Named and Default Arguments in Scala 2.8

Lukas Rytz

November 9, 2009

Contents

1 Introduction

Methods taking multiple parameters of the same type are a source of mistakes which cannot be detected during compile time: exchanging two arguments of the same type does not yield an error, but can produce unexpected results. This problem can be avoided elegantly by using named arguments. Furthermore, named arguments improve the readability of method calls with a large number of arguments.

The second language feature discussed in this document, default arguments, in general does not depend on having named arguments. But combining the two features actually improves their usefulness, for instance it avoids the requirement of putting the parameters with defaults to the end of the parameter list of a method. For that reason, the two features are introduced together.

2 Named arguments

In Scala 2.8, method arguments can be specified in *named style* using the same syntax as variable assignments:

```
def f[T](a: Int, b: T)
f(b = getT(), a = getInt())
```

The argument expressions are evaluated in call-site order, so in the above example `getT()` is executed before `getInt()`. Mixing named and positional arguments is allowed as long as the positional part forms a prefix of the argument list:

```
f(0, b = "1")     // valid
f(b = "1", a = 0) // valid
// f(b = "1", 0)  // invalid, a positional after named argument
// f(0, a = 1)    // invalid, parameter 'a' specified twice
```

If an argument expression has the form "`x = expr`" and `x` is not a parameter name of the method, the argument is treated as an assignment expression to some variable `x`, i.e. the argument type is `Unit`. So the following example will continue to work as expected:

```
def twice(op: => Unit) = { op; op }
var x = 1
twice(x = x + 1)
```

It is an error if the expression "`x = expr`" can be interpreted as both a named argument (parameter name `x`) and an assignment (variable `x` in scope). If the expression is surrounded by an additonal set of parenthesis or braces, it is never treated as a named argument. Also, if the application argument is a block expression (as in `f{ arg }`), `arg` is never treated as a named argument.

```
def twice(op: => Unit) = { op; op }
var op = 1
// twice(op = op + 1)  // error: reference to 'op' is ambiguous
twice((op = op + 1))   // assignment, not a named argument
twice({op = op + 1})   // assignment
twice{ op = op + 1 }  // assignment
```

2.1 Integration with other features

The following list shows how named arguments interfere with other language features of Scala:

By-Name Parameters continue to work as expected when using named arguments. The expression is only (and repeatedly) evaluated when the body of the method accesses the parameter.

Repeated Parameters When an application uses named arguments, the repeated parameter has to be specified exactly once. Using the same parameter name multiple times is disallowed.

Functional values A functional value in Scala is an instance of a class which implements a method called `apply`. One can use the parameter names of that apply method for a named application. For functional values whose static type is `scala.FunctionN`, the parameter names of that apply method can be used.

```
val f1 = new { def apply(x: Int) = x + 1 }
val f2 = (x: Int) => x + 1 // instance of Function1[Int, Int]
f1(x = 2)    // OK
// f2(x = 2) // "error: not found: value x"
f2(v1 = 2)   // OK, 'v1' is the parameter name in Function1
```

Overriding When a method is overridden (or an abstract method is implemented) in a subclass, the parameter names don't have to be the same as in the superclass. For type-checking an application which uses named arguments, the static type of the method determines which names have to be used.

```
trait A          { def f(a: Int): Int }
class B extends A { def f(x: Int) = x }
val a: A = new B
a.f(a = 1) // OK
```

Overloading Resolution When a method application refers to an overloaded method, first the set of applicable alternatives is determined and then the most specific alternative is chosen (see [1], Chapter 6.25.3).

The presence of named argument influences the set of applicable alternatives, the argument types have to be matched against the corresponding parameter types based on the names. In the following example, the second alternative is applicable:

```
def f()                    // #1
def f(a: Int, b: String)   // #2
f(b = "someString", a = 1) // using #2
```

If multiple alternatives are applicable, the most specific one is determined. This process is independent of the argument names used in a specific application and only looks at the method signature (for a detailed description, see [1], Chapter 6.25.3).

In the following example, both alternatives are applicable, but none of them is more specific than the other because the argument types are compared based on their position, not on the argument name:

```
def f(a: Int, b: String)   // #1
def f(b: Object, a: Int)   // #2
f(a = 1, b = "someString") // "error: ambiguous reference to
                           //  overloaded definition"
```

Anonymous functions The placeholder syntax syntax for creating anonymous functions is extended to work with named arguments.

```
def f(x: Int, y: String)
val g1: Int => Int = f(y = "someString", x = _)
val g2 = f(y = "someString", x = _: Int)
```

3 Default arguments

A method parameter with a default argument has the form "`p: T = expr`", where `expr` is evaluated every time a method application uses the default argument. To use a default argument, the corresponding parameter has to be omitted in the method application.

```
def f(a: Int, b: String = "defaultString", c: Int = 5)
```

```
f(1)
f(1, "otherString")
f(1, c = 10) // c needs to be specified as named argument
```

For every parameter with a default argument, a synthetic method which computes the default expression is generated. When a method application uses default arguments, the missing parameters are added to the argument list as calls to the corresponding synthetic methods.

```
def f(a: Int = 1, b: String)
// generates a method: def f$default$1 = 1

f(b = "3")
// transformed to: f(b = "3", a = f$default$1)
```

A default argument may be an arbitrary expression. Since the scope of a parameter extends over all subsequent parameter lists (and the method body), default expressions can depend on parameters of preceding parameter lists (but not on other parameters in the same parameter list). Note that when using a default value which depends on earlier parameters, the actual arguments are used, not the default arguments.

```
def f(a: Int = 0)(b: Int = a + 1) = b // OK
// def f(a: Int = 0, b: Int = a + 1)  // "error: not found: value a"

f(10)() // returns 11 (not 1)
```

A special expected type is used for type-checking the default argument `expr` of a method parameter "`x: T = expr`": it is obtained by replacing all occurrences of type parameters of the method (type parameters of the class for constructors) with the undefined type. This allows specifying default arguments for polymorphic methods and classes:

```
def f[T](a: T = 1) = a
f()    // returns 1: Int
f("s") // returns "s": String

def g[T](a: T = 1, b: T = "2") = b
g(a = "1")  // OK, returns "2": String
g(b = 2)    // OK, returns 2: Int
g()         // OK, returns "2": Any
// g[Int]() // "error: type mismatch; found: String, required: Int"

class A[T](a: T = "defaultString")
new A()  // creates an instance of A[String]
new A(1) // creates an instance of A[Int]
```

3.1 Integration with other features

The following list shows how default arguments interfere with other language features of Scala:

By-Name Parameters Default arguments on by-name parameters work as expected.

If an application does not specify a by-name parameter with a default argument, the default expression is evaluated every time the method body refers to that parameter.

Repeated Parameters It is not allowed to specify any default arguments in a parameter section which ends in a repeated parameter.

Overriding When a method with default arguments is overridden or implemented in a subclass, all defaults are inherited and available in the subclass. The subclass can also override default arguments and add new ones to parameters which don't have a default in the superclass.

During type-checking, the static type is used to determine whether a parameter has a default value or not. At run-time, since the usage of a default is translated to a method call, the default value is determined by the dynamic type of the receiver object.

```
trait A { def f(a: Int = 1, b: Int): (Int, Int) }
// B: inherit & add a default
class B extends A { def f(a: Int,     b: Int = 2) = (a, b) }
// C: override a default
class C extends A { def f(a: Int = 3, b: Int    ) = (a, b) }

val a1: A = new B
val a2: A = new C
// a1.f()   // "error: unspecified parameter: value b"
a2.f(b = 2) // returns (3, 2)
```

Overloading If there are multiple overloaded alternatives of a method, at most one is allowed to specify default arguments.

Overloading Resolution In a method application expression, when multiple overloaded alternatives are applicable, the alternative which use default arguments is never selected.

```
def f(a: Object)             // #1
def f(a: String, b: Int = 1) // #2
f("str") // both are applicable, #1 is selected
```

Case Classes For every case class, a method named "copy" is now generated which allows to easily create modified copies of the class's instances. The copy method takes the same type and value parameters as the primary constructor of the case class, and every parameter defaults to the corresponding constructor parameter.

```
case class A[T](a: T, b: Int) {
//  def copy[T'](a': T' = a, b': Int = b): A[T'] =
//    new A[T'](a', b')
}
val a1: A[Int] = A(1, 2)
val a2: A[String] = a1.copy(a = "someString")
```

The copy method is only added to a case class if no member named "copy" already exists in the class or in one of its parents. This implies that when a case

class extends another case class, there will only be one copy method, namely the one from the lowest case class in the hierarchy.

Implicit Parameters It is allowed to specify default arguments on implicit parameters. These defaults are used in case no implicit value matching the parameter type can be found.

```
def f(implicit a: String = "value", y: Int = 0) = a +": "+ y
implicit val s = "size"
println(f)              // prints "size: 0"
```

4 Implementation

4.1 Named arguments

When using named arguments, the argument order does not have to match the parameter order of the method definition. To evaluate the argument expressions in call-site order, the method application is transformed to a block in the following way:

```
class A {
  def f(a: Int, b: Int)(c: Int)
}
(new A).f(b = getB(), a = getA())(c = getC())
// transformed to
// {
//   val qual$1 = new A()
//   val x$1 = getB()
//   val x$2 = getA()
//   val x$3 = getC()
//   qual$1.f(x$2, x$1)(x$3)
// }
```

4.2 Default arguments

For every default argument expression the compiler generates a method computing that expression. These methods have deterministic names composed of the method name, the string "`$default$`" and a number indicating the parameter position. Each method is parametrized by the type parameters of the original method and by the value parameter sections preceding the corresponding parameter:

```
def f[T](a: Int = 1)(b: T = a + 1)(c: T = b)
// generates:
// def f$default$1[T]: Int = 1
// def f$default$2[T](a: Int): Int = a + 1
// def f$default$3[T](a: Int)(b: T): T = b
```

For constructor defaults, these methods are added to the companion object of the class (which is created if it does not exist). For other methods, the default methods are generated at the same location as the original method.

Method calls which use default arguments are transformed into a block of the same form as described above for named arguments:

```
f()("str")()
// transformed to:
// {
//   val x$1 = f$default$1
//   val x$2 = "str"
//   val x$3 = f$default$3(x$1)(x$2)
//   f(x$1)(x$2)(x$3)
// }
```

References

[1] Odersky, M. *The Scala Language Specification, Version 2.7*. Available online at http://www.scala-lang.org/docu/files/ScalaReference.pdf