

# New in Scala 2.13

© 2019 Hermann Hueck

<https://github.com/hermannhueck/new-in-scala213>

# Abstract

This presentation shows the feature updates from Scala 2.12 to 2.13. The list of features is not comprehensive, but it is my personal selection of favorites. I will focus on those which IMO impact/ease the programmers live most.

I will look at 5 feature areas: compiler, standard library, language changes, Future and finally collections which have been redesigned in 2.13.

# Agenda

1. Release Summary
2. Compiler
3. Standard Library
4. Concurrency / Future
5. Language Changes
6. Collections
7. Architecture of Collections
8. Migration
9. Resources

# 1. Release Summary

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.
- Future: is faster and more robust.

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.
- Future: is faster and more robust.
- Standard library: Elsewhere, useful classes and methods have been added.

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.
- Future: is faster and more robust.
- Standard library: Elsewhere, useful classes and methods have been added.
- Language: Literal types, partial unification, by-name implicits, more.

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.
- Future: is faster and more robust.
- Standard library: Elsewhere, useful classes and methods have been added.
- Language: Literal types, partial unification, by-name implicits, more.
- Compiler: 5-10% faster, deterministic output, improved optimizer.

# Release Summary

Release 2.13 improves Scala in the following areas:

- Collections: Standard library collections have been overhauled for simplicity, performance, and safety. This is the centerpiece of the release.
- Future: is faster and more robust.
- Standard library: Elsewhere, useful classes and methods have been added.
- Language: Literal types, partial unification, by-name implicits, more.
- Compiler: 5-10% faster, deterministic output, improved optimizer.

From the Scala release description:

<https://github.com/scala/scala/releases/tag/v2.13.0>

# 2. Compiler

# Compiler

- Performance improved
- Some improvements already flew back to 2.12.8. More to come in 2.12.9.
- Deterministic, reproducible compilation
- Optimizer improvements (collections, arrays, inlining)

# 3. Standard Library

# Smaller Footprint

No longer included:

- *scala-parallel-collections*
- *scala-xml*
- *scala-parser-combinators*
- *scala-swing*

# Integrated Java Interop

- The old `scala-java8-compat` module is now part of the standard library.
- This provides converters for options, function types and Java streams.
- `scala.collection.JavaConversions` removed (already deprecated in 2.12).

# Chaining: *pipe* and *tap*

```
import scala.util.chaining._

val x: Int = 5 tap println

val y: Int = 5 pipe (_ * x) tap println

List(1, 2, 3) tap (ys => println("debug: " + ys.toString))

val times6 = (_: Int) * 6
(1 - 2 - 3) pipe times6 pipe scala.math.abs tap println
```

# Chaining: *pipe* and *tap*

```
import scala.util.chaining._

val x: Int = 5 tap println

val y: Int = 5 pipe (_ * x) tap println

List(1, 2, 3) tap (ys => println("debug: " + ys.toString))

val times6 = (_: Int) * 6
(1 - 2 - 3) pipe times6 pipe scala.math.abs tap println
```

$x \text{ pipe } f$  is a replacement for  $f(x)$ , where  $f$  is a *Function1*[ $A \Rightarrow B$ ].

$x \text{ tap } f$  is a replacement for  $x \Rightarrow \{ f(x); x \}$ , where  $f$  is a side-effecting *Function1*[ $A \Rightarrow \text{Unit}$ ]. *tap* performs the side effect in  $f(x)$  and returns  $x$  unchanged.

# Backport to 2.12: *pipe* and *tap*

```
package compat213

package object chaining {

    implicit class ChainingOps[A](private val a: A) {
        def pipe[B](f: A => B): B = f(a)
        def |>[B](f: A => B): B = a pipe f
        def tap[B](f: A => Unit): A = a pipe ( x => { f(x); x } )
    }
}
```

# Backport to 2.12: *pipe* and *tap*

```
package compat213

package object chaining {

    implicit class ChainingOps[A](private val a: A) {
        def pipe[B](f: A => B): B = f(a)
        def |>[B](f: A => B): B = a pipe f
        def tap[B](f: A => Unit): A = a pipe ( x => { f(x); x } )
    }
}
```

```
import compat213.chaining._

val x: Int = 5 tap println

val y: Int = 5 pipe (_ * x) tap println

List(1, 2, 3) tap (ys => println("debug: " + ys.toString))

val times6 = (_: Int) * 6
(1 - 2 - 3) pipe times6 pipe scala.math.abs tap println
```

# Either: *Right.withLeft* and *Left.withRight*

- The constructor *Right* leaves the left type unspecified.
- The constructor *Left* leaves the right type unspecified.

```
sbt:New in Scala 2.13> ++2.13.0 console
...
[info] Starting scala interpreter...
Welcome to Scala 2.13.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_212).
Type in expressions for evaluation. Or try :help.

scala> Right(5)
res0: scala.util.Right[Nothing,Int] = Right(5)

scala> Right(5).withLeft[String]
res1: scala.util.Either[String,Int] = Right(5)

scala> Left("some error")
res2: scala.util.Left[String,Nothing] = Left(some error)

scala> Left("some error").withRight[Int]
res3: scala.util.Either[String,Int] = Left(some error)
```

# Backport to 2.12: *Right.withLeft* and *Left.withRight*

```
package compat213

package object either {

    implicit class RightOps[L, R](private val right: Right[L, R]) {
        def withLeft[LL](implicit ev: L <:< LL): Either[LL, R] =
            right.asInstanceOf[Either[LL, R]]
    }

    implicit class LeftOps[L, R](private val left: Left[L, R]) {
        def withRight[RR](implicit ev: R <:< RR): Either[L, RR] =
            left.asInstanceOf[Either[L, RR]]
    }
}
```

## *Either.flatten*

- available since 2.13
- in 2.12 you can flatten a nested *Either* with *flatMap*.

# *Either.flatten*

- available since 2.13
- in 2.12 you can flatten a nested *Either* with *flatMap*.

## Scala 2.12

```
val nestedEither = Right(Right(5))
nestedEither.flatMap(x => x)
```

## Scala 2.13

```
val nestedEither = Right(Right(5))
nestedEither.flatten
```

# String Operations: *toIntOption* etc.

- convert *String* literals to *Int*, *Double*, *Boolean* without throwing exceptions
- return *Some(value)* if the conversion succeeds, *None* if it fails.

# String Operations: *toIntOption* etc.

- convert *String* literals to *Int*, *Double*, *Boolean* without throwing exceptions
- return *Some(value)* if the conversion succeeds, *None* if it fails.

```
sbt:New in Scala 2.13> ++2.13.0 console
...
Welcome to Scala 2.13.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_212).

scala> "42".toIntOption
res0: Option[Int] = Some(42)

scala> "42.0".toIntOption
res1: Option[Int] = None

scala> "42.0".toDoubleOption
res2: Option[Double] = Some(42.0)

scala> "true".toBooleanOption
res3: Option[Boolean] = Some(true)
```

# Backport to 2.12: String Operations

```
package compat213

import scala.util.Try

package object string {

    implicit class StringOps(private val s: String) {
        def toIntOption: Option[Int] = Try(s.toInt).toOption
        def toDoubleOption: Option[Double] = Try(s.toDouble).toOption
        def toBooleanOption: Option[Boolean] = Try(s.toBoolean).toOption
    }
}
```

# *scala.util.Using.apply* for resource management

Similar to *try ... catch ... finally*,  
but guarantees to release/close the used resource.

# *scala.util.Using*.apply for resource management

Similar to *try ... catch ... finally*,  
but guarantees to release/close the used resource.

```
def bufferedReader(fileName: String): BufferedReader =
  new BufferedReader(new FileReader(fileName))

def readLines(reader: BufferedReader): Seq[String] =
  ???

def tryLines(fileName: String): Try[Seq[String]] =
  Using(bufferedReader(fileName)) { reader => readLines(reader) }

def catFile(fileName: String): Unit =
  tryLines(fileName) match {
    case Failure(exception) => exception.toString tap println
    case Success(lines) => lines foreach println
  }

catFile("README.md")
```

## *scala.util.Using.resource*

returns an  $A$ , not a  $\text{Try}[A]$

# *scala.util.Using.resource*

returns an *A*, not a *Try[A]*

```
package scala.util
object Using { // simplified
  def apply[R, A](resource: => R)(f: R => A): Try[A]
  def resource[R, A](resource: => R)(f: R => A): A
  ...
}
```

# *scala.util.Using.resource*

returns an *A*, not a *Try[A]*

```
package scala.util
object Using { // simplified
  def apply[R, A](resource: => R)(f: R => A): Try[A]
  def resource[R, A](resource: => R)(f: R => A): A
  ...
}
```

```
def bufferedReader(fileName: String): BufferedReader = ???

def readLines(reader: BufferedReader): Seq[String] = ???

def lines(fileName: String): Seq[String] =
  Using.resource(bufferedReader(fileName))(readLines)

def catFile2(fileName: String): Unit = { // might throw an exception
  lines(fileName) foreach println
}

catFile2("README.md")
```

# Backport to 2.12: *Using*

```
package compat213

import scala.util.Try
import scala.language.reflectiveCalls

object Using {

  def apply[A, R <: {def close(): Unit}](resrc: R)(use: R => A): Try[A] =
    Try(resource(resrc)(use))

  def resource[A, R <: {def close(): Unit}](resrc: R)(use: R => A): A =
    try {
      use(resrc)
    } finally {
      resrc.close()
    }
}
```

# Backport to 2.12: *Using*

```
package compat213

import scala.util.Try
import scala.language.reflectiveCalls

object Using {

  def apply[A, R <: {def close(): Unit}](resrc: R)(use: R => A): Try[A] =
    Try(resource(resrc)(use))

  def resource[A, R <: {def close(): Unit}](resrc: R)(use: R => A): A =
    try {
      use(resrc)
    } finally {
      resrc.close()
    }
}
```

This impl is a bit simplistic, but should work for all resources which provide a method *close*.

# s-Interpolator in Pattern matches

```
val dateString = "11-June-2019"  
  
val s"$day-$month-$year" = dateString  
  
year tap println  
month tap println  
day tap println
```

# Named Product Elements

Case classes and other *Products* (Tuples) now have methods *productElementNames* and *productElementName*.

# Named Product Elements

Case classes and other *Products* (Tuples) now have methods *productElementNames* and *productElementName*.

```
sealed trait Gender extends Product with Serializable
case object Male extends Gender
case object Female extends Gender

case class Person(name: String, age: Int, gender: Gender, email: String) {
  def tupled: (String, Int, Gender, String) = (name, age, gender, email)
}

val johndoe = Person("John Doe", 42, Male, "john@doe.com")
```

# Named Product Elements

Case classes and other *Products* (Tuples) now have methods *productElementNames* and *productElementName*.

```
sealed trait Gender extends Product with Serializable
case object Male extends Gender
case object Female extends Gender

case class Person(name: String, age: Int, gender: Gender, email: String) {
  def tupled: (String, Int, Gender, String) = (name, age, gender, email)
}

val johndoe = Person("John Doe", 42, Male, "john@doe.com")
```

```
johndoe.productElementNames foreach println
johndoe.productElementName(0) tap (name => print(s"$name: "))
johndoe.productElement(0) tap println
johndoe.productElementName(1) tap (name => print(s"$name: "))
johndoe.productElement(1) tap println
```

# Named Product Elements

## Naive JSON Serialization

```
def pairToJson(name: String, value: Any): String =
  s"""{ "$name": $value }"""

def productElementToJson(p: Product, index: Int): String =
  pairToJson(p.productElementName(index), p.productElement(index))

def productToJson(product: Product): String =
  (0 until product.productArity)
    .toList
    .map { index => productElementToJson(product, index) }
    .mkString("{ , , , , }")

implicit class ProductOps(private val product: Product) {
  def toJsonString: String = productToJson(product)
}

johndoe.toJsonString tap println
// { { "name": John Doe }, { "age": 42 }, { "gender": Male }, { "email": john@doe.
johndoe.tupled.toJsonString tap println
// { { "_1": John Doe }, { "_2": 42 }, { "_3": Male }, { "_4": john@doe.com } }
```

# 4. Concurrency / Future

# *Future + ExecutionContext* Changes Overview

- API nearly unchanged
- Massive performance improvements under the hood (*Future*, *Promise*, *ExecutionContext*)
- Improved handling of failures (*InterruptedException*, *RejectedExecutionException*)
- Made the global *ExecutionContext* “batched”
- Added synchronous (“parasitic”) *ExecutionContext* (releases you from writing your own synchronous *ExecutionContext*)

For more details on the internals of the improved implementation see Viktor Klang's talk at Scala Days 2019: Making Our Future Better

<https://www.youtube.com/watch?v=5FTJUUoT6y4>

# *Future*: Minor API Changes

- *removed* onSuccess *and* onFailure\* (already deprecated in 2.12)
- *Future.delegate* - new factory method

# *Future.delegate*

```
object Future {  
    def apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]  
    def delegate[T](body: => Future[T])(implicit executor: ExecutionContext): Future  
    ...  
}
```

# *Future.delegate*

```
object Future {  
    def apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]  
    def delegate[T](body: => Future[T])(implicit executor: ExecutionContext): Future  
    ...  
}
```

The following expressions are semantically equivalent:

```
def expr[T]: Future[T] = ???  
  
val f1 = Future.delegate(expr)  
val f2 = Future.apply(expr).flatten  
val f3 = Future.unit.flatMap(_ => expr)
```

# *Future.delegate*- Example

```
import scala.concurrent._  
import scala.concurrent.duration._  
import scala.util.chaining._  
  
implicit lazy val ec: ExecutionContext = ExecutionContext.global  
  
def plus17(x: Int): Int = x + 17  
  
def squaredAsync(value: Int) = Future { value * value }  
  
val f1: Future[Int] = Future.apply { squaredAsync(5) }.flatten map plus17  
val f2: Future[Int] = Future.unit.flatMap { _ => squaredAsync(5) } map plus17  
val f3: Future[Int] = Future.delegate { squaredAsync(5) } map plus17  
  
Await.result(f1, 3.seconds) tap println //=> 42  
Await.result(f2, 3.seconds) tap println //=> 42  
Await.result(f3, 3.seconds) tap println //=> 42
```

# 5. Language Changes

# Language Changes Overview

- Literal types: Literals (for strings, integers etc.) now have associated literal types.
- Partial unification: enabled by default
- By-name implicit parameters: enable implicit search to construct recursive values.
- Underscores in numeric literals
- Procedure syntax deprecated:  
Deprecated: `def m() { ... }`      Use instead: `def m(): Unit = { ... }`
- View bounds deprecated:  
Deprecated: `A <% B`                  Use instead: `(implicit ev: A => B)`
- Symbol literals deprecated:  
Deprecated: `'foo`                          Use instead: `Symbol("foo")`

# Underscores in Number Literals

```
val int0: Int = 1000000
val int1: Int = 1_000_000
val int2: Int = 1_0_0_0_0_0_0

// val int3: Int = 1_0_0_0_0_0_0_
// compile error: trailing separator is not allowed

val long: Long = 1_000_000_000L
val float: Float = 1_000.99f
val double: Double = 1_000_000.999_999
```

# Partial unification

- Partial unification is on by default in 2.13.
- The compiler no longer accepts *-Ypartial-unification*.
- The following code compiles in 2.12 only with *-Ypartial-unification*.

# Partial unification

- Partial unification is on by default in 2.13.
- The compiler no longer accepts *-Ypartial-unification*.
- The following code compiles in 2.12 only with *-Ypartial-unification*.

```
import scala.language.higherKinds

def foo[F[_], A](fa: F[A]): String =
  fa.toString

val either: Either[String, Int] = Right(42).withLeft[String]
foo { either }

val intToInt: Function1[Int, Int] = x => x * 2
foo { intToInt }
```

# Partial unification

- Partial unification is on by default in 2.13.
- The compiler no longer accepts *-Ypartial-unification*.
- The following code compiles in 2.12 only with *-Ypartial-unification*.

```
import scala.language.higherKinds

def foo[F[_], A](fa: F[A]): String =
  fa.toString

val either: Either[String, Int] = Right(42).withLeft[String]
foo { either }

val intToInt: Function1[Int, Int] = x => x * 2
foo { intToInt }
```

Detailed explanation of partial unification here:

<https://gist.github.com/djspiewak/7a81a395c461fd3a09a6941d4cd040f2>

# Literal Types

- Literals (for strings, integers etc.) now have associated literal types.
- The compiler will provide instances of a new typeclass `scala.ValueOf[T]` for all singleton types  $T$ .
- The value of a singleton type can be accessed by calling method `valueOf[T]`.

# Literal Types

- Literals (for strings, integers etc.) now have associated literal types.
- The compiler will provide instances of a new typeclass *scala.ValueOf[T]* for all singleton types *T*.
- The value of a singleton type can be accessed by calling method *valueOf[T]*.

```
val wahr: true = true
val foo: "foo" = "foo"
val one: 1 = 1
val other_one: one.type = one
implicitly[other_one.type =:= 1]
val x1: Int = valueOf[42] // valueOf[42] yields an Int and is the same as ...
val x2: Int = new scala.ValueOf(42).value
```

# By-name Implicit Parameters

- were not allowed in 2.12.
- They enable implicit search to construct recursive values.
- The following code will not compile  
if you remove the `=>` in `(implicit rec: => Foo)` .

# By-name Implicit Parameters

- were not allowed in 2.12.
- They enable implicit search to construct recursive values.
- The following code will not compile  
if you remove the `=>` in `(implicit rec: => Foo)` .

```
trait Foo {
  def next: Foo
}

object Foo {
  // wouldn't compile, if rec were a call by value parameter
  // remove the => and try to compile ...
  implicit def foo(implicit rec: => Foo): Foo =
    new Foo { def next = rec }
}

val foo = implicitly[foo]
assert(foo eq foo.next)
```

# 6. Collections

# Principles of the Collections Redesign

- simplicity
  - better error messages
  - easier to implement your own collection (but still complex)
- performance
- type safety, better type inference
- smaller footprint: parallel collections moved to a module of its own, etc.
- source code compatibility - as much as possible  
Most ordinary code that used the old collections will continue to work as-is. But of course ... there are breaking changes.

# Simpler Method Signatures

- No more *CanBuildFrom*
- Without *CanBuildFrom* method signatures became much simpler.

# Simpler Method Signatures

- No more *CanBuildFrom*
- Without *CanBuildFrom* method signatures became much simpler.

## *List#map* in 2.12

```
trait List[+A] extends ... {  
    def map[B, That](f: A => B)(implicit bf: CanBuildFrom[List[A], B, That]): That =  
}
```

## *List#map* in 2.13

```
trait List[+A] extends ... {  
    def map[B](f: A => B): List[B] = ???  
}
```

# Removed *collection.breakOut*

- *collection.breakOut* in 2.12 inferred the return type of a collection operation from the expected result type.
- It was based on *CanBuildFrom* which is gone in 2.13.
- To avoid constructing intermediate collections, use *.view* and *.to(Collection)* instead.

# Removed *collection.breakOut*

- *collection.breakOut* in 2.12 inferred the return type of a collection operation from the expected result type.
- It was based on *CanBuildFrom* which is gone in 2.13.
- To avoid constructing intermediate collections, use *.view* and *.to(Collection)* instead.

```
val list = List(1, 2, 3) tap println  
val toPair: Int => (Int, Int) = x => x -> x
```

```
// Scala 2.12 - type annotations required to infer the result type of list.map  
val indexedSeq          = list.map(toPair)(collection.breakOut)  
val array    : Array[(Int, Int)] = list.map(toPair)(collection.breakOut)  
val seq      : Seq[(Int, Int)]  = list.map(toPair)(collection.breakOut)  
val set       : Set[(Int, Int)] = list.map(toPair)(collection.breakOut)  
val map      : Map[Int, Int]   = list.map(toPair)(collection.breakOut)
```

```
// Scala 2.13 - type annotations not required  
val list2           = list.iterator.map(toPair)  
val array          = list.iterator.map(toPair).to(Array)  
val seq            = list.iterator.map(toPair).to(Seq)  
val set             = list.iterator.map(toPair).to(Set)
```

# Simpler Type Hierarchy

- No more *Traversable* and *TraversableOnce*.
  - They remain only as deprecated aliases for *Iterable* and *IterableOnce*.
- Parallel collections are now a separate module.
  - As a result, *GenSeq*, *GenTraversableOnce*, et al. are gone.

# New, Faster HashMap/Set Implementations

- Both immutable and mutable versions were completely replaced.
- They substantially outperform the old implementations in most scenarios.
- The mutable versions now perform on par with the Java standard library's implementations.

# Immutable *scala.Seq* and *scala.IndexedSeq*

- *Seq* is now an alias for *collection.immutable.Seq*.
  - Before, it was an alias for the possibly-mutable *collection.Seq*.
- *IndexedSeq* is now an alias for *collection.immutable.IndexedSeq*.
  - Before, it was an alias for the possibly-mutable *collection.IndexedSeq*.
- This also changes the type of varargs in methods and pattern matches.
- Arrays passed as varargs are defensively copied.

# *Seq* is immutable in 2.13 (not in 2.12)

```
trait Order
trait Food

def orderFood(order: Seq[Order]): Seq[Food] = {
    Seq(new Food{})
}
```

# *Seq* is immutable in 2.13 (not in 2.12)

```
trait Order
trait Food

def orderFood(order: Seq[Order]): Seq[Food] = {
    Seq(new Food{})
}
```

## Passing a mutable *ArrayBuffer*...

```
// We can NOT pass a mutable ArrayBuffer where an immutable Seq is expected.
val food1 = orderFood(ArrayBuffer(new Order{})) // DOES NOT COMPILE!
// [error] found  : scala.collection.mutable.ArrayBuffer[Order]
// [error] required: Seq[Order]
```

## Passing a mutable *Array* ...

We can pass a mutable *Array* where an immutable *Seq* is expected.  
*Array* (unlike *ArrayBuffer*) is implicitly converted (and copied).  
But the compiler spits out a warning.

```
val orderArray = Array(new Order {})

val food2 = orderFood(orderArray)      // COMPILES!
// [warn] Implicit conversions from Array to immutable.IndexedSeq
// [warn] are implemented by copying; Use the more efficient non-copying
// [warn] ArraySeq.unsafeWrapArray or an explicit toIndexedSeq call.
```

## Passing a mutable *Array* ...

We can pass a mutable *Array* where an immutable *Seq* is expected.  
*Array* (unlike *ArrayBuffer*) is implicitly converted (and copied).  
But the compiler spits out a warning.

```
val orderArray = Array(new Order {})

val food2 = orderFood(orderArray)      // COMPILES!
// [warn] Implicit conversions from Array to immutable.IndexedSeq
// [warn] are implemented by copying; Use the more efficient non-copying
// [warn] ArraySeq.unsafeWrapArray or an explicit toIndexedSeq call.
```

*toSeq* (or *toIndexedSeq*) wraps the mutable *Array* in an immutable *Seq*.

```
val food3 = orderFood(orderArray.toSeq)      // COMPILES!
val food4 = orderFood(orderArray.toIndexedSeq) // COMPILES!
```

## Passing a immutable *ArraySeq*...

- *ArraySeq* is a new collection of Scala 2.13.
- *ArraySeq* is an immutable array with efficient indexed access and a small memory footprint.
- A *mutable.ArraySeq* is also available in the new collections library.
- For Scala 2.12 an *ArraySeq* backport is provided in the *scala-collection-compat* library.
- *ArraySeq.unsafeWrapArray* wraps an *Array* in an *ArraySeq*.

```
val food5 = orderFood(ArraySeq(new Order{}))
val food6 = orderFood(ArraySeq.unsafeWrapArray(Array(new Order{})))
```

# Cross-compiling *Seq* for 2.12 and 2.13

## Seq Recap

- *scala.collection.Seq* is a base class for *scala.collection.immutable.Seq* and *scala.collection.mutable.Seq* in Scala 2.12 and 2.13.
- *scala.Seq* is an alias for *scala.collection.Seq* in Scala 2.12.
- *scala.Seq* is an alias for *scala.collection.immutable.Seq* in Scala 2.13.

# Cross-compiling *Seq* for 2.12 and 2.13

## Seq Recap

- *scala.collection.Seq* is a base class for *scala.collection.immutable.Seq* and *scala.collection.mutable.Seq* in Scala 2.12 and 2.13.
- *scala.Seq* is an alias for *scala.collection.Seq* in Scala 2.12.
- *scala.Seq* is an alias for *scala.collection.immutable.Seq* in Scala 2.13.

To make your 2.12 code cross-compilable for 2.12 and 2.13 you have

3 Options described below ...

# Cross-compiling *Seq* (1st option)

Explicitly use *scala.collection.Seq* in method parameters and return types.

```
import scala.collection

def orderFood(order: collection.Seq[Order]): collection.Seq[Food] = ???
```

# Cross-compiling *Seq* (1st option)

Explicitly use *scala.collection.Seq* in method parameters and return types.

```
import scala.collection

def orderFood(order: collection.Seq[Order]): collection.Seq[Food] = ???
```

- You don't force your code into immutable semantics.
- *orderFood* accepts mutable and immutable *Seqs*.
- mutability / immutability is unspecified for the return type.
- Caller must call *.toSeq* if only she needs an immutable result. (*.toSeq* only copies elements if the result is not yet immutable.)

Simplest migration strategy!

No changes at the call site!

# Cross-compiling *Seq* (2nd option)

Explicitly use *scala.collection.Seq* in parameters and *scala.collection.immutable.Seq* in return types.

```
import scala.collection
import scala.collection.immutable

def orderFood(order: collection.Seq[Order]): immutable.Seq[Food] = ???
```

# Cross-compiling *Seq* (2nd option)

Explicitly use *scala.collection.Seq* in parameters and *scala.collection.immutable.Seq* in return types.

```
import scala.collection
import scala.collection.immutable

def orderFood(order: collection.Seq[Order]): immutable.Seq[Food] = ???
```

- You force your code into immutable semantics only for return types.
- *orderFood* accepts mutable and immutable *Seqs*.
- immutability is fixed for the return type.

Still simple migration strategy!

Mostly no changes at the call site!

# Cross-compiling *Seq* (3rd option)

Use *scala.immutable.collection.Seq* in method parameters and return types.

```
import scala.collection.immutable  
  
def orderFood(order: immutable.Seq[Order]): immutable.Seq[Food] = ???
```

# Cross-compiling *Seq* (3rd option)

Use *scala.immutable.collection.Seq* in method parameters and return types.

```
import scala.collection.immutable  
  
def orderFood(order: immutable.Seq[Order]): immutable.Seq[Food] = ???
```

- You force your code into immutable semantics.
- *orderFood* accepts only immutable *Seqs*.
- immutability is also fixed for the return type.

Possibly many changes at the call site to make the arguments immutable!

Use Scalafix to automate this rewrite for a large code base.

# Simplified Views that Work

- Views have been vastly simplified and should now work reliably.
- `scala.collection.View` has two sub classes: `scala.collection.SeqView` and `scala.collectionMapView`
- Views are lazy. They record the operations (like `filter`, `map` etc.) and do not execute them before invoking a terminal operation (`foreach`, `toSeq`, `toMap` etc.).

# *Map#mapValues* and *Map#filterKeys*

- *Map#mapValues* and *Map#filterKeys* in 2.13 return *MapView*, not *Map*.
- These methods are also deprecated.
- Prefer using *MapView#mapValues* and *MapView#filterKeys*

# *Map#mapValues* and *Map#filterKeys*

- *Map#mapValues* and *Map#filterKeys* in 2.13 return *MapView*, not *Map*.
- These methods are also deprecated.
- Prefer using *MapView#mapValues* and *MapView#filterKeys*

```
val kvs = Map("one" -> 1, "two" -> 2, "three" -> 3)
val kvsFlipped: Map[Int, String] = kvs.toList.map { case (fst, snd) => (snd, fst)}
```

## Scala 2.12

```
val mappedValues: Map[String, Int] =
  kvs.mapValues(_ + 10)

val keysFiltered: Map[Int, String] =
  kvsFlipped.filterKeys(_ % 2 != 0)
```

## Scala 2.13

```
val mapView: MapView[String, Int] =
  kvs.view.mapValues(_ + 10)
val mappedValues: Map[String, Int] =
  mapView.toMap

val mapView2: MapView[Int, String] =
  kvsFlipped.view.filterKeys(_ % 2 != 0)
val keysFiltered: Map[Int, String] =
  mapView2.toMap
```

# *LazyList* replaces *Stream*

- *Stream* is lazy in it's tail, but eager in it's head.
- *LazyList* is lazy in it's head and tail.
- *Stream* is deprecated in 2.13.

Scala 2.12

```
val stream: Stream[(Int, Int)] =  
  Stream.continually(42)  
    .take(10)  
    .zipWithIndex  
    .map { case (value, index) =>  
      (index, value)  
    }
```

Scala 2.13

```
val ll: LazyList[(Int, Int)] =  
  LazyList.continually(42)  
    .take(10)  
    .zipWithIndex  
    .map { case (value, index) =>  
      (index, value)  
    }
```

# New Abstract and Concrete Collections

- *immutable.LazyList* replaces *immutable.Stream*.
- *immutable.ArraySeq* is an immutable wrapper for an array; there is also a mutable version.
- *mutable.CollisionProofHashMap* guards against denial-of-service attacks.
- *mutable.ArrayDeque* is a double-ended queue that internally uses a resizable circular buffer.
- *mutable.Stack* was reimplemented (and undeprecated), *immutable.Stack* was removed.
- *immutable.SeqMap* (abstract) provides immutable maps which maintain insertion order.
- Implementations: *VectorMap* and *TreeSeqMap* (in addition to the already existing *ListMap*)

# *Coll#to* converts one collection to another one.

- *Coll#to* in 2.12 received the target type in square brackets.
- *Coll#to* in 2.13 receives the target type's companion in parens.
- The *scala-collection-compat* library provides the new behaviour in 2.12.

```
val map = Map("one" -> 1, "two" -> 2, "three" -> 3)
```

## Scala 2.12

```
val l1 = map.toList
val l2 = map.to[List]

import scala.collection.compat._
val l3 = map.to(List)
```

## Scala 2.13

```
val l1 = map.toList
val l2 = map.to(List)

//
```

## *Added `.lengthIs` / `.sizeIs` and `.sizeCompare`*

- Allow fluent size comparisons without traversing the whole collection.

# Added *.lengthIs* / *.sizeIs* and *.sizeCompare*

- Allow fluent size comparisons without traversing the whole collection.

```
val xs = List.fill(5000)(scala.util.Random.nextInt)

// lengthIs or sizeIs traverse no more than 101 element
if (xs.lengthIs > 100) {
    new IllegalArgumentException("Too many elements!") tap println
} else {
    s"The list has ${xs.length} elements." tap println
}
```

# New *.tapEach* method for side-effects

- Allows inserting side-effects in a chain of method calls on a collection or view.

# New *.tapEach* method for side-effects

- Allows inserting side-effects in a chain of method calls on a collection or view.

```
val doubledAndSquared =  
  List(1, 2, 3)  
    .tapEach(x => println(s"value: $x"))  
    .map(x => x * 2)  
    .tapEach(x => println(s"doubled: $x"))  
    .map(x => x * x)  
    .tapEach(x => println(s"squared: $x"))
```

# New `.unfold` method

- This allows constructing a collection or iterator from an initial element and a repeated *Option*-returning operation, terminating on *None*.
- This was added to collection companion objects and to *Iterator*.

# New *.unfold* method

- This allows constructing a collection or iterator from an initial element and a repeated *Option*-returning operation, terminating on *None*.
- This was added to collection companion objects and to *Iterator*.

```
def bufferedReader(fileName: String) =  
  new BufferedReader(new FileReader(fileName))  
  
def readLines(reader: BufferedReader) =  
  Iterator.unfold(())(_ => Option(reader.readLine()).map(_ -> ())).toList  
  
def readLines_dissected(reader: BufferedReader): List[String] = {  
  val initialState: Unit = ()  
  val iterator: Iterator[String] = Iterator.unfold(initialState) { _ =>  
    val maybeLine: Option[String] = Option(reader.readLine())  
    val maybeLineState: Option[(String, Unit)] = maybeLine.map(_ -> ())  
    maybeLineState  
  }  
  iterator.toList  
}  
  
val lines: Seq[String] =  
  Using.resource(bufferedReader("README.md"))(readLines)  
  
lines foreach println
```

# Two overloaded *Map#map* operations! Why ???

```
def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]
```

```
def map[B](f: ((K, V)) => B): Iterable[B]
```

# Two overloaded $\text{Map} \# \text{map}$ operations! Why ???

```
def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]
```

```
def map[B](f: ((K, V)) => B): Iterable[B]
```

- If the mapping function  $f$  transforms a key value pair  $(K, V)$  into another key value pair  $(K2, V2)$ ,  $\text{map}$  returns a  $\text{Map}[K2, V2]$ .
- OTOH if the mapping function  $f$  transforms a key value pair  $(K, V)$  into some other value  $B$ ,  $\text{map}$  returns a  $\text{Iterable}[B]$ .
- Roughly the same holds for the two  $\text{Map} \# \text{flatMap}$  operations.
- To understand this we have to take a glimpse into the collections' architecture.

# 7. Architecture of Collections

See also:

<https://docs.scala-lang.org/overviews/core/architecture-of-scala-213-collections.html>

# Problem to solve

Define the return type of a collection operation in a generic way?

- This is not a problem for operations, which do not return a collection, but a single value like *isEmpty*, *length*, *find*, *foldLeft*, *sum*, *exists*, *forall* etc.
- This is difficult for operations that return a collection like *filter*, *take*, *drop*, *map*, *flatMap*, *flatten* etc.

# Problem to solve

Define the return type of a collection operation in a generic way?

- This is not a problem for operations, which do not return a collection, but a single value like *isEmpty*, *length*, *find*, *foldLeft*, *sum*, *forall* etc.
- This is difficult for operations that return a collection like *filter*, *take*, *drop*, *map*, *flatMap*, *flatten* etc.

A Non-solution with simple inheritance

```
trait Iterable[A] {  
    def filter(f: A => Boolean): Iterable[A]  
    def map[B](f: A => B): Iterable[B]  
}
```

```
trait List[A] extends Iterable[A] { ... }  
trait Vector[A] extends Iterable[A] { ... }
```

The inherited methods would return an *Iterable*, not a *List* or *Vector*.

# What we need ...

```
trait List[A] extends MagicBaseTrait[???, ???, ???] {  
    def filter(f: A => Boolean): List[A]  
    def map[B](f: A => B): List[B]  
}  
  
trait Vector[A] extends MagicBaseTrait[???, ???, ???] {  
    def filter(f: A => Boolean): Vector[A]  
    def map[B](f: A => B): Vector[B]  
}  
  
trait Map[K, V] extends MagicBaseTrait[???, ???, ???] {  
    def filter(f: (K, V) => Boolean): List[A]  
    def map[K2, V2](f: (K, V) => (K2, V2)): Map[K2, V2]  
    def map[B](f: (K, V) => B): Iterable[B]  
}
```

... without being forced to reimplement the operations in every collection.

# What we need ...

```
trait List[A] extends MagicBaseTrait[???, ???, ???] {
  def filter(f: A => Boolean): List[A]
  def map[B](f: A => B): List[B]
}

trait Vector[A] extends MagicBaseTrait[???, ???, ???] {
  def filter(f: A => Boolean): Vector[A]
  def map[B](f: A => B): Vector[B]
}

trait Map[K, V] extends MagicBaseTrait[???, ???, ???] {
  def filter(f: (K, V) => Boolean): List[A]
  def map[K2, V2](f: (K, V) => (K2, V2)): Map[K2, V2]
  def map[B](f: (K, V) => B): Iterable[B]
}
```

... without being forced to reimplement the operations in every collection.

In the old collection implementation (up to 2.12) this problem has been solved with *CanBuildFrom*.

# Selections and Transformations

- Selection operations (*filter*, *take*, *drop* etc.) do not change the elements.
  - The target type of the operation is exactly the same as the source type, e.g. *filter* on a *List[A]* returns a *List[A]*.
  - We must abstract over the source collection type (*List[A]*) in this case to generalize this.
- Transformation operations (*map*, *flatMap* etc.) do change the elements and in some cases (*Map* and some others) also the collection type.
  - The target type of the operation must be derived from the type constructor of the required result collection type.

# *IterableOps*

```
trait IterableOps[+A, +CC[_], +C] {  
    def filter(p: A => Boolean): C = ???  
    def map[B](f: A => B): CC[B] = ???  
}
```

```
trait List[+A] extends Iterable[A] with IterableOps[A, List, List[A]] {...}  
trait Vector[+A] extends Iterable[A] with IterableOps[A, Vector, Vector[A]] {...}
```

# *IterableOps*

```
trait IterableOps[+A, +CC[_], +C] {  
    def filter(p: A => Boolean): C = ???  
    def map[B](f: A => B): CC[B] = ???  
}
```

```
trait List[+A] extends Iterable[A] with IterableOps[A, List, List[A]] {...}  
trait Vector[+A] extends Iterable[A] with IterableOps[A, Vector, Vector[A]] {...}
```

- *IterableOps* is the MagicBaseTrait we are looking for.
- *IterableOps* is called a template trait.
- *IterableOps* has 3 type parameters, one for the element type (*A*), one for the collection type (*C*) and one for the collection's type constructor type (*CC*).
- Leaf collection types with one type parameter (*List*, *Vector*) extend *IterableOps*.
- This does not work for collections like *Map* with two type parameters.

# *MapOps*

```
trait MapOps[K, +V, +CC[_, _], +C] extends IterableOps[(K, V), Iterable, C] {  
    def map[K2, V2](f: ((K, V)) => (K2, V2)): CC[K2, V2] = ???  
}
```

```
trait Map[K, V] extends Iterable[(K, V)] with MapOps[K, V, Map, Map[K, V]]
```

# *MapOps*

```
trait MapOps[K, +V, +CC[_, _], +C] extends IterableOps[(K, V), Iterable, C] {  
    def map[K2, V2](f: ((K, V)) => (K2, V2)): CC[K2, V2] = ???  
}
```

```
trait Map[K, V] extends Iterable[(K, V)] with MapOps[K, V, Map, Map[K, V]]
```

- *MapOps* extends *IterableOps* and hence inherits all its operations.
- *MapOps* instantiates the collection's type constructor *CC* with *Iterable*.
- *MapOps* inherits a *map* operation from *IterableOps* returning *Iterable[B]*.
- *MapOps* defines another *map* operation overload returning *Map[K2, V2]*.
- *Map* inherits both *map* operations.

# Which *map* is chosen ...

... when you invoke *Map#map* at the call site?

```
// from IterableOps
def map[B](f: ((K, V)) => B): Iterable[B]

// from MapOps
def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]
```

- *map* from *MapOps* is more specific by the rules of overloading resolution.
- It will be chosen, if the *map* operation returns a pair of values.
- Otherwise the operation from *IterableOps* applies.

# Which *map* is chosen ...

... when you invoke *Map#map* at the call site?

```
// from IterableOps
def map[B](f: ((K, V)) => B): Iterable[B]

// from MapOps
def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]
```

- *map* from *MapOps* is more specific by the rules of overloading resolution.
- It will be chosen, if the *map* operation returns a pair of values.
- Otherwise the operation from *IterableOps* applies.

“same-result-type” principle:

Wherever possible a transformation method on a collection  
yields a collection of the same type.

# 8. Migration

See also:

<https://docs.scala-lang.org/overviews/core/collections-migration-213.html>

# When to Migrate?

- when all *libraryDependencies* are available for 2.13.
- when all transitive dependencies are available for 2.13.

# Migrate an Application w/o Cross Compilation?

- Upgrade your project to the latest 2.12.x version.
- Upgrade your sbt *libraryDependencies* to versions which are available in both binary versions: 2.12 and 2.13.
- Upgrade the project to 2.13.  
You loose backward compatibility to 2.12.  
Use all the new features of 2.13.
- Use *scalafix* to automate as much as possible!
- Rewrite the rest by hand.

# Migrate a Library with Cross Compilation?

- Upgrade your project to the latest 2.12.x version.
- Upgrade your sbt *libraryDependencies* to versions which are available in both binary versions: 2.12 and 2.13.
- Upgrade the project for cross-compilation to 2.13, 2.12 (possibly 2.11).
- Keep source compatibility as much as possible w/o using the new features of 2.13.
- Use version specific source folders: *src/main/scala-2.12* and *src/main/scala-2.13*
- Use *scala-collection-compat* library which backports many parts of the collection API to 2.12.
- Use *scalafix* to automate as much as possible!
- Rewrite the rest by hand.

# Migration Automation with Scalafix (Setup)

# Migration Automation with Scalafix (Setup)

Add scalafix plugin

```
// project/plugins.sbt
addSbtPlugin("ch.epfl.scala" % "sbt-scalafix" % "0.9.5")
```

# Migration Automation with Scalafix (Setup)

Add scalafix plugin

```
// project/plugins.sbt
addSbtPlugin("ch.epfl.scala" % "sbt-scalafix" % "0.9.5")
```

Change *build.sbt*

```
// build.sbt
scalafixDependencies += "org.scala-lang.modules" %% "scala-collection-migrations"
// scala-collection-compat needed only for cross compilation
libraryDependencies += "org.scala-lang.modules" %% "scala-collection-compat" % "2.
scalacOptions ++= List("-Yrangeapos", "-P:semanticdb:synthetics:on")
```

# Migration Automation with Scalafix (Run)

Run scalafix in sbt shell (for upgrade to 2.13)

```
> ;test:scalafix Collection213Upgrade ;scalafix Collection213Upgrade
```

Run scalafix in sbt shell (for cross compilation to 2.12/2.13)

```
> ;test:scalafix Collection213CrossCompat ;scalafix Collection213CrossCompat
```

Scala 2.13 Collection Compatibility Library and Migration Tool:

<https://github.com/scala/scala-collection-compat>

# 21. Resources

# Resources

- Code and Slides of this Talk:  
<https://github.com/hermannhueck/new-in-scala213>
- Official Release Description  
<https://github.com/scala/scala/releases/tag/v2.13.0>
- Making Our Future Better  
Viktor Klang's talk at Scala Days 2019  
<https://www.youtube.com/watch?v=5FTJUUoT6y4>

# Scala 2.13 Collection Related Links

- Implementing the Scala 2.13 collections  
Stefan Zeiger's talk at Scala Days 2019 in Lausanne  
<https://www.youtube.com/watch?v=L1lxZ1LBuGI>
- The Architecture of Scala 2.13's Collections  
<https://docs.scala-lang.org/overviews/core/architecture-of-scala-213-collections.html>
- Migrating a project to Scala 2.13's Collections  
<https://docs.scala-lang.org/overviews/core/collections-migration-213.html>
- Scala 2.13 Collection Compatibility Library  
<https://github.com/scala/scala-collection-compat>
- Let them be Lazy  
Julien Richard Foy's blog on the lazy collection (Views and LazyList)  
<https://www.scala-lang.org/blog/2017/11/28/view-based-collections.html>
- Scala 2.13's Collections Rework  
Stefan Zeiger's blog on the collections rework  
<https://www.scala-lang.org/blog/2017/02/28/collections-rework.html>

# Thank You

## Q & A

<https://github.com/hermannhueck/new-in-scala213>

