pre-existing ones...

## ScalaTest

- ScalaTest is a TDD framework for scala in the "xUnit" famility of TDD frameworks
- add the jar to the build path & run a simple test

## FlatSpec

```
import collection.mutable.Stack
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should be (1)
    stack.pop() should be (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[Int]
    a [NoSuchElementException] should be thrownBy {
      emptyStack.pop()
    }
```

```
  }
}
```

### FeatureSpec

### Assertions and Matchers

## Type Systems are Tests# Implicits

- implicit parameters are execution contexts (eg. types)
- implicit functions are capabilities
- implicit classes are abstractions over types (type classes)

## Implicit Parameters

“‘{.scala caption=“”} scala> implicit val prefix = “Logging:” prefix: String = “Logging:”

scala> def myPrinter(s: String)(implicit prefix: String) = println(prefix + s) myPrinter: (s: String)(implicit prefix: String)Unit

scala> myPrinter(“Hello”) Logging: Hello “‘

## Implicitly

“‘{.scala caption=“” label=“PS2-p129” caption=“named vs annoymous implict parameters”} import math.Ordering

case class MyListA { //named def sortBy1B(implicit ord: Ordering[B]): List[A] = list.sortBy(f)(ord)

//annoymous def sortBy2B : Ordering: List[A] = list.sortBy(f)(implicitly[Ordering[B]]) }

val list = MyList(List(1,3,5,2,4))

list sortBy1 (i => -i) list sortBy2 (i => -i) “‘

- `B : Ordering` means *an* `Ordering[B]` exists
  - this compiles (is true) when there's an implicit value of type `Ordering[B]` available

## Implicit Conversions

- automatic “lift”ing

```
object Implicits {
  implicit final class SpaceShip[A](val self: A) {
    def <-*-> [B](y: B): Tuple2[A, B] = Tuple2(self, y)
```

```
  }
}

import Implicits._

// converted to a SpaceShip because <-*-> exists on SpaceShips
// then <-*-> is called

val m = Map("one" <-*-> 1, "two" <-*-> 2)
```

## Implicit Patterns

## Type Classes

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

trait ToJSON {
  def toJSON(level: Int = 0): String

  val INDENTATION = "  "
  def indentation(level: Int = 0): (String,String) =
    (INDENTATION * level, INDENTATION * (level+1))
}

implicit class AddressToJSON(address: Address) extends ToJSON {
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"street": "${address.street}",
      |${indent}"city":   "${address.city}"
      |$outdent}""".stripMargin
  }
}

implicit class PersonToJSON(person: Person) extends ToJSON {
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"name":    "${person.name}",
      |${indent}"address": ${person.address.toJSON(level + 1)}
      |$outdent}""".stripMargin
  }
}

val a = Address("1 Scala Lane", "Anytown")
```

```
val p = Person("Buck Trends", a)

println(a.toJSON())
println()
println(p.toJSON())
```

# The Design of Scala Programs

- How should we define a functional program?
- In general, the aim is to phrase the design of a program as a series of function calls…

```
DataOut program(DataIn input) {
    return f( g( h( input ) ) );
}
```

In scala, depending on our definitions,

```
val program: Din => Dout = h andThen g andThen f

def program(input: Din): Dout = f(g(h(input)))

val program: Din => Dout = f _ compose g _ compose h _

val functions = List(f, g, h)

val program = functions.reduceLeft(functions)
```

- where f, g, h are referentially transparent
- as we have seen things are not so simple
- let's review the core features of the language so far and how we might use them to design programs in scala

## Patterns of the Language

The fundamental patterns are:

- compose
- decompose
- group
- recurse
- abstract
- aggregate
- mutate

each pattern corresponds to scala language constructs

## Composition

- everything is an expression
- expressions are composable
- they go everywhere

```
val result = if(x) "T" else "F"
```

## Decomposition

- what has been composed?
- achieved through pattern matching

```
trait StrNumber

case class StrInt(s: String) extends StrNumber
case class StrFloat(s: String) extends StrNumber


def printNumber(s: StrNumber) {
  s match {
    case StrInt(n) => println("Int " + n)
    case StrFloat(n) => println("Float " + n)
  }
}
```

- printNumber can tell which kind of StrNumber its argument is
- without each class having its own print method
- per-class operations "OK" if number is known and fixed
- if not, then extending classes/etc. is a nightmare
- if data structure is fixed, and operations is variable then case classes make most sense
- scala offers both approaches
- matches type and extracts content — decomposes
- somewhat verbose compared to Haskell, but small % application anyway

## Grouping

- everything can be grouped / nested inside everything
- static scoping
- two namespaces: types & values (aka terms & tokens)
- same rules for each

```
def solve(problem: Problem): Solution = {
   def isSolution(g: Guess): Bool = lib.test(g.someOperation())``
```

```
   problem.allGuesses.filter(isSolution)
}
```

- nest whatever you need where you need it
- "rookie mistake for functional programming (**in scala**)" is long-chaining:

```
  get(
    something.filter(
      somethingElse.flatMap( x =>
        x.fubar
      )
    )
```

- (**might be neater in other languages...**)

```
get $ filter something $ map fubar x
```

- every intermediate result needs a "meaningful name"
- (**programs should express intent**)
- grouping/nesting allows placing constructs in 'natural scope'
  - not just on class because it has to be

## Recursion

- compositional
- to arbitrary depths
- fallback if combinator approach fails (eg. series of flatMaps, ...)
- tail-rec functions reduce to loops
- can check if tail-rec with `@tailrec`

## Abstraction

- functions are values
- functions *abstract computations into values* ie. turn process into data

## Aggregation

- collect data
- transform (*vs CRUD = BAD*)
- uniform set of operators & apply everywhere

```
val numbers: List[Int] = List(1, 4, 5)

numbers.map( _ / 2 ) == List(0, 2, 2)   //note integer division
```

reusing map across different kinds of collections...

```
val numbers: Set[Int] = Set(1, 4, 5)

numbers.map( _ / 2 ) == Set(0, 2)       // sets are not seqs --- remove duplicate
```

## Mutation

- requires restraint
- minimize rather than eliminate, eg. uses of mutation in scala compiler
  - caching (optimization)
  - persisting (optimization)
  - copy on write (optimization)
  - unique ids
  - diagnostics
  - NB. total state for type checker is 2 `var`s

## A Focus on Modules

a module can be a

- function
- object
- class
- stream
- microservice
- actor
- focus is on how modules can be combined

## Features for Modularity

- vocabulary & semantics provided by rich type system
- start with Objects — atomic modules (ie. smallest whole module)
- parameterize modules via classes, generics, etc.
- traits —for slice of behaviour

## Nominal Abstraction

- ie. (by name, ie. traditional OO)

very general definition of graph library:

```
trait Graphs {
  type Node       //specific types as yet unknown, only that it exists
  type Edge

  def succ(e: Edge): Node
  def pred(e: Edge): Node
```

```
  //...

  //merely a "promise" that a type exists
  type Graph <: GraphAPI  //a graph has the API

  trait GraphAPI {
    def sources: Nodes
    def nodes: Set[Node]
    //...
  }
}
```

- pure interface
- ..provides vocabulary (ontology)
- we can then specify a particular implementation,

```
trait AbstractModel extends Graphs {

  //here we dont yet know what Nodes & Edges are
  class Graph(val nodes: Set[Node], val edges: Set[Edges]) extends GraphAPI {

      lazy val sources = nodes filter (incoming(_).isEmpty)

      //...
  }
}


//and we can provide an implementation of Node/Edge too

trait ConcreteModel extends Graph {
  type Node = Person
  type Edge = (Person, Person)

  def succ(e: Edge) = e._1
  def pred(e: Edge) = e._2
}


class PersonGraph extends AbstractModel with ConcreteModel
```

- we encapsulate by parameterized abstraction
- ie. in this case, we do not know what Node/Edge is in general —-thus abstracting
- we then provide them later as parameters

## Structural Abstraction

- ie. ( by position, ie. typical functional )

- parameterize our abstractions by position

```
class List[+T]
class Set[T]
class Function1[-T, +R]

List[Number]
Set[String]
Function1[String, Int]
```

## Positions *are* Names

```
class Set[T]      ===     class Set  { type $T }

class List[+T]    ===     class List { type $T }

List[Number]      ~==     List { type $T <: Number}
```

- cf. this with tuples being anonymos classes, tuples are structural / classses are nominal

## Implicits & Boilerplate

- implicit parameters —"simple concept"
- (… were intended to model typeclasses in Haskell

```
def min(x: A, b: A)(implicit cmp: Ordering[A]): A
```

- min needs to find a way to compare, so provided implicitly
- to be explicit would require passing the same compare function lots of times
- and probably given its the same type, that will be the same function
- …)
- can also represent a context (something the same across function calls )

```
def compile(cmdLine: String)(implicit defaultOptions: List[String]): Unit
```

- can represent a capability

```
def accessResource(id: CustomerID)(implicit adminACL: AdminRights): Resource
```

- implicit parameters also provides easy dependency injection

## Scala Extras

Additional elements to the language that provide more "rough edges" as they are mixed together with the above

- implicit conversions
  - != implicit parameters
  - implied way of using As as Bs

  - the more implied conversions, the stranger the interactions
- existential types
- structural types
  - sometimes complicated and require reflective access
- higher-kinded types
  - "in the end we need them"
  - not well integrated with other types & type system
  - no clear role in type parameter / fields system
- macros
  - very powerful, but complex

(All of these are under feature/experimental flags)

- experimental features require clear use case
- eg. scala as a host for a DSL

# Resources

# Books

## Guides

- Programming in Scala, Odersky et al., — 2nd Ed., 2011, artima;
- Programming Scala, Wampler & Payne — 2nd Ed., 2014, O'Rielly;
- Scala for the Impatient, Hortsmann — 1st Ed., 2012, Addison-Wesley;

## Core

- Scala By Example, Odersky — 1st Ed., 2014, scala-lang.org;
- Scala Language Specification, Odersky — 2.12 Ed., 201?, scala-lang.org;

## Functional Programming

- Functional Programming in Scala, Chiusano & Bjarnason — 1st Ed., Manning, 2015;

# Videos

## Odersky on the Languge

- Scala with Style, Odersky — 2013, ScalaDays, youtube
  - https://www.youtube.com/watch?v=kkTFx3-duc8
- Scala: The Simple Parts, Odersky — 2014, SF Scala Meetup, youtube
  - https://www.youtube.com/watch?v=ecekSCX3B4Q
- The Trouble with Types, Odersky — 2013, StrangeLoop, InfoQ

  - http://www.infoq.com/presentations/data-types-issues
- Binary Compatibility in Scala, Odersky — 2015, SF Scala, youtube
  - https://www.youtube.com/watch?v=TXNs51UII60

## And More

- "SF Scala"
  - https://www.youtube.com/results?search_query=sf+scala