Changelog

Changelog

Changes in Version 2.8.0

Trailing commas Trailing commas in expression, argument, type or pattern sequences are no longer supported.

Changes in Version 2.8

Changed visibility rules for nested packages (where done?)

Changed visibility rules so that packages are no longer treated specially.

Added section on weak conformance. Relaxed type rules for conditionals, match expressions, try expressions to compute their result type using least upper bound wrt weak conformance. Relaxed type rule for local type inference so that argument types need only weekly conform to inferred formal parameter types. Added section on numeric widening to support weak conformance.

Tightened rules to avoid accidental overrides.

Removed class literals.

Added section on context bounds.

Clarified differences between `isInstanceOf` and pattern matches.

Allowed `implicit` modifier on function literals with a single parameter.

Changes in Version 2.7.2

(10-Nov-2008)

Precedence of Assignment Operators The precedence of assignment operators has been brought in line with. From now on `+=`, has the same precedence as `=`.

Wildcards as function parameters A formal parameter to an anonymous function may now be a wildcard represented by an underscore.

```
_ => 7   // The function that ignores its argument
         // and always returns 7.
```

Unicode alternative for left arrow The Unicode glyph '\(←\)' \(\u2190\) is now treated as a reserved identifier, equivalent to the ASCII symbol '`<-`'.

Changes in Version 2.7.1

(09-April-2008)

Change in Scoping Rules for Wildcard Placeholders in Types A wildcard in a type now binds to the closest enclosing type application. For example `List[List[_]]` is now equivalent to this existential type:

```
List[List[t] forSome { type t }]
```

In version 2.7.0, the type expanded instead to:

```
List[List[t]] forSome { type t }
```

The new convention corresponds exactly to the way wildcards in Java are interpreted.

No Contractiveness Requirement for Implicits The contractiveness requirement for implicit method definitions has been dropped. Instead it is checked for each implicit expansion individually that the expansion does not result in a cycle or a tree of infinitely growing types.

Changes in Version 2.7.0

(07-Feb-2008)

Java Generics Scala now supports Java generic types by default:

- A generic type in Java such as `ArrayList<String>` is translated to a generic type in Scala: `ArrayList[String]`.
- A wildcard type such as `ArrayList<? extends Number>` is translated to `ArrayList[_ <: Number]`. This is itself a shorthand for the existential type `ArrayList[T] forSome { type T <: Number }`.
- A raw type in Java such as `ArrayList` is translated to `ArrayList[_]`, which is a shorthand for `ArrayList[T] forSome { type T }`.

This translation works if `-target:jvm-1.5` is specified, which is the new default. For any other target, Java generics are not recognized. To ensure upgradability of Scala codebases, extraneous type parameters for Java classes under `-target:jvm-1.4` are simply ignored. For instance, when compiling with `-target:jvm-1.4`, a Scala type such as `ArrayList[String]` is simply treated as the unparameterized type `ArrayList`.

Changes to Case Classes The Scala compiler generates a [companion extractor object for every case class] (05-classes-and-objects.html#case-classes) now. For instance, given the case class:

```
case class X(elem: String)
```

the following companion object is generated:

```
object X {
  def unapply(x: X): Some[String] = Some(x.elem)
  def apply(s: String): X = new X(s)
}
```

If the object exists already, only the `apply` and `unapply` methods are added to it.

Three restrictions on case classes have been removed.

1. Case classes can now inherit from other case classes.
2. Case classes may now be `abstract`.
3. Case classes may now come with companion objects.

Changes in Version 2.6.1

(30-Nov-2007)

Mutable variables introduced by pattern binding [Mutable variables can now be introduced by a pattern matching definition] (04-basic-declarations-and-definitions.html#variable-declarations-and-definitions), just like values can. Examples:

```
var (x, y) = if (positive) (1, 2) else (-1, -3)
var hd :: tl = mylist
```

Self-types Self types can now be introduced without defining an alias name for `this`. Example:

```
class C {
  type T <: Trait
  trait Trait { this: T => ... }
}
```

Changes in Version 2.6

(27-July-2007)

Existential types It is now possible to define existential types. An existential type has the form `T forSome {Q}` where `Q` is a sequence of value and/or type declarations. Given the class definitions

```
class Ref[T]
abstract class Outer { type T }
```

one may for example write the following existential types

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
```

Lazy values It is now possible to define lazy value declarations using the new modifier `lazy`. A `lazy` value definition evaluates its right hand side \(e\) the first time the value is accessed. Example:

```
import compat.Platform._
val t0 = currentTime
lazy val t1 = currentTime
val t2 = currentTime

println("t0 <= t2: " + (t0 <= t2))  //true
println("t1 <= t2: " + (t1 <= t2))  //false (lazy evaluation of t1)
```

Structural types It is now possible to declare structural types using [type refinements] (03-types.html#compound-types). For example:

```
class File(name: String) {
  def getName(): String = name
  def open() { /*..*/ }
  def close() { println("close file") }
}
def test(f: { def getName(): String }) { println(f.getName) }

test(new File("test.txt"))
test(new java.io.File("test.txt"))
```

There's also a shorthand form for creating values of structural types. For instance,

```
new { def getName() = "aaron" }
```

is a shorthand for

```
new AnyRef{ def getName() = "aaron" }
```

Changes in Version 2.5

(02-May-2007)

Type constructor polymorphism *Implemented by Adriaan Moors*

Type parameters and abstract type members can now also abstract over type constructors.

This allows a more precise `Iterable` interface:

```
trait Iterable[+T] {
  type MyType[+T] <: Iterable[T] // MyType is a type constructor

  def filter(p: T => Boolean): MyType[T] = ...
  def map[S](f: T => S): MyType[S] = ...
}

abstract class List[+T] extends Iterable[T] {
  type MyType[+T] = List[T]
}
```

This definition of `Iterable` makes explicit that mapping a function over a certain structure (e.g., a `List`) will yield the same structure (containing different elements).

Early object initialization Early object initialization makes it possible to initialize some fields of an object before any parent constructors are called. This is particularly useful for traits, which do not have normal constructor parameters. Example:

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting {
  println(msg)
}
```

In the code above, the field is initialized before the constructor of is called. Therefore, field `msg` in class is properly initialized to .

For-comprehensions, revised The syntax of for-comprehensions has changed. In the new syntax, generators do not start with a `val` anymore, but filters start with an `if` (and are called guards). A semicolon in front of a guard is optional. For example:

```
for (val x <- List(1, 2, 3); x % 2 == 0) println(x)
```

is now written

```
for (x <- List(1, 2, 3) if x % 2 == 0) println(x)
```

The old syntax is still available but will be deprecated in the future.

Implicit anonymous functions It is now possible to define [anonymous functions using underscores] (06-expressions.html#placeholder-syntax-for-anonymous-functions) in parameter position. For instance, the expressions in the left column are each function values which expand to the anonymous functions on their right.

```
_ + 1                  x => x + 1
_ * _                  (x1, x2) => x1 * x2
(_: int) * 2           (x: int) => (x: int) * 2
if (_) x else y        z => if (z) x else y
_.map(f)               x => x.map(f)
_.map(_ + 1)           x => x.map(y => y + 1)
```

As a special case, a partially unapplied method is now designated `m _` instead of the previous notation `&m`.

The new notation will displace the special syntax forms `.m()` for abstracting over method receivers and `&m` for treating an unapplied method as a function value. For the time being, the old syntax forms are still available, but they will be deprecated in the future.

Pattern matching anonymous functions, refined It is now possible to use [case clauses to define a function value] (08-pattern-matching.html#pattern-matching-anonymous-functions) directly for functions of arities greater than one. Previously, only unary functions could be defined that way. Example:

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

Changes in Version 2.4

(09-Mar-2007)

Object-local private and protected The `private` and `protected` modifiers now accept a `[this]` qualifier. A definition \(M\) which is labelled `private[this]` is private, and in addition can be accessed only from within the current object. That is, the only legal prefixes for \(M\) are `this` or `$C$.this`. Analogously, a definition \(M\) which is labelled `protected[this]` is protected, and in addition can be accessed only from within the current object.

Tuples, revised The syntax for tuples has been changed from \(\{…\}\) to \((…)\). For any sequence of types \(T_1 , … , T_n\),

\((T_1 , … , T_n)\) is a shorthand for `Tuple$n$[$T_1 , … , T_n$]`.

Analogously, for any sequence of expressions or patterns \(x_1 , … , x_n\),

\((x_1 , … , x_n)\) is a shorthand for `Tuple$n$($x_1 , … , x_n$)`.

Access modifiers for primary constructors The primary constructor of a class can now be marked [`private` or `protected`] (05-classes-and-objects.html#class-definitions). If such an access modifier is given, it comes between the name of the class and its value parameters. Example:

```
class C[T] private (x: T) { ... }
```

Annotations The support for attributes has been extended and its syntax changed. Attributes are now called *annotations*. The syntax has been changed to follow Java's conventions, e.g. `@attribute` instead of `[attribute]`. The old syntax is still available but will be deprecated in the future.

Annotations are now serialized so that they can be read by compile-time or run-time tools. Class has two sub-traits which are used to indicate how annotations are retained. Instances of an annotation class inheriting from trait will be stored in the generated class files. Instances of an annotation class inheriting from trait will be visible to the Scala type-checker in every compilation unit where the annotated symbol is accessed.

Decidable subtyping The implementation of subtyping has been changed to prevent infinite recursions. Termination of subtyping is now ensured by a new restriction of class graphs to be finitary.

Case classes cannot be abstract It is now explicitly ruled out that case classes can be abstract. The specification was silent on this point before, but did not explain how abstract case classes were treated. The Scala compiler allowed the idiom.

New syntax for self aliases and self types It is now possible to give an explicit alias name and/or type for the self reference `this`. For instance, in

```
class C { self: D =>
  ...
}
```

the name `self` is introduced as an alias for `this` within `C` and the self type of `C` is assumed to be `D`. This construct is introduced now in order to replace eventually both the qualified this construct and the clause in Scala.

Assignment Operators It is now possible to [combine operators with assignments] (06-expressions.html#assignment-operators). Example:

```
var x: int = 0
x += 1
```

Changes in Version 2.3.2

(23-Jan-2007)

Extractors It is now possible to define patterns independently of case classes, using `unapply` methods in extractor objects. Here is an example:

```
object Twice {
  def apply(x:Int): int = x*2
  def unapply(z:Int): Option[int] = if (z%2==0) Some(z/2) else None
}
val x = Twice(21)
x match { case Twice(n) => Console.println(n) } // prints 21
```

In the example, `Twice` is an extractor object with two methods:

- The `apply` method is used to build even numbers.
- The `unapply` method is used to decompose an even number; it is in a sense the reverse of `apply`. `unapply` methods return option types: `Some(...)` for a match that succeeds, `None` for a match that fails. Pattern variables are returned as the elements of `Some`. If there are several variables, they are grouped in a tuple.

In the second-to-last line, `Twice`'s method is used to construct a number `x`. In the last line, `x` is tested against the pattern `Twice(n)`. This pattern succeeds for even numbers and assigns to the variable `n` one half of the number that was tested. The pattern match makes use of the `unapply` method of object `Twice`. More details on extractors can be found in the paper "Matching Objects with Patterns" by Emir, Odersky and Williams.

Tuples A new lightweight syntax for tuples has been introduced. For any sequence of types \(T_1 , … , T_n\),

\({T_1 , … , T_n }\) is a shorthand for `Tuple$n$[$T_1 , … , T_n$]`.

Analogously, for any sequence of expressions or patterns \(x_1, … , x_n\),

\({x_1 , … , x_n }\) is a shorthand for `Tuple$n$($x_1 , … , x_n$)`.

Infix operators of greater arities It is now possible to use methods which have more than one parameter as infix operators. In this case, all method arguments are written as a normal parameter list in parentheses. Example:

```
class C {
  def +(x: int, y: String) = ...
}
val c = new C
c + (1, "abc")
```

Deprecated attribute A new standard attribute `deprecated` is available. If a member definition is marked with this attribute, any reference to the member will cause a "deprecated" warning message to be emitted.

Changes in Version 2.3

(23-Nov-2006)

Procedures A simplified syntax for [methods returning `unit`] (04-basic-declarations-and-definitions.html#procedures) has been introduced. Scala now allows the following shorthands:

`def f(params)` \(for\) `def f(params): unit` `def f(params) { ... }` \(for\) `def f(params): unit = { ... }`

Type Patterns The syntax of types in patterns has been refined. Scala now distinguishes between type variables (starting with a lower case letter) and types as type arguments in patterns. Type variables are bound in the pattern. Other type arguments are, as in previous versions, erased. The Scala compiler will now issue an "unchecked" warning at places where type erasure might compromise type-safety.

Standard Types The recommended names for the two bottom classes in Scala's type hierarchy have changed as follows:

```
All      ==>     Nothing
AllRef   ==>     Null
```

The old names are still available as type aliases.

Changes in Version 2.1.8

(23-Aug-2006)

Visibility Qualifier for protected Protected members can now have a visibility qualifier, e.g. `protected[<qualifier>]`. In particular, one can now simulate package protected access as in Java writing

```
protected[P] def X ...
```

where would name the package containing `X`.

Relaxation of Private Access Private members of a class can now be referenced from the companion module of the class and vice versa.

Implicit Lookup The lookup method for implicit definitions has been generalized. When searching for an implicit definition matching a type \(T\), now are considered

1. all identifiers accessible without prefix, and
2. all members of companion modules of classes associated with \(T\).

(The second clause is more general than before). Here, a class is *associated* with a type \(T\) if it is referenced by some part of \(T\), or if it is a base class of some part of \(T\). For instance, to find implicit members corresponding to the type

```
HashSet[List[Int], String]
```

one would now look in the companion modules (aka static parts) of `HashSet`, `List`, `Int`, and `String`. Before, it was just the static part of .

Tightened Pattern Match A typed pattern match with a singleton type `p.type` now tests whether the selector value is reference-equal to `p`. Example:

```
val p = List(1, 2, 3)
val q = List(1, 2)
val r = q
r match {
  case _: p.type => Console.println("p")
  case _: q.type => Console.println("q")
}
```

This will match the second case and hence will print "q". Before, the singleton types were erased to `List`, and therefore the first case would have matched, which is non-sensical.

Changes in Version 2.1.7

(19-Jul-2006)

Multi-Line string literals It is now possible to write [multi-line string-literals] (01-lexical-syntax.html#string-literals) enclosed in triple quotes. Example:

```
"""this is a
   multi-line
   string literal"""
```

No escape substitutions except for unicode escapes are performed in such string literals.

Closure Syntax The syntax of closures has been slightly restricted. The form

```
  x: T => E
```

is valid only when enclosed in braces, i.e. `{ x: T => E }`. The following is illegal, because it might be read as the value x typed with the type `T => E`:

```
  val f = x: T => E
```

Legal alternatives are:

```
  val f = { x: T => E }
  val f = (x: T) => E
```

Changes in Version 2.1.5

(24-May-2006)

Class Literals There is a new syntax for class literals: For any class type \(C\), `classOf[$C$]` designates the run-time representation of \(C\).

Changes in Version 2.0

(12-Mar-2006)

Scala in its second version is different in some details from the first version of the language. There have been several additions and some old idioms are no longer supported. This appendix summarizes the main changes.

New Keywords The following three words are now reserved; they cannot be used as identifiers:

```
implicit    match     requires
```

Newlines as Statement Separators Newlines can now be used as statement separators in place of semicolons.

Syntax Restrictions There are some other situations where old constructs no longer work:

Pattern matching expressions The `match` keyword now appears only as infix operator between a selector expression and a number of cases, as in:

```
  expr match {
    case Some(x) => ...
    case None => ...
  }
```

Variants such as `expr.match {...}` or just `match {...}` are no longer supported.

"With" in extends clauses The idiom

```
class C with M { ... }
```

is no longer supported. A `with` connective is only allowed following an `extends` clause. For instance, the line above would have to be written

```
class C extends AnyRef with M { ... } .
```

However, assuming `M` is a trait, it is also legal to write

```
class C extends M { ... }
```

The latter expression is treated as equivalent to

```
class C extends S with M { ... }
```

where `S` is the superclass of `M`.

Regular Expression Patterns The only form of regular expression pattern that is currently supported is a sequence pattern, which might end in a sequence wildcard . Example:

```
case List(1, 2, _*) => ... // will match all lists starting with 1, 2, ...
```

It is at current not clear whether this is a permanent restriction. We are evaluating the possibility of re-introducing full regular expression patterns in Scala.

Selftype Annotations The recommended syntax of selftype annotations has changed.

```
class C: T extends B { ... }
```

becomes

```
class C requires T extends B { ... }
```

That is, selftypes are now indicated by the new `requires` keyword. The old syntax is still available but is considered deprecated.

For-comprehensions For-comprehensions now admit value and pattern definitions. Example:

```
for {
  val x <- List.range(1, 100)
  val y <- List.range(1, x)
  val z = x + y
  isPrime(z)
} yield Pair(x, y)
```

Note the definition `val z = x + y` as the third item in the for-comprehension.

Conversions The rules for [implicit conversions of methods to functions] (06-expressions.html#method-conversions) have been tightened. Previously, a parameterized method used as a value was always implicitly converted to a function. This could lead to unexpected results when method arguments where forgotten. Consider for instance the statement below:

```
show(x.toString)
```

where `show` is defined as follows:

```
def show(x: String) = Console.println(x) .
```

Most likely, the programmer forgot to supply an empty argument list `()` to `toString`. The previous Scala version would treat this code as a partially applied method, and expand it to:

```
show(() => x.toString())
```

As a result, the address of a closure would be printed instead of the value of `s`.

Scala version 2.0 will apply a conversion from partially applied method to function value only if the expected type of the expression is indeed a function type. For instance, the conversion would not be applied in the code above because the expected type of `show`'s parameter is `String`, not a function type.

The new convention disallows some previously legal code. Example:

```
def sum(f: int => double)(a: int, b: int): double =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)

val sumInts  =  sum(x => x)  // error: missing arguments
```

The partial application of `sum` in the last line of the code above will not be converted to a function type. Instead, the compiler will produce an error message which states that arguments for method `sum` are missing. The problem can be fixed by providing an expected type for the partial application, for instance by annotating the definition of `sumInts` with its type:

```
val sumInts: (int, int) => double  =  sum(x => x)  // OK
```

On the other hand, Scala version 2.0 now automatically applies methods with empty parameter lists to `()` argument lists when necessary. For instance, the `show` expression above will now be expanded to

```
show(x.toString()) .
```

Scala version 2.0 also relaxes the rules of overriding with respect to empty parameter lists. The revised definition of *matching members* makes it now possible to override a method with an explicit, but empty parameter list `()` with a parameterless method, and *vice versa*. For instance, the following class definition is now legal:

```
class C {
  override def toString: String = ...
}
```

Previously this definition would have been rejected, because the `toString` method as inherited from `java.lang.Object` takes an empty parameter list.

Class Parameters A class parameter may now be prefixed by `val` or `var`.

Private Qualifiers Previously, Scala had three levels of visibility: *private*, *protected* and *public*. There was no way to restrict accesses to members of the current package, as in Java.

Scala 2 now defines access qualifiers that let one express this level of visibility, among others. In the definition

```
private[C] def f(...)
```

access to `f` is restricted to all code within the class or package `C` (which must contain the definition of `f`).

Changes in the Mixin Model The model which details [mixin composition of classes] (05-classes-and-objects.html#templates) has changed significantly. The main differences are:

1. We now distinguish between *traits* that are used as mixin classes and normal classes. The syntax of traits has been generalized from version 1.0, in that traits are now allowed to have mutable fields. However, as in version 1.0, traits still may not have constructor parameters.
2. Member resolution and super accesses are now both defined in terms of a *class linearization.*
3. Scala's notion of method overloading has been generalized; in particular, it is now possible to have overloaded variants of the same method in a subclass and in a superclass, or in several different mixins. This makes method overloading in Scala conceptually the same as in Java.

Implicit Parameters Views in Scala 1.0 have been replaced by the more general concept of implicit parameters.

Flexible Typing of Pattern Matching The new version of Scala implements more flexible typing rules when it comes to [pattern matching over heterogeneous class hierarchies] (08-pattern-matching.html#pattern-matching-expressions). A *heterogeneous class hierarchy* is one where subclasses inherit a common superclass with different parameter types. With the new rules in Scala version 2.0 one can perform pattern matches over such hierarchies with more precise typings that keep track of the information gained by comparing the types of a selector and a matching pattern. This gives Scala capabilities analogous to guarded algebraic data types.