What is new since "Programming in Scala"

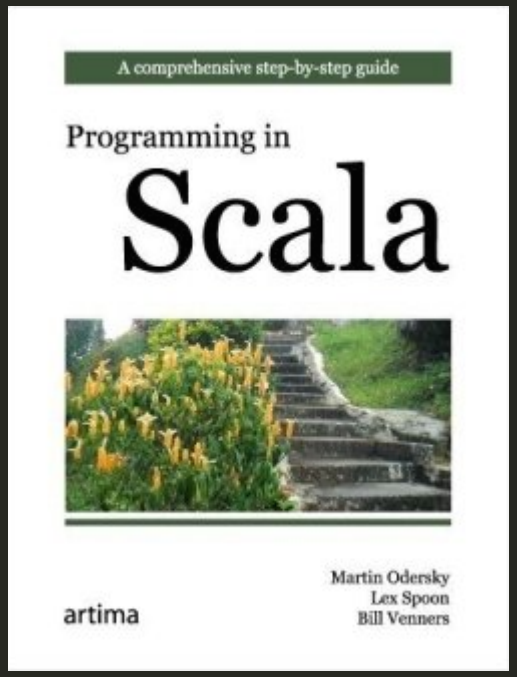
Marconi Lanna

Originate

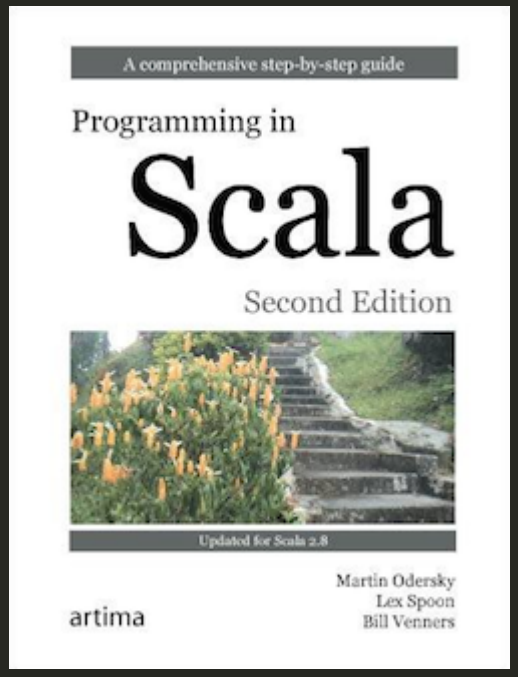
Originate

Programming in Scala

Martin Odersky, Lex Spoon, Bill Venners



December 10, 2008
Scala 2.7.2



December 13, 2010
Scala 2.8.1

Originate

Scala timeline

- 2003: 0.8, 0.9
- 2004: 1.0, 1.1, 1.2, 1.3
- 2005: 1.4
- 2006: 2.0, 2.1, 2.2, 2.3
  - `scalac` written in Scala
- 2007: 2.4, 2.5, 2.6
  - Lift
- 2008: 2.7
- 2010: 2.8
  - Play 1.1: Scala support via plug-in
  - Akka
- 2011: 2.9
  - Typesafe
- 2012
  - Play 2.0: native Scala
- 2013: 2.10
- 2014: 2.11
- 2016: 2.12

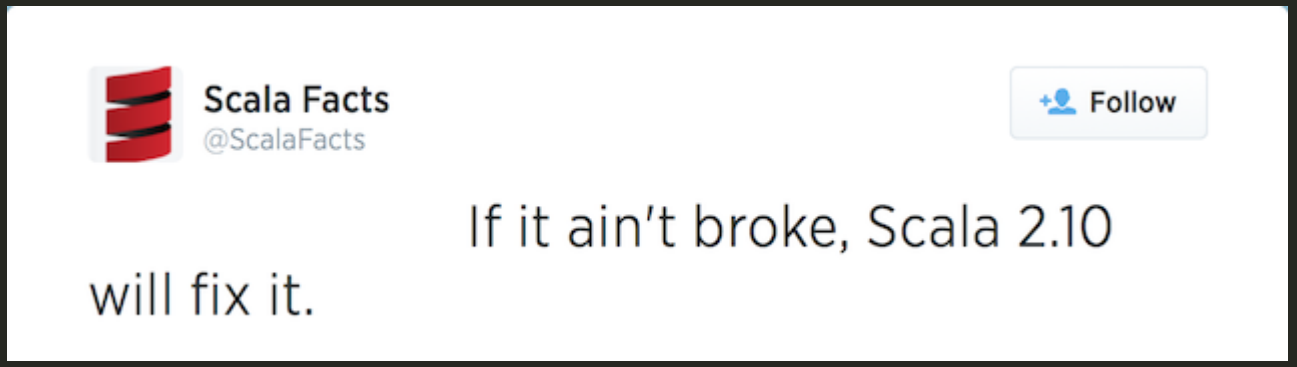
What was new in Scala 2.8

> "A huge number of bug fixes with respect to 2.7.7, and an impressive amount of new features."
>
> *Scala 2.8.0 Release Notes*

- Redesigned collection library
- Named and default parameters
  - `copy` methods for case classes
- Package objects
- Nested annotations
- Type specialization (boxed primitives)
- `JavaConverters`
  - Explicit implicit Java <-> Scala conversions: `asScala` and `asJava`
- Revamped REPL
  - Enhanced tab-completion
  - Searchable history (`Ctrl-R`)
- Scaladoc 2
  - New look-and-feel
  - Wiki-like syntax
- Binary compatibility (*for minor revisions*)

What is new in Scala 2.9, 2.10, 2.11

Originate


Scala Facts
@ScalaFacts
Follow
If it ain't broke, Scala 2.10 will fix it.

`DelayedInit` and the `App` trait (2.9)

```
object Hello extends Application {
    println("hello, world")
}
```

Not thread safe, not optimized by the JVM.

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

Command line arguments accessible via `val args`.

DelayedInit and the App trait (2.9)

```
import System.{currentTimeMillis => now}

object application extends Application {
    val n = 4 * 1000 * 1000

    val start = now

    var sum = 0L
    for (i <- 1 to n) sum += i
    val middle = now

    sum = 0L
    for (i <- 1 to n) sum += i
    val end = now

    println("application results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

`DelayedInit` and the `App` trait (2.9)

```
import System.{currentTimeMillis => now}

object app extends App {
    val n = 4 * 1000 * 1000

    val start = now

    var sum = 0L
    for (i <- 1 to n) sum += i
    val middle = now

    sum = 0L
    for (i <- 1 to n) sum += i
    val end = now

    println("app results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

`DelayedInit` and the `App` trait (2.9)

```
application results:
1st run: 6514ms
2nd run: 7249ms


app results:
1st run: 7ms
2nd run: 5ms
```

DelayedInit and the App trait (2.9)

```
object application2 extends Application {
    val n = 4 * 1000 * 1000

    val start = now

    var i, sum = 0L
    while (i < n) {
        i += 1
        sum += i
    }
    val middle = now

    i = 0L
    sum = 0L
    while (i < n) {
        i += 1
        sum += i
    }
    val end = now

    println("application2 results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

DelayedInit and the App trait (2.9)

```
object app2 extends App {
    val n = 4 * 1000 * 1000

    val start = now

    var i, sum = 0L
    while (i < n) {
        i += 1
        sum += i
    }
    val middle = now

    i = 0L
    sum = 0L
    while (i < n) {
        i += 1
        sum += i
    }
    val end = now

    println("app2 results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

`DelayedInit` and the `App` trait (2.9)

`for`

```
application results:
1st run: 6514ms
2nd run: 7249ms

app results:
1st run: 7ms
2nd run: 5ms
```

`while`

```
application2 results:
1st run: 45ms
2nd run: 41ms

app2 results:
1st run: 8ms
2nd run: 3ms
```

`Range.foreach` optimization (2.10)

"Makes code like

```
0 to 100 foreach (x += _)
```

as fast as (often faster than, in fact) a while loop."

Paul Phillips, Git commit message

Parallel collections (2.9)

> "An effort to facilitate parallel programming by sparing users from low-level parallelization details, meanwhile providing them with a familiar and simple high-level abstraction."

Efficient and transparent: `collection.par` and `collection.seq`.

Simply invoke the `par` method on the sequential collection and use the parallel collection in the same way one would normally use a sequential collection.

Depending on the collection, `par` may be a constant time operation: the underlying dataset is shared between sequential and parallel collections. `seq` is always O(1).

- Array
- Iterable
- Map
- Range
- Seq
- Set
- Trie (2.10)
- Vector

Parallel collections (2.9)

- Concurrent, out-of-order semantics.
- The order in which functions are applied to the elements is arbitrary.
- Side effects are prone to race conditions.
- Side effects and non-associative operations can lead to non-determinism.
- Unlike non-associative, non-commutative operations are deterministic.

Associative and commutative: addition

```
(1 + 2) + 3 == 1 + (2 + 3) == (2 + 1) + 3 // Works
```

Associative, non-commutative: string concatenation

```
("a" + "b") + "c" == "a" + ("b" + "c") != ("b" + "a") + "c" // Works
```

Non-associative, non-commutative: subtraction

```
(1 - 2) - 3 != 1 - (2 - 3) != (2 - 1) - 3 // Does not work
```

Parallel collections (2.9)

```
import System.{currentTimeMillis => now}
import scala.util.Random

object seq extends App {
    val n = 50 * 1000 * 1000
    val max = 2 * n

    def random = Random nextInt max

    val col = Vector.fill(n)(random)
    val target = random

    val start = now

    col.count(math.sqrt(_) == target)
    val middle = now

    col.count(math.sqrt(_) == target)
    val end = now

    println("seq results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

Parallel collections (2.9)

```
import System.{currentTimeMillis => now}
import scala.util.Random

object par extends App {
    val n = 50 * 1000 * 1000
    val max = 2 * n

    def random = Random nextInt max

    val col = Vector.fill(n)(random).par
    val target = random

    val start = now

    col.count(math.sqrt(_) == target)
    val middle = now

    col.count(math.sqrt(_) == target)
    val end = now

    println("par results:")
    println("1st run: " + (middle-start) + "ms")
    println("2nd run: " + (end-middle) + "ms")
}
```

Originate

Parallel collections (2.9)

```
seq results:
1st run: 431ms
2nd run: 416ms


par results:
1st run: 116ms
2nd run: 104ms
```

Generalized try-catch-finally (2.9)

Reusable exception handling.

```
try
    body
catch
    handler
finally
    cleanup
```

Where `body` and `cleanup` are arbitrary expressions, and `handler` is a `PartialFunction[Throwable, T]`.

```
import scala.util.control.NonFatal

val defaultHandler: PartialFunction[Throwable, Unit] = {
    case NonFatal(t) => println(t)
}

try 1 / 0 catch defaultHandler

try "a".toInt catch defaultHandler
```

Error handling with `Try` (2.10)

> "`Try` represents a computation that may either result in an exception, or return a successfully computed value."

`Try` is used to perform operations without the need to do explicit exception-handling in all places that an exception might occur.

Instances of `Try[T]` are either `Success[T]` or `Failure[T]`.

Only non-fatal exceptions are caught. System errors are thrown.

- `map`
- `flatMap`
- `recover`
- `recoverWith`
- `filter`
- `getOrElse`
- `toOption`

Error handling with `Try` (2.10)

```
def div(x: String, y: String): Try[Int] = {
    for {
        a <- Try(x.toInt)
        b <- Try(y.toInt)
    } yield a / b
}


div("a", "0")
// Failure(java.lang.NumberFormatException: For input string: "a")


div("1", "b")
// Failure(java.lang.NumberFormatException: For input string: "b")


div("1", "0")
// Failure(java.lang.ArithmeticException: / by zero)


assert(div("1", "0").toOption == None)

assert(div("1", "0").getOrElse(-1) == -1)

assert(div("42", "3").get == 14)
```

Value classes, implicit classes, and extension methods (2.10)

Originate

Implicit classes (2.10)

A more convenient syntax for defining extension methods.

Implicit classes have a primary constructor with exactly one parameter.

```
implicit class A(n: Int) {
    def x = ???
    def y = ???
}
```

They are desugared into a class and an implicit method pairing.

```
class A(n: Int) { ... }

implicit def A(n: Int) = new A(n)
```

Case classes cannot be implicit.

```
implicit class IntOps(n: Int) {
    def stars = "*" * n
}

assert(5.stars == "*****")
```

Value classes (2.10)

Value classes are used to avoid object allocation. (*Conditions apply*)

The type safety of custom data types without the runtime overhead.

`Celsius`, `Fahrenheit`, `Weight`, `Height`, `FirstName`, `Email`, `Age` etc.

Only a primary constructor with exactly one `val` parameter.

Only methods (`def`). No `var`, `val`, `lazy val`, nested classes, traits, or objects.

May not define `equals` or `hashCode` methods.

Cannot be extended by another class.

```
case class Age(age: Int) extends AnyVal

val age = Age(18)
```

At compile time `age` is of type `Age`, but at runtime it is an `Int`.

```
age + 1 // error: type mismatch
```

Extension methods (2.10)

Value classes and implicit classes can be combined to produce allocation-free extension methods.

Equivalent to using an object with static helper methods.

A simple mechanical transformation performed by the compiler.

```
implicit class IntOps(n: Int) {
    def stars = "*" * n
}

5.stars

// equivalent to
new IntOps(5).stars
```

```
implicit class IntOps(val n: Int)
    extends AnyVal {
    def stars = "*" * n
}

5.stars

// equivalent to
object IntOps {
    def stars(n: Int) = "*" * n
}

IntOps.stars(5)
```

String interpolation (2.10)

```
val name = "world"
assert(s"hello, $name" == "hello, world")
```

Supports arbitrary expressions:

```
assert(s"${2 + 2} = 4" == "4 = 4")
```

Works with triple quotes:

```
s"""Dear $victim,

I am $name, the only $heir of late $deceased.
My $relative was a very wealthy $occupation in
$location, the economic capital of $country
before he was $cause to death by $murderer.

Before the death of my $relative on $date, he secretly
told me that he has a sum of $value left in a
suspense account in a local Bank here in $country."""
```

```
val a = 0.02
assert(s"US$$$a" == "US$0.02")
```

String interpolation (2.10)

Formatted strings:

```
assert(f"${math.Pi}%.2f" == "3.14")
```

The `f` interpolator is typesafe:

```
f"${math.Pi}%d"
// error: type mismatch;
// found   : Double
// required: Int
```

At *compile* time!

Custom interpolators:

```
sql"insert into City(name, country) values ($name, $country)"

json"""{ "foo" : $foo, "bar" : $bar }"""
```

`Futures` and `Promises` (2.10, 2.9.3)

`Future` is a way to perform many operations in parallel in an efficient and non-blocking (asynchronous) way.

`Future` is a placeholder for a result that does not yet exist, but which may become available at some point.

Callbacks (`onComplete`, `onSuccess`, `onFailure`) are executed **eventually**.

The order in which callbacks are executed is not deterministic.

Callbacks may not be called sequentially, but execute concurrently at the same time.

Futures can be combined and transformed with `map`, `flatMap`, `recover`, `recoverWith`, `filter`, `foreach`, `andThen`, `collect`, `fallbackTo`, etc. and used in for-comprehensions.

Futures and Promises (2.10, 2.9.3)

```
def f(tag: String, ms: Int) = Future {
    println(s"$tag started")
    Thread.sleep(ms)
    println(s"$tag ended")
    tag
}
```

Futures and Promises (2.10, 2.9.3)

```
for {
    a <- f("a", 500)
    b <- f("b", 300)
    c <- f("c", 200)
} yield a + b + c
```

```
a started
a ended
b started
b ended
c started
c ended
```

```
val fa = f("a", 500)
val fb = f("b", 300)
val fc = f("c", 200)

for {
    a <- fa
    b <- fb
    c <- fc
} yield a + b + c
```

```
a started
c started
b started
c ended
b ended
a ended
```

Futures and Promises (2.10, 2.9.3)

`Future` is a read-only placeholder for a result which does not yet exist.

`Promise` is a writable, single-assignment container which completes a `Future`.

```
val p = Promise[Int]
val f = p.future

assert(!f.isCompleted)

p success 42

assert(f.isCompleted)
```

`Future.sequence` converts a `Seq[Future[T]]` to a `Future[Seq[T]]`.

`Dynamic` trait (2.10)

Syntax sugar. A simple mechanical transformation performed by the compiler.

It is not any sort of "*dynamic*" type; it is not any sort of "*optional*" static typing.

Enable flexible DSLs and convenient interfacing with dynamic languages and data formats like JSON.

Extend `Dynamic` and implement at least one of the following methods:

- `applyDynamic`
- `applyDynamicNamed`
- `selectDynamic`
- `updateDynamic`

```
foo.bar               -> foo.selectDynamic("bar")
foo.bar = 42          -> foo.updateDynamic("bar")(42)
foo.bar(0) = 42       -> foo.selectDynamic("bar").update(0, 42)
foo.bar("baz")        -> foo.applyDynamic("bar")("baz")
foo.bar(baz = "qux") -> foo.applyDynamicNamed("bar")(("baz", "qux"))
foo.bar(baz = 1, 2)  -> foo.applyDynamicNamed("bar")(("baz", 1), ("", 2))
```

Akka actors (2.10)

Since Scala 2.10, Akka is the default actor library.

The legacy Scala Actors library is deprecated in 2.11.

Please refer to “The Scala Actors Migration Guide” for details :-)

Modularization (2.10)

Some of the more advanced language features have to be explicitly enabled.

```
import language.X
```

- `dynamics`
- `existentials`
- `higherKinds`
- `implicitConversions`
- `postfixOps`
- `reflectiveCalls`
- `experimental.macros`

```
scalac -language:dynamics
```

To enable all features: `import language._` or `scalac -language:_`

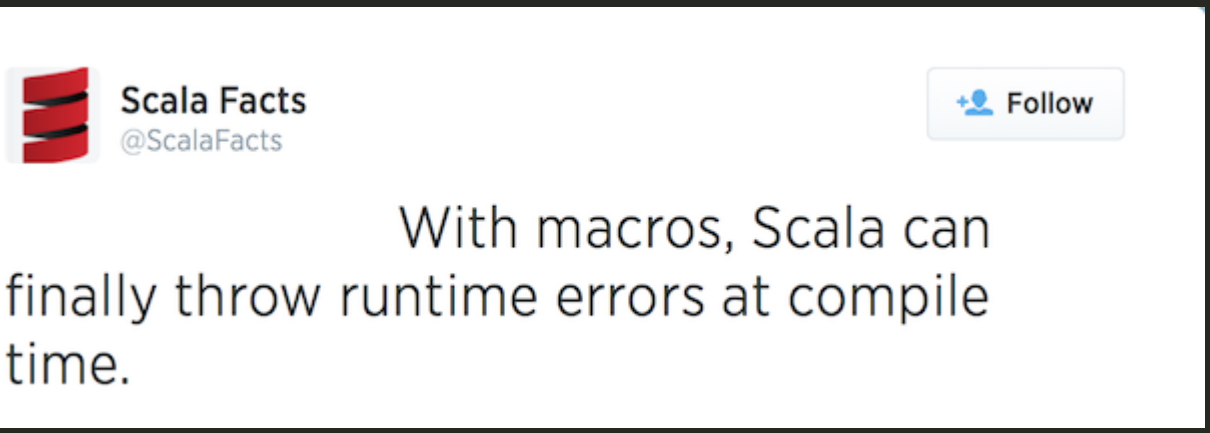
`implicitConversions` is only needed when *defining* new implicit conversions.

Not needed to use existing conversions.

Not needed to define implicit classes.

Reflection, macros, and quasiquotes (2.10, experimental)

Originate


Scala Facts
@ScalaFacts
Follow
With macros, Scala can finally throw runtime errors at compile time.

Originate

Reflection, macros, and quasiquotes (2.10, experimental)

Metaprogramming: programs that modify themselves at compile time.

Code generation and advanced DSLs.

Compile-time (macros) and runtime reflection.

Scala specific elements are unavailable under the Java reflection API: functions, traits, generics, existential, higher-kinded, path-dependent and abstract types, etc.

Reified Scala expressions.

Reflection, macros, and quasiquotes (2.10, experimental)

Quasiquotes (2.11) are a *significantly simplified* notation to manipulate Scala syntax trees with ease.

```
q"foo + bar"
```

```
assert(q"foo + bar" equalsStructure q"foo.+(bar)")
```

Quasiquotes can be decomposed via pattern matching:

```
val q"$foo + $bar" = q"1 + 2 * 3"

assert(foo equalsStructure q"1")

assert(bar equalsStructure q"2*3")
assert(bar equalsStructure q"2.$$times(3)")
```

Reflection, macros, and quasiquotes (2.10, experimental)

```
log("We have a problem")


if (Logger.enabled)
    Logger.log("We have a problem")


def log(msg: String): Unit = macro impl


def impl(c: Context)(msg: c.Expr[String]): c.Expr[Unit] = {
    import c.universe._

    q"""
        if (Logger.enabled)
            Logger.log($msg)
    """
}
```

Reflection, macros, and quasiquotes (2.10, experimental)

Play JSON API

```
case class ABC(a: Int, b: String, c: Boolean)

implicit val xfmt = Json.format[ABC]

toJson(), fromJson()
```

- Type safe
- No runtime reflection
- No bytecode enhancement

Scala Pickling

“A fast, boilerplate-free, automatic serialization framework for Scala supporting different serialization formats”

```
val pckl = List(1, 2, 3, 4).pickle
val lst = pckl.unpickle[List[Int]]
```

Case classes with more than 22 parameters (2.11)

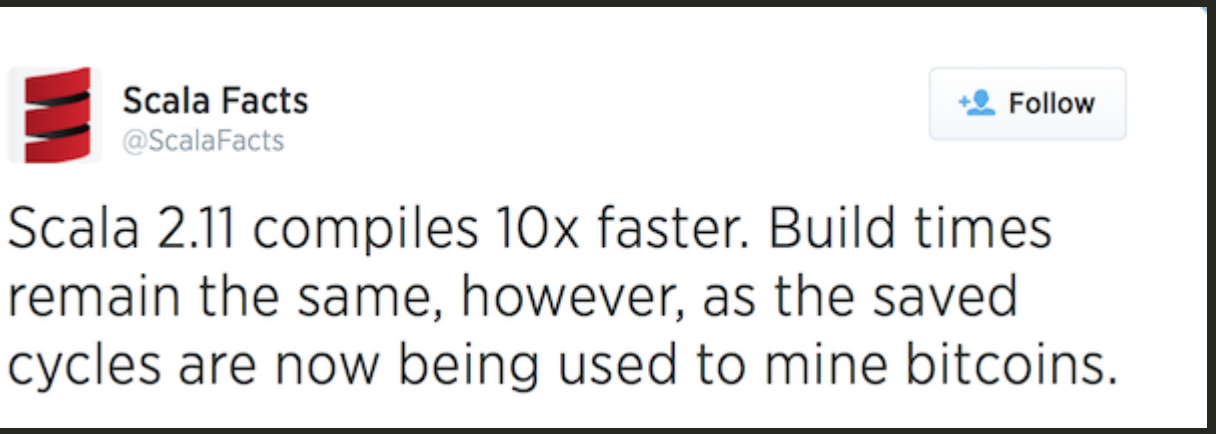
```
case class Alphabet(
      a: Int, b: Int, c: Int, d: Int, e: Int
    , f: Int, g: Int, h: Int, i: Int, j: Int
    , k: Int, l: Int, m: Int, n: Int, o: Int
    , p: Int, q: Int, r: Int, s: Int, t: Int
    , u: Int, v: Int, w: Int, x: Int, y: Int
    , z: Int)

Alphabet(
       0,  1,  2,  3,  4,  5,  6,  7,  8,  9
    , 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
    , 20, 21, 22, 23, 24, 25)
```

New methods in collections

Non-exhaustive list:

- Iterator
  - `span` (2.9)
- Option
  - `filterNot` (2.9)
  - `flatten`, `fold`, `forall`, `nonEmpty` (2.10)
  - `contains` (2.11)
- Seq
  - `permutations` (2.9)
- SeqLike
  - `combinations` (2.9)
- mutable.SeqLike
  - `transform` (2.9)
- SetLike
  - `subsets` (2.9)
- Traversable
  - `inits`, `tails`, `unzip3` (2.9)
- TraversableOnce
  - `collectFirst`, `maxBy`, `minBy` (2.9)

sbt incremental compilation (2.11)


Scala Facts
@ScalaFacts
Follow
Scala 2.11 compiles 10x faster. Build times remain the same, however, as the saved cycles are now being used to mine bitcoins.

sbt incremental compilation (2.11)

sbt 0.13.2

```
incOptions := incOptions.value.withNameHashing(true)
```

Originate

Predef.??? (2.10)

Placeholder for methods not yet implemented.

Useful for TDD, code samples, presentations, blogs, implementing abstract methods in a rush...

```
object A {
    def x: Int = ???
}

A.x // scala.NotImplementedError: an implementation is missing
```

2.12 and beyond: what is in Scala future?

- Java 8 support (2.12)
  - Java 8-style lambdas
  - Java streams
  - Bidirectional interoperability
  - Java 8+ only (2.12)
- Compiler-based code style checker
- Improved lazy vals initialization (SIP-20)


Scala Facts
@ScalaFacts
Follow
Lazy vals are the sausages of Scala. They look delicious at first, until you learn how they are made.

2.12 and beyond: what is in Scala future?

- Java 8 support (2.12)
  - Java 8-style lambdas
  - Java streams
  - Bidirectional interoperability
  - Java 8+ only (2.12)
- Compiler-based code style checker
- Improved lazy vals initialization (SIP-20)
- Spores (SIP-21): safer closures for concurrent and distributed environments
- `async` & `await` (SIP-22)
- Collections library cleanup and simplification
- No procedure syntax (`def a {...}`)
- XML string interpolation replaces XML literals
- `scala.meta`
- Scala.js
- Dotty

References

- **github.com/marconilanna/ScalaByTheBay2014**
- **blog.originate.com**
- Scala 2.8.0 Release Notes
- Scala 2.9.0 Release Notes
- Scala 2.10.0 Release Notes
- Scala 2.11.0 Release Notes
- Scala Documentation
  - Scala Improvement Process
  - Parallel Collections
  - Implicit Classes
  - Value Classes
  - String Interpolation
  - Futures and Promises
  - The Scala Actors Migration Guide
  - Reflection
  - Macros
  - Quasiquotes
- Range.foreach optimization

Thank you! Questions?

Originate