Pragmatic real-world Scala (Advanced Scala)

Heiko Seeberger

Skills Matter

2011-08-22 to 2011-08-23

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

Explicitly implicit

Internal DSLs

Contributing to the Scala collections

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

Explicitly implicit

Internal DSLs

Contributing to the Scala collections

Scala distribution

Exercise: Install the Scala distribution

- Download the current stable release as an archive for your platform (*.tgz* or *.zip*) from www.scala-lang.org/downloads
- Unpack the archive to a suitable location, e.g. *~/tools/scala*
- Add the *bin* directory to your path
- Verify the installation by opening a terminal and entering *scala -version*:

```
tmp$ scala -version
Scala code runner version 2.9.0.1 -- Copyright
    2002-2011, LAMP/EPFL
```

- Also download the Scala API documentation, unpack and browse it

Exercise: "Hello World!" on the command line

- Create the file *Hello.scala*[1] using an arbitrary text editor:

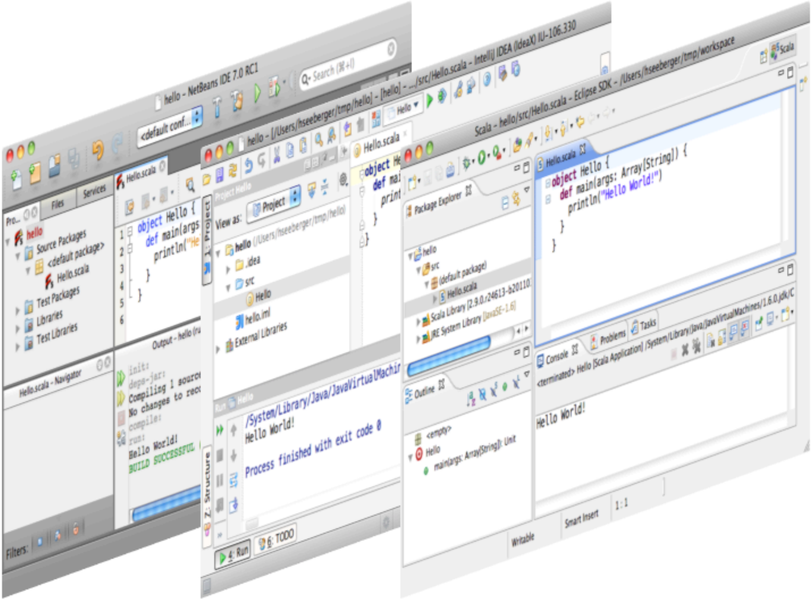
```scala
object Hello {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

- Compile and run it:

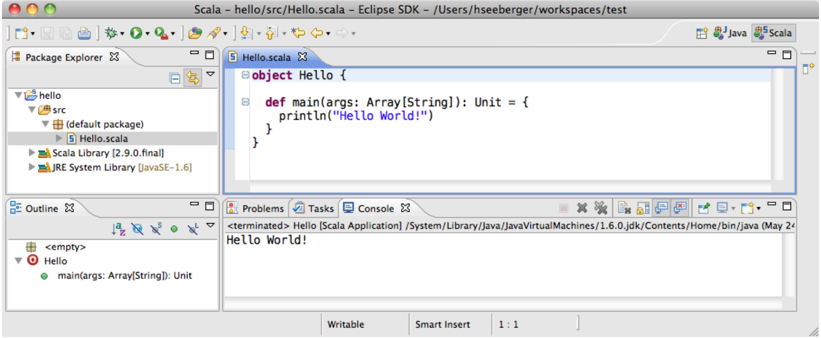
```
tmp$ scalac Hello.scala
tmp$ scala Hello
Hello World!
```

[1] You don't have to understand the code yet!

There are plugins for all major IDEs



Scala IDE for Eclipse



- We, the trainer(s), will use Eclipse for this course
- Feel free to use another IDE or none at all, but we will only be able to offer limited support

Exercise: Install Eclipse and the Scala IDE for Eclipse

- Download and install Eclipse Indigo (3.7) Classic for your platform from www.eclipse.org/downloads/
- Install the Scala plugin via the menu *"Help > Install New Software ..."* using the update site download.scala-ide.org/releases-29/2.0.0-beta
- Verify the installation by opening a fresh workspace, e.g. ~/*workspaces/fasttracktoscala* and switching to the Scala perspective

Exercise: "Hello World!" in Eclipse

- Create a *"New > Scala Project"* with name *hello*
- Create a *"New > Scala Object"* with name *Hello*
- Copy the code from the previous exercise
- Select *"Run As > Scala Application"* from the context menu of the editor or package explorer
- In order to avoid conflicts with other future projects we suggest you now close or delete this project

Simple Build Tool (sbt)

```
tmp$ cd scalatrain
scalatrain$ sbt
[info] Set current project to default (in build file:/Users/hseeberger/.sbt/plugins/)
[info] Set current project to default (in build file:/Users/hseeberger/tmp/scalatrain/)
> compile
[success] Total time: 0 s, completed May 24, 2011 1:14:42 PM
```

- THE build tool for Scala
- Writen in Scala and specifically for Scala
- Used by most real-world projects

Exercise: Install sbt

- Download the launcher: repo.typesafe.com/typesafe/releases/org.scala-tools.sbt/sbt-launch/0.10.1/sbt-launch.jar
- Create the following file as a start script for sbt:
  - *sbt* on Mac/Linux: *java -Xmx512M -jar LAUNCHER "$@"*
  - *sbt.bat* on Windows: *java -Xmx512M -jar LAUNCHER %**

Exercise: Create a sbt project

- Create a fresh project directory, e.g. *~/projects/scalatrain* and cd into it
- Attention: Do not create this in your Eclipse workspace!
- Starting sbt will take you to an interactive session
- Execute the following three commands at the sbt prompt:

```
> set name := "scalatrain"
...
> set scalaVersion := "2.9.0-1"
...
> session save
...
```

- Take a look at the fresh file *build.sbt*
- Keep the sbt session running!

sbt commands - quick overview

- General commands:
  - *exit* or *quit* ends the current session
  - *help* lists available commands
- Build commands:
  - *compile* compiles main sources
  - *test:compile* compiles test sources
  - *test* runs tests
  - *console* starts the REPL
  - *run* looks for a main class and runs it
  - Triggered execution: Prefix a command with $\sim$
- Other commands:
  - *clean* deletes all output in the *target* directory
  - *reload* reloads the build

Exercise: Install the sbt-Eclipse integration

- The sbteclipse plugin let's you create Eclipse project files from an sbt project
- In the project directory create the file *project/plugins/build.sbt* with the below contents
- Attention: Copy and paste is your friend, but pay attention to the details:
  - The blank line between the blocks is important!
  - Sometimes the quotes are not copied correctly!
  - You could also copy from github.com/typesafehub/sbteclipse

```
resolvers += {
  val typesafeRepoUrl = new java.net.URL("http://repo.typesafe.com/typesafe/releases")
  val pattern = Patterns(false,
          "[organisation]/[module]/[sbtversion]/[revision]/[type]s/[module](-[classifier])-[revision].[ext]")
  Resolver.url("Typesafe Repository", typesafeRepoUrl)(pattern)
}

libraryDependencies <<= (libraryDependencies, sbtVersion) { (deps, version) =>
  deps :+ ("com.typesafe.sbteclipse" %% "sbteclipse" % "1.3-RC2" extra("sbtversion" -> version))
}
```

Exercise: Create Eclipse project files

- In the sbt session execute the commands *reload* and the now available *eclipse* with argument *create-src*

```
> reload
...
> eclipse create-src
...
[info] Successfully created Eclipse project files ...
```

- Import the new Eclipse project using *"Import..."* > *"Existing Projects into Workspace"*
- Verify the import by inspecting the project, e.g. the source folders *src/main/scala* etc.

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

Explicitly implicit

Internal DSLs

Contributing to the Scala collections

ScalaTrain case study

- Objectives: Journey planner for trains
- Covers all of the Scala basics discussed on the next slide
- Initial state provided on the accompanying USB stick

Exercise: Copy ScalaTrain case study

- Copy the all contents of the directory *solutions/000_Exercise__Add_XML_serialization_to_Time* from the accompanying USB stick to the project directory *scalatrain* on your local disc
- Cd into the project directory, reload the sbt session and recreate the Eclipse project files
- Update the Eclipse project if necessary

Short recap of important basics

- Basic OO features
- Testing using specs2
- Collections and functional programming
- For-expressions and for-loops
- Inheritance and traits
- Pattern matching
- XML support

Agenda

- Setting up the development environment
- Bootstrapping
- Functional programming in depth
- Mastering the type system
- Explicitly implicit
- Internal DSLs
- Contributing to the Scala collections

How can we avoid call site evaluation?

- Sometimes we want arguments not be evaluated at call site, e.g. when evaluation is costly and possibly superfluous:

```
debug("This" + " is" + " very" + " costly!")
```

- We could use arity-0 functions:

```
def debug(msg: () => String) { println(msg()) }
debug(() => "This" + " is" + " very" + " ugly!")
```

- But that is an ugly workaround, isn't it?

By-name parameters

- We can do better!
- Use => in a type annotation to define a by-name parameter:

```scala
def debug(msg: => String) { println(msg) }
debug("This" + " is" + " a" + " by-name" + " param.")
```

- The given argument will be evaluated every time it is used
- Attention: This is not lazy evaluation but like calling a function!

Exercise: Create a wrapper around SLF4J (1)

- Add dependencies to SLF4J and Logback:

```
libraryDependencies ++= Seq(
    "org.slf4j" % "slf4j-api" % "1.6.1",
    "ch.qos.logback" % "logback-classic" % "0.9.28",
    "org.specs2" %% "specs2" % "1.4" % "test")
```

- Use the package *org.scalatrain.util* for the class and trait to be created on next slide ...

Exercise: Create a wrapper around SLF4J (2)

- Create the class *Logger*:
  - Rename SLF4J *Logger* to *Slf4jLogger* using an import selector clause
  - Add a class parameter of type *Slf4jLogger*
  - Write a delegate method for *debug* (and optionally for *error*, *warn*, etc.) using a by-name parameter for the message
  - Only delegate to the wrapped *Slf4jLogger* if the respective log level is enabled
- Create the trait *Logging*:
  - Add a lazy immutable field *logger* initialized with a *Logger*
  - Initialize the wrapped *Slf4jLogger* with the fully qualified name of the class into which *Logging* is mixed-in
- Mix *Logging* into *JourneyPlanner* and log a debug message about the *Train*s the *JourneyPlanner* was initialized with

Local methods

- Just like local variables we can define local methods:

```
scala> def foo = {
     |   def bar = "bar"
     |   "foo" + bar
     | }
foo: java.lang.String

scala> foo
res0: java.lang.String = foobar
```

- Note: Local methods are translated to functions

Recursion by example: Factorial

- The factorial of an natural number is defined recursively:
  - factorial(0) = 1
  - factorial(n) = n * factorial(n - 1) for n $>$ 0
- We can implement it accordingly:

```
def factorial(n: BigInt): BigInt = {
  require(n >= 0, "n must not be negative!")
  if (n == 0) 1 else n * factorial(n - 1)
}
```

- Attention: Type annotation required!

Stack overflows and tail recursion

- Recursion might be stack intensive and even cause stack overflows:

```
scala> factorial(100000)
java.lang.StackOverflowError
at scala.math.BigInt.longValue(BigInt.scala:335)
...
```

- The Scala complier can optimize tail recursive algorithms, i.e. operations where the recursive call is the last instruction
- Therefore make your recursive operations tail recursive!

Use *@tailrec*

- Use the annotation *@tailrec* to make the compiler verify that an operation is tail recursive:

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> @tailrec def factorial(n: BigInt): BigInt = {
     |   require(n >= 0, "n must not be negative!")
     |   if (n == 0) 1 else n * factorial(n - 1)
     | }
<console>:10: error: could not optimize @tailrec
    annotated method factorial: it contains a
    recursive call not in tail position
```

Transforming to tail recursion by example

- Use an accumulator parameter:

```scala
@tailrec def factorial(acc: BigInt, n: BigInt):
    BigInt = {
  require(n >= 0, "n must not be negative!")
  if (n == 0) acc else factorial(n * acc, n - 1)
}
```

- Use a local method to prevent API pollution:

```scala
scala> def factorial(n: BigInt): BigInt = {
  require(n >= 0, "n must not be negative!")
  @tailrec def factorial(acc: BigInt, n: BigInt):
      BigInt =
    if (n == 0) acc else factorial(n * acc, n - 1)
  factorial(1, n)
}
```

Exercise: Check that *Time.schedule* is increasing

- Add the method *isIncreasing* taking a *Seq[Time]* as argument to the singleton object *Time*
- Return *true* only if the given *Seq[Time]* is increasing
- Implement it in a tail recursive fashion
- Use *Time.isIncreasing* to check that the *Train.schedule* is increasing in time (TODO comment)
- Add tests to *TimeSpec* and *TrainSpec*

Partial functions

- A partial function need not be defined on its whole domain
- In Scala a *PartialFunction* is a subtype of *Function1*:

```
trait PartialFunction [-A, +B] extends (A) => B
```

- Use the method *isDefinedAt* to determine whether a partial function is defined for a given value:

```
scala> val pf = Map(1 -> "a")
pf: ...Map[Int,...String] = Map(1 -> a)

scala> pf isDefinedAt 1
res0: Boolean = true

scala> pf isDefinedAt 2
res1: Boolean = false
```

Partial function literals

- Use a block of *case* alternatives to define a partial function literal:

```
scala> ('a' to 'f').zipWithIndex filter {
     |   case (_, i) if i % 2 == 0 => true
     |   case _ => false
     | }
res0: ... = Vector((a,0), (c,2), (e,4))
```

- Sometimes this syntax is more lightweight to define functions than the “usual” way, thanks to pattern matching
- Pay attention to exhaustive matches to avoid runtime errors!

A powerful collection method: *sliding*

- Groups elements in fixed size blocks by passing a "sliding window" over them:

```scala
def sliding(size: Int): Iterator[Seq[A]]
```

- Example:

```scala
scala> 1 to 4 sliding 2 foreach println
Vector(1, 2)
Vector(2, 3)
Vector(3, 4)
```

Exercise: Reimplement *Time.isIncreasing* using *sliding*

- Add the method *isIncreasingSliding* taking a *Seq[Time]* as argument to the singleton object *Time*
- Return *true* only if the given *Seq[Time]* is increasing
- Implement it using *sliding*
- Add tests to *TimeSpec*

Curried methods

- A method can have more than one parameter list, which is called currying[2]:

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

scala> add(1)(2)
res0: Int = 3
```

- Give all argument lists to invoke a curried method:

```scala
scala> add(1)(2)
res0: Int = 3
```

[2] After the mathematician Haskell Curry.

Partially applied functions

- Replacing one or more argument lists with an underscore yields a partially applied function:

```
scala> val addOne = add(1) _
addOne: (Int) => Int = <function1>

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

- The underscore can be omitted when the Scala compiler expects a function with “matching” signature:

```
scala> def isEven(i: Int) = i % 2 == 0
isEven: (i: Int)Boolean

scala> 1 to 4 filter isEven
res0: ...IndexedSeq[Int] = Vector(2, 4)
```

A powerful collection method: *foldLeft*

- *foldLeft* transforms a collection into a single value
- Thereto it applies the given arity-2 function to a start value and all elements, going left to right:

```scala
def foldLeft[B](b: B)(f: A => B): B
```

- Examples:

```scala
scala> Seq(1, 2, 3).foldLeft(0) { _ + _ }
res0: Int = 6

scala> Seq(1, 2, 3).foldLeft(1) { _ * _ }
res1: Int = 6
```

Exercise: Reimplement *Time.isIncreasing* using *foldLeft*

- Add the method *isIncreasingFold* taking a *Seq[Time]* as argument to the singleton object *Time*
- Return *true* only if the given *Seq[Time]* is increasing
- Implement it using *foldLeft*
- Hint: Use a *Pair* as start value
- Add tests to *TimeSpec*

Group exercise: Power of folding

- Folding is very powerful and allows to implement almost every other collection method in terms of a fold
- Let's try to implement *map*, *flatMap* and *filter* for *List*s using *foldLeft*

Exercise: Calculate connections (1)

- Add the field *backToBackStations* of type *Seq[(Station, Station)]* to *Train*:
  - Initialize it with the sequence of all pairs of consecutive *Station*s, e.g. *Seq(a -> b, b -> c)* for *Seq(a, b, c)*
  - Add tests to *TrainSpec* verifying that *Train.backToBackStations* is initialized correctly
- Add the field *departureTimes* of type *Map[Station, Time]* to *Train*:
  - Initialize it with all *Station*s mapped to the according *Time*s
  - Add tests to *TrainSpec* verifying that *Train.departureTimes* is initialized correctly

Exercise: Calculate connections (2)

- Create the case class *Hop* with the class parameters:
  - *from* of type *Station*, must not be *null*
  - *to* of type *Station*, must not be *null* and not equal to *from*
  - *train* of type *Train*, *from* and *to* must be back-to-back *Station*s of *train*
  - Add checks for the above preconditions
  - Create the test specification *HopSpec* and add tests for the precondition checks
- Add the fields *departureTime* and *arrivalTime* of type *Time* to *Hop*:
  - Use *Train.departureTimes* with *from* and *to* respectively
  - Add tests to *HopSpec* verifying that these fields are initialized correctly

Exercise: Calculate connections (3)

- Add the field *hops* of type *Map[Station, Set[Hop]]* to *JourneyPlanner*
- Initialize it with all *Hop*s grouped by the departure *Station*
- Add tests to *JourneyPlannerSpec* verifying that *JourneyPlanner.hops* is initialized correctly

Exercise: Calculate connections (4)

- Add the method *connections* to *JourneyPlanner* with the following parameters:
  - *from* of type *Station*
  - *to* of type *Station*
  - *departureTime* of type *Time*
- Return a *Set[Seq[Hop]]*, for the time being *Set.empty*
- Add precondition checks for:
  - *from*, *to* and *departureTime* must not be *null*
  - *from* and *to* must not be equal
- Add tests for the precondition checks to *JourneyPlannerSpec*

Exercise: Calculate connections (5)

- Finalize the method *JourneyPlanner.connections*:
  - Drop connections starting earlier that the given *departureTime*
  - Drop connections where the *departureTime* of a subseqent *Hop* is earlier that the *arrivalTime* of the current one
  - Drop connections with duplicate *Station*s
  - Hint: Use a recursive algorithm
- Add tests to *JourneyPlannerSpec* verifying that this method is implemented correctly

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

Explicitly implicit

Internal DSLs

Contributing to the Scala collections

Type parameters

- Classes, traits and methods can be (type-)parameterized
- Use square brackets after the identifier to define a single type parameter or a list of such:

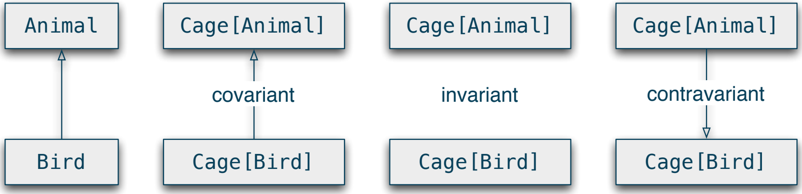
```
trait Set[A]
trait Map[A, B]
def map[B](f: A => B): Set[B]
```

- Convention: Single capital letters starting with *A*

Variance

- Subtyping immediately gives rise to the question of variance:
- “Given two types with a subtype relationship, what is the relation of a type parameterized with each of those?”

```scala
class Animal
class Bird extends Animal
class Cage[A]
```



Variance and mutability

- At first approximation variance results from mutability:
  - Read-only (immutable) => Covariant
  - Write-only => contravriant
  - Read-write (mutable) => Invariant
- Examle: Try to put an elephant into a bird cage!

```scala
val birdCage = new Cage[Bird]
val cage: Cage[Animal] = birdCage // OK if covariant
cage.put(new Elephant) // Oops!
```

Variance declarations

- By default parameterized types are invariant:

```
scala> val birdCage: Cage[Animal] = new Cage[Bird]
<console>:13: error: type mismatch;
 found   : Cage[Bird]
 required: Cage[Animal]
```

- Use $+$ to declare a covariant type parameter:

```
class Cage[+A]
```

- Use - to declare a contravariant type parameter:

```
class Cage[-A]
```

When can we use variance declarations?

- The answer depends on whether type variables are used in positive or negative occurrences:

```
class Cage[A] {
  get: A // positive
  put(animal: A): Unit // negative
  val animal: A // positive
  var animal: A // positive and negative
}
```

- Only positive occurences $\sim$ immutable => covariant
- Only negative occurences $\sim$ write-only => contravariant

Variance declarations and the Scala compiler

- Don't worry, the Scala compiler knows the rules!
- Examle: Try to make a negatively used type parameter covariant

```
scala> class Cage[-A] {
     |   def get: A = sys.error("Not implemented")
     | }
<console>:8: error: contravariant type A occurs in
    covariant position in type => A of method get
```

Exercise: Create an immutable queue

- Create the class *Queue* with a type parameter *A* and a class parameter *elements* of type *List[A]]*
- Make the constructor private and add a companion object defining the factory method *apply* with a type parameter *A* and repeated parameters *elements* of type *A*
- Add the method *dequeue* returning a *Tuple2* of the first element and a new *Queue* without the dequeued element or throwing a *NoSuchElementException* for an empty *Queue*
- Create the specification *QueueSpec* and add tests for *Queue.dequeue*

Exercise: Make *Queue* covariant

- In the REPL create the class *Animal* and its subclass *Bird*
- Create a *val* of type *Queue[Bird]*
- Try to assign it to a *val* of type *Queue[Animal]*
- Make *Queue* covariant
- Try again to use a *Queue[Bird]* as a *Queue[Animal]*

How can we add the method *enqueue* to *Queue*?

- We would like to add the following method:

```
def enqueue(element: A): Queue[A] = ...
```

- But the Scala compiler will complain:

```
[error] ... covariant type A occurs in contravariant
    position in type A of value element
```

Lower bounds

- Use $>:$ to define a lower bound on a type parameter:

```scala
case class Cage[A >: Animal](a: A)
```

- The type will automatically be widened as far as necessary:

```scala
scala> Cage(new Bird)
res0: Cage[Animal] = Cage(Bird@370c488c)

scala> Cage(new Animal)
res1: Cage[Animal] = Cage(Animal@53bb112d)

scala> Cage("String")
res2: Cage[java.lang.Object] = Cage(String)

scala> Cage(1)
res3: Cage[Any] = Cage(1)
```

Exercise: Add the method *enqueue* to *Queue*

- This method shall create a new *Queue* with the given element added to the end
- Hint: Use a lower bounded type parameter
- Add tests to *QueueSpec* verifying that the resulting *Queue* is correct

Upper bounds

- An upper bound expresses an "is a" relation
- Use $<:$ to define an upper bound on a type parameter:

```scala
case class Cage[A <: Animal](a: A)
```

- Only subtypes of the given upper bound are accepted:

```scala
scala> Cage(new Bird)
res0: Cage[Bird] = Cage(Bird@3b517b79)

scala> Cage(new Animal)
res1: Cage[Animal] = Cage(Animal@61b1acc3)

scala> Cage("String")
<console>:11: error: inferred type arguments
    [java.lang.String] do not conform to method
    apply's type parameter bounds [A <: Animal]
```

Digression: Package objects

- Sometimes a singleton object isn't the perfect place, but since Scala 2.8 we can add arbitrary members directly to packages
- Use the keywords *package object* to define a package object:

```scala
package foo
package object bar {
  def baz = ...
}
```

- Convention: Use a file named *package.scala*

Exercise: Generalize the methods *Time.isIncreasing**

- There is no reason why these methods sould be restricted to *Time*
- Create the package object *org.scalatrain.util* and move these methods there
- Create the new specification *utilSpec* in package *org.scalatrain.util* and move the tests there
- Now make the methods work with any sequence of *Ordered* elements

A word or two about contravariance

- At first glance contravariance might looks strange, but sometimes it is just natural:

```scala
class Cage[-A <: Animal] {
  def put(animal: A): Unit = ...
}
```

- If we have a *Cage[Animal]* which is suitable for any animal, we can certainly use it as a *Cage[Bird]*, i.e. put a *Bird* into it
- Functions are contravariant in their argument types and covariant in their result type:

```scala
trait Function1[-A , +B] {
  def apply(a: A): B
}
```

Type members

- Classes, traits and singleton/package objects can have type members in addition to fields and methods[3]
- Use the keyword *type* to define a type member:

```
1 type List[+A] = scala.collection.immutable.List[A]
2 type Date
3 override type Date = java.util.Date
```

- (1) is taken from the package object *scala* and shows a type alias
- (2) shows an abstract type member
- (3) shows a concrete type member implementing or overriding a supertype's type member

[3] We will talk about inner classes etc. soon.

Inner types

- Classes, traits and singleton/package objects can also have inner types, i.e. classes, traits or type members:

```scala
class Outer {
  class InnerClass
  trait InnerTrait
  type InnerType = String
}
```

Path-dependent types I

- To access an inner type you need an outer instance:

```
scala> class Outer {
     |   class Inner
     |   def put(inner: Inner): Unit = ()
     | }
defined class Outer

scala> val outer1 = new Outer
outer1: Outer = Outer@184307f2

scala> val inner1 = new outer1.Inner
inner1: outer1.Inner = Outer$Inner@57c93cf5
```

- There is no way to access an inner type like this:

```
scala> new Outer.Inner
<console>:8: error: not found: value Outer
```

Path-dependent types II

- The type of an inner instance depends on the outer instance:

```
scala> val outer2 = new Outer
outer2: Outer = Outer@4b24e48f

scala> val inner2 = new outer2.Inner
inner2: outer2.Inner = Outer$Inner@155a792d

scala> outer1.put(inner2)
<console>:12: error: type mismatch;
 found   : outer2.Inner
 required: outer1.Inner
```

- Therefore the inner types of two outer instances are different

Type selections

- The inner types share a common supertype, denoted by a type selection *Outer#Inner*:

```
scala> val inner1: Outer#Inner = new outer1.Inner
inner1: Outer#Inner = Outer$Inner@19f9bdc4

scala> val inner2: Outer#Inner = new outer2.Inner
inner2: Outer#Inner = Outer$Inner@50b98ef4
```

- Type selections (*Outer#Inner*) operate on types, whereas path-dependent types (*outer.Inner*) operate on instances
- You cannot create an instance of a type selection

Singleton types

- Use *.type* to access the type that represents a given object:

```
scala> inner1.isInstanceOf[inner1.type]
res0: Boolean = true

scala> inner2.isInstanceOf[inner1.type]
res1: Boolean = false
```

- A path-dependent type is just a short-hand notation for a singleton type and a type selection:

```
scala> val inner1 = new outer1.type#Inner
inner1: outer1.Inner = Outer$Inner@2ce90f2d
```

Refinements and structural typing

- An existing type can be refined anonymously:

```
type Date = java.util.Date {
  def time: Long
}

val date: Date = new java.util.Date {
  def time: Long = getTime
}
```

- A refinement defining new fields or methods is called a structural type

Structural typing and static duck typing

- "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."[4]
- Simply omit the type to be refined and just declare some fields or methods:

```
def using[A <: { def close(): Unit }, B](
    resource: A)(block: A => B): B = {
  try {
    block(resource)
  } finally {
    if (resource != null) resource.close()
  }
}

using(new FileWriter("./test.txt")) { out =>
  out.write("test")
}
```

[4]Wikipedia: Duck test

Group exercise: Feeding animals properly

- Let there be animals that eat specific food, e.g.:
  - Birds eat grains
  - Cows eat grass
- Next let there be fish that eat fish
- Then let fish eat all that can swim
- Finally add the method *id* to *Animal* that just returns *this*:
  - Try to chain method calls, e.g. *bird.id.eat(Grains)*
  - Why doesn't this work? How can it be fixed?

Phantom types

- Phantom types are never instantiated
- They are used as compile-time constraints, e.g. to impose a certain order on chained method calls:

```
phantom.first.second.third // OK
phantom.first.first // Won't compile!
```

Group exercise: Waking up animals properly

- Add the methods *wakeUp* and *goToSleep* to the class *Animal*
- Apply a phantom type based approch to ensure that:
  - *wakeUp* can only be called for asleep animals
  - *goToSleep* can only be called for awake animals

Self types I

- Use the keyword *this* followed by a type annotation and => to declare a self type:

```
trait Foo {
  def foo = "foo"
}

class Bar {
  this: Foo =>
  def bar = foo
}
```

- Technically a self type is an assumed type for *this* within the enclosing type

Self types II

- Practically a self type is a requirement for the type of an object being created in a *new* expression:

```
scala> new Bar
<console>:10: error: class Bar cannot be instantiated
    because it does not conform to its self-type Bar
    with Foo

scala> new Bar with Foo
res0: Bar with Foo = $anon$1@4a4cfc65
```

- It determines the traits to be mixed it
- But it does not expose these in the type

Self names

- Use an arbitrary identifier instead of *this* to declare a named self type:

```
object Foo {
  self =>
  object Bar {
    val fooClass = self.getClass
  }
}
```

- When omitting the type annotation the self type is taken to be the enclosing type

Exercise: Self types

- Change *Logger* from a class to a trait
- Apply the neccessary changes to *Logging*
- Attention: Log output must still contain the correct *Logger*
- Wrong:

```
scala> new JourneyPlanner(Set())
17:18:41.785 [Thread-23] DEBUG
    org.scalatrain.util.Logging$$anon$1 ...
```

- Right:

```
scala> new JourneyPlanner(Set.empty)
17:19:18.549 [Thread-27] DEBUG
    org.scalatrain.JourneyPlanner ...
```

Agenda

Sometimes things don't fit

- Why does the following compile?

```
'2' -> "ABC"

"org.slf4j" % "slf4j-api" % "1.6.1"

planner.isShortTrip(null, frankfurt) must throwA[IAE]
```

- There is no operator -> on *Char*
- There is no operator *%* on *String*
- There is no method *must* on *Boolean*

If it doesn't fit, use a bigger hammer

- The Scala compiler doesn't give up too soon:
- If an actual type doesn't match the expected, it looks for an implicit conversion to the expected type

```
2 - "1"
```

- If a non-existing method/field is called/accessed, it looks for an implicit conversion of the receiver

```
"2" - 1
```

What is an implicit conversion?

- An implicit conversion is an arity-1 method from one type to another[5]
- Use the keyword *implicit* to define a implicit conversion:

```scala
implicit def fromString(s: String): Int = s.toInt
```

- The name is irrelevant, but by convention it should be descriptive, e.g. from<source> or <source>To<target>

[5] Actually this kind of implicit conversions is called views; for details see the Scala language specification, section 7.3.

When do implicit conversions apply?

- In order to be applied, an implicit conversion must be in scope
- Current scope: Identifiers accessible without prefix
  - Local identifiers
  - Members of an enclosing scope
  - Imported identifiers
- Implicit scope: Members of companion objects of associated[6] types
  - The type in question itself
  - All parts of a parameterized type, e.g. A[B, C]
  - All parts of a compound type, e.g. A with B with C

[6] Slightly simplified; for details see the Scala language specification, section 7.2.

What if multiple implicit conversions apply?

- Precedence rules (from highest to lowest)[7]:
  - Local implicits
  - Imported implicits
  - Companion object of the type
  - Companion object of type arguments of the type
  - Outer types of compound types
- Best practice: Go for companion objects!
  - No imports needed
  - Local overrides possible

[7] Again slightly simplified.

There can only be one!

```
scala> def foo(bar: String) = bar.reverse
foo: (bar: String)String

scala> implicit def fromInt(i: Int) = i.toString
fromInt: (i: Int)java.lang.String

scala> implicit def fromInt2(i: Int) = i.toString
fromInt2: (i: Int)java.lang.String

scala> foo(1)
<console>:11: error: type mismatch;
 found   : Int(1)
 required: String
Note that implicit conversions are not applicable because
    they are ambiguous:
...
```

Exercise: Implicitly convert a *String* to a *Station*

- Motivation: There are many places where we need *Station*s. Wouldn't it be nice if we could just use *String*s instead of *Station* instances wrapping a *String*? Example:

```
planner trainsAt "Munich"
```

- Add an implicit conversion from *String* to *Station*
- Which is the best location?
- Add tests to *StationSpec* verifying that the implicit conversion is working correctly
- Also give it a try in the REPL

Digression: Regular expressions

- Regular expressions in Scala build upon *java.util.regex*
- Use the method *r* to easily create a pattern[8] from a *String*:

```
scala> val date = """(\d{1,2})/(\d{1,2})""".r
date: scala.util.matching.Regex = (\d{1,2})/(\d{1,2})
```

- Take a look at the API documentation to discover the available methods, e.g. *findAllIn*, *findFirstIn*, *replaceAllIn*, etc.
- Regular expressions can also be used in pattern matching, whereby capturing groups define the elements of the pattern:

```
scala> val date(mm, dd) = "12/24"
mm: String = 12
dd: String = 24
```

[8] *scala.util.matching.Regex* wraps a *java.util.regex.Pattern*.

Exercise: Implicitly convert a *String* to a *Time*

- Motivation: There are many places where we need *Time*s. Wouldn't it be nice if we could just use *String*s like "9:45" instead? Example:

```
"13:15" - "9:45"
```

- Add an implicit conversion from *String* to *Time*
- Implement it using regular expressions
- Add tests to *TimeSpec* verifying that the implicit conversion is working correctly
- Also give it a try in the REPL

Implicit parameters

- If implicit conversions are a big hammer, implicit parameters are an executive briefcase: They let you pass easily
- Use the keyword *implicit* to define an implicit parameter list:

```
def pow(x: Int)(implicit y: Int) = math.pow(x, y)
```

- This only works for the last parameter list!

Implicit values

- Use the keyword *implicit* to define an implicit value:

```scala
implicit val defaultExponent = 2
```

- Arguments for implicit parameters can be omitted, if implicit values are in scope:

```scala
scala> pow(3)
<console>:9: error: could not find implicit value for
    parameter y: Int

scala> implicit val defaultExponent = 2
defaultExponent: Int = 2

scala> pow(3)
res1: Double = 9.0
```

- For looking up implicit values the same rules apply like for implicit conversions

Type classes and ad-hoc polymorphism

- The concept of type classes[9] allows for ad-hoc polymorphism: Functions (methods) can be applied to different types

```
def equal(a1: A, a2: A): Boolean
equal(1, 1) // true
equal("a", "b") // false
equal(1, "1") // Won't compile
```

- This is different from subtype polymorphism!

[9]Type classes were first intorduced in the Haskell programming language.

Parametrically polymorphic types

- A type class is a parametrically polymorphic type, declaring the generic functions:

```
trait Equal[A] {
  def equal(a1: A, a2: A): Boolean
}
```

- A type class instance defines the functions for a specific type argument:

```
new Equal[Int] {
  override def equal(a1: Int, a2: Int): Boolean =
    a1 == a2
}
```

Implicit parameters/values are the type class glue

- Use an implicit parameter for the type class:

```
object Equal {
  def equal[A](a1: A, a2: A)(implicit e: Equal[A]):
      Boolean =
    e.equal(a1, a2)
}
```

- Define implicit values for all supported type class instances:

```
object Equal {
  ...
  implicit val intEqual = new Equal[Int] { ...
  implicit val stringEqual = new Equal[String] { ...
}
```

OO flavored type classes

- We would like the polymorphic functions to be methods on the respective types:

```
1 === 1 // true
"a" === "b" // false
1 === "1" // Won't compile
```

- Use an implicit conversion to a polymorphic class that defines the polymorphic functions:

```
implicit def to_===[A](a: A)(implicit e: Equal[A]) =
  new ===(a)

class ===[A](a: A) {
  def ===(a2: A)(implicit e: Equal[A]): Boolean =
    e.equal(a, a2)
}
```

Group exercise: Implement type-safe equality

- Let's provide the polymorpic operators $===$ and $!==$ via type classes to any type
- Let's provide type class instances for *Int* and *String*:

```
1 === 1 // true
"a" !== "b" // true
1 === "1" // Won't compile
```

Exercise: Generalize XML serialization

- Currently XML serialization in ScalaTrain isn't generalized, but tied to only *Time*
- Further on the current state doesn't separate concerns
- Apply a type class based refactoring to gerneralize XML serialization:
  - Remove *toXml* from the singleton object *Time*
  - Remove *fromXml* from the class *Time*
- The tests should compile and run unmodified!

View bounds

- A view bound expresses a “can be viewed as” relation
- Use <% to define a view bound on a type parameter:

```scala
case class Cage[A <% Animal](a: A)
```

- Only types that can be “viewed” as the given bound, either by a “is a” relation or an implicit conversion are accepted:

```scala
scala> Cage("String")
<console>:11: error: No implicit view available from
    java.lang.String => Animal.

scala> implicit def toAnimal(s: String) = new Animal
stringToAnimal: (s: String)Animal

scala> Cage("String")
res0: Cage[java.lang.String] = Cage(String)
```

Exercise: Fully generalize the methods *Time.isIncreasing**

- There is no reason why these methods should be restricted to *Ordered* elements
- Make them work with any sequence of types that can be viewed as *Ordered*
- Add tests showing that they now work with *Int* and *String*

Question

Is *List* a type?

Digression: Values, types and type constructors

- Types (aka data types) abstract over values, e.g.:
  - *Int* can have the values *0*, *1*, etc.
  - *Boolean* can have the values *true* and *false*
- Type constructors abstract over types, e.g.:
  - *: kind of all data types (nullary type constructor)
  - * => *: kind of a unary type constructor, e.g. *List*
  - * => * => *: kind of a binary type constructor, e.g. *Tuple2*
- In other words:
  - Parameterized types aren't data types but type constructors
  - Only by applying all type arguments you get a data type
  - E.g.: Apply *Int* to *List[A]* returns the data type *List[Int]*

Context bounds

- A context bound expresses a “has a” relation to the given unary type constructor
- Use : to define a context bound on a type parameter:

```
def max[A : Ordering](a1: A, a2: A): A
```

- This is equivalent to using an implicit parameter:

```
def max[A](a1: A, a2: A)(implicit ev: Ordering[A]): A
```

implicitly

- Use the method *implicitly* to access the implicit argument of a context bound:

```scala
def max[A : Ordering](a1: A, a2: A) =
  implicitly[Ordering[A]].max(a1, a2)
```

- Usage example:

```
scala> max(1, 2)
res0: Int = 2

scala> max(Some(1), Some(2))
<console>:9: error: No implicit Ordering defined for
    Some[Int].
```

Tricking type erasure with *Manifests*

- Due to type erasure type arguments are lost at runtime, e.g.:

```
scala> def foo[A](bar: Any) = bar.isInstanceOf[A]
<console>:7: warning: abstract type A in type A is
    unchecked since it is eliminated by erasure
```

- Use a *Manifest* as context bound to save the type information:

```
def foo[A : Manifest](any: Any) =
  manifest[A].erasure isAssignableFrom any.getClass
```

- Usage example:

```
scala> foo[java.util.Date](new java.sql.Date(0))
res0: Boolean = true
```

Group exercise: Investigate the standard library

- Why can we use *isIncreasing* for a *Seq[Int]*?

```
scala> isIncreasing(Seq(1, 2, 3))
res0: Boolean = true
```

- Let's look at all the details!

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

Explicitly implicit

Internal DSLs

Contributing to the Scala collections

What's an internal DSL?

- A domain specific language (DSL) is "a programming language ... dedicated to a particular problem domain ... The concept isn't new — special-purpose programming languages ... have always existed ..."[10]
- An internal DSL is embedded in a host language: "Internal DSLs are particular ways of using a host language to give the host language the feel of a particular language."[11]

[10] Wikipedia, 2011-08-03

[11] Martin Fowler, martinfowler.com/bliki/DomainSpecificLanguage.html

Library versus internal DSL

- What's the difference between a library and an internal DSL?
- Technically-wise there is no difference!
- These DSL properties might help to differentiate:
  - Intutively understandable for domain experts
  - High-level and easy to use
  - Robust

Example for an internal DSL: ScalaModules[12]

High-level and intuitive DSL usage:

```
context findService withInterface[Foo] andApply { _.bar }
```

Low-level and not so easy to understand DSL usage:

```
ServiceReference reference =
    context.getServiceReference(Foo.class.getName());
if (reference != null) {
    try {
        Object service = context.getService(reference);
        Foo foo = (Foo) service;
        if (foo != null) System.out.println(foo.bar());
    } finally {
        context.ungetService(reference);
    }
}
```

[12] ScalaModules is a Scala DSL for OSGi; see www.scalamodules.org.

Typical building blocks for internal DSLs

- Currying
- By-name parameters
- Higher-order functions
- Advanced features of the type system
- “Dot-free” operator notation
- Implicit conversions

Custom control abstractions

- Currying is useful to define methods that look like built-in control constructs:

```scala
withOut(nullOut) { println("This won't be printed!") }
```

- This could be an appropriate implementation:

```scala
def withOut[A](out: PrintStream)(a: => A): A = {
  val formerOut = Console.out
  try {
    Console.setOut(out)
    a
  } finally {
    Console.setOut(formerOut)
  }
}
```

Group exercise: Automated resource management

- Let's write a custom control abstraction for automated resource management:
- Resources are "something that can be closed"
- The "new keyword" *withResource* shall close the given resource automatically
- Usage example:

```
scala> withResource(in) { res =>
  Source.fromInputStream(res).getLines.size
}
res0: Int = 13
```

Exercise: Create loops as custom control abstractions

- Create a *repeat-while* loop:

```
repeatWhile(x < 10) {
  println(x)
  x += 1
}
```

- Now gain some extra points by creating a *repeat-until* loop:

```
repeat {
  println(x)
  x += 1
} until(x >= 10)
```

Operator notation

- "Dot-free" operator notation is useful to define fluid interfaces and code that reads like natural language
- Infix operator notation:

```
1 :: 2 :: Nil
```

- Postfix operator notation:

```
1 :: 2 :: Nil size
```

Group exercise: A DSL for *Time*

- Let's write a DSL to create *Time* instances like this:

```
scala> 2 h 30 m
res0: org.scalatrain.Time = 02:30
```

- First try: Use only implicit conversions and operator notation
- Why is this not robust?
- How can we make it robust?

Exercise: A DSL for *Train*

- Create a DSL to create *Train* instances like this:

```
scala> Ice("722") startsAt "9:55" startsFrom "Munich"
    at "13:25" from "Frankfurt"
res0: org.scalatrain.Train =
    Train(Ice(722,false),List((09:55,Station(Munich)),
    (13:25,Station(Frankfurt))))
```

- Make it robust!

Agenda

Setting up the development environment

Bootstrapping

Functional programming in depth

Mastering the type system

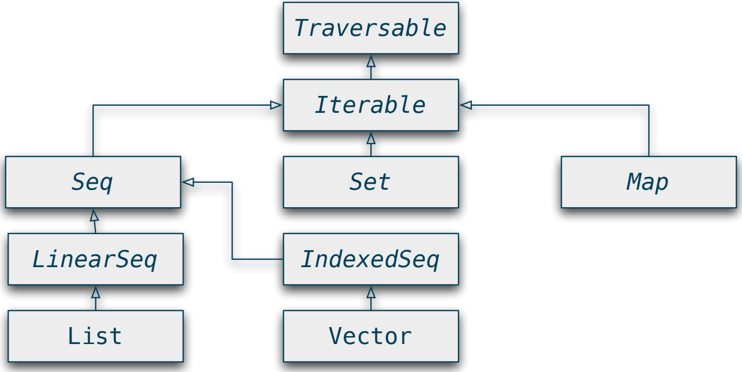
Explicitly implicit

Internal DSLs

Contributing to the Scala collections

Recap: Collection hierachy

- The Scala collections are very rich
- Some of the most important are show here:



- Lots of methods are already defined on *Traversable*

Uniform return type principle

- Common methods like *filter*, *take* and (in many cases) *map* preserve the collection type:

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

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

scala> Traversable(1, 2, 3) take 2
res2: Traversable[Int] = List(1, 2)
```

- How can this be achieved?

Uniform return type principle through duplication

- Duplication ensures the correct return type:

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

trait Iterable[A] extends Traversable[A] {
  override def filter(p: A => Boolean): Iterable[A] =
    ...
```

- Cumbersome approach: Often the implementations are identical, just the signature (the return type) is different
- Even worse, duplication could lead to bit rot: Inconsistencies, broken windows effect, etc.

Uniform return type principle through clever design

- Duplication was reality until Scala 2.7
- Scala 2.8 introduced a completely redesigned collection library based on these core principles:
  - Abstract over return types using builders
  - Abstract over return types using type classes

Abstract over return types using builders

- For many methods the return type can be abstracted out by introducing a builder that is specific for a given collection
- A builder is an implementation of the trait *Builder*:

```scala
trait Builder[-Elem, +To] {
  def +=(elem: Elem): this.type
  def clear()
  def result(): To
  ...
```

- A *Builder* is a mutable (package *scala.collection.mutable*) data structure, e.g. *ListBuffer*

*Builder*s in action

- For many collection types there are *Like*-traits parameterized with the element type and the collection type
- These *Like*-traits implement many methods in terms of the method *newBuilder*, e.g.:

```
trait TraversableLike[+A, +Repr] {
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    for (x <- this)
      if (p(x)) b += x
    b.result
  }
  def newBuilder: Builder[A, Repr]
  ...
```

Exercise: *Queues* like Scala *Seqs*

- Mix *SeqLike* into *Queue*
- Implement the abstract methods *apply*, *length* and *iterator* in terms of the wrapped *List* and return *this* for *seq*
- Delegate *newBuilder* to an identically named method in the companion object that returns a new *Builder* for *Queue*
- Now the following should be possible:

```
scala> Queue(1, 2, 3) filter { _ > 1 }
res0: misc.Queue[Int] = Queue(2, 3)
```

- But we are not there yet:

```
scala> Queue(1, 2, 3) map { _ + 1 }
res1: scala.collection.TraversableOnce[Int] =
    non-empty iterator
```

The collection type cannot always be preserved

- For some methods the collection type cannot always be preserved:

```
scala> BitSet(1, 2, 3) map { _ / 0.5 }
res0: ...Set[Double] = Set(2.0, 4.0, 6.0)

scala> "abc" map { _ + 1 }
res1: ...IndexedSeq[Int] = Vector(98, 99, 100)
```

- But sometimes it works:

```
scala> BitSet(1, 2, 3) map { _ * 2 }
res0: ...BitSet = BitSet(2, 4, 6)

scala> "abc" map { _ + 1 toChar }
res1: String = bcd
```

Abstract over return types using type classes

- The issue boils down to the following question:
- For a given collection type *From* and a given new element type *Elem*, what collection types *To* can be constructed?
- This can be encoded in the type class *CanBuildFrom*:

```
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(from: From): Builder[Elem, To]
  def apply(): Builder[Elem, To]
}
```

- *CanBuildFrom* is a *Builder* factory

CanBuildFrom in action

- Methods, that might not preserve the collection type, are implemented using implicit *CanBuildFrom* parameters, e.g.:

```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: A => B)(implicit bf:
      CanBuildFrom[Repr, B, That]): That = {
    val b = bf(repr)
    for (x <- this) b += f(x)
    b.result
  }
  ...
```

- Examples for type class instances of *CanBuildFrom*:

```
implicit def cbf1[A] =
  new CanBuildFrom[BitSet, B, Set[A]] { ... }
implicit def cbf2 =
  new CanBuildFrom[BitSet, Int, BitSet] { ... }
```

Exercise: *Queue* can be built from *Queue*

- Add a type class instance for *CanBuildFrom* to *Queue*
- Implement it such that for all *A* and *B* a *Queue[B]* can be built from a *Queue[A]*
- Now the following should also work:

```
scala> Queue(1, 2, 3) map { _ + 1 }
res0: misc.Queue[Int] = Queue(2, 3, 4)
```