

Introduction to Scala

George Ball
J&G Services Ltd

Contents

1. Introduction and Background
2. The Scala Language
3. Object Oriented Scala
4. Inheritance
5. Functional Programming with Scala
6. Collections
7. More Functional Programming
8. Exceptions

Introduction and Background

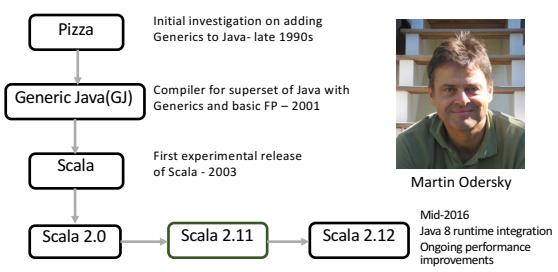


Welcome to Scala

- Scalable Language
- A modern language developed for the Java™ platform
 - Interoperates with Java
 - Also .Net, JavaScript versions
- Supports Object Oriented and Functional paradigms
- Many toolkits/frameworks built on Scala
 - Akka
 - Play
 - Slick
 - Spark



A Short History



Concise Syntax

- Java

```
public class Person {
    private String firstName;
    private String lastName;
    public Person() {
        this("John", "Doe");
    }
    public Person( String first, String last ) {
        this.firstName = first;
        this.lastName = last;
    }
    public String first() {
        return this.firstName;
    }
    public String last() {
        return this.lastName;
    }
}
```

Concise Syntax

- Scala

```
class Person( val first: String = "John",
             val last: String = "Doe" )
```

Getting Started

- A first Scala program

A "Singleton object, define class and instance"

No requirements on file naming

Hello scala

```
object HelloWorld {
    def main( args: Array[String] ) {
        println("Hello from scala")
    }
}
```

Further simplification possible:

```
object HelloWorld extends App {
    println("Hello from scala")
}
```

Semicolon optional as separator at end of line

Type follows identifier in declarations (where type is needed)

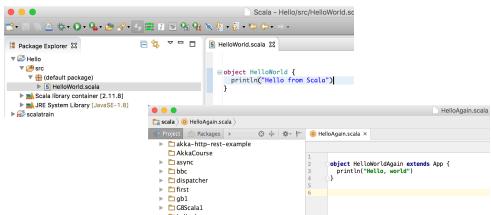
Running the Program

- Standard Scala compile/execution tools available
 - similar to JDK

```
$ scalac Hello.scala
$ ls -l
-rw-r--r-- 1 george staff 604 18 Sep 09:27 HelloWorld$.class
-rw-r--r-- 1 george staff 632 18 Sep 09:27 HelloWorld.class
$ scala HelloWorld
Hello from scala
```

Using an IDE

- Plugins available for common IDEs



The Scala REPL

- An interactive mode for experimenting with Scala

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51).
Type in expressions for evaluation. Or try :help.

scala> println("Hello to all...")
Hello to all...

scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
...
:load <path>           interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:quit                  exit the interpreter
:replay [options]       reset the repl and replay all previous commands
:require <path>         add a jar to the classpath
:type [-v] <expr>       display the type of an expression without evaluating it
...
```

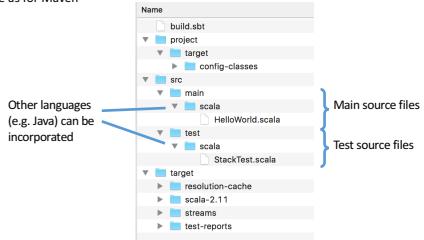
sbt: The Scala Build Tool

- Like Gradle
- Build files use Scala based DSL
 - Unlike Maven which uses XML
- Leverages Ivy for dependencies
- Incremental compilation reduces build times
 - Also server mode of operation
- Basis of Lightbend Activator tool



sbt Project Layout

- sbt has a basic structure for projects
 - Same as for Maven



Working with sbt

```
$ sbt
...
> run
[info] Compiling 1 Scala source to .../target/scala-2.11/classes...
[info] Running HelloWorld
Hello all
[success] Total time: 2 s, completed 08-Aug-2016 15:42:10
>

-> test
[info] Run completed in 324 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 1 s, completed 08-Aug-2016 15:50:41
>
```

The sbt Build File

- build.sbt
 - Written using a Scala DSL

Possible to manage build
for several Scala versions

```
name := "Hello World"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.specs2" %% "specs2-core" % "3.8.4" % "test"
libraryDependencies += Seq("org.scalatest" % "scalatest_2.11" % "3.0.0" % "test",
  "org.scalactic" %% "scalactic" % "3.0.0")
```

Dependencies can be specified
individually or as a list

The sbt Console

- Allows Scala REPL interaction with dependencies resolved

```
$ sbt
[info] Set current project to Hello World (in build file:./latest/)
> console
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51).
Type in expressions for evaluation. Or try :help.

scala> println("Hello there")
Hello there

scala> :quit
[success] Total time: 4 s, completed 08-Aug-2016 16:17:33
>
```

Continuous Mode

- sbt can monitor for changes to files in the build
 - rerun the task if any change is detected
 - prepend ~ to task

```
$ sbt
[info] Set current project to Hello World (in build file
> ~run
[info] Running HelloWorld
Hello all
[success] Total time: 0 s, completed 08-Aug-2016 16:18:46
1. Waiting for source changes... (press enter to interrupt)
[info] Running HelloWorld
Hello all
[success] Total time: 0 s, completed 08-Aug-2016 16:19:11
2. Waiting for source changes... (press enter to interrupt)
>
```

Introducing the Scala Language



Basic Principles

- Three language "rules"
- Everything is an expression
 - Should yield a value
 - If no value returned, expression is known as a statement
- Every value is an object
 - Compiler optimises to use JVM primitive types when appropriate
- Every operation is a method call

Values

- Named pieces of *immutable* storage

```

scala> val myVal = 10
myVal: Int = 10
          Type inferred from
          initialising expression

scala> myVal += 20
<console>:9: error: reassignment to val
           myVal += 20
                   ^
          val introduces immutable data

scala> val y: Int = "Hello"
<console>:7: error: type mismatch;
           found   : String("Hello")
           required: Int
           val y: Int = "Hello"
                   ^
          Explicitly typed ~
          compiler ensures initialising
          expression is compatible

scala> val z = { val a = 5; a + 3 }
z: Int = 8
          Expression may be compound
  
```

Variables

- Named pieces of *mutable* storage
 - may be initialised after definition
 - must be initialised before use
- Use of mutable data is discouraged in Scala

```
scala> var myVar: Int = 20
myVar: Int = 20

scala> myVar += 10

scala> myVar
res3: Int = 30

scala> myVar + myVal
res4: Int = 40
```

var and *val* may be mixed
in expressions

Methods/Functions

- Use *def* keyword

```
scala> def times2 ( i: Int ) = i * 2
times2: (i: Int)Int
scala> times2( 3 )
res17: Int = 6

scala> def upper ( strings: String* ) = strings.map( _ .toUpperCase() )
upper: (Strings: String*)Seq[Java.lang.String]
scala> upper ( "one", "two" )
res18: Seq[java.lang.String] = ArrayBuffer(ONE, TWO)

scala> def sayHello = println( "Hello everyone" )
sayHello: Unit
scala> sayHello
Hello everyone
```

No args so()
not required

Unit similar to
Java void type

About Strings

- Scala String type based on `java.lang.String`
 - Some additional capabilities
- String interpolation allows Scala expressions to be evaluated inside String literals

```
scala> val a = 1
a: Int = 1
scala> val b = 3
b: Int = 3
scala> "$a + $b = ${a + b}"
res2: String = 1 + 3 = 4
scala> f"$a%d + ${b%2d} = ${a + b}%3.2f"
res3: String = " 1 + 3 = 4.00"
```

Simple substitution

Formatted substitution

Regular Expressions

- Powerful notation for working with Strings

```
scala> val s1 = "I like coffee before lunch, and Tea after lunch"
s1: String = I like coffee before lunch, and Tea after lunch
scala> s1.matches( raw".*[tT]ea.*" )                                raw string interpolator prevents
res14: Boolean = true                                                 expansion of \.. sequences in string.
scala> s1.replaceAll( raw"[tT]ea", "coffee" )                         res16: String = I like coffee before lunch, and coffee after lunch
scala> s1.replaceAll( raw"[tT]ea", "coffee" ).replaceFirst( raw"[Cc]offee", "tea" ) res12: String = I like tea before lunch, and coffee after lunch
```

Regular Expressions

- RE Capture groups may be accessed
 - Slightly unusual syntax

```
scala> val s1 = "I like coffee before lunch, and Tea after lunch"
s1: String = I like coffee before lunch, and Tea after lunch
scala> val drink = raw".*([tT]ea|[Cc]offee) before.*([tT]ea|[Cc]offee) after.*" r
drink: scala.util.matching.Regex = .*(?i)ea|(?c)offee before.*(?i)ea|(?c)offee after.*
scala> val drink(morningDrink, afternoonDrink) = s1
morningDrink: String = coffee
afternoonDrink: String = Tea
```

Conditional Expressions

- if expression**
 - Similar syntax to Java/C/C++
 - Similar semantics to ?: operator
 - Yields a value

```
scala> val amount = 25000
amount: Int = 25000
scala> val taxRate = if ( amount < 41000 ) 0.25 else 0.4
taxRate: Double = 0.25
scala> val x = 10
x: Int = 10
scala> if ( x % 2 == 0 ) println("even") else println("odd")
even
```

Unit valued expression/statement

Pattern Matching

- **match expression**

- similar to switch statement
- each branch is an expression

```
scala> val month = 5
month: Int = 5

scala> month match {
|   case 1 => println("January")
|   case 2 => println("February")
|   case 3 => println("March")
|   case 4 => println("April")
|   case 5 => println("May")
|   ...
|   case 12 => println("December")
|   case _ => println("Ooops")
}

May
```

No automatic fall-through,
so no need for break

Default case represented
using `_` wildcard

Pattern Matching

- **match** is an operator (method)

- Part of expression yielding a value

```
scala> def season ( m: Int ) =
|   m match {
|     case 1 | 2 | 3 => "Winter"
|     case 4 | 5 | 6 => "Spring"
|     case 7 | 8 | 9 => "Summer"
|     case 10 | 11 | 12 => "Autumn"
|     case _ => "Weird"
|   }
season: (m: Int)java.lang.String

scala> season(2)
res5: java.lang.String = Winter
```

- Example also shows matching multiple values

Pattern Matching

- Case options can have guards associated

- Allows continuous ranges of values to be matched

```
scala> def state ( t: Int ) =
|   t match {
|     case i if ( i < 0 ) => "ice"
|     case i if ( i >= 0 && i < 100 ) => "water"
|     case i if ( i >= 100 ) => "steam"
|   }
state: (t: Int)java.lang.String

scala> state (120)
res9: java.lang.String = steam

scala> state (-10)
res10: java.lang.String = ice
```

While Loop

- Conventional behaviour

- Imperative style

```
scala> var j = 1
j: Int = 1

scala> var tot = 0
tot: Int = 0

scala> while (j <= 5) {
|   tot += j
|   j += 1
| }

scala> tot
res10: Int = 15
```

Note var used as variables are mutable

```
scala> var j = 1
j: Int = 1

scala> while (j <= 5) {
|   if (j % 2 == 0)
|     println(j + ": even")
|   else
|     println(j + ": odd")
|   j += 1
| }

1: odd
2: even
3: odd
4: even
5: odd
```

Basic for Loop

- Special case of for comprehension

- Body is a statement

- Evaluated for its side effects
- Control variable is immutable

- Equivalent to foreach on input Seq

```
scala> for (a <- 1 to 5) println(a)
1
2
3
4
5
```

```
scala> 1 to 5 foreach (println(_))
1
2
3
4
5
```

Object Oriented Scala



Object Oriented Scala

- Scala supports the OO paradigm
- Every value is an object
 - Compiler uses JVM primitive types when appropriate
- Scala follows the Java approach to OO
 - Classes and traits define object types
 - Singleton objects available
 - Single inheritance of classes
 - Mixin inheritance provided through traits

```
scala> 350.toString
res0: String = 350
scala> :type 350
Int
scala> 350 + 1
res1: Int = 351
scala> (350).+(1)
res2: Int = 351
```

Defining a Class

- Easy to define a new class
 - Instantiate using new
 - REPL provides javap command to examine bytecodes

```
scala> :javap -c MyClass
Compiled from "<console>"
public class MyClass {
    public MyClass();
    Code:
        0: aload_0
        1: invokespecial #9
        4: return
    }

scala> class MyClass
defined class MyClass
scala> val myObj = new MyClass
myObj: MyClass = MyClass@14bf9759
scala> myObj.toString
res3: String = MyClass@14bf9759
```

Constructor

- Every class has a "primary" constructor
 - Code embedded between { ... }

```
class MyClass {
  println("Building a MyClass")
}

scala> val myObj = new MyClass
Building a MyClass
myObj: MyClass = MyClass@5db45159
```

- Class parameters may be defined
 - Accessible in constructor code

```
class Message( head: String, body: String ) {
  println(s"$head $body")
}

scala> val myMsg = new Message("Hello", "world")
Hello world
myMsg: Message = Message@5c669da8
```

Constructor

- Other constructors may be defined
 - Specify as method called `this`
 - Allows chaining, as per Java
 - Must eventually execute primary constructor

```
class Message( head: String, body: String ) {
  def this( bd: String ) = this("Hello", bd )
  println( s"Primary: $head $body" )
}

scala> val myMsg = new Message("Class")
Primary: Hello Class
myMsg: Message = Message@ff6077
```

Class Properties

- Class may have immutable or mutable data properties
 - Use `val` or `var` as appropriate

```
class Message( head: String, body: String ) {
  val msg = s"$head $body"
}

scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@6f9ad11c

scala> myMsg.msg
res9: String = Hello world

scala> myMsg.msg = "Goodnight everyone"
<console>:13: error: reassignment to val
          myMsg.msg = "Goodnight everyone"
               ^
```

Class Parameters and Properties

- Class parameters are not stored as properties
 - Only available in primary constructor

```
class Message ( head: String, body: String ) {
  val msg = s"$head $body"
}

scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@6f9ad1c

scala> myMsg.head
<console>:14: error: value head is not a member of Message
           myMsg.head
                  ^
```

Class Parameters and Properties

- Use val or var to promote parameters to properties

```
class Message( val head: String, val body: String ) {
  val msg = s"$head $body"
}

scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@5f59ea8c

scala> myMsg.head
res12: String = Hello
scala> myMsg.msg
res14: String = Hello world
```

Adding Methods

- Methods define calculations on instances of the class
 - Use def to define methods

```
class Arith ( val i: Int, val j: Int ) {
  def add = i + j
  def sub: Int = i - j
  def addAndMult( by: Int ) = (i + j) * by
}

scala> val arith = new Arith( 5, 2 )
arith: Arith = Arith@6ac4c3f7

scala> arith.add
res15: Int = 7

scala> arith.addAndMult(2)
res16: Int = 14
```

Compiler infers return type

Explicit return type
(recommended for public interface)

Parameter types must be specified

val and def – the Uniform Access Principle

- val and def are similar syntactically
 - If def has no parameters

```
class VClass {
  val now = new java.util.Date().toString
}

scala> val vObj = new VClass
vObj: VClass = VClass@16d07cf3

scala> vObj.now
res17: String = Wed Aug 10 11:45:46 CST 2016

scala> vObj.now
res18: String = Wed Aug 10 11:45:46 CST 2016
```

val and def – the Uniform Access Principle

- val causes expression to be evaluated *eagerly*
 - def causes expression to be evaluated *lazily*
 - Caller should not notice different syntax

```
class DClass {
  def now = new java.util.Date().toString
}

scala> val dObj = new DClass
dObj: DClass = DClass@7b53b1ad

scala> dObj.now
res20: String = Wed Aug 10 11:46:00 CST 2016

scala> dObj.now
res21: String = Wed Aug 10 11:46:16 CST 2016
```

A Complete Example

- Complex numbers:

◦ A simple (sic) implementation

Default value
for property

```
class Complex ( val re: Int, val im: Int = 0 ) {

  def + ( that: Complex ) =
    new Complex( this.re + that.re, this.im + that.im )

  def - ( that: Complex ) =
    new Complex ( this.re - that.re, this.im - that.im )

  override def toString = s"${re} + ${im}i"
}
```

Keyword is mandatory when
overriding method from base class

A Complete Example

- Single argument methods can be invoked as operators
 - "Normal" precedence applies for methods named using operator characters

```
scala> val c1 = new Complex(1,2)
c1: Complex = 1 + 2i
REPL uses toString
method for display

scala> val c2 = new Complex(3)
c2: Complex = 3 + 0i

scala> c1.(c2)
res23: Complex = 4 + 2i

scala> c1 + c2
res24: Complex = 4 + 2i
```

A Complete Example

- Method with no arguments can be written as postfix

```
scala> val c1 = new Complex(1,2)
c1: Complex = 1 + 2i

scala> c1 toString
res25: String = 1 + 2i
```

- Define unary_- methods for prefix +, -, !, ~

```
...
def unary_- = new Complex(-re, -im)
...

scala> -c1
res26: Complex = -1 + -2i
```

Equality

- Scala == and != operators check state equality
 - Compiler translates to call equals() method for compatibility with Java
- Use eq and ne methods to check identity equality
 - For reference types (see later)

```
scala> 3 == 3
res27: Boolean = true

scala> new String("foobar") == new String("foobar")
res28: Boolean = true

scala> new String("foobar") eq new String("foobar")
<console>:12: warning: comparing a fresh object using `eq` will always yield false
           new String("foobar") eq new String("foobar")
                           ^
res29: Boolean = false
```

Singleton Objects

- Class and single automatically created instance
 - Cannot create further instances

```
object MessageObj {
  val hd = "Hello"
  val bd = "World"
  def showMessage = s"$hd $bd"
}

scala> MessageObj.hd
res34: String = Hello

scala> MessageObj.showMessage
res35: String = Hello World
```

```
scala> new MessageObj
<console>:12: error: not found: type MessageObj
          ^
          ^
```

Companion Objects

- Singleton object compiled in same unit as a class is called a Companion Object

- Has access to all private items in the class
- Can be viewed as a container for static data and methods

```
class Message ( val hd: String, val bd: String ) {
  Message.count += 1
}
object Message {
  var count = 0
}

scala> val m1 = new Message( "hello", "world" )
m1: Message = Message@3292eff7

scala> val m2 = new Message("I see", "no ships")
m2: Message = Message@59014efe

scala> Message.count
res37: Int = 2
```

- Use :paste mode to have REPL compile both together

Object Factories

- Uses companion object and associated apply() method
 - Compiler rewrites () "operator" to call this method

```
class Message private ( val hd: String, bd: String )
object Message {
  var count = 0
  def apply( h: String, b: String ): Message = {
    count += 1
    new Message(h, b)
  }
}

scala> val m1 = new Message("from", "the constructor")
<console>:13: error: constructor Message in class Message
                                cannot be accessed in object $iw
                                ^
                                ^

scala> val m1 = Message("from", "the factory")
m1: Message = Message@18b45500
```

Case Classes

- Useful for representing types that present "value" semantics
- Compiler provides boilerplate implementation of methods
 - equals, hashCode, toString, copy
- Class parameters are automatically promoted to properties
- Companion object and apply method provided for construction

```
scala> case class Complex (re: Int, im: Int)
defined class Complex

scala> val c1 = Complex(1,2)
c1: Complex = Complex(1,2)

scala> c1 == Complex(1,2)
res39: Boolean = true
```

Case Classes

- Copy method provides safe copying
 - Possibly altering properties
- Case classes not universally applicable
 - Additional bytecode generated
 - Case class cannot inherit from another case class
 - Normally used for value objects, not service objects

```
scala> c1.copy() == c1
res41: Boolean = true

scala> c1.copy() eq c1
res42: Boolean = false

scala> val c3 = c1.copy(re = 4)
c3: Complex = Complex(4,2)
```

Named parameter passing

Packages

- Define namespaces, as in Java
- More flexible
 - Can be nested
 - Many packages per source file

```
package myPackage {
  class AA

  package mySubPackage {
    class AB1
    class AB2
  }
}

class C

object foo extends App {
  val a = new myPackage.AA
  val b = new myPackage.mySubPackage.AB1
  val c = new C
}
```

Package Objects

- Allow definition of package-global data and methods

```
package object myPackage {
    val packageGlobalVal = 100;
    def packageGlobalDef = "Hello There"
}
```

- Use like any other package members

```
object MyProg extends App {
    val myVal = myPackage.packageGlobalVal
    println(myPackage.packageGlobalDef)
}
```

Import

- Allows symbols to be used without package qualification

- Like Java, but again more flexible

- E.g import can be scoped

- Use _ as wildcard
to import all symbols

Import valid
for block

Import valid for
entire source file

```
import myPackage._

object foo extends App {
    val a = new AA
    val b = new B.BB1
    def foo = {
        import myPackage.mySubPackage.AB2
        val ab2 = new AB2
        ...
    }
}
```

Import

- Import multiple symbols at one time

```
import java.io.{ InputReader, OutputWriter }
```

- Import symbol and create alias

- Helps deal with name clashes

```
import java.util.Date
import java.sql.{ Date => SqlDate }
```

- Import allowed from entities other than packages

- package objects
- singleton objects

Import

- Default imports performed by the compiler

```
import java.lang._
import scala._
import scala.Predef._
```

- scala.Predef defines many useful functions

- println
- printf
- Various read... functions
- assert
- require

```
scala> require (1 == 0, "WTF??")
java.lang.IllegalArgumentException: requirement failed: WTF??
at .<init>(<console>:8)
at .<clinit>(<console>)
-
```

Visibility

- By default all class members (properties and methods) are public

- Can be restricted to private

- Visible in containing type only

- ...or protected

- Visible in containing type and subtypes

```
class Person ( val firstName: String,
              val lastName: String,
              private val age: Int ) {

    protected val name = s"${firstName} ${lastName}"
}
```

Visibility

- Visibility restriction can be qualified

- Restricted to package

```
package myPackage

class Person ( val firstName: String,
              val lastName: String,
              private [myPackage] val age: Int ) {

    protected val name = s"${firstName} ${lastName}"
}
```

- Restricted to instance

```
class Person ( val firstName: String,
              val lastName: String,
              private [this] val age: Int ) {

    protected val name = s"${firstName} ${lastName}"
}
```

Inheritance in OO Scala

Inheritance

- Scala classes may inherit from one other class
 - Standard JVM model
 - Non-private members inherited
 - Non-final members may be overridden

```
class Animal ( val name: String ) {
  def eat = println("Hungry....")
  private def mySecret = println("This is my secret")
}

class Lion ( n: String ) extends Animal( n ) {
  def roar = println(name + " says Grrrr")
  override def eat = println("Rare steak please")
}
```

Invoke superclass constructor with arg

Preventing Subclassing or Overriding

- Use final to prevent a class from being extended

```
scala> final class Animal
defined class Animal

scala> class Lion extends Animal
<console>:12: error: illegal inheritance from final class Animal
          class Lion extends Animal
```

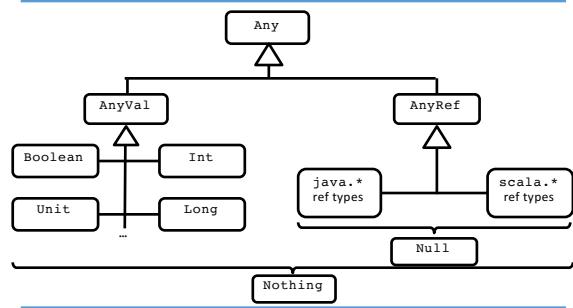
Preventing Subclassing or Overriding

- ...or to prevent a member being overridden

```
scala> class Animal { final def eat = println("Hungry") }
defined class Animal

scala> class Lion extends Animal {
    |   override def eat = println("WTF??")
    |
<console>:13: error: overriding method eat in class Animal of type => Unit;
method eat cannot override final member
        override def eat = println("WTF??")
```

Scala Class Inheritance Hierarchy



Value Types

- Subtypes of `AnyVal`
- Present as objects
 - Act as wrappers around JVM primitive types
 - Compiler removes object wrapper when possible
 - Balances efficiency with object capabilities
- New value types may be defined
 - Must wrap a single JVM primitive type
 - Must extend `AnyVal`

```
class Foo ( i: Int ) {
    def bar = i + 1
}

scala> javap -c Foo
Compiled from "<console>"
public class Foo {
    public int bar();
    Code:
        0: aload_0
        1: getfield    #11  // Field i:I
        4: iconst_1
        5: iadd
        6: ireturn
}
```

Abstract Classes

- Class need not provide full implementation of all properties
- Such a class must be marked as abstract

```
abstract class Animal {
  def eat: String
}

scala> new Animal
<console>:13: error: class Animal is abstract; cannot be instantiated
          ^
class Lion extends Animal {
  override def eat = "Yum"
}
scala> new Lion().eat
res48: String = Yum
```

Annotations for abstract classes:

- A callout points to the `abstract` keyword with the text: "Cannot be instantiated".
- A callout points to the class definition with the text: "Argument types and result type should be specified".
- A callout points to the `override` keyword with the text: "Keyword not mandatory but advised".

Overriding and the Uniform Access Principle

- `def` may be overridden with a `val`
 - Subclass causes the property value to become stable
- `val` may *not* be overridden by a `def`

```
class Person {
  def age: Int = // Some calculation
}

scala> val p = new Person
p: Person = Person@1c59d3ae
scala> p.age
res23: Int = 22

class DorianGray extends Person {
  override val age: Int = 21
}
```

Traits

- Trait is similar to Java 8 interface
 - May include implementation
 - May not define construction parameters
 - Use trait to interact with Java interfaces
- Scala class may "inherit" or "mix in" several traits
- Use trait to
 - Advertise public interface
 - Add behaviour to existing type

Why Traits?

- The case for multiple inheritance is normally based on mixins
 - Can help produce a more accurate model

```
scala> abstract class Animal
defined class Animal

scala> class Bird extends Animal { def fly = "wheee" }
defined class Bird

scala> class Fish extends Animal { def swim = "splash" }
defined class Fish

scala> class Seagull extends ????????
scala> class Penguin extends ????????
```

Why Traits?

- Start with (abstract) base class
 - Mix in required functionality

```
scala> abstract class Animal
defined class Animal

scala> trait CanSwim { def swim = "Splash" }
defined trait CanSwim

scala> trait CanFly { def fly = "Wheeee" }
defined trait CanFly

scala> class Fish extends Animal with CanSwim
defined class Fish

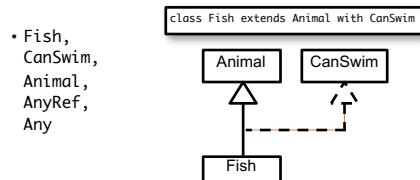
scala> class Seabird extends Animal with CanFly with CanSwim
```

First supertype can
be class or trait

Use "with" for
subsequent supertypes

Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:



Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:

Seabird, CanSwim, CanFly, Animal, AnyRef, Any

```
class Seagull extends Animal with CanFly with CanSwim
```

Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:

Seabird, CanFly, CanSwim, Animal, AnyRef, Any

```
class Seagull extends Animal with CanSwim with CanFly
```

Multiple Implementations in Supertypes

Ambiguity must be removed

```
(new C).bar???
```

```
scala> trait Foo { def bar = "FooBar" }
defined trait Foo
scala> trait Bar { def bar = "BarBar" }
defined trait Bar
scala> class C extends Foo with Bar
note: warning: error: class C inherits conflicting members:
method bar in trait Foo of type => String and
method bar in trait Bar of type => String
>Note: this can be resolved by declaring an override in class C.
class C extends Foo with Bar
scala> class C extends Foo with Bar {
|   override def bar = super[foo].bar
| }
defined class C
scala> (new C).bar
res49: String = FooBar
```

Trait Example

- Add ordering to the Complex class
 - with our own definition of "ordering" of complex numbers!!
- Trait extends java.lang.Comparable
 - provides compareTo() method
 - use [] instead of < > for type parameters
- Define two "operator" methods

```
trait Ordered[A] extends java.lang.Comparable[A] {
  def < (that: A) = compareTo(that) < 0
  def > (that: A) = compareTo(that) > 0
}
```

Using the Trait

- New type – Ordered Complex Numbers
- Extends basic class with capabilities of trait
 - provides implementation of abstract method compareTo()
 so that trait provided operators will work

```
class OrdComplex (r: Int = 0, i: Int = 0)
  extends Complex (r, i) with Ordered[OrdComplex] {
  def compareTo(that: OrdComplex): Int = this.re - that.re
}

scala> val oc1 = new OrdComplex(2,3)
oc1: OrdComplex = OrdComplex@69cab16f

scala> val oc2 = new OrdComplex(3,4)
oc2: OrdComplex = OrdComplex@19dbb0a1d

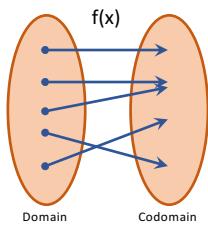
scala> oc1 < oc2
res37: Boolean = true
```

Functional Programming in Scala



What is Functional Programming?

- Functional Programming views computation as function evaluation
 - in the mathematical sense
 - avoiding state and mutable data
- FP started with Lisp in the 1950s
 - but the ideas are much older than that...
- Today, many languages support FP
 - Haskell
 - Clojure
 - Scala
 - Java
 - C#
 - C++
 - Python
 - ...



What is Functional Programming?



What is Functional Programming?



What is Functional Programming?



What is Functional Programming?

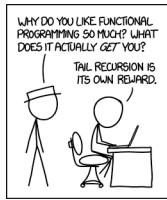


What is Functional Programming?



What Characterises Functional Programming?

- First-class and higher-order functions
 - lambdas
 - closures
- Immutable state
- Use of recursion
- Declarative style
- Lazy evaluation
- Type inference



What are FP's Strong Points?

- Higher-order functions and declarative style
 - concise expression of algorithms
 - efficient implementation
- Immutable state
 - less concern with concurrency problems
 - easy parallelization
- Especially well suited to Big Data processing
 - declarative style allows flexibility of implementation
 - thanks to ease of parallelization

Functional Programming in Scala

- We have already seen "functions" in Scala

```
scala> def twice ( i: Int ) = i * 2
twice: (i: Int)Int
scala> twice(3)
res59: Int = 6
```

- These are not strictly speaking first class functions
 - E.g. cannot be assigned

```
scala> val doubleIt = twice
<console>:12: error: missing argument list for method twice
      ^
```

Functional Programming in Scala

- First class functions are objects

```
scala> val doubleIt = (n: Int) => n * 2
doubleIt: Int => Int = <function1>
scala> doubleIt(3)
res60: Int = 6
```

- doubleIt is an instance of a type that extends Function1[Int, Int]
 - trait
 - How does the function get called?
- Many other Functionn types – up to Function22[...]

Function Literals

- AKA Lambda Expressions

```
scala> (n: Int) => n * n
res61: Int => Int = <function1>
scala> ( n:Int ) => n * n )(10)
res62: Int = 100
scala> val squareIt = (n: Int) => n * n
squareIt: Int => Int = <function1>
scala> (a: Int, b: Int) => a + b
res63: (Int, Int) => Int = <function2>
scala> ((a: Int, b: Int) => a + b)(4,3)
res66: Int = 7
```

Higher Order Functions

- Functions that take other functions as arguments

```
scala> val doIt = ( i: Int, f: Int => Int ) => f(i)
doIt: (Int, Int => Int) => Int = <function2>
scala> doIt(10, doubleIt)
res63: Int = 20
scala> doIt(10, squareIt)
res64: Int = 100
```

Function Composition

- Chaining function invocations, building a pipeline of processing
 - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>
scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>
scala> f(2)
res199: Int = 3
scala> g(2)
res200: Int = 4
```

Function Composition

- Chaining function invocations, building a pipeline of processing
 - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>
scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>
scala> val h = f compose g
h: Int => Int = <function1>
scala> f(2)
res199: Int = 3
scala> g(2)
res200: Int = 4
scala> h(2)
res202: Int = 5
```

Function Composition

- Chaining function invocations, building a pipeline of processing
 - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>

scala> val j = g compose f
j: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4

scala> j(2)
res203: Int = 6
```

Function Composition

- Chaining function invocations, building a pipeline of processing
 - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>

scala> val k = f andThen g
k: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4

scala> k(2)
res201: Int = 6
```

Higher Order Functions

- Functions that return functions

```
scala> val multBy = (n: Int) => ((x: Int) => x * n)
multBy: Int => (Int => Int) = <function1>

scala> val double = multBy(2)
double: Int => Int = <function1>

scala> val triple = multBy(3)
triple: Int => Int = <function1>

scala> double(4)
res67: Int = 8

scala> triple(4)
res68: Int = 12
```

Closures

- Function definition may refer to external values
- Function "captures" or "closes over" these external values
- Beware closing over mutable state!

```
scala> var extVal = 100
extVal: Int = 100
scala> def foo(n: Int): Int = n * extVal
foo: (Int)Int
scala> foo(2)
res0: Int = 200
scala> extVal = 200
extVal: Int = 200
scala> foo(2)
res1: Int = 400
```



Methods and Functions

- Method can be converted to a function object
 - "lifting"

```
def myMethod(a: Int) = a + 10
```

- Use explicit type

```
scala> val myMethod3: Int => Int = myMethod
myMethod3: Int => Int = <function0>
scala> myMethod3(10)
res10: Int = 20
```

- Use special syntax

```
scala> val myMethod2 = myMethod _
myMethod2: Int => Int = <function1>
scala> myMethod2(10)
res8: Int = 20
```

Collections



Tuples

- Aggregate type containing elements of possibly different types
 - Like struct

```
scala> val myTuple = ( 1, "One" )
myTuple: (Int, String) = (1,One)

scala> val myData = ( 3.5, 2, new java.util.Date )
myData: (Double, Int, java.util.Date) = (3.5,2,Thu Aug 11 22:04:28 CST 2016)
```

- Instances of a range of types
 - Tuple1, Tuple2, ... Tuple22
 - Tuple2 aliased to Pair
- Known as Product Types
 - Cartesian product

Tuples

- Syntactic sugar available for Pair (Tuple2)

```
scala> val myPair = "George" -> 21
myPair: (String, Int) = (George,21)
```

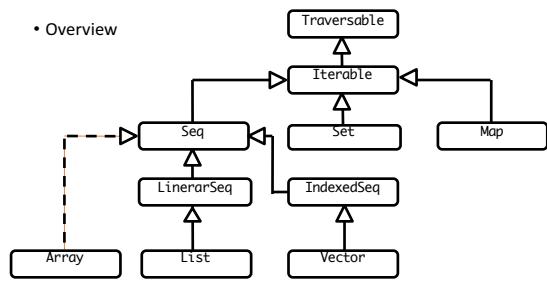
- Elements of tuple can be accessed using special names
 - _1 for first element
 - Compiler checks for "range"

```
scala> myPair._1
res87: String = George

scala> myPair._4
<console>:13: error: value _4 is not a member of (String, Int)
          myPair._4
                  ^
```

Collections Types

- Overview



Collections Packages

- Based on implementation characteristics

- Immutable

- Package `scala.collection.immutable`
- Cannot be changed in place (may generate new copies if updates allowed)
- `scala.Predef` defines type aliases so default collection types are immutable
- (Except for `Seq`, to allow for `Arrays`)

- Mutable

- Package `scala.collection.mutable`
- Collections may be changed in place

- Beware – this is not related to `val` / `var`

Working with Collections

- Syntactic sugar allows convenient notation to be used
 - Companion object and factory methods

```

scala> val mySeq = List(1,2,3)
mySeq: List[Int] = List(1, 2, 3)

scala> val s1 = Set(1, "Hello")
s1: scala.collection.immutable.Set[Any] = Set(1, Hello)
  
```

- Note collection types are parameterised

- Type inference engine can deduce element type from initialisation
- Can specify element type if extra checking required

```

scala> val v1 = Vector[Int](12, 13, 14)
v1: scala.collection.immutable.Vector[Int] = Vector(12, 13, 14)
  
```

Seq

- Ordered collection of items
 - Indexed using Int expressions

- Subtypes differ in implementation approach

- Array – use JVM array types and associated bytecodes
- IndexedSeq (Vector) – constant time access to elements
- LinearSeq (List) – Fast head/tail, length functions

- All collection types are parameterised

- Even Array

```
scala> val a = Array(1, 2, 3)
a: Array[Int] = Array(1, 2, 3)

scala> a(2)
res91: Int = 3

scala> a(1) = 0

scala> a
res93: Array[Int] = Array(1, 0, 3)
```

Seq

- Many methods and properties available

```
scala> val s = Seq( 1, 2, 3, 4 )
s: Seq[Int] = List(1, 2, 3, 4)

scala> s.length
res94: Int = 4

scala> s.indices
res95: scala.collection.immutable.Range = Range(0, 1, 2, 3)

scala> s.reverse
res96: Seq[Int] = List(4, 3, 2, 1)

scala> s.contains(2)
res97: Boolean = true
```

Seq

```
scala> s ::= 5
res98: Seq[Int] = List(1, 2, 3, 4, 5)
scala> 0 ::= s           Right binding operator
res99: Seq[Int] = List(0, 1, 2, 3, 4)

scala> s ::= Seq(4, 5, 6, 7)
res100: Seq[Int] = List(1, 2, 3, 4, 4, 5, 6, 7)

scala> s.count(_ == 4)
res101: Int = 1

scala> val names = Seq("tom", "dick", "harry")
names: Seq[String] = List(tom, dick, harry)

scala> names.sorted
res103: Seq[String] = List(dick, harry, tom)

scala> names.sortWith((a,b) => a.length.compareTo(b.length) < 0)
res105: Seq[String] = List(tom, dick, harry)
```

Note List is immutable type

Set

- Unordered collection, no duplicates

• Smaller number of methods

```
scala> val odds = Set(1, 3, 5, 7, 9)
odds: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3)
scala> odds.contains(3)
res10: Boolean = true
scala> odds & Set(3, 4, 5)
res10: scala.collection.immutable.Set[Int] = Set(5, 3)
scala> odds | Set(3, 4, 5)
res11: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 4)
scala> odds && Set(3, 4, 5)
res13: scala.collection.immutable.Set[Int] = Set(1, 9, 7)
scala> odds + 4
res15: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 4)
scala> odds + 11
res16: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 11)
```

Map

- Collection of (key, value) pairs

• Can be constructed out of a collection of Pair(Tuple2) objects

```
scala> val capitals = Map("Denmark" -> "Copenhagen",
  "Sweden" -> "Stockholm",
  "Norway" -> "Oslo")
capitals: scala.collection.immutable.Map[String,String] = Map(Denmark ->
Copenhagen, Sweden -> Stockholm, Norway -> Oslo)
scala> capitals.keys
res17: Iterable[String] = Set(Denmark, Sweden, Norway)
scala> capitals.values
res18: Iterable[String] = MapLike(Copenhagen, Stockholm, Oslo)
```

Map

- Lookup operation through apply method

• "Associative Array"

• Exception if no such key, alternative approaches available

```
scala> capitals("Norway")
res10: String = Oslo
scala> capitals("Finland")
java.util.NoSuchElementException: key not found: Finland
 at scala.collection.MapLike$class.default(MapLike.scala:228)
 at scala.collection.AbstractMap.default(Map.scala:59)
 at scala.collection.MapLike$class.apply(MapLike.scala:141)
 at scala.collection.AbstractMap.apply(Map.scala:59)
 ... 32 elided
scala> capitals.getOrElse("Finland", "I have no idea")
res23: String = I have no idea
```

Map

- Use MutableMap type to add or change entries

```
scala> val mCapitals = collection.mutable.Map() ++ capitals
mCapitals: scala.collection.mutable.Map[String,String] = Map(Norway -> Oslo,
Denmark -> Copenhagen, Sweden -> Stockholm)

scala> mCapitals("Finland") = "Helsinki"

scala> mCapitals += "Iceland" -> "Reykjavik"
res131: mCapitals.type = Map(Norway -> Oslo, Denmark -> Copenhagen, Iceland ->
Reykjavik, Sweden -> Stockholm, Finland -> Helsinki)

scala> val capitals2 = mCapitals toMap
capitals2: scala.collection.immutable.Map[String,String] = Map(Denmark ->
Copenhagen, Iceland -> Reykjavik, Finland -> Helsinki, Sweden -> Stockholm,
Norway -> Oslo)
```

Change back to
immutable Map

Dealing With Optional Values

- What should be returned when we use a non-existent key to retrieve value from a Map?
 - Throw Exception...
 - Return special value (e.g null)
- Neither of these is type safe
 - What is the type of null in Java???
- Need an equivalent to SQL's NULL
 - Representation of "no value"

Dealing With Optional Values

- Option[T] is the type of a value that may or may not be present
 - If present, the value will have type T
- Represented by a sealed type hierarchy
 - Two case class subtypes of Option[T]
 - Some[T] is a container for a value of type T
 - None represents no value

```
scala> capitals.get("Denmark")
res0: Option[String] = Some(Copenhagen)

scala> capitals.get("Finland")
res1: Option[String] = None
```

Success and failure
cases both have
the same type

Dealing With Optional Values

- Retrieve the actual value using the get method

- Returns value of type T if present

- Throws NoSuchElementException if no value

- Use getOrElse to return "default" value in case where no value is present

```
scala> capitals.get("Denmark").get
res2: String = Copenhagen

scala> capitals.get("Finland").get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:347)
  at scala.None$.get(Option.scala:345)
... 32 elided

scala> capitals.getOrElse("I don't know")
res4: String = I don't know
```

Dealing With Optional Values

- Pattern matching can also be used

```
scala> val capitalOf = (country: String) => capitals.get(country) match {
|   case Some(city) => city
|   case None => "Unknown"
| }
capitalOf: String => String = <function1>

scala> capitalOf("Norway")
res5: String = Oslo

scala> capitalOf("Iceland")
res6: String = Unknown
```

- Further possibilities (see later)

Functional Programming and Collections

- Conventional approach based on Iterator pattern

- Process each element in turn
- "External" iteration

- Collections lend themselves to functional programming style

- Algorithms can elegantly be specified
- "Internal" iteration

- Based on Higher Order Functions

- map
- flatMap
- filter
- ...

Using Higher Order Functions

- Large range of methods defined on collections

```
scala> val range1 = 1 to 5
range1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> range1.foreach( i => print(s"$i ") )
1 2 3 4 5

scala> range1 forall ( i => i <= 5 )
res133: Boolean = true

scala> range1 exists ( i => i == 10 )
res134: Boolean = false
```

Using map

- Return a new collection
 - Apply function to elements in source to generate elements in result

```
scala> val range1 = 1 to 5
range1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> val plusOne = range1.map( i => i + 1 )
plusOne: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 3, 4, 5, 6)

scala> val slist = List("one", "two", "three")
slist: List[String] = List(one, two, three)

scala> slist.map( s => s.toUpperCase )
res135: List[String] = List(ONE, TWO, THREE)
```

Combining Elements from a Collection

- foldLeft** and **foldRight**
 - Function argument determines how elements are combined
 - Notice functions are defined in "curried" form – two argument lists
 - foldRight** scans elements right to left

```
val theSum = range1.foldLeft(0)( (i, j) => i + j )
          ^           ^           ^
          Initial   Accumulating   Element
          Value     "Total"        to process

scala> val pathName = slist.foldLeft("")( (a, b) => s"$a/$b" )
pathName: String = /one/two/three

scala> val pathName = slist.foldRight("")( (a, b) => s"$a/$b" )
pathName: String = one/two/three/
          ^           ^           ^
          Initial   Element     Accumulating
          Value     to process  tail
```

Selecting Elements from a Collection

- filter

- Elements placed into new collection based on supplied predicate function

```
scala> val oddNumbers = range1.filter( _ % 2 != 0 )
oddNumbers: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 3, 5)

scala> val strings = slist.filter( s => s.length > 3 )
strings: List[String] = List(three)

scala> val word = "fortunately"
word: String = fortunately

scala> word.seq
res139: scala.collection.immutable.WrappedString = fortunately

scala> word.seq.filter("aeiou".contains(_))
res140: scala.collection.immutable.WrappedString = ouae
```

Partitioning a Collection

- partition method

- Return a Tuple2 of collections

```
scala> range1.partition( _ % 2 == 0 )
res141: (scala.collection.immutable.IndexedSeq[Int],
scala.collection.immutable.IndexedSeq[Int]) = (Vector(2, 4),Vector(1, 3, 5))

scala> word.seq.partition( "aeiou".contains(_) )
res142: (scala.collection.immutable.WrappedString,
scala.collection.immutable.WrappedString) = (ouae,frtntly)
```

- groupBy gives more general grouping/partitioning capability

```
scala> 1 to 10 groupBy ( _ % 3 )
res143: scala.collection.immutable.Map[Int,scala.collection.immutable.IndexedSeq[Int]] =
= Map(2 -> Vector(2, 5, 8), 1 -> Vector(1, 4, 7, 10), 0 -> Vector(3, 6, 9))
```

Joining Collections

- Zip

- Combines Seq[A] and Seq[B] objects into Seq[Tuple2[A,B]]

```
scala> 1 to 3 zip slist
res145: scala.collection.immutable.IndexedSeq[(Int, String)] =
= Vector((1,one), (2,two), (3,three))
```

- Also zipWithIndex

```
scala> val colours = List("Red", "Green", "Blue")
colours: List[String] = List(Red, Green, Blue)

scala> colours zipWithIndex
warning: there was one feature warning; re-run with -feature for details
res146: List[(String, Int)] = List((Red,0), (Green,1), (Blue,2))
```

Nested Collections

- Collections can contain other collections
 - Often the result of call to map

```
scala> val text = List("Here is the first line", "this is the second line")
text: List[String] = List(Here is the first line, this is the second line)

scala> val splitlines = text.map(_.split(" "))
splitlines: List[Array[String]] =
List(Array(Here, is, the, first, line), Array(this, is, the, second, line))
```

- Flatten method removes one level of nesting

```
scala> splitlines.flatten
res150: List[String] =
List(Here, is, the, first, line, this, is, the, second, line)
```

Combining map and flatten

flatMap

- Very important function
- Forms basis of many idioms in Scala and functional programming
- Used with many types, not just collections
- E.g. Option[T], Future[T], Try[T]
- Monads

```
scala> val words = text.flatMap(_.split(" "))
words: List[String] =
List(Here, is, the, first, line, this, is, the, second, line)
```

Option[T] and the Higher Order Functions

- map, flatMap, foreach, etc all defined on Option[T] type
 - Allow processing on results of operations without forcing null check

```
scala> capitals.get("Norway").map(_.toUpperCase)
res10: Option[String] = Some(OSLO)

scala> capitals.get("Finland").map(_.toUpperCase)
res11: Option[String] = None

scala> capitals.get("Norway").foreach(println(_))
Oslo

scala> capitals.get("Iceland").foreach(println(_))

scala>
```

The for Comprehension

- Used to generate a new instance of a container type
 - E.g. collections
 - Yield operator specifies how elements are generated

```
scala> val squares = for ( i <- 1 to 5 ) yield i * i
squares: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16, 25)
```

- Equivalent to map
 - Compiler rewrites for comprehension to use map

```
scala> val squares = 1 to 5 map ( i => i * i )
squares: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16, 25)
```

The for Comprehension

- Can also specify nested for comprehensions

```
scala> val coordinates = for ( x <- 1 to 3;
|   y <- 6 to 9 )
|   yield (x,y)
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((1,6), (1,7), (1,8), (1,9), (2,6), (2,7), (2,8), (2,9),
(3,6), (3,7), (3,8), (3,9))
```

- Equivalent to combination of flatMap and map

```
scala> val coordinates = 1 to 3 flatMap ( x => 6 to 9 map ( y => (x,y) ) )
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((1,6), (1,7), (1,8), (1,9), (2,6), (2,7), (2,8), (2,9),
(3,6), (3,7), (3,8), (3,9))
```

The for Comprehension

- for comprehension can include guard

```
scala> val coordinates = for ( x <- 1 to 3 if x % 2 != 0;
|   y <- 6 to 9 if y % 2 == 0 )
|   yield (x, y)
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((1,6), (1,8), (3,6), (3,8))
```

- Equivalent to use of filter

```
scala> val coordinates = 1 to 3 filter ( _ % 2 != 0 ) flatMap (
|   x => 6 to 9 filter ( _ % 2 == 0 ) map (
|     y => (x,y) )
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((1,6),
(1,8), (3,6), (3,8))
```

The for Loop

- Special case of for comprehension
 - Function argument returns Unit

```
scala> for ( i <- 1 to 5 ) print(s"$i ")
1 2 3 4 5
```

- Equivalent to foreach higher order function

```
scala> 1 to 5 foreach { i => print(s"$i ") }
1 2 3 4 5
```

- Can also be used with guards

Option[T] with for Comprehensions

- Recall that for comprehension is rewritten as combinations of map, flatMap and filter
 - Option[T] supports these methods

```
scala> val countries = List("Denmark", "USA", "Sweden")
countries: List[String] = List(Denmark, USA, Sweden)

scala> for ( c <- countries;
    |   cap <- capitals.get(c) )
    |   yield(c, cap)
res14: List[(String, String)] = List((Denmark,Copenhagen), (Sweden,Stockholm))
```

```
scala> countries flatMap ( c => capitals.get(c) map ( cap => (c, cap) ) )
res15: List[(String, String)] = List((Denmark,Copenhagen), (Sweden,Stockholm))
```

More About Functional Programming in Scala



Recursion

- Functional programming paradigm prefers recursion to iteration

- Less dependency on mutable state
- Supports declarative style

- Many algorithms can be expressed in a recursive way

```
scala> def factorial ( i: Int ): BigInt = i match {
    | case 0 => 1
    | case _ => i * factorial(i - 1)
}
factorial: (i: Int)BigInt

scala> factorial(0)
res163: BigInt = 1

scala> factorial(10)
res168: BigInt = 3628800

scala> factorial(10000)
java.lang.StackOverflowError
...
```

Tail Recursion and Tail Call Elimination

- Every recursive call needs a new stack frame
 - Deep recursion can cause stack overflow error

- Tail recursion occurs when
 - Recursive call is last action of a function
 - Result of the function call is returned directly

```
scala> def gcd ( a: Int, b: Int ): Int = if ( b == 0 ) a else gcd( b, a % b )
gcd: (a: Int, b: Int)Int

scala> gcd(14,35)
res170: Int = 7
```

- Tail call elimination is an optimisation that can be used here
 - Reuse current stack frame for recursive call
 - Recursion effectively becomes iteration

Tail Recursion and Tail Call Elimination

- Not all algorithms are tail recursive by default
 - Scala compiler will generate tail call elimination code when possible
 - Otherwise normal recursion
- @tailrec annotation causes compilation failure if no tail recursion in annotated function

```
scala> @tailrec
    def factorial( i: Int ): BigInt = i match {
      case 0 => 1
      case _ => i * factorial(i-1)
    }
<console>:16: error: could not optimize @tailrec annotated method factorial:
  it contains a recursive call not in tail position
    case _ => i * factorial(i-1)
           ^

```

Tail Recursion and Tail Call Elimination

- Non tail-recursive function can often be transformed into tail-recursive
 - Introduce argument to pass accumulated value to next call

```
scala> def factorial( n: Int ): BigInt = {
    |   @tailrec def fact( n: Int, accum: BigInt ): BigInt = n match {
    |     case 0 => accum
    |     case _ => fact( n-1, n * accum )
    |   }
    |   fact(n,1)
  }
factorial: (n: Int)BigInt

scala> factorial(10000)
res176: BigInt = 284625968091705451890641321211986889014805140170279923079417999427
4411340003764443772990786757784775815884062142317528830042339940153518739052421
...
```

Currying

- Currying is a technique to transform a function taking 2 or more arguments into a composition of functions taking 1 argument

Function that takes one Int parameter and returns another function from Int to Int
 Function that adds 2 to its Int argument

```
scala> val add = ( a: Int, b: Int ) => a + b
add: (Int, Int) => Int = <function2>

scala> add.curried
res182: Int => (Int => Int) = <function1>

scala> add.curried(2)
res183: Int => Int = <function1>

scala> add.curried(2)(3)
res184: Int = 5
```

Partial Function Application

- Currying a function allows partial application
 - Generate a function that effectively fixes the first argument(s)
 - Use this generated function further

```
scala> val add2 = add.curried(2)
add2: Int => Int = <function1>
scala> add2(3)
res185: Int = 5
scala> val foo = (a: Int, b: Int, c: Int) => a + b + c
foo: (Int, Int, Int) => Int = <function3>
scala> def add23 = foo.curried(2)(3)
add23: Int => Int
scala> add23(4)
res186: Int = 9
```

Lazy Parameter Evaluation

- AKA Pass By Name
- Suppresses evaluation of actual parameter expression until parameter is used in the function

```
def showTheTime ( t: java.util.Date ) = {
  println(s"The time is $t")
  Thread.sleep(1000)
  println(s"The time is $t")
}

scala> showTheTime( new java.util.Date )
The time is Sat Aug 13 20:52:49 CST 2016
The time is Sat Aug 13 20:52:49 CST 2016
```

Evaluated on call
Value used inside function

Lazy Parameter Evaluation

- Parameter passing is normally by value
- Actual parameter expression is evaluated as part of call

```
def showTheTime ( t: => java.util.Date ) = {
  println(s"The time is $t")
  Thread.sleep(1000)
  println(s"The time is $t")
}

scala> showTheTime(new java.util.Date)
The time is Sat Aug 13 21:23:56 CST 2016
The time is Sat Aug 13 21:23:57 CST 2016
```

Evaluated each time
parameter is used
during the function call

Managing Resources

- Curried function/method definition and pass by name parameters allow definition of powerful structures within a program

- Resource management

Must be passed by name

```
def sync ( l: Lock ) ( f: -> Unit ) = {
  l.acquire()
  try {
    f
  } finally {
    l.release()
  }
}
```

Code in f is evaluated here

```
val myLock = new Lock()
sync ( myLock ) {
  println("I am holding the lock")
}
```

Partial Functions

- Partially defined, not partially evaluated (curried)
- Function not defined over its entire domain
 - E.g. sqrt over Int
- Scala provides type level support for partial functions
- PartialFunction type is subtype of Function1
 - Defines two additional methods
 - def isDefinedAt(a: A): Boolean
 - def orElse(that: PartialFunction[A,B]): PartialFunction[A,B]

Defining a Partial Function

- Case alternatives

```
scala> val pf: PartialFunction[Int, String] = {
    |   case 1 => "one"
    |   case 2 => "two"
    |   case 3 => "three"
    | }
pf: PartialFunction[Int,String] = <function1>

scala> pf.isDefinedAt(2)
res55: Boolean = true
scala> pf(2)
res56: String = two
scala> pf.isDefinedAt(4)
res57: Boolean = false
scala> pf(4)
scala.MatchError: 4 (of class java.lang.Integer)
at
scala.PartialFunction$anon$1.apply(PartialFunction.scala:248)
-
```

Defining a Partial Function

- Implement the required methods

```
scala> object SquareRoot extends PartialFunction[Int, Double] {
|   def apply ( i: Int ) = if ( i >= 0 ) Math.sqrt(i) else
|     throw new IllegalArgumentException("_")
|   def isDefinedAt(i: Int) = ( i >= 0 )
| }
defined module SquareRoot

scala> SquareRoot.isDefinedAt(4)
res62: Boolean = true
scala> SquareRoot(4)
res63: Double = 2.0
scala> SquareRoot.isDefinedAt(-1)
res64: Boolean = false
scala> SquareRoot(-1)
java.lang.IllegalArgumentException: Negative sqrt not cool
  at SquareRoot$.apply$mcD$sp(<console>:17)
```

Map as a Partial Function

- Map type is a mapping from keys to values
 - Not every value from the key type may have a mapping
 - Hence Map is a partial function

```
scala> val squares = Map( 1->1, 2->4, 3->9 )
squares: scala.collection.immutable.Map[Int,Int] = Map(1 -> 1, 2 -> 4, 3 -> 9)

scala> squares.isDefinedAt(2)
res190: Boolean = true

scala> squares.isDefinedAt(4)
res191: Boolean = false

scala> squares(2)
res192: Int = 4

scala> squares(4)
java.util.NoSuchElementException: key not found: 4
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  ...
```

Composing Partial Functions

- Composition of partial functions means providing mappings for previously undefined values
 - orElse method
 - Does not overwrite existing values if key already present

```
scala> val moreSquares = squares.orElse Map( 4->16, 5->25 )
moreSquares: PartialFunction[Int,Int] = <function1>

scala> moreSquares.isDefinedAt(4)
res195: Boolean = true

scala> moreSquares(4)
res196: Int = 16
```

Dealing with Exceptions in Scala

Scala and Exceptions

- Scala designed to run on the JVM
 - Exceptions are a basic feature of the JVM
 - Many Java methods will throw exceptions
- Scala does not support checked exceptions
 - Handling is not enforced
- try { ... } catch { ... } syntax supported
 - But with some differences

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

scala> try {
    |   s.toInt
    | } catch {
    |   case e: Exception => println("oops")
    | }
res0: AnyVal = 123
```

Scala and Exceptions

- try { ... } catch { ... } is an expression
 - Yields a value, but what is the type of this value?

```
scala> val i = try {
    |   s.toInt
    | } catch {
    |   case e: Exception => println("oops")
    | }
i: AnyVal = 123
```

```
scala> val s = "Foobar"
s: String = xxx

scala> val i = try {
    |   s.toInt
    | } catch {
    |   case e: Exception => println("oops")
    | }
oops
i: AnyVal = ()
```

Using Option[T]

- Encapsulate result in Option[T] type

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

scala> val i = try {
    | Some(s.toInt)
    | } catch {
    | case e: Exception => None
    | }
i: Option[Int] = Some(123)
```

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

scala> val i = try {
    | Some(s.toInt)
    | } catch {
    | case e: Exception => None
    | }
i: Option[Int] = None
```

- Further processing of result can take place

```
scala> i map (_ + 4)
res1: Option[Int] = Some(127)
```

```
scala> i map (_ + 4)
res2: Option[Int] = None
```

Using Option[T]

- Use of Option[T] may lead to loss of information
 - Details of exceptions

```
scala> val data = List("0", "1", "blah", "2")
data: List[String] = List(0, 1, blah, 2)

scala> for (
    | a <- data;
    | b <- try { Some(a.toInt) }           "blah" fails here
    |   catch { case ex: Exception => None };
    | c <- try { Some(12 / b) }            "0" fails here
    |   catch { case ex2: Exception => None } )
    | yield (a, c)
res3: List[(String, Int)] = List((1,12), (2,6))
```

The Try[T] Type

- Sealed ADT like Option[T]

- Captures details of non-fatal exceptions
- Serious faults (e.g. Errors) will still be thrown
- scala.util.control.NonFatal used to determine if Fatal or Nonfatal

```
scala> import scala.util.{Try, Success, Failure}
import scala.util.{Try, Success, Failure}

scala> Try("123".toInt)
res4: scala.util.Try[Int] = Success(123)

scala> Try("Foobar".toInt)
res5: scala.util.Try[Int] = Failure(java.lang.NumberFormatException:
For input string: "Foobar")
```

Working with Try[T]

- Higher order functions can be used

```
scala> val result = Try { "123".toInt } map { n => n * 2 }
result: scala.util.Try[Int] = Success(246)

scala> val result = Try { "blah".toInt } map { n => n * 2 }
result: scala.util.Try[Int] = Failure(java.lang.NumberFormatException:
For input string: "blah")

scala> val result = Try { "0".toInt } flatMap { i => Try { 12 / i } }
result: scala.util.Try[Int] = Failure(java.lang.ArithmetricException: / by zero)
```

Working with Try[T]

- Use get method to retrieve value
 - Throws exception if one exists
- Allows "effect" to be exposed at appropriate stage

```
scala> val result = Try { "123".toInt } map { n => n * 2 } get
result: Int = 246

scala> val result = Try { "blah".toInt } map { n => n * 2 } get
java.lang.NumberFormatException: For input string: "blah"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
...

scala> val result = Try { "0".toInt } flatMap { i => Try { 12 / i } } get
java.lang.ArithmetricException: / by zero
at $anonfun$2$$anonfun$apply$1.apply$mcI$sp(<console>:14)
...
```

Chaining Calls Using Try[T]

- Common Scala idiom
 - Follow the "happy path"
 - Keep track of failure details

```
scala> val dividend = "foobar"
dividend: String = foobar

scala> val divisor = "3"
divisor: String = 3

scala> for ( x <- Try { dividend.toInt };
|   y <- Try { divisor.toInt } )
|   yield ( x / y )
res8: scala.util.Try[Int] = Failure(java.lang.NumberFormatException:
For input string: "foobar")
```

```
scala> val dividend = "15"
dividend: String = 15

scala> val divisor = "3"
divisor: String = 3

scala> for ( x <- Try { dividend.toInt };
|   y <- Try { divisor.toInt } )
|   yield ( x / y )
res7: scala.util.Try[Int] = Success(5)
```