Lightbend Scala Language Professional

March 2020

Scala Background

Objectives

- Scala: a JVM language
- Why use Scala?
- Scala's origin

The REPL

```
$ sbt console
<elided>
Welcome to Scala 2.12.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> "Hello," + " world!"
res0: String = Hello, world!
```

- The **R**ead **E**val **P**rint **L**oop is an interactive shell, started via
  - *scala* from the command line
  - *console* in sbt
- Compiles and evaluates Scala code immediately
- Very helpful for experiment-driven development

Useful REPL Commands

```
scala> :help
All commands can be abbreviated, e.g. :he instead of :help.

:cp <path>          add a jar or directory to the classpath
:help [command]     print this summary or command-specific help
:history [num]      show the history (optional num is commands to show)
:javap <path|class>  disassemble a file or class name
:load <path>        load and interpret a Scala file
:paste              enter paste mode: all input up to ctrl-D compiled together
:quit               exit the interpreter
:replay             reset execution and replay all previous commands
:reset              reset the repl to its initial state, forgetting all session entries
:sh <command line>   run a shell command (result is implicitly => List[String])
:warnings           show the suppressed warnings from the most recent line which had an
```

Immutable Values

```
scala> val message = "Hello, world!"
message: String = Hello, world!

scala> message = "Trying to reassign ..."
<console>:12: error: reassignment to val
       message = "Trying to reassign ..."
               ^
```

An immutable value is defined with the *val* keyword

Type Inference

```
scala> val message = "Hello, world!"
message: String = Hello, world!

scala> val message: String = "Hello, world!"
message: String = Hello, world!

scala> val message: Int = "Hello, world!"
<console>:11: error: type mismatch;
 found   : String("Hello, world!")
 required: Int
       val message: Int = "Hello, world!"
                          ^
```

- The compiler can infer the type
- For public API you should provide the types explicitly
- Of course Scala is statically typed

Mutable Variables

```
scala> var message = "Hello, world!"
message: String = Hello, world!

scala> message = "Trying to reassign ..."
message: String = Trying to reassign ...
```

- A mutable variable is defined with the *var* keyword
- Can be reassigned a new value

Expression-oriented Programming

What's the difference between a statement and an expression?

- A *statement* gets executed with no return value, purely for its *side effects*.
- An *expression* gets evaluated to a value.

```
scala> while (false) {
         // Do something ...
       }

scala> if (1 == 1) "correct" else "weird"
res1: String = correct
```

- The REPL prefixes each line in a multi-line expressions with a "|", but that is not part of the actual Scala code
- In Scala many language constructs are expressions, e.g. blocks of code, *if* statements or *try-catch* statements

Multi-line Expressions

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

val value = 1 +
  2 +
  3

// Exiting paste mode, now interpreting.

value: Int = 6
```

- Expressions can span multiple lines
- **Best practice**: Use dangling operators

Code Blocks

```
scala> val value = {
         val x = 1
         val y = 2
         x + y
       }
value: Int = 3
```

- Code blocks are expressions
- The return value of a block is determined by the last expression in the block
- In this example, the result of the block is the value of the nested expression x + y
- **Best practice**: Keep opening braces on the same line

Lightweight Language

```
scala> val lightweight =
  if ("Scala" startsWith "S") {
    val scala = "Scala"
    val is = "is"
    val lightweight = "lightweight"
    scala + " " + is + " " + lightweight
  } else
    "Cannot happen, because Scala != Java"
lightweight: String = Scala is lightweight
```

- No curly braces needed for single line expressions
- Type annotations can be omitted
- No dots and parentheses needed
- Semicolons can be omitted
- No *return* statement needed

OO Basics

Objectives

- Create and usage of classes
- Fields and methods
- Singleton objects
- Case classes

Object-orientation in Scala

- Classes and traits
  - Fields keep state
  - Methods provide operations (encapsulation)
  - Access modifiers declare visibility (information hiding)
- Singleton objects are first-class objects
- Inheritance
  - Single inheritance, i.e. extend exactly one superclass
  - Multiple traits can be mixed in

The Simplest Class

```
scala> class Hello
defined class Hello

scala> val hello = new Hello
hello: Hello = Hello@33bd6867

scala> hello.toString
res0: String = Hello@33bd6867
```

- Defined with *class* keyword
- Class instance (object) is created with the *new* keyword
- *AnyRef* is used if no superclass is extended explicitly

The Primary Constructor

```
scala> class Hello {
         println("Hello")
       }
defined class Hello

scala> new Hello
Hello
res0: Hello = Hello@7f68f33c
```

- Each class gets a **primary constructor** automatically
- Class definition (name and parameters) => signature
- Class body (except for field and method definitions) => implementation

ScalaTrain case study

- ScalaTrain is an application for scheduling trains
- It has Trains, Stations, Times, a JourneyPlanner and Schedules
- We will be building these during the course exercises

Working on Exercises

- `man e` - Displays current exercise instructions.
- `showExerciseID` - Displays the current exercise name.
- `listExercises` - Lists all exercises in course.
- `nextExercise` - Brings new tests and instructions into scope, while preserving your code.
- `prevExercise` - Reverts tests and instructions to the previous state, while preserving your code.
- `gotoExerciseNr <ex Nr>` - Jump to exercise `Nr`, bring in tests for that exercise while preserving code.
- `pullSolution` - Overwrites your code with the official solution.
- `saveState` - Create a snapshot of your current code.
- `restoreState <ex Id>` - Restore the code from a saved snapshot.
- `savedStates` - List all saved states.

Class Parameters

```
scala> class Hello(message: String) {
         println(message)
       }
defined class Hello

scala> new Hello("Hello, world!")
Hello, world!
res0: Hello = Hello@5daef86b

scala> new Hello
<console>:13: error: not enough arguments for constructor Hello: (message: Strir
Unspecified value parameter message.
       new Hello
       ^
```

- Classes can have one or more parameters with format **name: type**
- Class parameters are parameters of the primary constructor

Inspecting the Bytecode

```
scala> :javap -p Hello
Compiled from "<pastie>"
public class $line3.$read$$iw$$iw$Hello {
  public $line3.$read$$iw$$iw$Hello(java.lang.String);
}
```

- *javap* disassembles Java class files
- Show structure or bytecode
- Start *javap* from the REPL with *:javap*

Useful javap Options

```
scala> :javap -c Hello
Compiled from "<pastie>"
public class $line3.$read$$iw$$iw$Hello {
  public $line3.$read$$iw$$iw$Hello(java.lang.String);
    Code:
       0: aload_0
       1: invokespecial #19  // Method java/lang/Object."<init>":()V
       4: getstatic     #25  // Field scala/Predef$.MODULE$:Lscala/Predef$;
       7: aload_1
       8: invokevirtual #29  // Method scala/Predef$.println:(Ljava/lang/Object;)V
      11: return
}
```

- *-p* shows all members
- *-c* shows disassembled bytecode
- *-v* shows everything

Methods

```
scala> def hello = "Hello"
hello: String

scala> def echo(message: String): String = message
echo: (message: String)String
```

- Methods are class members providing operations
- Use *def* to define a method, followed by
  - Name
  - Optional parameter list
  - Optional type annotation
  - Expression (method body) after an equals sign

Infix Operators

```
scala> "Martin Odersky".split(" ")
res0: Array[String] = Array(Martin, Odersky)

scala> "Martin Odersky" split " "
res1: Array[String] = Array(Martin, Odersky)
```

- Operators are methods used in operator notation
- Operator notation means omitting dots and parentheses
- Methods with one parameter can be used in **infix** notation

Postfix Operators

```
scala> "Martin Odersky" split " " size
warning: there were 1 feature warnings; re-run with -feature for details
res0: Int = 2
```

- Methods without parameters can be used in **postfix** notation
- In general, avoid using **postfix** notation

Prefix Operators

```
scala> ! true
res0: Boolean = false

scala> true.unary_!
res1: Boolean = false
```

Methods starting with *unary_* followed by +, -, *!* or ~ can be used in **prefix** notation

Conventions for Operator Notation

```
1 + 1
if(!true)
"Martin Odersky".split(" ").size
```

- Use infix notation for methods with symbolic names
- Use prefix notation for *unary_* methods
- Otherwise use *dot* (method invocation) notation

Equality I

```
scala> 3 == 3
res0: Boolean = true

scala> new String("Scala") == new String("Scala")
res1: Boolean = true
```

- Use "==" for equality (and "!=" for inequality)
- No difference between "primitive" or "reference" types (Java)
- Use "==" or "!=" when comparing. When overriding, overide "equals"
- There's no type check, the argument for "==" / equals is "Any"

Equality II

```
scala> null == new String("Scala")
res2: Boolean = false

scala> new String("Scala") eq new String("Scala")
res4: Boolean = false

scala> new String("Scala") ne null
res5: Boolean = true
```

- Use "eq" and "ne" for checking identity (comparing references)
- Comparisons in Scala are null-safe (make sure it stays that way if you override)

Default Arguments

```
def name(first: String =  "", last: String = ""): String =
  first + " " + last

scala> name("Martin")
res0: String = Martin
```

- Lets you omit trailing arguments
- How can you omit leading arguments?

Named Arguments

```
scala> name(last = "Odersky")
res0: String = Odersky

scala> name(first = "Martin", last = "Odersky")
res0: String = Martin Odersky
```

- Leading arguments can be omitted by giving trailing arguments by name
- You can always give all arguments by name and you can also mix
- Attention: Parameter names are part of the API and should follow the coding conventions used for other names

Packages

```
package com.lightbend.training.scalatrain
```

- Packages organize non trivial code bases
- **Best practice**: The file structure should reflect the package structure

Imports I

```
import com.lightbend.training.scalatrain.Train
import com.lightbend.training.scalatrain._
```

- Use *import* if you don't want to use the fully qualified name
- Use the underscore _ to import all members of a package

Imports II

```
import scala.concurrent.{ Future, Promise }
import java.sql.{ Date => SqlDate }

def method = {
  import com.lightbend.training.scalatrain.Time
  new Time(10, 30)
}
```

- Import multiple members with *{...}*
- Rename imported objects
- Scope imports for methods

Access Modifiers

```
class Hello {
  private val message = "Hello!"
}

class Welcome {
  protected val message = "Hello!"
}
```

- By default all members are public
- Use *private* to restrict access
- Using *protected* makes a member visible within the enclosing entity and subclasses

Qualified Access Modifiers

```
package hello
class Hello {
  private[hello] val message = "Hello!"
}

class Hello {
  private[this] val message = "Hello!"
  def messagesEqual(that: Hello) =
    this.message == that.message // Won't compile!
}
```

- Relax access up to a given entity through a qualifer
- Use *this* to restrict access to the instance

Singleton Objects

```
object Hello {
  def message = "Hello!"
}

scala> Hello.message
res0: java.lang.String = "Hello!"
```

- Create only one single instance of a class
- Access a singleton object with its name
- Quiz: Why can you not create an instance of a singleton object?

Use Cases for Singleton Objects

```
object Hello {
  val defaultMessage = "Hello!"
}
```

- Singletons are first-class objects replacing *static*, e.g. holding constants
- Use cases
  - Class Factories (companion objects)
  - Util classes
  - Applications

Applications and the Main Method

```
object Hello {
  def main(args: Array[String]): Unit =
    println("Hello!")
}
```

Application is a singleton object with a main method

Running an Application

```
$ scala -cp target/scala-2.11/classes/ Hello
Hello
```

```
> run
[info] Running Hello
Hello
[success] Total time: 0 s, completed Jul 20, 2012 6:00:20 PM
```

- Run an application with *scala* and the fully qualified name
- Alternatively use the run command in *sbt*

Companion Objects

```
object Hello {
  private val defaultMessage = "Hello!"
}

class Hello(message: String = Hello.defaultMessage) {
  println(message)
}
```

- If a singleton object and a class or trait share the same name, package and file, they are called **companions**
- Companions can access their private members

Meet Predef

```
scala> require(1 == 2, "WEIRD!")
java.lang.IllegalArgumentException: requirement failed: WEIRD!
        at scala.Predef$.require(Predef.scala:224)
        ...
```

- The standard library contains the *Predef* singleton object
- All its members are imported automatically
- One example is the *require* method, which is used to check preconditions

Case Classes

```
case class Time(hours: Int = 0, minutes: Int = 0)

scala> val time = Time(9)
time: Time = Time(9,0)
```

- *case* keyword adds several additional features:
  - 1. Create new instances without *new*

Applying Syntactic Sugar I

```
scala> Time(9)
res0: Time = Time(9,0)

scala> Time.apply(9)
res1: Time = Time(9,0)
```

- Each case class has a companion object with an *apply* method
- In this context apply is used as a factory method

Applying Syntactic Sugar II

```
object Reverse {
  def apply(s: String): String =
    s.reverse
}

scala> Reverse("Hello")
res0: String = olleH
```

- Calling *apply* works for any object
- We will see more examples, e.g. for collections and functions

More Case Class Features I

```
scala> time == Time(9)
res0: Boolean = true

scala> time == Time(9, 30)
res1: Boolean = false

scala> time.hours
res2: Int = 9
```

- 2. Compiler creates nice *toString*, *equals* and *hashCode* implementations
- 3. Class parameters are promoted to immutable fields automatically

More Case Class Features II

```
scala> time.copy(minutes = 45)
res0: Time = Time(9,45)
```

- 4. *copy* method is automatically implemented
- 5. Use case classes in pattern matching (covered a little later)

Why are not all Classes Case Classes?

- Overhead in bytecode size
- Can not inherit a case class from another one
- **Best practice**
  - Value objects are perfect candidates for case classes
  - Service objects should not be case classes

Testing

Objectives

- Scala testing ecosystem
- Introduction to Behavior Driven Development
- ScalaTest overview and usage

Testing Libraries

- ScalaTest, Specs2, ScalaCheck, JUnit, TestNG
- Mocking frameworks: Mockito, ScalaMock
- In this course we use ScalaTest

ScalaTest

```
libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "3.0.5" % "test"
)
```

- ScalaTest provides different styles to write your tests
- We will use a behavior-driven style of development (BDD), in which tests are combined with text that describes the behavior the tests verify
- ScalaTest library dependency has already been added to the *build.sbt*

Structure of a ScalaTest BDD Specification

```
class EchoSpec extends WordSpec with Matchers {

  "Calling echo" should {
    "return the given value" in {
      val message = "message"
      Echo.echo(message) shouldEqual message
    }
    "throw an exception for a null argument" in {
      an[Exception] should be thrownBy Echo.echo(null)
    }
  }
}
```

ScalaTest Benefits

- Testing code is well structured and documented
- Descriptions of subjects and behaviors are part of the test report
- Matchers give you very descriptive error messages
- Integration with ScalaCheck and Mockito

Important Matchers

```
value shouldEqual expected
emptyCollection shouldBe 'empty
emptyCollection should be('empty)
collection should not be 'empty
collection should have size 20
collection should contain(value)
boolean shouldBe true
an[IllegalArgumentException] should be thrownBy expr
```

Or use this approach

```
assert(value === expected)
assert(emptyCollection.empty)
assert(collection.empty === false)
assert(collection.size === 20)
assert(collection.contains(2) === true)
intercept[IndexOutOfBoundsException] {
  collection.foo(badArgument)
}
```

More matchers can be found on the ScalaTest documentation

ScalaTest Exercises

```
man [e] > scalatrain > tests > test
...
[error] (test:compile) Compilation failed
```

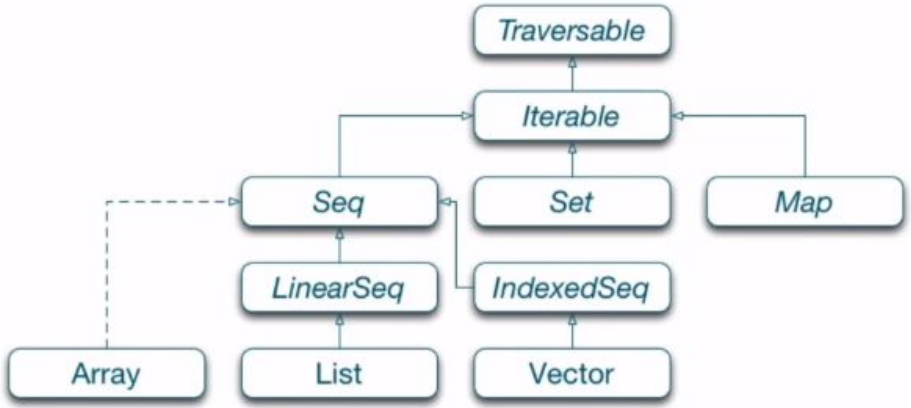
- For the exercises from here on, we have written tests for your guidance
  - nextExercise will bring the tests for the next exercise into your project under *src/test*
  - Your task: Make the tests compile and succeed
- **Attention**: The tests make some assumptions about the code you write, in particular naming and scoping; please adjust your code accordingly

Collections

Objectives

- Overview of Collection hierarchy
- Creating Collection instances
- Important Collections & methods
- Tuples
- Collections & (im)mutability

Collection Hierarchy Extract



- Scala collection library is very comprehensive
- We cover some basics, for more see the online documentation

Creating Collection Instances

```
scala> Vector(1, 2, 3)
res0: Vector[Int] = Vector(1, 2, 3)

scala> Seq(1, 2, 3)
res1: Seq[Int] = List(1, 2, 3)

scala> Set(1, 2, "3")
res2: Set[Any] = Set(1, 2, 3)
```

Applying Syntactic Sugar

```
scala> Vector(1, 2, 3)
res0: Vector[Int] = Vector(1, 2, 3)

scala> Vector.apply(1, 2, 3)
res1: Vector[Int] = Vector(1, 2, 3)
```

- Each collection has a companion object with an *apply* method
- Calling an object is translated into calling *apply*
- *apply* is used as a factory method

Collections Take Type Parameters

```
Seq[A]
Map[A, B]
```

```
scala> Vector(1, 2, 3)
res0: Vector[Int] = Vector(1, 2, 3)

scala> Vector[Int](1, 2, 3)
res1: Vector[Int] = Vector(1, 2, 3)
```

- In Scala there are no raw types, you must specify the element types
- Type parameters are declared in square brackets
- Type arguments can be inferred or given explicitly

Important Collection Methods

- ++ → Concatenates two collections
- *toSeq*, *toSet*, etc. → Convert between collection types
- *isEmpty* and *size* → Provide size information
- *contains* → Tests whether a collection contains an element
- *head* → Returns the first element, *last* the last
- *tail* → Returns all elements except for the first, *init* vice versa
- *take n* → Returns the first *n* elements, *drop n* vice versa
- *zip* → Combines corresponding elements in pairs
- *groupBy* → Partitions a collection into a map of collections

Sequences

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

scala> numbers(0)
res0: Int = 1

scala> numbers :+ 4
res1: Seq[Int] = List(1, 2, 3, 4)

scala> 0 +: numbers
res2: Seq[Int] = List(0, 1, 2, 3)

scala> Seq(1, 2, 1, 3).distinct
res3: Seq[Int] = List(1, 2, 3)
```

- Sequence is a mapping from an index position to an element
- Sequences derive from *Seq*

Right-binding Operators

```
scala> numbers.+:(0)
res0: Seq[Int] = List(0, 1, 2, 3)

scala> 0 +: numbers
res1: Seq[Int] = List(0, 1, 2, 3)

scala> numbers +: 0
<console>:9: error: value +: is not a member of Int
```

- Binary operator ending with a colon *:* binds to the right
- Target and argument of the call are swapped

Sets

```
scala> val numbers = Set(1, 2, 3)
numbers: Set[Int] = Set(1, 2, 3)

scala> numbers(1)
res0: Boolean = true

scala> numbers.contains(1)
res1: Boolean = true

scala> numbers + 4
res2: Set[Int] = Set(1, 2, 3, 4)

scala> numbers + 3
res3: Set[Int] = Set(1, 2, 3)
```

Set is a collection that contains no duplicates

Tuples I

```
scala> Tuple2(1, "a")
res0: (Int, String) = (1,a)

scala> Tuple2(1, 2)
res1: (Int, Int) = (1,2)
```

```
scala> (1, "a")
res0: (Int, String) = (1,a)
```

- Tuples are not collections, but finite heterogeneous containers
- Tuples are case classes parameterized in each field
- The standard library contains *Tuple1* to *Tuple22*
- There is syntactic sugar to ease the creation of tuples

Tuples II

```
scala> val pair = (1, "a")
pair: (Int, String) = (1,a)

scala> pair._2
res0: String = a

scala> 1 -> "a"
res1: (Int, String) = (1,a)
```

- Tuple fields are named *_1*, *_2*, etc.
- *Tuple2*s (Pairs) can be created with the -> operator

Maps

```
scala> val map = Map(1 -> "a", 2 -> "b")
map: Map[Int,String] = Map(1 -> a, 2 -> b)

scala> map(1)
res0: String = a

scala> map.get(9)
res1: Option[String] = None

scala> map.getOrElse(1, "z")
res2: String = a

scala> map.getOrElse(9, "z")
res3: String = z
```

Map is a collection of key-value Pairs with unique keys

Collections and (Im)mutability

- Scala collections come in three flavors
  - *scala.collection* package, the abstract base
  - *scala.collection.immutable* package
  - *scala.collection.mutable* package
- By default (without imports) the immutable collections are used
  - See type aliases in *Predef* and the *scala* package object
  - One exception: *Seq* refers to *scala.collection.Seq* because of arrays

Mutating Immutable Collections

```
scala> val numbers = Vector(1, 2, 3)
numbers: Vector[Int] = Vector(1, 2, 3)

scala> 0 +: numbers
res0: Vector[Int] = Vector(0, 1, 2, 3)

scala> numbers
res1: Vector[Int] = Vector(1, 2, 3)
```

- Immutable collections can't be mutated **in place**
- Mutating operations return a new instance
- Immutable collections are **persistent** (share data structurally)
- **Best practice**: Start immutable and know the performance characteristics

Functional Programming Basics

Objectives

- Higher order functions
- Function literals
- Function types
- Important higher order functions

Functional Collections

- Collections contains a lot of **higher order functions**
  - Methods which take functions as arguments or
  - Methods returning a function
- Intuition for a **function**: Something that can be called
  - A function takes zero or more arguments
  - A function returns a value

Higher Order Means Higher Abstraction

```
scala> val numbers = Vector(1, 2, 3)
numbers: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> numbers.map(number => number + 1)
res0: Vector[Int] = Vector(2, 3, 4)
```

- Higher order functions let you raise the level of abstraction
  - You just tell **what** to do (declarative programming)
  - You don't have to specify **how** it should be done
- Example: *map* takes a function which takes an element of the collection and returns some value

Function Literals

```
numbers.map((number: Int) => number + 1)
numbers.map(number => number + 1)
numbers.map(_ + 1)
```

- Function literal is an anonymous function
  - Useful to pass a function as an argument to a method
- Short notation
  - Each underscore _ stands for one function parameter
  - Can only be used if you use each parameter exactly once

Function Values

```
scala> val addOne = (n: Int) => n + 1
addOne: Int => Int = <function1>

scala> numbers.map(addOne)
res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

- Scala has first-class functions → Function values are objects
- Assign function values to variables
- Pass function values as arguments to higher order functions

Function Types

- If functions are objects, which are their types?
- The REPL shows *Int => Int*, which is syntactic sugar for *Function1[Int, Int]*

```
scala> val addOne = (n: Int) => n + 1
addOne: Int => Int = <function1>
```

- The standard library contains *Function0* to *Function22*
- All function types define an *apply* method

```
scala> addOne(2)
res0: Int = 3
```

Using Methods as Functions

```
scala> def addOne(n: Int) = n + 1
addOne: (n: Int)Int

scala> val addOneFun = addOne
<console>:12: error: missing argument list for method addOne
Unapplied methods are only converted to functions when a function type is expec
<elided>

scala> val addOneFun: Int => Int = addOne
addOne: Int => Int = <function1>

scala> numbers.map(addOne)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

- Methods and functions share a lot, but they are different
- If the compiler expects a function, but you give a method with a matching signature, the method gets **lifted** into a function

Higher order Function: map

```
trait Traversable[A] {
def map[B](f: A => B): Traversable[B]
...
```

map transforms a collection by applying a function to each element

```
scala> val languages = Vector("Scala", "Java", "JavaScript")
languages: Vector[String] = Vector(Scala, Java, JavaScript)

scala> languages.map(language => language.toLowerCase)
res0: Vector[String] = Vector(scala, java, javascript)

scala> languages.map(language => language.length)
res1: Vector[Int] = Vector(5, 4, 10)
```

The collection type remains, but the element type may change

Higher Order Function: flatMap

```
trait Traversable[A] {
def flatMap[B](f: A => Traversable[B]): Traversable[B]
...
```

Like *map*, *flatMap* transforms a collection by applying a function to each element

```
scala> Seq("Now is", "the time").map(s => s.split(" "))
res0: Seq[Array[String]] = List(Array(Now, is), Array(the, time))

scala> Seq("Now is", "the time").flatMap(s => s.split(" "))
res1: Seq[String] = List(Now, is, the, time)
```

Each collection for each element is expanded into the result

Higher Order Function: filter

```
trait Traversable[A] {
def filter(f: A => Boolean): Traversable[A]
...
```

- **predicate** is a unary (one-argument) function returning a *Boolean* value
- *filter* selects elements which satisfy a predicate
- *filterNot* selects elements which don't satisfy the predicate

For Loops and Expressions

Objectives

- Difference between for loops and for expressions
- generators, filters, definitions & yield
- Translation of for loops & for expresssions

For Loops

```
for (seq) block
```

```
scala> for (n <- 1 to 3) println(n)
1
2
3
```

- *for*-loops return *Unit* and execute side-effects
- *seq* contains **generators**, **filters** and **definitions**
- *block* may execute side-effects, its result is ignored

For Expressions

```
for (seq) yield expr
```

- *for*-expression isn't a loop, but returns a result
- *seq* contains **generators**, **filters** and **definitions**
- *expr* creates an element of the result

Generators

```
elem <- coll
```

```
scala> for (n <- Vector(1, 2, 3)) yield n + 1
res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)

scala> for (n <- Set(1, 2, 3)) yield "#" + n
res1: scala.collection.immutable.Set[String] = Set(#1, #2, #3)
```

- Generators drive the iteration
- *coll* is the data to be iterated
- *elem* is a local variable bound to the current element of the iteration
- The (first) generator determines the type of the result

Multiple Generators

```
for {
  n <- 1 to 3
  m <- 1 to n
} yield n * m
```

- Multiple data structures can be iterated in a nested fashion
- Separate multiple generators by semicolon or better use curly braces and new lines
- Quiz
  - What's *to*?
  - What's the result of the above *for*-expression?

Filters

```
if expr
```

```
for {
  n <- 1 to 3 if n % 2 == 1
  m <- 1 to n
} yield n * m
```

- Filters control the iteration
- *expr* must evaluate to a *Boolean* value
- Filters can follow generators on the same line
- Quiz: What's the result of the above *for*-expression?

Translation of For Expressions

- *for*-expressions are translated into possibly nested calls of *flatMap*, *map* and *withFilter*

```
for (n <- 1 to 3) yield n + 1
(1 to 3).map(n => n + 1)

for (n <- 1 to 3; m <- 1 to n) yield n * m
(1 to 3).flatMap(n => (1 to n).map(m => n * m))
```

- Quiz: How does the last exercise get translated?
- **Best practice**: If it comes to iteration, start with a *for*-expression

Translation of For Loops

```
for (n <- 1 to 3) println(n)
(1 to 3).foreach(n => println(n))
```

for-loops are translated into possibly nested calls of *foreach* and *withFilter*

Inheritance and Traits

Objectives

- Scala inheritance basics
- Abstract classes and members
- Scala type system
- Mix-in traits

Inheritance Basics

- Scala supports inheritance
  - Code reuse
  - Specialization
- Each class, except for *Any*, has **exactly one superclass**
  - Non private members are inherited
  - Non final members can be overridden
- Subtype polymorphism
  - Polymorph means "having many forms"
  - The subclass type **conforms** to the superclass type (subclass type **is a** superclass type)

Subclasses

```
class Animal
class Bird extends Animal
```

- *extends* defined a subclass
- *AnyRef* is used as superclass if you omit the *extends* part
- You can only extend exactly one class

Implementation Inheritance

```
class Animal {
  def eat(): Unit = println("Yum yum!")
  private def secret(): Unit = println("Noone must see!")
}

class Bird extends Animal {
  eat() // <-- Inherited
  secret() // <-- Not Inherited
}

<console>:19: error: not found: value secret
  secret() // <-- Not Inherited
```

All non-private members are inherited

The Superclass Constructor

```
class Animal(val name: String)

class Bird(name: String) extends Animal(name)
```

```
scala> class Bird(name: String) extends Animal
<console>:8: error: not enough arguments for constructor Animal: (name: String)A
Unspecified value parameter name.
```

- Subclass must call a superclass constructor with *extends*
- Makes sure that all class parameters are initialized
- **Attention**: Superclasses are initialized first

Final Classes

```
final class Animal
```

```
scala> class Bird extends Animal
<console>:8: error: illegal inheritance from final class Animal
```

Use *final* to prevent a class from being extended

Sealed Classes

```
sealed class Animal

class Bird extends Animal
final class Fish extends Animal
```

- *Sealed* classes can only be extended within the same source file
- Use it to create an **algebraic data type** (ADT) called tagged union
- We can think of sealed classes (ADTs) as being a class that has a compiler enforced limited number of possible subtypes
- Example: *Option* with *Some* and *None*

Overriding

```
class Animal {
  def eat(): Unit = println("Yum yum!")
}

class Bird extends Animal {
  override def eat(): Unit = println("Beep!")
}
```

- Use *override* keyword to override a member
  - It redefines an existing implemented member
- All non final inherited members can be overridden
- Use *final* to prevent a member from being overridden

Accessing Superclass Members

```
class Animal {
  def eat(): Unit = println("Yum yum!")
}

class Bird extends Animal {
  override def eat(): Unit = {
    super.eat()
    println("Beep!")
  }
}
```

Use *super* to access the superclass members

Uniform Access Principle

```
class Animal {
  def name: String = "Arnold"
  // val name: String = "Arnold"
}
```

- Do you remember the uniform access principle: For the client it should make no difference whether a "property" is implemented through storage or through computation
- Properties can be implemented as *def* without parentheses or as *val*
- Quiz: What are the benefits?

Overriding defs with vals

```
class Animal {
  def name: String  = scala.util.Random.shuffle("Arnold".toSeq).mkString
}

class Bird extends Animal {
  override val name: String = "Bill"
}
```

- A *val* is stable, but a parameterless *def* could return different results on different calls
- You can decide to become stable and override a *def* with a *val*
- Why is the other way around not possible?

Lazy vals

```
scala> lazy val lazySeven = {
         println("I am very lazy ;-)")
         7
       }
lazySeven: Int = <lazy>

scala> lazySeven
I am very lazy ;-)
res1: Int = 7
```

- All *val*s are initialized during object construction
- Use *lazy* keyword to defer initialization until first usage
- **Attention**: Lazy *val*s are not final and therefore might show performance drawbacks

Digression: String Interpolation

```
scala> val n = 20
n: Int = 20

scala> s"Value = $n"
res0: String = Value = 20

scala> f"Hex value = $n%02x"
res1: String = Hex value = 14
```

- Since Scala 2.10 you can define processed strings
- String starting with *s* embeds expressions using *$id* or *${ expr }*
- String starting with *f* formats the results of the expressions

Abstract Classes

```
abstract class Animal {
  def name: String
  def eat(): Unit
}
```

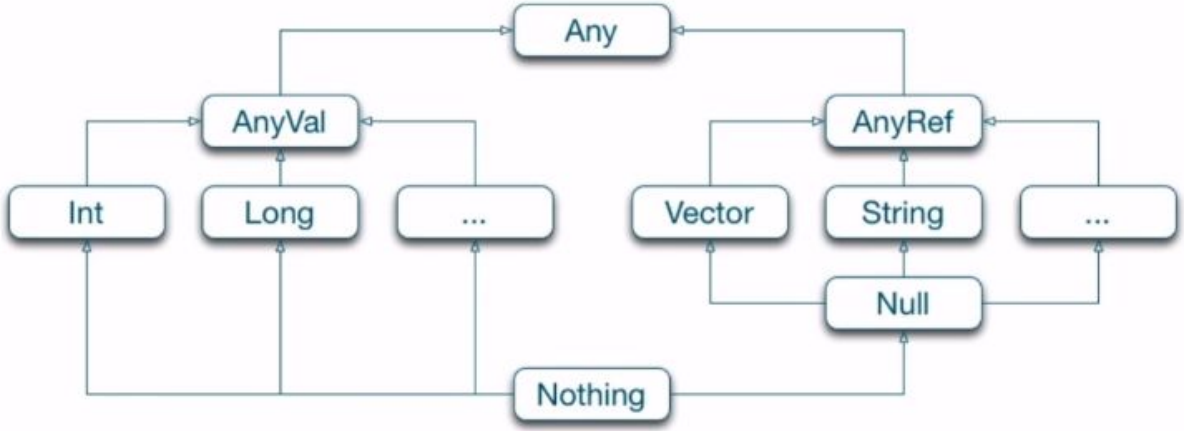
- Abstract classes cannot be instantiated
- Abstract members define only the signature
- Abstract classes can have a mix of abstract and concrete members

Abstract Members

```
class Bird extends Animal {
  def name: String = "Bill"
  override def eat(): Unit = println("Beep!")
}
```

- Implement the body of an abstract member of a superclass
- *override* is optional
- **Best practice for override**: It depends!
  - Use it to indicate member that were declared on super classes
  - Compile errors during refactoring

Scala Type Hierarchy



- *AnyVal* is for value types and *AnyRef* for reference types
- *Null* is a bottom type for reference types
- *Nothing* is the general bottom type

Traits Overview

- On the JVM there is only **single class inheritance**
  - No problems with multiple inheritance like complexity, ambiguity, etc.,
  - Limited possibilities: No multiple inheritance
- Scala introduces **traits** to overcome this limitation
  - Inherit from exactly one superclass
  - **Mix-in** multiple traits

Use Case for Traits

```
abstract class Animal

class Bird extends Animal {
  def fly: String = "I am flying!"
}

class Fish extends Animal {
  def swim: String = "I am swimming!"
}

class Duck // We got stuck with the duck!!
```

Assuming ducks and fish swim alike, it would be good to inherit the *swim* method instead of outsourcing or even duplicating code

Create a trait

```
trait Swimmer {
  def swim: String = "I am swimming!"
}
```

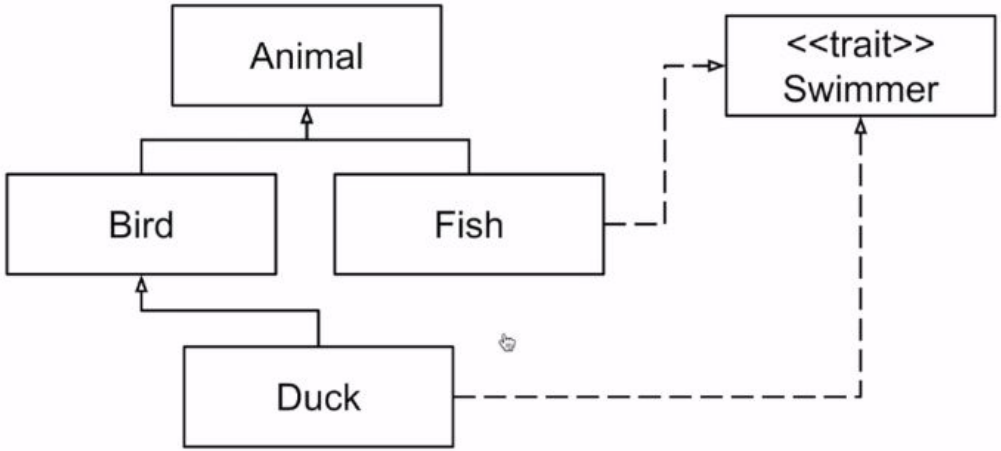
- A Swimmer trait contains the *swim* method

Use Mix-in Composition I

```
abstract class Animal

class Bird extends Animal {
  def fly: String = "I am flying!"
}

trait Swimmer {
  def swim: String = "I am swimming!"
}

class Fish extends Animal with Swimmer
class Duck extends Bird with Swimmer
```

- Use *with* to mix-in a trait

Use Mix-in Composition II



- Fish and Duck can now *swim* via *Swimmer* trait
- And Duck can also *fly* via *Bird* class

Trait Characteristics

```
class Mobility {
  ...
}

trait Swimmer extends Mobility {
  def swim: String = "I am swimming!"
  def float: String
}
```

- Traits may contain concrete and abstract members
- Traits are abstract and can not have parameters
- Like classes, traits extend exactly one superclass

with vs. extends I

```
trait Swimmer
trait Flyer
class Fish extends Swimmer
class Duck extends Swimmer with Flyer
```

- *extends* mixes in the first trait
- *extends* creates either a subclass or mixes-in the first trait
- *with* mixes in subsequent traits

with vs. extends II

```
trait Swimmer extends AnyRef
class Fish extends AnyRef with Swimmer
```

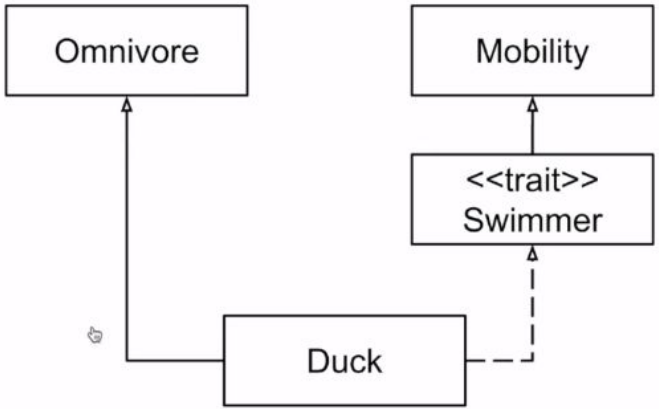
- Bottom Line: First keyword has to be *extends*, regardless whether a class or trait, then use *with* for further mix-ins
- Remember that mixing-in a trait means extending the superclass of the trait
- Quiz: Is *extends AnyRef* needed?

Mix-in Rules: Inheritance Hierarchy I

```
scala> class Omnivore; class Mobility
...
scala> trait Swimmer extends Mobility
...

scala> class Duck extends Omnivore
defined class Duck

scala> class Duck extends Omnivore with Swimmer
<console>:13: error: illegal inheritance; superclass Omnivore
 is not a subclass of the superclass Mobility
 of the mixin trait Swimmer
       class Duck extends Omnivore with Swimmer
                                        ^
```

- A class mixing-in a trait is also extending the superclass of the trait
- Traits must respect the inheritance hierarchy, i.e. a subclass must not extend two incompatible superclasses

Mix-in Rules: Inheritance Hierarchy II



- `class Duck extends Omnivore with Swimmer` has two superclasses
- This is the classic "Diamond Problem"

Mix-in Rules: Concrete Members

```
scala> trait Swimmer { def move = "swimming" }
defined trait Swimmer

scala> trait Flyer { def move = "flying" }
defined trait Flyer

scala> class Duck extends Swimmer with Flyer
<console>:13: error: class Duck inherits conflicting members:
  method move in trait Swimmer of type => String  and
  method move in trait Flyer of type => String
(Note: this can be resolved by declaring an override in class Duck.)
       class Duck extends Swimmer with Flyer
             ^
```

- If multiple traits define the same member, you must use *override*
- To be able to use *override* on *Swimmer* or *Flyer*, either trait has to extend the other or you create a common parent with an abstract member

Pattern Matching

Objectives

- Usage and benefits
- Catch exceptions
- Deconstruct tuples

Match Expressions

```
expr match {
  case pattern1 => result1
  case pattern2 => result2
}
```

- Expressions are matched against a pattern
- Difference to the *switch* statement of C or Java
  - No fall through to the next alternative
  - Scala's match expressions have a very rich set of "patterns" for matching
  - Match expressions return a value
- *MatchError* is thrown if no pattern matches

Match Pattern

```
case pattern => result
```

- *case* declares a match pattern
- *pattern* is not arbitrary code, but one of various pattern types
- *result* is an arbitrary expression
- If *pattern* matches, *result* will be evaluated and returned

Wildcard Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case _ => s"$any is no time!"
}
```

```
scala> whatTimeIsIt("foo")
res0: String = foo is no time!
```

- Use _ as a wildcard to match everything
- Use the wildcard as last alternative to avoid *MatchError*s

Variable Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case x => s"$x is no time!"
}
```

```
scala> whatTimeIsIt("winter time")
res0: String = winter time is no time!
```

- Use an identifier starting with a **small** letter to
  - Match everything
  - Capture the value
- The variable pattern is useful in combination with other patterns

Typed Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case time: Time => s"It is ${time.hours}:${time.minutes}"
  case _ => s"$any is no time!"
}
```

```
scala> whatTimeIsIt(Time(9, 30))
res0: String = It is 9:30
```

- Use a type annotation to match only certain types
- Typed pattern needs to be composed with the wildcard or variable pattern

Constant Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case "12:00" => "High noon"
  case _ => s"$any is no time!"
}
```

```
scala> whatTimeIsIt("12:00")
res0: String = High noon

scala> whatTimeIsIt("10:00")
res1: String = 10:00 is no time!
```

Use a **stable identifier** to match something constant

Stable Identifiers

```
val highNoon = "12:00"

def whatTimeIsIt(any: Any): String = any match {
  case `highNoon` => "High noon"
  case TestData.munich => "Not a time"
  case _ => s"$any is no time!"
}
```

- Stable identifiers are
  - Literals
  - Identifiers for vals or singleton objects starting with
    - Capital letter
    - Small letter enclosed in backticks

Tuple Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case (x, "12:00") => s"From $x to high noon"
  case _ => s"$any is no time!"
}
```

```
scala> whatTimeIsIt(("dusk", "12:00"))
res0: String = From dusk to high noon
```

- Use tuple syntax to match and decompose tuples
- Tuple pattern is composed with other patterns, e.g. with the constant or variable pattern

Constructor Pattern

```
def whatTimeIsIt(any: Any): String = any match {
  case Time(12, 0) => "High noon"
  case Time(12, m) => s"It is $m minutes past 12"
  case _ => s"$any is no time!"
}
```

```
scala> whatTimeIsIt(Time(12, 15))
res0: String = It is 15 minutes past 12
```

- Use constructor syntax to match and decompose case classes
- The constructor pattern is composed with other patterns
- You can build deeply nested structures

Sequence Pattern

```
def matchSeq[A](seq: Seq[A]): String = seq match {
  case Seq(1, 2, 3) => "1 to 3"
  case x +: Nil => s"Only element is $x"
  case _ :+ x => s"Last element is $x"
  case Nil => "Empty sequence"
}
```

```
scala> matchSeq(1 to 3)
res0: String = 1 to 3

scala> matchSeq(Vector(1))
res1: String = Only element is 1

scala> matchSeq(Array(1, 2, 3, 4))
res2: String = Last element is 4
```

Use sequence constructors or append or prepend operators to match and decompose sequences

Pattern Alternatives

```
def whatTimeIsIt(any: Any): String = any match {
  case "00:00" | "12:00" => "Midnight or high noon"
  case _ => s"$any is no time!"
}
```

Use | to combine various pattern alternatives

Pattern Binders

```
def whatTimeIsIt(any: Any): String = any match {
  case time @ Time(_, 0) => s"$time with 0 minutes"
  case time @ Time(_, m) => s"$time with $m minutes"
  case _ => s"$any is no time!"
}
```

- Use @ to bind a variable to a pattern
- This is useful if a combined pattern like a constructor pattern or sequence pattern must be captured as a whole

Pattern Guards

```
def isAfternoon(any: Any) = any match {
  case Time(h, m) if h >=12 => s"Yes, it is ${h-12}:$m pm"
  case Time(h, m) => s"No, it is $h:$m am"
  case _ => s"$any is no time!"
}

scala> isAfternoon(Time(15, 30))
res0: String = Yes, it is 3:30 pm

scala> isAfternoon(Time(9, 30))
res1: String = No, it is 9:30 am
```

- Composing patterns gives you great control over matching
- If that's not enough, use the *if* keyword to define a pattern guard

Catching Exceptions

```
def toInt(s: String): Int =
  try {
    s.toInt
  } catch {
    case _: NumberFormatException => 0
  }
```

- No checked exceptions in Scala (you don't have to catch exceptions)
- Use *try-catch*-expression to catch an exception
- In the *catch*-clause define one or more match alternatives
- *finally*-clause is optional

vals with Patterns

```
scala> val (morning, highNoon) = Time(6) -> Time(12)
morning: Time = Time(6,0)
highNoon: Time = Time(12,0)

scala> val (morning, _) = Time(6) -> Time(12)
morning: Time = Time(6,0)

scala> val midnight = Time()
Midnight: Time = Time(0,0)

scala> val (morning, `midnight`) = Time(6) -> Time(12)
scala.MatchError: (Time(6,0),Time(12,0)) (of class scala.Tuple2)
```

- Use patterns to define vals
- Pay attention to non exhaustive matches

Patterns in Generators

```
for (keyAndValue <- Vector(1 -> "a", 2 -> "b"))
  yield keyAndValue._1 + keyAndValue._2

for ((key, value) <- Vector(1 -> "a", 2 -> "b"))
  yield key + value
```

- Patterns can simplify generators involving tuples
- Decompose the tuple into named elements

Dealing with Optional Values

Objectives

- Introduction to Option
- Pattern Matching on Option
- Higher Order Function on Option

Optional Values

Why is using *null* for a non existing optional value a bad idea?

```
scala> val languages = Map("s" -> "Scala", "j" -> "Java")
languages: Map[String,String] = Map(s -> Scala, j -> Java)

scala> languages.get("s")
res0: Option[String] = Some(Scala)

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

- The standard library has a better alternative: The *Option* class
- It is an ADT with two possible types
  - *Some* case class, wrapping a value
  - *None* singleton object

Creating Option Instances

```
scala> Some("Scala")
res0: Some[String] = Some(Scala)

scala> Option(null)
res1: Option[Null] = None
```

- Use *Some* and *None* directly
- Use *apply* method of the *Option* companion object to wrap *null*ish (Java) APIs to avoid *Some(null)*

Pattern Matching on Option

```
scala> def languageByFirst(s: String): String =
         languages.get(s) match {
           case Some(language) => language
           case None => s"No language starting with $s!"
         }
languageByFirst: (s: String)String

scala> languageByFirst("s")
res0: String = Scala

scala> languageByFirst("c")
res1: String = No language starting with c!
```

- The type system enforces us to handle an optional value properly
- One way is with pattern matching

Higher Order Functions on Option I

```
scala> Option("Scala").map(_.reverse)
res0: Option[String] = Some(alacS)

scala> for {
         language <- Some("Scala")
         behavior <- Some("rocks")
       } yield s"$language $behavior"
res1: Option[String] = Some(Scala rocks)
```

- *Option* offers higher order functions known from collections
- Using higher order functions such as *map* and *flatMap* on *Option* lets you easily compose a chain of calls that handles missing values gracefully

Higher Order Functions on Option II

- *foreach* calls a function only if a value is present. As for collections, *foreach* executes the function for its side-effects, any return value would be discarded.

```
scala> val languages = Map("s" -> "Scala", "j" -> "Java")
languages: scala.collection.immutable.Map[String,String] = Map(s -> Scala, j -> J

scala> languages.get("s").foreach(println)
Scala

scala> languages.get("c").foreach(println)

scala>
```

Obtaining the Value

- *getOrElse* extracts the wrapped value or returns a default

```
def languageByFirst(s: String): String =
  languages.get(s).getOrElse(s"No language starting with $s!")
```

Handling Failure

Objectives

- Using Try
- Pattern Matching on Try
- Higher Order Function on Try
- Chaining Calls with Try

Exceptions

```
def toInt(s: String): Int =
  try {
    s.toInt
  } catch {
    case _: NumberFormatException => 0
  }
```

- You can use *try/catch* in Scala
- If you don't need *finally*, Scala provides a very useful alternative: *Try*

Using Try

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

scala> Try("100".toInt)
res0: scala.util.Try[Int] = Success(100)

scala> Try("Martin".toInt)
res1: scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input stri
```

Try executes the given expression and returns *Success* or *Failure*

Pattern Matching on Try

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

scala> def makeInt(s: String): Int = Try(s.toInt) match {
     |   case Success(n) => n
     |   case Failure(_) => 0
     | }
makeInt: (s: String)Int

scala> makeInt("35")
res2: Int = 35

scala> makeInt("James")
res3: Int = 0
```

- Use *Try* in the same way you use *Option*
- One way is with pattern matching

Higher Order Functions on Try

```
def makeInt(s: String): Int =
  Try(s.toInt).getOrElse(0)
```

```
scala> Success("Scala").map(_.reverse)
res0: scala.util.Try[String] = Success(alacS)

scala> for {
         language <- Success("Scala")
         behavior <- Success("rocks")
       } yield s"$language $behavior"
res1: scala.util.Try[String] = Success(Scala rocks)
```

- *getOrElse* extracts the wrapped value or returns a default
- *Try* offers higher order functions known from collections

Chaining Calls with Try

- Ability to pipeline, chain operations, catching exceptions along the way
- *flatMap* and *map* can be called on Failure
- Important: Only non-fatal exceptions are caught by Try (see *scala.util.control.NonFatal*). Serious system errors will still be thrown.