

# Scala (programming language)

**Scala** (/ˈskɑːlɑː/ *SKAH-lah*)<sup>[7]</sup> is a general-purpose programming language providing support for both object-oriented programming and functional programming. The language has a strong static type system. Designed to be concise,<sup>[8]</sup> many of Scala's design decisions are aimed to address criticisms of Java.<sup>[6]</sup>

Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Scala provides language interoperability with Java, so that libraries written in either language may be referenced directly in Scala or Java code.<sup>[9]</sup> Like Java, Scala is object-oriented, and uses a curly-brace syntax reminiscent of the C programming language. Unlike Java, Scala has many features of functional programming languages like Scheme, Standard ML and Haskell, including currying, immutability, lazy evaluation, and pattern matching. It also has an advanced type system supporting algebraic data types, covariance and contravariance, higher-order types (but not higher-rank types), and anonymous types. Other features of Scala not present in Java include operator overloading, optional parameters, named parameters, and raw strings. Conversely, a feature of Java not in Scala is checked exceptions, which has proved controversial.<sup>[10]</sup>

The name Scala is a portmanteau of *scalable* and *language*, signifying that it is designed to grow with the demands of its users.<sup>[11]</sup>

## Contents

### History

### Platforms and license

Other compilers and targets

### Examples

"Hello World" example

Basic example

Example with classes

### Features (with reference to Java)

Syntactic flexibility

Unified type system

For-expressions

Functional tendencies

Everything is an expression

Type inference

Anonymous functions

## Scala



<b>Paradigm</b>	Multi-paradigm: <u>concurrent</u> , <u>functional</u> , <u>imperative</u> , <u>object-oriented</u>
<b>Designed by</b>	Martin Odersky
<b>Developer</b>	Programming Methods Laboratory of École polytechnique fédérale de Lausanne
<b>First appeared</b>	20 January 2004
<b>Stable release</b>	2.13.3 / 25 June 2020 <sup>[1]</sup>
<b>Typing discipline</b>	Inferred, static, <u>strong</u> , <u>structural</u>
<b>Implementation language</b>	Scala
<b>Platform</b>	<ul style="list-style-type: none"> <li><u>JVM</u></li> <li><u>JavaScript</u> (Scala.js (<u>http://scala-js.org</u>))</li> <li><u>LLVM</u> (Scala Native (<u>http://scala-native.org</u>)) (experimental)</li> </ul>
<b>License</b>	<u>Apache License</u> 2.0 <sup>[2]</sup>
<b>Filename</b>	.scala, .sc

[Immutability](#)

[Lazy \(non-strict\) evaluation](#)

[Tail recursion](#)

[Case classes and pattern matching](#)

[Partial functions](#)

[Object-oriented extensions](#)

[Expressive type system](#)

[Type enrichment](#)

**[Concurrency](#)**

**[Cluster computing](#)**

**[Testing](#)**

**[Versions](#)**

**[Comparison with other JVM languages](#)**

**[Adoption](#)**

[Language rankings](#)

[Companies](#)

**[Criticism](#)**

**[See also](#)**

**[References](#)**

**[Further reading](#)**

**[extensions](#)**


**Website** [www.scala-lang.org](http://www.scala-lang.org) (<https://www.scala-lang.org>)

**Influenced by**

[Common Lisp](#),<sup>[3]</sup> [Eiffel](#), [Erlang](#), [Haskell](#),<sup>[4]</sup> [Java](#),<sup>[5]</sup> [OCaml](#),<sup>[5]</sup> [Oz](#), [Pizza](#),<sup>[6]</sup> [Scheme](#),<sup>[5]</sup> [Smalltalk](#), [Standard ML](#)<sup>[5]</sup>

**Influenced**

[Ceylon](#), [Fantom](#), [F#](#), [Kotlin](#), [Lasso](#), [Red](#)

 [Scala](#) at Wikibooks

## History

The design of Scala started in 2001 at the [École Polytechnique Fédérale de Lausanne](#) (EPFL) (in [Lausanne](#), [Switzerland](#)) by [Martin Odersky](#). It followed on from work on [Funnel](#), a programming language combining ideas from functional programming and [Petri nets](#).<sup>[12]</sup> Odersky formerly worked on [Generic Java](#), and [javac](#), Sun's Java compiler.<sup>[12]</sup>

After an internal release in late 2003, Scala was released publicly in early 2004 on the [Java platform](#).<sup>[13][6][12][14]</sup> A second version (v2.0) followed in March 2006.<sup>[6]</sup>

On 17 January 2011, the Scala team won a five-year research grant of over €2.3 million from the [European Research Council](#).<sup>[15]</sup> On 12 May 2011, Odersky and collaborators launched Typesafe Inc. (later renamed [Lightbend Inc.](#)), a company to provide commercial support, training, and services for Scala. Typesafe received a \$3 million investment in 2011 from [Greylock Partners](#).<sup>[16][17][18][19]</sup>

## Platforms and license

Scala runs on the [Java platform](#) (Java virtual machine) and is compatible with existing Java programs.<sup>[13]</sup> As [Android](#) applications are typically written in Java and translated from Java bytecode into [Dalvik](#) bytecode (which may be further translated to native machine code during installation) when packaged, Scala's Java compatibility makes it well-suited to Android development, more so when a functional approach is preferred.<sup>[20]</sup>

The reference Scala software distribution, including compiler and libraries, is released under the [Apache license](#).<sup>[21]</sup>

## Other compilers and targets

*Scala.js* is a Scala compiler that compiles to JavaScript, making it possible to write Scala programs that can run in web browsers or *Node.js*.<sup>[22]</sup> The compiler was in development since 2013, was announced as no longer experimental in 2015 (v0.6). Version v1.0.0-M1 was released in June 2018. In September 2020 it is at version 1.1.1.<sup>[23]</sup>

Scala Native is a Scala [compiler](#) that targets the [LLVM](#) compiler infrastructure to create executable code that uses a lightweight managed runtime, which uses the [Boehm garbage collector](#). The project is led by Denys Shabalin and had its first release, 0.1, on 14 March 2017. Development of Scala Native began in 2015 with a goal of being faster than [just-in-time compilation](#) for the JVM by eliminating the initial runtime compilation of code and also providing the ability to call native routines directly.<sup>[24][25]</sup>

A reference Scala compiler targeting the [.NET Framework](#) and its [Common Language Runtime](#) was released in June 2004,<sup>[12]</sup> but was officially dropped in 2012.<sup>[26]</sup>

## Examples

---

### "Hello World" example

The [Hello World program](#) written in Scala has this form:

```
object HelloWorld extends App {  
  println("Hello, World!")  
}
```

Unlike the [stand-alone Hello World application for Java](#), there is no class declaration and nothing is declared to be static; a [singleton object](#) created with the **object** keyword is used instead.

When the program is stored in file *HelloWorld.scala*, the user compiles it with the command:

```
$ scalac HelloWorld.scala
```

and runs it with

```
$ scala HelloWorld
```

This is analogous to the process for compiling and running Java code. Indeed, Scala's compiling and executing model is identical to that of Java, making it compatible with Java build tools such as [Apache Ant](#).

A shorter version of the "Hello World" Scala program is:

```
println("Hello, World!")
```

Scala includes interactive shell and scripting support.<sup>[27]</sup> Saved in a file named `HelloWorld2.scala`, this can be run as a script with no prior compiling using:

```
$ scala HelloWorld2.scala
```

Commands can also be entered directly into the Scala interpreter, using the option `-e`:

```
$ scala -e 'println("Hello, World!")'
```

Expressions can be entered interactively in the REPL:

```
$ scala
Welcome to Scala 2.12.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> List(1, 2, 3).map(x => x * x)
res0: List[Int] = List(1, 4, 9)

scala>
```

## Basic example

The following example shows the differences between Java and Scala syntax:

```
// Java:
int mathFunction(int num) {
    int numSquare = num*num;
    return (int) (Math.cbrt(numSquare) +
        Math.log(numSquare));
}
```

```
// Scala: Direct conversion from Java

// no import needed; scala.math
// already imported as `math`
def mathFunction(num: Int): Int = {
    var numSquare: Int = num*num
    return (math.cbrt(numSquare) +
        math.log(numSquare)).
        asInstanceOf[Int]
}
```

```
// Scala: More idiomatic
// Uses type inference, omits `return`
// statement,
// uses `toInt` method, declares numSquare
// immutable

import math._
def mathFunction(num: Int) = {
    val numSquare = num*num
    (cbrt(numSquare) + log(numSquare)).toInt
}
```

Some syntactic differences in this code are:

- Scala does not require semicolons to end statements.
- Value types are capitalized: `Int`, `Double`, `Boolean` instead of `int`, `double`, `boolean`.
- Parameter and return types follow, as in Pascal, rather than precede as in C.
- Methods must be preceded by `def`.
- Local or class variables must be preceded by `val` (indicates an immutable variable) or `var` (indicates a mutable variable).

- The `return` operator is unnecessary in a function (although allowed); the value of the last executed statement or expression is normally the function's value.
- Instead of the Java cast operator `(Type)` `foo`, Scala uses `foo.asInstanceOf[Type]`, or a specialized function such as `toDouble` or `toInt`.
- Instead of Java's `import foo.*;`, Scala uses `import foo._`.
- Function or method `foo()` can also be called as just `foo`; method `thread.send(signo)` can also be called as just `thread send signo`; and method `foo.toString()` can also be called as just `foo toString`.

These syntactic relaxations are designed to allow support for domain-specific languages.

Some other basic syntactic differences:

- Array references are written like function calls, e.g. `array(i)` rather than `array[i]`. (Internally in Scala, the former expands into `array.apply(i)` which returns the reference)
- Generic types are written as e.g. `List[String]` rather than Java's `List<String>`.
- Instead of the pseudo-type `void`, Scala has the actual singleton class `Unit` (see below).

## Example with classes

The following example contrasts the definition of classes in Java and Scala.

```
// Java:
public class Point {
    private final double x, y;

    public Point(final double x, final double
y) {
        this.x = x;
        this.y = y;
    }

    public Point(
        final double x, final double y,
        final boolean addToGrid
    ) {
        this(x, y);

        if (addToGrid)
            grid.add(this);
    }

    public Point() {
        this(0.0, 0.0);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    double distanceToPoint(final Point other)
    {
        return distanceBetweenPoints(x, y,
            other.x, other.y);
    }

    private static Grid grid = new Grid();

    static double distanceBetweenPoints(
        final double x1, final double y1,
        final double x2, final double y2
    ) {
```

```
// Scala
class Point(
    val x: Double, val y: Double,
    addToGrid: Boolean = false
) {
    import Point._

    if (addToGrid)
        grid.add(this)

    def this() = this(0.0, 0.0)

    def distanceToPoint(other: Point) =
        distanceBetweenPoints(x, y, other.x,
other.y)
}

object Point {
    private val grid = new Grid()

    def distanceBetweenPoints(x1: Double, y1:
Double,
        x2: Double, y2: Double) = {
        math.hypot(x1 - x2, y1 - y2)
    }
}
```

```

    }
    return Math.hypot(x1 - x2, y1 - y2);
}

```

The code above shows some of the conceptual differences between Java and Scala's handling of classes:

- Scala has no static variables or methods. Instead, it has *singleton objects*, which are essentially classes with only one instance. Singleton objects are declared using `object` instead of `class`. It is common to place static variables and methods in a singleton object with the same name as the class name, which is then known as a *companion object*.<sup>[13]</sup> (The underlying class for the singleton object has a `$` appended. Hence, for `class Foo` with companion object `object Foo`, under the hood there's a class `Foo$` containing the companion object's code, and one object of this class is created, using the singleton pattern.)
- In place of constructor parameters, Scala has *class parameters*, which are placed on the class, similar to parameters to a function. When declared with a `val` or `var` modifier, fields are also defined with the same name, and automatically initialized from the class parameters. (Under the hood, external access to public fields always goes through accessor (getter) and mutator (setter) methods, which are automatically created. The accessor function has the same name as the field, which is why it's unnecessary in the above example to explicitly declare accessor methods.) Note that alternative constructors can also be declared, as in Java. Code that would go into the default constructor (other than initializing the member variables) goes directly at class level.
- Default visibility in Scala is `public`.

## Features (with reference to Java)

---

Scala has the same compiling model as Java and C#, namely separate compiling and dynamic class loading, so that Scala code can call Java libraries.

Scala's operational characteristics are the same as Java's. The Scala compiler generates byte code that is nearly identical to that generated by the Java compiler.<sup>[13]</sup> In fact, Scala code can be decompiled to readable Java code, with the exception of certain constructor operations. To the Java virtual machine (JVM), Scala code and Java code are indistinguishable. The only difference is one extra runtime library, `scala-library.jar`.<sup>[28]</sup>

Scala adds a large number of features compared with Java, and has some fundamental differences in its underlying model of expressions and types, which make the language theoretically cleaner and eliminate several *corner cases* in Java. From the Scala perspective, this is practically important because several added features in Scala are also available in C#. Examples include:

### Syntactic flexibility

As mentioned above, Scala has a good deal of syntactic flexibility, compared with Java. The following are some examples:

- Semicolons are unnecessary; lines are automatically joined if they begin or end with a token that cannot normally come in this position, or if there are unclosed parentheses or brackets.
- Any method can be used as an infix operator, e.g. `"%d apples".format(num)` and `"%d apples" format num` are equivalent. In fact, arithmetic operators like `+` and `<<` are treated just like any other methods, since function names are allowed to consist of sequences of arbitrary symbols (with a few exceptions made for things like parens, brackets and braces that

must be handled specially); the only special treatment that such symbol-named methods undergo concerns the handling of precedence.

- Methods apply and update have syntactic short forms. `foo()`—where `foo` is a value (singleton object or class instance)—is short for `foo.apply()`, and `foo() = 42` is short for `foo.update(42)`. Similarly, `foo(42)` is short for `foo.apply(42)`, and `foo(4) = 2` is short for `foo.update(4, 2)`. This is used for collection classes and extends to many other cases, such as [STM](#) cells.
- Scala distinguishes between no-parens (`def foo = 42`) and empty-parens (`def foo() = 42`) methods. When calling an empty-parens method, the parentheses may be omitted, which is useful when calling into Java libraries that do not know this distinction, e.g., using `foo.toString` instead of `foo.toString()`. By convention, a method should be defined with empty-parens when it performs [side effects](#).
- Method names ending in colon (`:`) expect the argument on the left-hand-side and the receiver on the right-hand-side. For example, the `4 :: 2 :: Nil` is the same as `Nil :: (2) :: (4)`, the first form corresponding visually to the result (a list with first element 4 and second element 2).
- Class body variables can be transparently implemented as separate getter and setter methods. For `trait FooLike { var bar: Int }`, an implementation may be `object Foo extends FooLike { private var x = 0; def bar = x; def bar_=(value: Int) { x = value } }`. The call site will still be able to use a concise `foo.bar = 42`.
- The use of curly braces instead of parentheses is allowed in method calls. This allows pure library implementations of new control structures.<sup>[29]</sup> For example, `breakable { ... if (...) break() ... }` looks as if `breakable` was a language defined keyword, but really is just a method taking a [thunk](#) argument. Methods that take thunks or functions often place these in a second parameter list, allowing to mix parentheses and curly braces syntax: `Vector.fill(4) { math.random }` is the same as `Vector.fill(4) (math.random)`. The curly braces variant allows the expression to span multiple lines.
- For-expressions (explained further down) can accommodate any type that defines monadic methods such as `map`, `flatMap` and `filter`.

By themselves, these may seem like questionable choices, but collectively they serve the purpose of allowing [domain-specific languages](#) to be defined in Scala without needing to extend the compiler. For example, [Erlang's](#) special syntax for sending a message to an actor, i.e. `actor ! message` can be (and is) implemented in a Scala library without needing language extensions.

## Unified type system

Java makes a sharp distinction between primitive types (e.g. `int` and `boolean`) and reference types (any [class](#)). Only reference types are part of the inheritance scheme, deriving from `java.lang.Object`. In Scala, all types inherit from a top-level class `Any`, whose immediate children are `AnyVal` (value types, such as `Int` and `Boolean`) and `AnyRef` (reference types, as in Java). This means that the Java distinction between primitive types and boxed types (e.g. `int` vs. `Integer`) is not present in Scala; boxing and unboxing is completely transparent to the user. Scala 2.10 allows for new value types to be defined by the user.

## For-expressions



Instead of the Java "foreach" loops for looping through an iterator, Scala has `for`-expressions, which are similar to list comprehensions in languages such as Haskell, or a combination of list comprehensions and generator expressions in Python. For-expressions using the `yield` keyword allow a new collection to be generated by iterating over an existing one, returning a new collection of the same type. They are translated by the compiler into a series of `map`, `flatMap` and `filter` calls. Where `yield` is not used, the code approximates to an imperative-style loop, by translating to `foreach`.

A simple example is:

```
val s = for (x <- 1 to 25 if x*x > 50) yield 2*x
```

The result of running it is the following vector:

```
Vector(16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50)
```

(Note that the expression `1 to 25` is not special syntax. The method `to` is rather defined in the standard Scala library as an extension method on integers, using a technique known as implicit conversions<sup>[30]</sup> that allows new methods to be added to existing types.)

A more complex example of iterating over a map is:

```
// Given a map specifying Twitter users mentioned in a set of tweets,
// and number of times each user was mentioned, look up the users
// in a map of known politicians, and return a new map giving only the
// Democratic politicians (as objects, rather than strings).
val dem_mentions = for {
  (mention, times) <- mentions
  account <- accounts.get(mention)
  if account.party == "Democratic"
} yield (account, times)
```

Expression `(mention, times) <- mentions` is an example of pattern matching (see below). Iterating over a map returns a set of key-value tuples, and pattern-matching allows the tuples to easily be deconstructed into separate variables for the key and value. Similarly, the result of the comprehension also returns key-value tuples, which are automatically built back up into a map because the source object (from the variable `mentions`) is a map. Note that if `mentions` instead held a list, set, array or other collection of tuples, exactly the same code above would yield a new collection of the same type.

## Functional tendencies

While supporting all of the object-oriented features available in Java (and in fact, augmenting them in various ways), Scala also provides a large number of capabilities that are normally found only in functional programming languages. Together, these features allow Scala programs to be written in an almost completely functional style and also allow functional and object-oriented styles to be mixed.

Examples are:

- No distinction between statements and expressions
- Type inference
- Anonymous functions with capturing semantics (i.e., closures)
- Immutable variables and objects
- Lazy evaluation
- Delimited continuations (since 2.8)



- Higher-order functions
- Nested functions
- Currying
- Pattern matching
- Algebraic data types (through *case classes*)
- Tuples

## Everything is an expression

Unlike C or Java, but similar to languages such as Lisp, Scala makes no distinction between statements and expressions. All statements are in fact expressions that evaluate to some value. Functions that would be declared as returning `void` in C or Java, and statements like `while` that logically do not return a value, are in Scala considered to return the type `Unit`, which is a singleton type, with only one object of that type. Functions and operators that never return at all (e.g. the `throw` operator or a function that always exits non-locally using an exception) logically have return type `Nothing`, a special type containing no objects; that is, a bottom type, i.e. a subclass of every possible type. (This in turn makes type `Nothing` compatible with every type, allowing type inference to function correctly.)

Similarly, an `if-then-else` "statement" is actually an expression, which produces a value, i.e. the result of evaluating one of the two branches. This means that such a block of code can be inserted wherever an expression is desired, obviating the need for a ternary operator in Scala:

<pre>// Java: int hexDigit = x &gt;= 10 ? x + 'A' - 10 : x + '0';</pre>	<pre>// Scala: val hexDigit = if (x &gt;= 10) x + 'A' - 10 else x + '0'</pre>
---	---

For similar reasons, `return` statements are unnecessary in Scala, and in fact are discouraged. As in Lisp, the last expression in a block of code is the value of that block of code, and if the block of code is the body of a function, it will be returned by the function.

To make it clear that all functions are expressions, even methods that return `Unit` are written with an equals sign

```
def printValue(x: String): Unit = {
  println("I ate a %s".format(x))
}
```

or equivalently (with type inference, and omitting the unnecessary braces):

```
def printValue(x: String) = println("I ate a %s" format x)
```

## Type inference

Due to type inference, the type of variables, function return values, and many other expressions can typically be omitted, as the compiler can deduce it. Examples are `val x = "foo"` (for an immutable constant or immutable object) or `var x = 1.5` (for a variable whose value can later be changed). Type inference in Scala is essentially local, in contrast to the more global Hindley-Milner algorithm used in Haskell, ML and

other more purely functional languages. This is done to facilitate object-oriented programming. The result is that certain types still need to be declared (most notably, function parameters, and the return types of recursive functions), e.g.

```
def formatApples(x: Int) = "I ate %d apples".format(x)
```

or (with a return type declared for a recursive function)

```
def factorial(x: Int): Int =  
  if (x == 0)  
    1  
  else  
    x*factorial(x - 1)
```

## Anonymous functions

In Scala, functions are objects, and a convenient syntax exists for specifying anonymous functions. An example is the expression `x => x < 2`, which specifies a function with one parameter, that compares its argument to see if it is less than 2. It is equivalent to the Lisp form `(lambda (x) (< x 2))`. Note that neither the type of `x` nor the return type need be explicitly specified, and can generally be inferred by type inference; but they can be explicitly specified, e.g. as `(x: Int) => x < 2` or even `(x: Int) => (x < 2): Boolean`.

Anonymous functions behave as true closures in that they automatically capture any variables that are lexically available in the environment of the enclosing function. Those variables will be available even after the enclosing function returns, and unlike in the case of Java's *anonymous inner classes* do not need to be declared as `final`. (It is even possible to modify such variables if they are mutable, and the modified value will be available the next time the anonymous function is called.)

An even shorter form of anonymous function uses placeholder variables: For example, the following:

```
list map { x => sqrt(x) }
```

can be written more concisely as

```
list map { sqrt(_) }
```

or even

```
list map sqrt
```

## Immutability

Scala enforces a distinction between immutable and mutable variables. Mutable variables are declared using the `var` keyword and immutable values are declared using the `val` keyword. A variable declared using the `val` keyword can not be reassigned in the same way that a variable declared using the `final` keyword can't be reassigned in Java. It should be noted however that `val`'s are only shallowly immutable, that is, an object referenced by a `val` is not guaranteed to itself be immutable.

Immutable classes are encouraged by convention however, and the Scala standard library provides a rich set of immutable collection classes. Scala provides mutable and immutable variants of most collection classes, and the immutable version is always used unless the mutable version is explicitly imported.<sup>[31]</sup> The immutable

variants are persistent data structures that always return an updated copy of an old object instead of updating the old object destructively in place. An example of this is immutable linked lists where prepending an element to a list is done by returning a new list node consisting of the element and a reference to the list tail. Appending an element to a list can only be done by prepending all elements in the old list to a new list with only the new element. In the same way, inserting an element in the middle of a list will copy the first half of the list, but keep a reference to the second half of the list. This is called structural sharing. This allows for very easy concurrency — no locks are needed as no shared objects are ever modified.<sup>[32]</sup>

## Lazy (non-strict) evaluation

Evaluation is strict ("eager") by default. In other words, Scala evaluates expressions as soon as they are available, rather than as needed. However, it is possible to declare a variable non-strict ("lazy") with the `lazy` keyword, meaning that the code to produce the variable's value will not be evaluated until the first time the variable is referenced. Non-strict collections of various types also exist (such as the type `Stream`, a non-strict linked list), and any collection can be made non-strict with the `view` method. Non-strict collections provide a good semantic fit to things like server-produced data, where the evaluation of the code to generate later elements of a list (that in turn triggers a request to a server, possibly located somewhere else on the web) only happens when the elements are actually needed.

## Tail recursion

Functional programming languages commonly provide tail call optimization to allow for extensive use of recursion without stack overflow problems. Limitations in Java bytecode complicate tail call optimization on the JVM. In general, a function that calls itself with a tail call can be optimized, but mutually recursive functions cannot. Trampolines have been suggested as a workaround.<sup>[33]</sup> Trampoline support has been provided by the Scala library with the object `scala.util.control.TailCalls` since Scala 2.8.0 (released 14 July 2010). A function may optionally be annotated with `@tailrec`, in which case it will not compile unless it is tail recursive.<sup>[34]</sup>

## Case classes and pattern matching

Scala has built-in support for pattern matching, which can be thought of as a more sophisticated, extensible version of a switch statement, where arbitrary data types can be matched (rather than just simple types like integers, booleans and strings), including arbitrary nesting. A special type of class known as a *case class* is provided, which includes automatic support for pattern matching and can be used to model the algebraic data types used in many functional programming languages. (From the perspective of Scala, a case class is simply a normal class for which the compiler automatically adds certain behaviors that could also be provided manually, e.g., definitions of methods providing for deep comparisons and hashing, and destructuring a case class on its constructor parameters during pattern matching.)

An example of a definition of the quicksort algorithm using pattern matching is this:

```
def qsort(list: List[Int]): List[Int] = list match {  
  case Nil => Nil  
  case pivot :: tail =>  
    val (smaller, rest) = tail.partition(_ < pivot)  
    qsort(smaller) ::: pivot :: qsort(rest)  
}
```

The idea here is that we partition a list into the elements less than a pivot and the elements not less, recursively sort each part, and paste the results together with the pivot in between. This uses the same divide-and-conquer strategy of mergesort and other fast sorting algorithms.

The `match` operator is used to do pattern matching on the object stored in `list`. Each `case` expression is tried in turn to see if it will match, and the first match determines the result. In this case, `Nil` only matches the literal object `Nil`, but `pivot :: tail` matches a non-empty list, and simultaneously *destructures* the list according to the pattern given. In this case, the associated code will have access to a local variable named `pivot` holding the head of the list, and another variable `tail` holding the tail of the list. Note that these variables are read-only, and are semantically very similar to variable bindings established using the let operator in Lisp and Scheme.

Pattern matching also happens in local variable declarations. In this case, the return value of the call to `tail.partition` is a tuple — in this case, two lists. (Tuples differ from other types of containers, e.g. lists, in that they are always of fixed size and the elements can be of differing types — although here they are both the same.) Pattern matching is the easiest way of fetching the two parts of the tuple.

The form `_ < pivot` is a declaration of an anonymous function with a placeholder variable; see the section above on anonymous functions.

The list operators `::` (which adds an element onto the beginning of a list, similar to `cons` in Lisp and Scheme) and `:::` (which appends two lists together, similar to `append` in Lisp and Scheme) both appear. Despite appearances, there is nothing "built-in" about either of these operators. As specified above, any string of symbols can serve as function name, and a method applied to an object can be written "infix"-style without the period or parentheses. The line above as written:

```
qsort(smaller) ::: pivot :: qsort(rest)
```

could also be written thus:

```
qsort(rest).:::(pivot).:::(qsort(smaller))
```

in more standard method-call notation. (Methods that end with a colon are right-associative and bind to the object to the right.)

## Partial functions

In the pattern-matching example above, the body of the `match` operator is a partial function, which consists of a series of `case` expressions, with the first matching expression prevailing, similar to the body of a switch statement. Partial functions are also used in the exception-handling portion of a `try` statement:

```
try {  
  ...  
} catch {  
  case nfe:NumberFormatException => { println(nfe); List(0) }  
  case _ => Nil  
}
```

Finally, a partial function can be used alone, and the result of calling it is equivalent to doing a `match` over it. For example, the prior code for quicksort can be written thus:

```
val qsort: List[Int] => List[Int] = {  
  case Nil => Nil  
  case pivot :: tail =>
```

```

    val (smaller, rest) = tail.partition(_ < pivot)
    qsort(smaller) ::: pivot :: qsort(rest)
  }

```

Here a read-only *variable* is declared whose type is a function from lists of integers to lists of integers, and bind it to a partial function. (Note that the single parameter of the partial function is never explicitly declared or named.) However, we can still call this variable exactly as if it were a normal function:

```

scala> qsort(List(6,2,5,9))
res32: List[Int] = List(2, 5, 6, 9)

```

## Object-oriented extensions

Scala is a pure object-oriented language in the sense that every value is an object. Data types and behaviors of objects are described by classes and traits. Class abstractions are extended by subclassing and by a flexible mixin-based composition mechanism to avoid the problems of multiple inheritance.

Traits are Scala's replacement for Java's interfaces. Interfaces in Java versions under 8 are highly restricted, able only to contain abstract function declarations. This has led to criticism that providing convenience methods in interfaces is awkward (the same methods must be reimplemented in every implementation), and extending a published interface in a backwards-compatible way is impossible. Traits are similar to mixin classes in that they have nearly all the power of a regular abstract class, lacking only class parameters (Scala's equivalent to Java's constructor parameters), since traits are always mixed in with a class. The `super` operator behaves specially in traits, allowing traits to be chained using composition in addition to inheritance. The following example is a simple window system:

```

abstract class Window {
  // abstract
  def draw()
}

class SimpleWindow extends Window {
  def draw() {
    println("in SimpleWindow")
    // draw a basic window
  }
}

trait WindowDecoration extends Window { }

trait HorizontalScrollbarDecoration extends WindowDecoration {
  // "abstract override" is needed here in order for "super()" to work because the parent
  // function is abstract. If it were concrete, regular "override" would be enough.
  abstract override def draw() {
    println("in HorizontalScrollbarDecoration")
    super.draw()
    // now draw a horizontal scrollbar
  }
}

trait VerticalScrollbarDecoration extends WindowDecoration {
  abstract override def draw() {
    println("in VerticalScrollbarDecoration")
    super.draw()
    // now draw a vertical scrollbar
  }
}

trait TitleDecoration extends WindowDecoration {
  abstract override def draw() {
    println("in TitleDecoration")
    super.draw()
    // now draw the title bar
  }
}

```

A variable may be declared thus:

```
val mywin = new SimpleWindow with VerticalScrollbarDecoration with HorizontalScrollbarDecoration
with TitleDecoration
```

The result of calling `mywin.draw()` is:

```
in TitleDecoration
in HorizontalScrollbarDecoration
in VerticalScrollbarDecoration
in SimpleWindow
```

In other words, the call to `draw` first executed the code in `TitleDecoration` (the last trait mixed in), then (through the `super()` calls) threaded back through the other mixed-in traits and eventually to the code in `Window`, *even though none of the traits inherited from one another*. This is similar to the decorator pattern, but is more concise and less error-prone, as it doesn't require explicitly encapsulating the parent window, explicitly forwarding functions whose implementation isn't changed, or relying on run-time initialization of entity relationships. In other languages, a similar effect could be achieved at compile-time with a long linear chain of implementation inheritance, but with the disadvantage compared to Scala that one linear inheritance chain would have to be declared for each possible combination of the mix-ins.

## Expressive type system

Scala is equipped with an expressive static type system that mostly enforces the safe and coherent use of abstractions. The type system is, however, not sound.<sup>[35]</sup> In particular, the type system supports:

- Classes and abstract types as object members
- Structural types
- Path-dependent types
- Compound types
- Explicitly typed self references
- Generic classes
- Polymorphic methods
- Upper and lower type bounds
- Variance
- Annotation
- Views

Scala is able to infer types by usage. This makes most static type declarations optional. Static types need not be explicitly declared unless a compiler error indicates the need. In practice, some static type declarations are included for the sake of code clarity.

## Type enrichment

A common technique in Scala, known as "enrich my library"<sup>[36]</sup> (originally termed "pimp my library" by Martin Odersky in 2006;<sup>[30]</sup> concerns were raised about this phrasing due to its negative connotations<sup>[37]</sup> and immaturity<sup>[38]</sup>), allows new methods to be used as if they were added to existing types. This is similar to the C# concept of extension methods but more powerful, because the technique is not limited to adding methods and can, for instance, be used to implement new interfaces. In Scala, this technique involves declaring an

implicit conversion from the type "receiving" the method to a new type (typically, a class) that wraps the original type and provides the additional method. If a method cannot be found for a given type, the compiler automatically searches for any applicable implicit conversions to types that provide the method in question.

This technique allows new methods to be added to an existing class using an add-on library such that only code that *imports* the add-on library gets the new functionality, and all other code is unaffected.

The following example shows the enrichment of type `Int` with methods `isEven` and `isOdd`:

```
object MyExtensions {  
  implicit class IntPredicates(i: Int) {  
    def isEven = i % 2 == 0  
    def isOdd  = !isEven  
  }  
}  
  
import MyExtensions._ // bring implicit enrichment into scope  
4.isEven             // -> true
```

Importing the members of `MyExtensions` brings the implicit conversion to extension class `IntPredicates` into scope.<sup>[39]</sup>

## Concurrency

Scala's standard library includes support for the actor model, in addition to the standard Java concurrency APIs. Lightbend Inc. provides a platform<sup>[40]</sup> that includes Akka,<sup>[41]</sup> a separate open-source framework that provides actor-based concurrency. Akka actors may be distributed or combined with software transactional memory (*transactors*). Alternative communicating sequential processes (CSP) implementations for channel-based message passing are Communicating Scala Objects,<sup>[42]</sup> or simply via JCSP.

An Actor is like a thread instance with a mailbox. It can be created by `system.actorOf`, overriding the `receive` method to receive messages and using the `!` (exclamation point) method to send a message.<sup>[43]</sup> The following example shows an `EchoServer` that can receive messages and then print them.

```
val echoServer = actor(new Act {  
  become {  
    case msg => println("echo " + msg)  
  }  
})  
echoServer ! "hi"
```

Scala also comes with built-in support for data-parallel programming in the form of Parallel Collections<sup>[44]</sup> integrated into its Standard Library since version 2.9.0.

The following example shows how to use Parallel Collections to improve performance.<sup>[45]</sup>

```
val urls = List("https://scala-lang.org", "https://github.com/scala/scala")  
  
def fromURL(url: String) = scala.io.Source.fromURL(url)  
  .getLines().mkString("\n")  
  
val t = System.currentTimeMillis()  
urls.par.map(fromURL(_)) // par returns parallel implementation of a collection  
println("time: " + (System.currentTimeMillis - t) + "ms")
```

Besides actor support and data-parallelism, Scala also supports asynchronous programming with Futures and Promises, software transactional memory, and event streams.<sup>[46]</sup>



## Cluster computing

---

The most well-known open-source cluster-computing solution written in Scala is [Apache Spark](#). Additionally, [Apache Kafka](#), the [publish-subscribe message queue](#) popular with Spark and other stream processing technologies, is written in Scala.

## Testing

---

There are several ways to test code in Scala. [ScalaTest](#) supports multiple testing styles and can integrate with Java-based testing frameworks.<sup>[47]</sup> [ScalaCheck](#) is a library similar to Haskell's [QuickCheck](#).<sup>[48]</sup> [specs2](#) is a library for writing executable software specifications.<sup>[49]</sup> [ScalaMock](#) provides support for testing high-order and curried functions.<sup>[50]</sup> [JUnit](#) and [TestNG](#) are popular testing frameworks written in Java.

## Versions

---

Version	Released	Features	Status
<b>1.0.0-b2</b> <sup>[51]</sup>	8-Dec-2003	–	–
<b>1.1.0-b1</b> <sup>[51]</sup>	19-Feb-2004	<ul style="list-style-type: none"> <li>scala Enumeration</li> <li>Scala license was changed to the revised BSD license</li> </ul>	–
<b>1.1.1</b> <sup>[51]</sup>	23-Mar-2004	<ul style="list-style-type: none"> <li>Support for Java static inner classes</li> <li>Library class improvements to Iterable, Array, xml.Elem, Buffer</li> </ul>	–
<b>1.2.0</b> <sup>[51]</sup>	9-Jun-2004	<ul style="list-style-type: none"> <li>Views</li> <li>XML Literals</li> </ul>	–
<b>1.3.0</b> <sup>[51]</sup>	16-Sep-2004	<ul style="list-style-type: none"> <li>Support for Microsoft .NET</li> <li>Method closures</li> <li>Type syntax for parameterless methods changed from [ ] T to =&gt; T</li> </ul>	–
<b>1.4.0</b> <sup>[51]</sup>	20-Jun-2005	<ul style="list-style-type: none"> <li>Attributes</li> <li>match keyword replaces match method</li> <li>Experimental support for runtime types</li> </ul>	–
<b>2.0</b> <sup>[52]</sup>	12-Mar-2006	<ul style="list-style-type: none"> <li>Compiler completely rewritten in Scala</li> <li>Experimental support for Java generics</li> <li>implicit and requires keywords</li> <li>match keyword only allowed infix</li> <li>with connective is only allowed following an extends clause</li> <li>Newlines can be used as statement separators in place of semicolons</li> <li>Regular expression match patterns restricted to sequence patterns only</li> <li>For-comprehensions admit value and pattern definitions</li> <li>Class parameters may be prefixed by val or var</li> <li>Private visibility has qualifiers</li> </ul>	–
<b>2.1.0</b> <sup>[51]</sup>	17-Mar-2006	<ul style="list-style-type: none"> <li>sbaz tool integrated in the Scala distribution</li> <li>match keyword replaces match method</li> <li>Experimental support for runtime types</li> </ul>	–
<b>2.1.8</b> <sup>[53]</sup>	23-Aug-2006	<ul style="list-style-type: none"> <li>Protected visibility has qualifiers</li> <li>Private members of a class can be referenced from the companion module of the class and vice versa</li> <li>Implicit lookup generalised</li> <li>Typed pattern match tightened for singleton types</li> </ul>	–
<b>2.3.0</b> <sup>[54]</sup>	23-Nov-2006	<ul style="list-style-type: none"> <li>Functions returning Unit don't have to explicitly state a return type</li> <li>Type variables and types are distinguished between in pattern matching</li> <li>All and AllRef renamed to Nothing and Null</li> </ul>	–
<b>2.4.0</b> <sup>[55]</sup>	09-Mar-2007	<ul style="list-style-type: none"> <li>private and protected modifiers accept a [this] qualifier</li> <li>Tuples can be written with round brackets</li> <li>Primary constructor of a class can now be marked private or protected</li> <li>Attributes changed to annotations with new syntax</li> <li>Self aliases</li> </ul>	–

		<ul style="list-style-type: none"> <li>▪ Operators can be combined with assignment</li> </ul>	
2.5.0 <sup>[56]</sup>	02-May-2007	<ul style="list-style-type: none"> <li>▪ Type parameters and abstract type members can also abstract over type constructors</li> <li>▪ Fields of an object can be initialized before parent constructors are called</li> <li>▪ Syntax change for comprehensions</li> <li>▪ Implicit anonymous functions (with underscores for parameters)</li> <li>▪ Pattern matching of anonymous functions extended to support any arity</li> </ul>	—
2.6.0 <sup>[57]</sup>	27-Jul-2007	<ul style="list-style-type: none"> <li>▪ Existential types</li> <li>▪ Lazy values</li> <li>▪ Structural types</li> </ul>	—
2.7.0 <sup>[58]</sup>	07-Feb-2008	<ul style="list-style-type: none"> <li>▪ Java generic types supported by default</li> <li>▪ Case classes functionality extended</li> </ul>	—
2.8.0 <sup>[59]</sup>	14-Jul-2010	<ul style="list-style-type: none"> <li>▪ Revision the common, uniform, and all-encompassing framework for collection types.</li> <li>▪ Type specialisation</li> <li>▪ Named and <u>default arguments</u></li> <li>▪ Package objects</li> <li>▪ Improved annotations</li> </ul>	—
2.9.0 <sup>[60]</sup>	12-May-2011	<ul style="list-style-type: none"> <li>▪ Parallel collections</li> <li>▪ Thread safe App trait replaces Application trait</li> <li>▪ DelayedInit trait</li> <li>▪ Java Interop improvements</li> </ul>	—
2.10 <sup>[61]</sup>	04-Jan-2013	<ul style="list-style-type: none"> <li>▪ Value Classes<sup>[62]</sup></li> <li>▪ Implicit Classes<sup>[63]</sup></li> <li>▪ String Interpolation<sup>[64]</sup></li> <li>▪ Futures and Promises<sup>[65]</sup></li> <li>▪ Dynamic and applyDynamic<sup>[66]</sup></li> <li>▪ Dependent method types: <ul style="list-style-type: none"> <li>▪ <code>def identity(x: AnyRef): x.type = x // the return type says we return exactly what we got</code></li> </ul> </li> <li>▪ New ByteCode emitter based on ASM: <ul style="list-style-type: none"> <li>▪ Can target JDK 1.5, 1.6 and 1.7</li> <li>▪ Emits 1.6 bytecode by default</li> <li>▪ Old 1.5 backend is deprecated</li> </ul> </li> <li>▪ A new Pattern Matcher: rewritten from scratch to generate more robust code (no more exponential blow-up!) <ul style="list-style-type: none"> <li>▪ code generation and analyses are now independent (the latter can be turned off with -Xno-patmat-analysis)</li> </ul> </li> <li>▪ Scaladoc Improvements <ul style="list-style-type: none"> <li>▪ Implicits (-implicits flag)</li> <li>▪ Diagrams (-diagrams flag, requires graphviz)</li> <li>▪ Groups (-groups)</li> </ul> </li> <li>▪ Modularized Language features<sup>[67]</sup></li> <li>▪ Parallel Collections<sup>[68]</sup> are now configurable with custom thread pools</li> </ul>	—

		<ul style="list-style-type: none"> <li>▪ Akka Actors now part of the distribution <ul style="list-style-type: none"> <li>▪ <code>scala.actors</code> have been deprecated and the akka implementation is now included in the distribution.</li> </ul> </li> <li>▪ Performance Improvements <ul style="list-style-type: none"> <li>▪ Faster inliner</li> <li>▪ <code>Range#sum</code> is now <math>O(1)</math></li> </ul> </li> <li>▪ Update of ForkJoin library</li> <li>▪ Fixes in immutable TreeSet/TreeMap</li> <li>▪ Improvements to PartialFunctions</li> <li>▪ Addition of ??? and NotImplementedError</li> <li>▪ Addition of IsTraversableOnce + IsTraversableLike type classes for extension methods</li> <li>▪ Deprecations and cleanup</li> <li>▪ Floating point and octal literal syntax deprecation</li> <li>▪ Removed <code>scala.dbc</code></li> </ul> <p>Experimental features</p> <ul style="list-style-type: none"> <li>▪ Scala Reflection<sup>[69]</sup></li> <li>▪ Macros<sup>[70]</sup></li> </ul>	
<b>2.10.2</b> <sup>[71]</sup>	06-Jun-2013	—	—
<b>2.10.3</b> <sup>[72]</sup>	01-Oct-2013	—	—
<b>2.10.4</b> <sup>[73]</sup>	18-Mar-2014	—	—
<b>2.10.5</b> <sup>[74]</sup>	05-Mar-2015	—	—
<b>2.11.0</b> <sup>[75]</sup>	21-Apr-2014	<ul style="list-style-type: none"> <li>▪ Collection performance improvements</li> <li>▪ Compiler performance improvements</li> </ul>	—
<b>2.11.1</b> <sup>[76]</sup>	20-May-2014	—	—
<b>2.11.2</b> <sup>[77]</sup>	22-Jul-2014	—	—
<b>2.11.4</b> <sup>[78]</sup>	31-Oct-2014	—	—
<b>2.11.5</b> <sup>[79]</sup>	08-Jan-2015	—	—
<b>2.11.6</b> <sup>[80]</sup>	05-Mar-2015	—	—
<b>2.11.7</b> <sup>[81]</sup>	23-Jun-2015	—	—
<b>2.11.8</b> <sup>[82]</sup>	08-Mar-2016	—	—
<b>2.11.11</b> <sup>[83]</sup>	18-Apr-2017	—	—
<b>2.11.12</b> <sup>[84]</sup>	13-Nov-2017	—	—

<b>2.12.0</b> <sup>[85]</sup>	03-Nov-2016	<ul style="list-style-type: none"> <li>■ Java 8 required</li> <li>■ Java 8 bytecode generated</li> <li>■ Java 8 SAM (Functional interface) language support</li> </ul>	—
<b>2.12.1</b> <sup>[86]</sup>	05-Dec-2016	—	—
<b>2.12.2</b> <sup>[87]</sup>	18-Apr-2017	—	—
<b>2.12.3</b> <sup>[88]</sup>	26-Jul-2017	—	—
<b>2.12.4</b> <sup>[89]</sup>	17-Oct-2017	—	—
<b>2.12.5</b> <sup>[90]</sup>	15-Mar-2018	—	—
<b>2.12.6</b> <sup>[91]</sup>	27-Apr-2018	—	—
<b>2.12.7</b> <sup>[92]</sup>	27-Sep-2018	—	—
<b>2.12.8</b> <sup>[93]</sup>	04-Dec-2018	First Scala 2.12 release with the license changed to Apache v2.0	—
<b>2.13.0</b> <sup>[94]</sup>	11-Jun-2019	—	Current

## Comparison with other JVM languages

---

Scala is often compared with Groovy and Clojure, two other programming languages also using the JVM. Substantial differences between these languages are found in the type system, in the extent to which each language supports object-oriented and functional programming, and in the similarity of their syntax to the syntax of Java.

Scala is statically typed, while both Groovy and Clojure are dynamically typed. This makes the type system more complex and difficult to understand but allows almost all<sup>[35]</sup> type errors to be caught at compile-time and can result in significantly faster execution. By contrast, dynamic typing requires more testing to ensure program correctness and is generally slower in order to allow greater programming flexibility and simplicity. Regarding speed differences, current versions of Groovy and Clojure allow for optional type annotations to help programs avoid the overhead of dynamic typing in cases where types are practically static. This overhead is further reduced when using recent versions of the JVM, which has been enhanced with an *invoke dynamic* instruction for methods that are defined with dynamically typed arguments. These advances reduce the speed gap between static and dynamic typing, although a statically typed language, like Scala, is still the preferred choice when execution efficiency is very important.

Regarding programming paradigms, Scala inherits the object-oriented model of Java and extends it in various ways. Groovy, while also strongly object-oriented, is more focused in reducing verbosity. In Clojure, object-oriented programming is deemphasised with functional programming being the main strength of the language. Scala also has many functional programming facilities, including features found in advanced functional languages like Haskell, and tries to be agnostic between the two paradigms, letting the developer choose between the two paradigms or, more frequently, some combination thereof.

Regarding syntax similarity with Java, Scala inherits much of Java's syntax, as is the case with Groovy. Clojure on the other hand follows the Lisp syntax, which is different in both appearance and philosophy. However, learning Scala is also considered difficult because of its many advanced features. This is not the case

with Groovy, despite its also being a feature-rich language, mainly because it was designed to be mainly a scripting language.

## Adoption

---

### Language rankings

As of 2013, all JVM-based languages (Clojure, Groovy, Kotlin, Scala) are significantly less popular than the original Java language, which is usually ranked first or second,<sup>[95][96]</sup> and which is also simultaneously evolving over time.

The Popularity of Programming Language Index,<sup>[97]</sup> which tracks searches for language tutorials, ranked Scala 15th in April 2018 with a small downward trend. This makes Scala the most popular JVM-based language after Java, although immediately followed by Kotlin, a JVM-based language with a strong upward trend ranked 16th.

The TIOBE index<sup>[96]</sup> of programming language popularity employs internet search engine rankings and similar publication-counting to determine language popularity. As of April 2018, it shows Scala in 34th place, having dropped four places over the last two years, but—as mentioned under "Bugs & Change Requests"—TIOBE is aware of issues with its methodology of using search terms which might not be commonly used in some programming language communities. In this ranking Scala is ahead of some functional languages like Haskell (42nd), Erlang, but below other languages like Swift (15th), Perl (16th), Go (19th) and Clojure (30th).

The ThoughtWorks Technology Radar, which is an opinion based biannual report of a group of senior technologists,<sup>[98]</sup> recommended Scala adoption in its languages and frameworks category in 2013.<sup>[99]</sup> In July 2014, this assessment was made more specific and now refers to a "Scala, the good parts", which is described as "To successfully use Scala, you need to research the language and have a very strong opinion on which parts are right for you, creating your own definition of Scala, the good parts."<sup>[100]</sup>

The RedMonk Programming Language Rankings, which establishes rankings based on the number of GitHub projects and questions asked on Stack Overflow, ranks Scala 14th.<sup>[95]</sup> Here, Scala is placed inside a second-tier group of languages—ahead of Go, PowerShell and Haskell, and behind Swift, Objective-C, Typescript and R. However, in its 2018 report, the Rankings noted a drop of Scala's rank for the third time in a row, questioning "how much of the available oxygen for Scala is consumed by Kotlin as the latter continues to rocket up these rankings".<sup>[95]</sup>

In the 2018 edition of the "State of Java" survey,<sup>[101]</sup> which collected data from 5160 developers on various Java-related topics, Scala places third in terms of usage of alternative languages on the JVM. Compared to the last year's edition of the survey, Scala's usage among alternative JVM languages fell by almost a quarter (from 28.4% to 21.5%), overtaken by Kotlin, which rose from 11.4% in 2017 to 28.8% in 2018.

### Companies

- In April 2009, Twitter announced that it had switched large portions of its backend from Ruby to Scala and intended to convert the rest.<sup>[102]</sup>
- Gilt uses Scala and Play Framework.<sup>[103]</sup>
- Foursquare uses Scala and Lift.<sup>[104]</sup>
- Coursera uses Scala and Play Framework.<sup>[105]</sup>
- Apple Inc. uses Scala in certain teams, along with Java and the Play framework.<sup>[106][107]</sup>

- *The Guardian* newspaper's high-traffic website [guardian.co.uk](https://guardian.co.uk)<sup>[108]</sup> announced in April 2011 that it was switching from Java to Scala,<sup>[109][110]</sup>
- The *New York Times* revealed in 2014 that its internal content management system *Blackbeard* is built using Scala, Akka and Play.<sup>[111]</sup>
- The *Huffington Post* newspaper started to employ Scala as part of its contents delivery system *Athena* in 2013.<sup>[112]</sup>
- Swiss bank *UBS* approved Scala for general production usage.<sup>[113]</sup>
- *LinkedIn* uses the *Scalatra* microframework to power its Signal API.<sup>[114]</sup>
- *Meetup* uses *Unfiltered* toolkit for real-time APIs.<sup>[115]</sup>
- *Remember the Milk* uses *Unfiltered* toolkit, Scala and Akka for public API and real-time updates.<sup>[116]</sup>
- *Verizon* seeking to make "a next-generation framework" using Scala.<sup>[117]</sup>
- *Airbnb* develops open-source machine-learning software "Aerosolve", written in Java and Scala.<sup>[118]</sup>
- *Zalando* moved its technology stack from Java to Scala and Play.<sup>[119]</sup>
- *SoundCloud* uses Scala for its back-end, employing technologies such as *Finagle* (micro services),<sup>[120]</sup> *Scalding* and *Spark* (data processing).<sup>[121]</sup>
- *Databricks* uses Scala for the *Apache Spark* Big Data platform.
- *Morgan Stanley* uses Scala extensively in their finance and asset-related projects.<sup>[122]</sup>
- There are teams within *Google/Alphabet Inc.* that use Scala, mostly due to acquisitions such as *Firebase*<sup>[123]</sup> and *Nest*.<sup>[124]</sup>
- *Walmart Canada* Uses Scala for their back-end platform.<sup>[125]</sup>
- *Duolingo* uses Scala for their back-end module that generates lessons.<sup>[126]</sup>
- *HMRC* uses Scala for many UK Government Tax applications.<sup>[127]</sup>

## Criticism

---

In March 2015, former VP of the Platform Engineering group at Twitter *Raffi Krikorian*, stated that he would not have chosen Scala in 2011 due to its *learning curve*.<sup>[128]</sup> The same month, *LinkedIn* SVP *Kevin Scott* stated their decision to "minimize [their] dependence on Scala".<sup>[129]</sup> In November 2011, *Yammer* moved away from Scala for reasons that included the learning curve for new team members and incompatibility from one version of the Scala compiler to the next.<sup>[130]</sup>

## See also

---

- *sbt*, a widely used build tool for Scala projects
- *Play!*, an open-source Web application framework that supports Scala
- *Akka*, an open-source toolkit for building concurrent and distributed applications
- *Chisel*, an open-source language built upon Scala that is used for hardware design and generation.<sup>[131]</sup>

## References

---

1. "Scala 2.13.3 is now available!" (<https://www.scala-lang.org/news/2.13.3>). 2020-06-25. Retrieved 2020-06-27.



2. "NOTICE file" (<https://github.com/scala/scala/blob/2.13.x/NOTICE>). 2019-01-24. Retrieved 2019-12-04 – via [GitHub](#).
3. "Scala Macros" (<http://scalamacros.org>).
4. Fogus, Michael (6 August 2010). "MartinOdersky take(5) toList" (<http://blog.fogus.me/2010/08/06/martinodersky-take5-tolist/>). *Send More Paramedics*. Retrieved 2012-02-09.
5. Odersky, Martin (11 January 2006). "The Scala Experiment - Can We Provide Better Language Support for Component Systems?" (<https://lampwww.epfl.ch/~odersky/talks/pop106.pdf>) (PDF). Retrieved 2016-06-22.
6. Odersky, Martin; et al. (2006). "An Overview of the Scala Programming Language" (<https://www.scala-lang.org/docu/files/ScalaOverview.pdf>) (PDF) (2nd ed.). École Polytechnique Fédérale de Lausanne (EPFL). Archived (<https://web.archive.org/web/20200709211816/https://www.scala-lang.org/docu/files/ScalaOverview.pdf>) (PDF) from the original on 2020-07-09.
7. Odersky, Martin (2008). *Programming in Scala* (<https://books.google.com/books?id=MFjNhTjeQKkC&q=scala%20is%20pronounced%20skah-lah&pg=PA3>). Mountain View, California: Artima. p. 3. ISBN 9780981531601. Retrieved 12 June 2014.
8. Potvin, Pascal; Bonja, Mario (24 September 2015). *An IMS DSL Developed at Ericsson*. Lecture Notes in Computer Science. **7916**. arXiv:1509.07326 (<https://arxiv.org/abs/1509.07326>). doi:10.1007/978-3-642-38911-5 (<https://doi.org/10.1007%2F978-3-642-38911-5>). ISBN 978-3-642-38910-8. S2CID 1214469 (<https://api.semanticscholar.org/CorpusID:1214469>).
9. "Frequently Asked Questions - Java Interoperability" (<https://www.scala-lang.org/old/faq/4>). *scala-lang.org*. Retrieved 2015-02-06.
10. Friesen, Jeff (16 November 2016). "Are checked exceptions good or bad?" (<https://www.javaworld.com/article/3142626/core-java/are-checked-exceptions-good-or-bad.html>). *JavaWorld*. Retrieved 28 August 2018.
11. Loverdo, Christos (2010). *Steps in Scala: An Introduction to Object-Functional Programming* ([https://books.google.com/books?id=vZAfN\\_Vk2i0C&q=%22steps+in+scala%22&pg=PR13](https://books.google.com/books?id=vZAfN_Vk2i0C&q=%22steps+in+scala%22&pg=PR13)). Cambridge University Press. p. xiii. ISBN 9781139490948. Retrieved 31 July 2014.
12. Martin Odersky, "A Brief History of Scala" (<https://www.artima.com/weblogs/viewpost.jsp?thread=163733>), Artima.com weblogs, 9 June 2006
13. Odersky, M.; Rompf, T. (2014). "Unifying functional and object-oriented programming with Scala" (<https://doi.org/10.1145%2F2591013>). *Communications of the ACM*. **57** (4): 76. doi:10.1145/2591013 (<https://doi.org/10.1145%2F2591013>).
14. Martin Odersky, "The Scala Language Specification Version 2.7"
15. "Scala Team Wins ERC Grant" (<https://www.scala-lang.org/node/8579>). Retrieved 4 July 2015.
16. "Commercial Support for Scala" (<https://www.scala-lang.org/node/9484>). 2011-05-12. Retrieved 2011-08-18.
17. "Why We Invested in Typesafe: Modern Applications Demand Modern Tools" (<https://medium.com/@greylockvc/why-we-invested-in-typesafe-modern-applications-demand-modern-tools-b6f7deec8b89>). 2011-05-12. Retrieved 2018-05-08.
18. "Open-source Scala gains commercial backing" ([https://news.cnet.com/8301-13846\\_3-20062090-62.html](https://news.cnet.com/8301-13846_3-20062090-62.html)). 2011-05-12. Retrieved 2011-10-09.
19. "Cloud computing pioneer Martin Odersky takes wraps off his new company Typesafe" ([https://www.mercurynews.com/business/ci\\_18048434](https://www.mercurynews.com/business/ci_18048434)). 2011-05-12. Retrieved 2011-08-24.
20. "Scala on Android" (<https://scala-android.org/>). Retrieved 8 June 2016.
21. "Scala 2.12.8 is now available!" (<https://www.scala-lang.org/news/2.12.8>). 2018-12-04. Retrieved 2018-12-09.
22. "Scala Js Is No Longer Experimental | The Scala Programming Language" (<https://www.scala-lang.org/blog/2015/02/05/scala-js-no-longer-experimental.html>). Scala-lang.org. Retrieved 28 October 2015.

23. <https://github.com/scala-js/scala-js/releases>
24. Krill, Paul (15 March 2017). "Scaled-down Scala variant cuts ties to the JVM" (<https://www.info-world.com/article/3180823/application-development/scaled-down-scala-variant-cuts-ties-to-the-jvm.html>). InfoWorld. Retrieved 21 March 2017.
25. Krill, Paul (2016-05-11). "Scala language moves closer to bare metal" (<https://www.infoworld.com/article/3068669/application-development/scala-language-moves-closer-to-bare-metal.html>). InfoWorld.
26. Expunged the .net backend. by paulp · Pull Request #1718 · scala/scala · GitHub (<https://github.com/scala/scala/pull/1718>). Github.com (2012-12-05). Retrieved on 2013-11-02.
27. "Getting Started with Scala" (<https://www.scala-lang.org/old/node/166>). *scala-lang.org*. 15 July 2008. Retrieved 31 July 2014.
28. "Home" (<https://web.archive.org/web/20100831041226/http://blog.lostlake.org/index.php?%2Farchives%2F73-For-all-you-know%2C-its-just-another-Java-library.html>). Blog.lostlake.org. Archived from the original (<http://blog.lostlake.org/index.php?/archives/73-For-all-you-know,-its-just-another-Java-library.html>) on 31 August 2010. Retrieved 2013-06-25.
29. Scala's built-in control structures such as `if` or `while` cannot be re-implemented. There is a research project, Scala-Virtualized, that aimed at removing these restrictions: Adriaan Moors, Tiark Rompf, Philipp Haller and Martin Odersky. *Scala-Virtualized* (<https://dl.acm.org/citation.cfm?id=2103769>). *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, 117–120. July 2012.
30. "Pimp my Library" (<https://www.artima.com/weblogs/viewpost.jsp?thread=179766>). Artima.com. 2006-10-09. Retrieved 2013-06-25.
31. "Mutable and Immutable Collections - Scala Documentation" (<https://docs.scala-lang.org/overviews/collections/overview.html>). Retrieved 30 April 2020.
32. "Collections - Concrete Immutable Collection Classes - Scala Documentation" (<https://docs.scala-lang.org/overviews/collections/concrete-immutable-collection-classes.html>). Retrieved 4 July 2015.
33. Dougherty, Rich. "Rich Dougherty's blog" (<http://blog.richdougherty.com/2009/04/tail-calls-tailrec-and-trampolines.html>). Retrieved 4 July 2015.
34. "TailCalls - Scala Standard Library API (Scaladoc) 2.10.2 - scala.util.control.TailCalls" ([https://www.scala-lang.org/api/current/scala/util/control/TailCalls\\$.html](https://www.scala-lang.org/api/current/scala/util/control/TailCalls$.html)). Scala-lang.org. Retrieved 2013-06-25.
35. "Java and Scala's Type Systems are Unsound" (<https://raw.githubusercontent.com/namin/unsound/master/doc/unsound-oopsla16.pdf>) (PDF).
36. Giarrusso, Paolo G. (2013). "Reify your collection queries for modularity and speed!". *Proceedings of the 12th annual international conference on Aspect-oriented software development*. ACM. arXiv:1210.6284 (<https://arxiv.org/abs/1210.6284>). Bibcode:2012arXiv1210.6284G (<https://ui.adsabs.harvard.edu/abs/2012arXiv1210.6284G>). "Also known as pimp-my-library pattern"
37. Gilbert, Clint (2011-11-15). "What is highest priority for Scala to succeed in corporate world (Should be in scala-debate?) ?" (<https://www.scala-lang.org/old/node/11504%3Fpage=3.html#comment-51270>). *scala-lang.org*. Retrieved 2019-05-08.
38. "Should we "enrich" or "pimp" Scala libraries?" (<https://meta.stackexchange.com/questions/184514/should-we-enrich-or-pimp-scala-libraries>). *stackexchange.com*. 17 June 2013. Retrieved 15 April 2016.

39. Implicit classes were introduced in Scala 2.10 to make method extensions more concise. This is equivalent to adding a method `implicit def IntPredicate(i: Int) = new IntPredicate(i)`. The class can also be defined as `implicit class IntPredicates(val i: Int) extends AnyVal { ... }`, producing a so-called *value class*, also introduced in Scala 2.10. The compiler will then eliminate actual instantiations and generate static methods instead, allowing extension methods to have virtually no performance overhead.
40. "Lightbend Reactive Platform" (<https://www.lightbend.com/platform>). Lightbend. Retrieved 2016-07-15.
41. What is Akka? (<https://doc.akka.io/docs/akka/snapshot/intro/what-is-akka.html>), Akka online documentation
42. Communicating Scala Objects (<http://users.comlab.ox.ac.uk/bernard.sufrin/CSO/cpa2008-cso.pdf>), Bernard Sufrin, Communicating Process Architectures 2008
43. Yan, Kay. "Scala Tour" (<http://www.scala-tour.com/#/using-actor>). Retrieved 4 July 2015.
44. "ParallelCollections - Overview - Scala Documentation" (<https://docs.scala-lang.org/overviews/parallel-collections/overview.html>). Docs.scala-lang.org. Retrieved 2013-06-25.
45. Yan, Kay. "Scala Tour" (<http://www.scala-tour.com/#/parallel-collection>). Retrieved 4 July 2015.
46. Learning Concurrent Programming in Scala (<https://www.amazon.com/Learning-Concurrent-Programming-Aleksandar-Prokopec/dp/1783281413/>), Aleksandar Prokopec, Packt Publishing
47. Kops, Micha (2013-01-13). "A short Introduction to ScalaTest" (<https://www.hascode.com/2013/01/a-short-introduction-to-scalatest/>). *hascode.com*. Retrieved 2014-11-07.
48. Nilsson, Rickard (2008-11-17). "ScalaCheck 1.5" (<https://www.scala-lang.org/old/node/352.html>). *scala-lang.org*. Retrieved 2014-11-07.
49. "Build web applications using Scala and the Play Framework" (<http://workwithplay.com/blog/2013/05/22/testing-with-spec2/>). *workwithplay.com*. 2013-05-22. Retrieved 2014-11-07.
50. Butcher, Paul (2012-06-04). "ScalaMock 3.0 Preview Release" (<https://paulbutcher.com/2012/06/04/scalamock-3-0-preview-release/>). *paulbutcher.com*. Retrieved 2014-11-07.
51. "Scala Change History" (<https://web.archive.org/web/20071009004609/https://www.scala-lang.org/downloads/history.html>). *scala-lang.org*. Archived from the original (<https://www.scala-lang.org/downloads/history.html>) on 2007-10-09.
52. "Changes in Version 2.0 (12-Mar-2006)" (<https://www.scala-lang.org/download/changelog.html#changes-in-version-20-12-mar-2006--20->). *scala-lang.org*. 2006-03-12. Retrieved 2014-11-07.
53. "Changes in Version 2.1.8 (23-Aug-2006)" (<https://www.scala-lang.org/download/changelog.html#2.1.8>). *scala-lang.org*. 2006-08-23. Retrieved 2014-11-07.
54. "Changes in Version 2.3.0 (23-Nov-2006)" (<https://www.scala-lang.org/download/changelog.html#2.3.0>). *scala-lang.org*. 2006-11-23. Retrieved 2014-11-07.
55. "Changes in Version 2.4.0 (09-Mar-2007)" (<https://www.scala-lang.org/download/changelog.html#2.4.0>). *scala-lang.org*. 2007-03-09. Retrieved 2014-11-07.
56. "Changes in Version 2.5 (02-May-2007)" (<https://www.scala-lang.org/download/changelog.html#2.5.0>). *scala-lang.org*. 2007-05-02. Retrieved 2014-11-07.
57. "Changes in Version 2.6 (27-Jul-2007)" (<https://www.scala-lang.org/download/changelog.html#2.6.0>). *scala-lang.org*. 2007-06-27. Retrieved 2014-11-07.
58. "Changes in Version 2.7.0 (07-Feb-2008)" (<https://www.scala-lang.org/download/changelog.html#2.7.0>). *scala-lang.org*. 2008-02-07. Retrieved 2014-11-07.
59. "Changes in Version 2.8.0 (14-Jul-2010)" (<https://www.scala-lang.org/download/changelog.html#2.8.0>). *scala-lang.org*. 2010-07-10. Retrieved 2014-11-07.
60. "Changes in Version 2.9.0 (12-May-2011)" (<https://www.scala-lang.org/download/changelog.html#2.9.0>). *scala-lang.org*. 2011-05-12. Retrieved 2014-11-07.
61. "Changes in Version 2.10.0" ([https://www.scala-lang.org/download/changelog.html#changes\\_in\\_version\\_2100](https://www.scala-lang.org/download/changelog.html#changes_in_version_2100)). *scala-lang.org*. 2013-01-04. Retrieved 2014-11-07.

62. Harrah, Mark. "Value Classes and Universal Traits" (<https://docs.scala-lang.org/overviews/core/value-classes.html>). *scala-lang.org*. Retrieved 2014-11-07.
63. Suereth, Josh. "SIP-13 - Implicit classes" (<https://docs.scala-lang.org/sips/completed/implicit-classes.html>). *scala-lang.org*. Retrieved 2014-11-07.
64. Suereth, Josh. "String Interpolation" (<https://docs.scala-lang.org/overviews/core/string-interpolation.html>). *scala-lang.org*. Retrieved 2014-11-07.
65. Haller, Philipp; Prokopec, Aleksandar. "Futures and Promises" (<https://docs.scala-lang.org/overviews/core/futures.html>). *scala-lang.org*. Retrieved 2014-11-07.
66. "SIP-17 - Type Dynamic" (<https://docs.scala-lang.org/sips/completed/type-dynamic.html>). *scala-lang.org*. Retrieved 2014-11-07.
67. "SIP-18 - Modularizing Language Features" (<https://docs.scala-lang.org/sips/completed/modularizing-language-features.html>). *scala-lang.org*. Retrieved 2014-11-07.
68. Prokopec, Aleksandar; Miller, Heather. "Parallel Collections" (<https://docs.scala-lang.org/overviews/parallel-collections/overview.html>). *scala-lang.org*. Retrieved 2014-11-07.
69. Miller, Heather; Burmako, Eugene. "Reflection Overview" (<https://docs.scala-lang.org/overview/s/reflection/overview.html>). *scala-lang.org*. Retrieved 2014-11-07.
70. Burmako, Eugene. "Def Macros" (<https://docs.scala-lang.org/overviews/macros/overview.html>). *scala-lang.org*. Retrieved 2014-11-07.
71. "Scala 2.10.2 is now available!" (<https://web.archive.org/web/20141108081141/http://www.scala-lang.org/news/2013/06/06/release-notes-v2.10.2.html>). *scala-lang.org*. 2013-06-06. Archived from the original (<https://www.scala-lang.org/news/2013/06/06/release-notes-v2.10.2.html>) on 2014-11-08. Retrieved 2014-11-07.
72. "Scala 2.10.3 is now available!" (<https://web.archive.org/web/20141108081200/http://www.scala-lang.org/news/2013/10/01/release-notes-v2.10.3.html>). *scala-lang.org*. 2013-10-01. Archived from the original (<https://www.scala-lang.org/news/2013/10/01/release-notes-v2.10.3.html>) on 2014-11-08. Retrieved 2014-11-07.
73. "Scala 2.10.4 is now available!" (<https://www.scala-lang.org/news/2.10.4>). *scala-lang.org*. 2014-03-18. Retrieved 2015-01-07.
74. "Scala 2.10.5 is now available!" (<https://www.scala-lang.org/news/2.10.5>). *scala-lang.org*. 2015-03-04. Retrieved 2015-03-23.
75. "Scala 2.11.0 is now available!" (<https://www.scala-lang.org/news/2.11.0>). *scala-lang.org*. 2014-04-21. Retrieved 2014-11-07.
76. "Scala 2.11.1 is now available!" (<https://www.scala-lang.org/news/2.11.1>). *scala-lang.org*. 2014-05-20. Retrieved 2014-11-07.
77. "Scala 2.11.2 is now available!" (<https://www.scala-lang.org/news/2.11.2>). *scala-lang.org*. 2014-07-22. Retrieved 2014-11-07.
78. "Scala 2.11.4 is now available!" (<https://www.scala-lang.org/news/2.11.4>). *scala-lang.org*. 2014-10-30. Retrieved 2014-11-07.
79. "Scala 2.11.5 is now available!" (<https://www.scala-lang.org/news/2.11.5>). *scala-lang.org*. 2015-01-08. Retrieved 2015-01-22.
80. "Scala 2.11.6 is now available!" (<https://www.scala-lang.org/news/2.11.6>). *scala-lang.org*. 2015-03-05. Retrieved 2015-03-12.
81. "Scala 2.11.7 is now available!" (<https://www.scala-lang.org/news/2.11.7>). *scala-lang.org*. 2015-06-23. Retrieved 2015-07-03.
82. "Scala 2.11.8 is now available!" (<https://www.scala-lang.org/news/2.11.8>). *scala-lang.org*. 2016-03-08. Retrieved 2016-03-09.
83. "Three new releases and more GitHub goodness!" (<https://www.scala-lang.org/news/releases-1Q17.html>). *scala-lang.org*. 2017-04-18. Retrieved 2017-04-19.
84. "Security update: 2.12.4, 2.11.12, 2.10.7 (CVE-2017-15288)" (<https://www.scala-lang.org/news/security-update-nov17.html>). *scala-lang.org*. 2017-11-13. Retrieved 2018-05-04.

85. "Scala 2.12.0 is now available!" (<https://www.scala-lang.org/news/2.12.0>). *scala-lang.org*. 2016-11-03. Retrieved 2017-01-08.
86. "Scala 2.12.1 is now available!" (<https://www.scala-lang.org/news/2.12.1>). *scala-lang.org*. 2016-12-05. Retrieved 2017-01-08.
87. "Three new releases and more GitHub goodness!" (<https://www.scala-lang.org/news/releases-1Q17.html>). *scala-lang.org*. 2017-04-18. Retrieved 2017-04-19.
88. "SCALA 2.12.3 IS NOW AVAILABLE!" (<https://www.scala-lang.org/news/2.12.3>). *scala-lang.org*. 2017-07-26. Retrieved 2017-08-16.
89. "SCALA 2.12.4 IS NOW AVAILABLE!" (<https://www.scala-lang.org/news/2.12.4>). *scala-lang.org*. 2017-10-18. Retrieved 2017-10-26.
90. "SCALA 2.12.5 IS NOW AVAILABLE!" (<https://www.scala-lang.org/news/2.12.5>). *scala-lang.org*. 2018-03-15. Retrieved 2018-03-20.
91. "Scala 2.12.6 is now available!" (<https://www.scala-lang.org/news/2.12.6>). *scala-lang.org*. 2018-04-27. Retrieved 2018-05-04.
92. "Scala 2.12.7 is now available!" (<https://www.scala-lang.org/news/2.12.7>). *scala-lang.org*. 2018-09-27. Retrieved 2018-10-09.
93. "Scala 2.12.8 is now available!" (<https://www.scala-lang.org/news/2.12.8>). *scala-lang.org*. 2018-12-04. Retrieved 2018-12-09.
94. "Scala 2.13.0 is now available!" (<https://www.scala-lang.org/news/2.13.0>). *scala-lang.org*. 2019-06-11. Retrieved 2018-06-17.
95. "The RedMonk Programming Language Rankings: January 2018" (<https://redmonk.com/sograd/y/2018/03/07/language-rankings-1-18/>).
96. "TIOBE Index for April 2018" (<https://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>).
97. "Popularity of Programming Language Index" (<https://pypl.github.io/PYPL.html>).
98. "ThoughtWorks Technology Radar FAQ" (<https://martinfowler.com/articles/radar-faq.html>).
99. "ThoughtWorks Technology Radar MAY 2013" (<http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf>) (PDF).
00. "The RedMonk Programming Language Rankings: January 2018" (<https://www.thoughtworks.com/radar/languages-and-frameworks/scala-the-good-parts>).
01. "The State of Java in 2018" (<https://www.baeldung.com/java-in-2018>).
02. Greene, Kate (1 April 2009). "The Secret Behind Twitter's Growth, How a new Web programming language is helping the company handle its increasing popularity" (<https://www.technologyreview.com/blog/editors/23282/?nlid=1908>). *Technology Review*. MIT. Retrieved 6 April 2009.
03. "Play Framework, Akka and Scala at Gilt Groupe" ([https://www.lightbend.com/blog/play\\_framework\\_akka\\_and\\_scala\\_at\\_gilt](https://www.lightbend.com/blog/play_framework_akka_and_scala_at_gilt)). Lightbend. 15 July 2013. Retrieved 16 July 2016.
04. "Scala, Lift, and the Future" (<https://web.archive.org/web/20160113201402/http://www.grenadesandwich.com/blog/steven/2009/11/27/scala-lift-and-future>). Archived from the original (<http://www.grenadesandwich.com/blog/steven/2009/11/27/scala-lift-and-future>) on 13 January 2016. Retrieved 4 July 2015.
05. "Why we love Scala at Coursera" (<https://tech.coursera.org/blog/2014/02/18/why-we-love-scala-at-coursera/>). Coursera Engineering. Retrieved 4 July 2015.
06. "Apple Engineering PM Jarrod Nettles on Twitter" (<https://twitter.com/hayvok/status/705468085461889025>). Jarrod Nettles. Retrieved 2016-03-11.
07. "30 Scala job openings at Apple" (<https://alvinalexander.com/photos/30-scala-job-openings-apple>). Alvin Alexander. Retrieved 2016-03-11.
08. David Reid & Tania Teixeira (26 February 2010). "Are people ready to pay for online news?" ([https://news.bbc.co.uk/1/hi/programmes/click\\_online/8537519.stm](https://news.bbc.co.uk/1/hi/programmes/click_online/8537519.stm)). BBC. Retrieved 2010-02-28.

09. "Guardian switching from Java to Scala" (<http://www.h-online.com/open/news/item/Guardian-switching-from-Java-to-Scala-1221832.html>). Heise Online. 2011-04-05. Retrieved 2011-04-05.
10. "Guardian.co.uk Switching from Java to Scala" ([https://www.infoq.com/articles/guardian\\_scala](https://www.infoq.com/articles/guardian_scala)). InfoQ.com. 2011-04-04. Retrieved 2011-04-05.
11. Roy, Suman & Sundaresan, Krishna (2014-05-13). "Building Blackbeard: A Syndication System Powered By Play, Scala and Akka" (<https://open.blogs.nytimes.com/2014/05/13/building-blackbeard-a-syndication-system-powered-by-play-scala-and-akka>). Retrieved 2014-07-20.
12. Pavley, John (2013-08-11). "Sneak Peek: HuffPost Brings Real Time Collaboration to the Newsroom" ([https://www.huffingtonpost.com/john-pavley/huffpost-content-management-system\\_b\\_3739572.html](https://www.huffingtonpost.com/john-pavley/huffpost-content-management-system_b_3739572.html)). Retrieved 2014-07-20.
13. Binstock, Andrew (2011-07-14). "Interview with Scala's Martin Odersky" (<https://drdobbs.com/architecture-and-design/231001802>). Dr. Dobb's Journal. Retrieved 2012-02-10.
14. Synodinos, Dionysios G. (2010-10-11). "LinkedIn Signal: A Case Study for Scala, JRuby and Voldemort" (<https://www.infoq.com/articles/linkedin-scala-jruby-voldemort>). InfoQ.
15. "Real-life Meetups Deserve Real-time APIs" (<https://making.meetup.com/post/2929945070/real-life-meetups-deserve-real-time-apis>).
16. "Real time updating comes to the Remember The Milk web app" (<https://blog.rememberthemilk.com/2011/08/real-time-updating-comes-to-the-remember-the-milk-web-app>).
17. "Senior Scala Engineer" (<https://www.verizon.com/jobs/santa-clara/network/jobid362082-scala-engineer-verizon-jobs/>). Retrieved 2014-08-18.
18. Novet, Jordan (2015-06-04). "Airbnb announces Aerosolve, an open-source machine learning software package" (<https://venturebeat.com/2015/06/04/airbnb-introduces-aerosolve-a-open-source-machine-learning-software-package/>). Retrieved 2016-03-09.
19. Kops, Alexander (2015-12-14). "Zalando Tech: From Java to Scala in Less Than Three Months" (<https://www.slideshare.net/ZalandoTech/zalando-tech-from-java-to-scala-in-less-than-three-months>). Retrieved 2016-03-09.
20. Calçado, Phil (2014-06-13). "Building Products at SoundCloud—Part III: Microservices in Scala and Finagle" (<https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle>). Retrieved 2016-03-09.
21. Concurrent Inc. (2014-11-18). "Customer Case Studies: SoundCloud" (<http://www.concurrentinc.com/customer/soundcloud/>). Retrieved 2016-03-09.
22. Skills Matter (2015-12-03). "Scala at Morgan Stanley (Video)" (<https://vimeo.com/147697498>). Retrieved 2016-03-11.
23. Greg Soltis. "SF Scala, Greg Soltis: High Performance Services in Scala (Video)" (<https://www.youtube.com/watch?v=THcAwoA5G2w>). Retrieved 2016-03-11.
24. Lee Mighdoll. "Scala jobs at Nest" (<https://www.scala-lang.org/old/node/11547.html>). Retrieved 2016-03-11.
25. Nurun. "Nurun Launches Redesigned Transactional Platform With Walmart Canada" (<https://www.nurun.com/en/news/nurun-launches-redesigned-transactional-platform-with-walmart-canada/>). Retrieved 2013-12-11.
26. André K. Horie (2017-01-31). "Rewriting Duolingo's engine in Scala" (<https://making.duolingo.com/rewriting-duolingos-engine-in-scala>). Retrieved 2017-02-03.
27. "HMRC GitHub repository" (<https://github.com/hmrc?utf8=%E2%9C%93&q=&type=&language=scala>).
28. Krikorian, Raffi (17 March 2015). *O'Reilly Software Architecture Conference 2015 Complete Video Compilation: Re-Architecting on the Fly - Raffi Krikorian - Part 3* (<https://techbus.safaribooksonline.com/video/software-engineering-and-development/9781491924563>) (video). O'Reilly Media. Event occurs at 4:57. Retrieved 8 March 2016. "What I would have done differently four years ago is use Java and not used Scala as part of this rewrite. [...] it would take an engineer two months before they're fully productive and writing Scala code."

29. Scott, Kevin (11 Mar 2015). "Is LinkedIn getting rid of Scala?" (<https://www.quora.com/Is-Linked-In-getting-rid-of-Scala>). *quora.com*. Retrieved 25 January 2016.
30. Hale, Coda (29 November 2011). "The Rest of the Story" (<https://codahale.com/the-rest-of-the-story/>). *codahale.com*. Retrieved 7 November 2013.
31. "Chisel: Constructing Hardware in a Scala Embedded Language | ASPIRE" (<https://aspire.eecs.berkeley.edu/projects/chisel-constructing-hardware-in-a-scala-embedded-language/>). *UC Berkeley APSIRE*. Retrieved 27 May 2020.

## Further reading

---

- Odersky, Martin; Spoon, Lex; Venners, Bill (15 December 2019). *Programming in Scala: A Comprehensive Step-by-step Guide* ([https://www.artima.com/shop/programming\\_in\\_scala\\_4ed](https://www.artima.com/shop/programming_in_scala_4ed)) (4th ed.). Artima Inc. p. 896. ISBN 978-0-9815316-1-8.
- Horstmann, Cay (15 December 2016). *Scala for the Impatient* (<https://www.informit.com/title/9780134540566>) (2nd ed.). Addison-Wesley Professional. p. 384. ISBN 978-0-134-54056-6.
- Wampler, Dean; Payne, Alex (14 December 2014). *Programming Scala: Scalability = Functional Programming + Objects* (<https://oreilly.com/catalog/9781491949856/>) (2nd ed.). O'Reilly Media. p. 583. ISBN 978-1-491-94985-6.
- Suereth, Joshua D. (Spring 2011). *Scala in Depth* ([https://archive.org/details/scaladepth00suer\\_381](https://archive.org/details/scaladepth00suer_381)). Manning Publications. p. 225 ([https://archive.org/details/scaladepth00suer\\_381/page/n245](https://archive.org/details/scaladepth00suer_381/page/n245)). ISBN 978-1-935182-70-2.
- Meredith, Gregory (2011). *Monadic Design Patterns for the Web* (<https://github.com/leithaus/XTrace/blob/monadic/src/main/book/content/monadic.pdf>) (PDF) (1st ed.). p. 300.
- Odersky, Martin; Spoon, Lex; Venners, Bill (10 December 2008). *Programming in Scala, First Edition, eBook* (<https://www.artima.com/pins1ed>) (1st ed.). Artima Inc.

---

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Scala\\_\(programming\\_language\)&oldid=985886431](https://en.wikipedia.org/w/index.php?title=Scala_(programming_language)&oldid=985886431)"

---

This page was last edited on 28 October 2020, at 15:02 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.