Named and Default Arguments in Scala 2.8

Lukas Rytz
École Polytechnique Fédérale de Lausanne

Motivation

- Why named and default arguments?
  - Documentation / readability

```
frame.setSize(300, 200)
```

- Avoid arity-based method overloading

```
def f(x: Int, y: Int) = x+y
def f(x: Int)         = f(x, 1)
f(2)
```

Motivation

- Why named and default arguments?
  - Documentation / readability

```
frame.setSize(width = 300, height = 200)
```

- Avoid arity-based method overloading

```
def f(x: Int, y: Int) = x+y
def f(x: Int)         = f(x, 1)
f(2)
```

Motivation

- Why named and default arguments?
  - Documentation / readability

```
frame.setSize(width = 300, height = 200)
```

- Avoid arity-based method overloading

```
def f(x: Int = 1, y: Int = 1) = x+y
f(2)
f(y = 3)
```

Parameters have expressive defaults

- Default arguments can be arbitrary expressions
- In curried method definitions, defaults can refer to previous parameters

Parameters have expressive defaults

- Default arguments can be arbitrary expressions
- In curried method definitions, defaults can refer to previous parameters

```
def resize(i: Image)(h: Int = i.h,
                     w: Int = i.w) { .. }

resize(img)(w = img.w*2) // stretch
```

Defaults integrate with inheritance

- Overriding methods can inherit, add and override default arguments
- Virtual dispatch for default arguments

Virtual dispatch on defaults

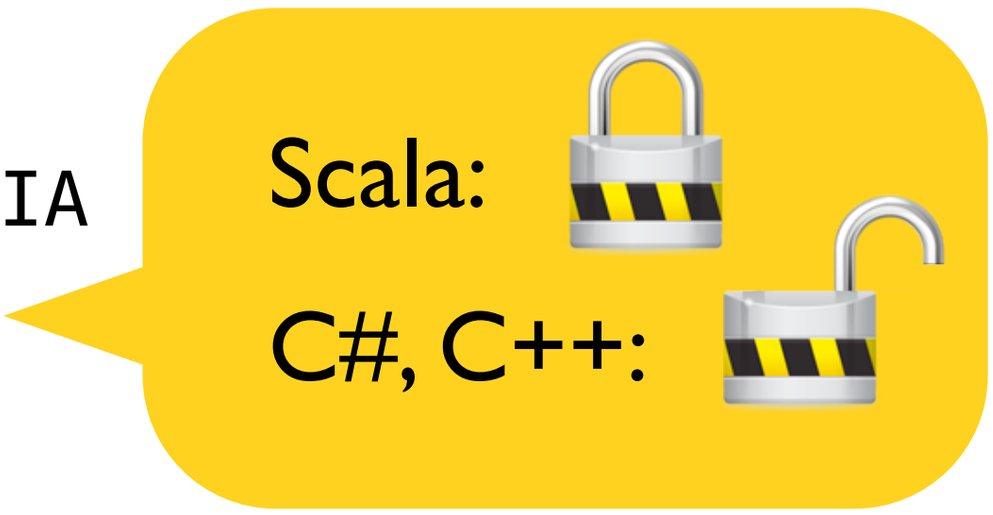
```
trait Office {
  def call(to: Number,
           encrypt: Boolean = false)
}
```

```
class CIA extends Office {
  def call(to: Number,
           encrypt: Boolean = true) { .. }
}

val office: Office = new CIA
office.call(president)
```

Virtual dispatch on defaults

```
trait Office {
  def call(to: Number,
           encrypt: Boolean = false)
}
```

```
class CIA extends Office {
  def call(to: Number,
           encrypt: Boolean = true) { .. }
}

val office: Office = new CIA
office.call(president)
```



Named arguments have intuitive syntax

- No new syntax to make parameter names public, all parameters are named

```
def setSize(width: Int, height: Int)
```

- Assignment syntax for named arguments

```
frame.setSize(height = 200, width = 300)
```

Named arguments are flexible

- Using named arguments is optional
- Overriding methods can chose parameter names freely

Evaluation order

- Left-to-right for named arguments

```
m(y = a(), x = b())
```

- Defaults are evaluated last, but per parameter list

```
def f(x: Int = e1)(y: Int = e2, z: Int)

f()(z = e3)  // evaluates e1, e3, e2
f(1)(z = e4) // evaluates e4, e2
```

Defaults are allowed on generic parameters

- A relaxed expected type is used to type check the default expression

```
def id[T](x: T = “scala”) = x
```

- Type of the default is checked at call-site

```
id(2)
id[Int]() // type mismatch error.
          // found: String, required: Int
```

Motivation: Functional update for datatypes

```
scala> case class Pair[A,B](first: A, second: B)
defined class Pair

scala> val p1 = Pair(1, “scala”)
p1: Pair[Int,java.lang.String] = Pair(1,scala)

scala> val p2 = p1.copy(second = 2)
p2: Pair[Int,Int] = Pair(1,2)
```

Motivation: Functional update for datatypes

```
scala> case class Pair[A,B](first: A, second: B)
defined class Pair

scala> val p1 = Pair(1, “scala”)
p1: Pair[Int,java.lang.String] = Pair(1,scala)

scala> val p2 = p1.copy(second = 2)
p2: Pair[Int,Int] = Pair(1,2)
```

so “copy” is a language extension?

Copy is implemented with default arguments

- No language extension required: you can write "copy" yourself!

```
case class Pair[A,B](first: A, second: B) {
  def copy(first: A  = this.first,
           second: B = this.second) = {
    Pair(first, second)
  }
}
```

- Compiler-generated for case classes

Copy is implemented with default arguments

- No language extension required: you can write "copy" yourself!

```
case class Pair[A,B](first: A, second: B) {
  def copy[U,V](first: U  = this.first,
                second: V = this.second) = {
    Pair(first, second)
  }
}
```

- Compiler-generated for case classes

Assignments might become ambiguous

The assignment syntax for named arguments can lead to ambiguities with variable assignments

```
def id(x: Int) = x
var x: Int = 1

id(x = 2) // non-ambiguous
```

Assignments might become ambiguous

The assignment syntax for named arguments can lead to ambiguities with variable assignments

```
def id(x: Int)
var x: Int = 1

id(x = 2) // non-ambiguous
```

expected: Int

assignment: type Unit

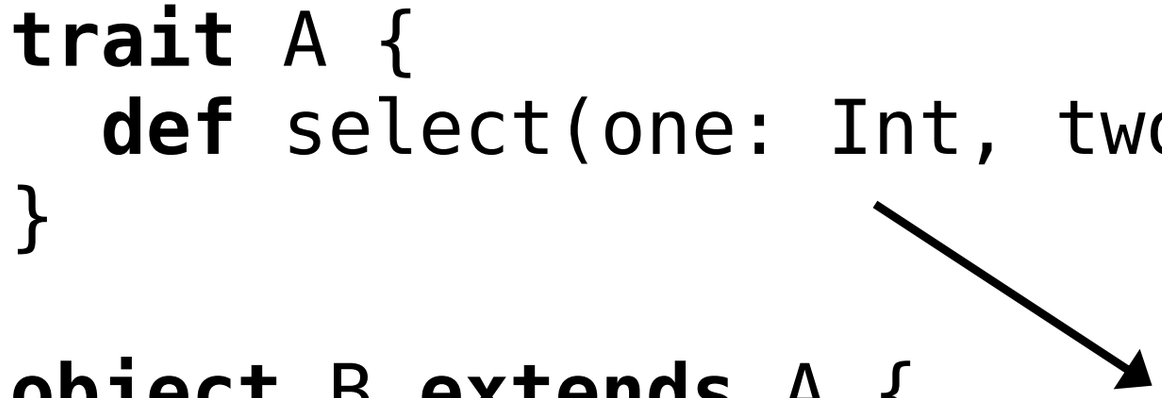
Assignments might become ambiguous

The assignment syntax for named arguments can lead to ambiguities with variable assignments

```
def id[T](x: T) = x
var x: Int = 1

id(x = 2)

  // error: reference to x is ambiguous;
  // it is both a parameter and a variable
```

Name changes

Using a parameter name on a different position in the subclass is problematic:



```
trait A {
  def select(one: Int, two: Int): Int
}

object B extends A {
  def select(two: Int, one: Int) = one
}

    B.select(one = 1, two = 2) // 1
(B: A).select(one = 1, two = 2) // 2
```

Name changes

Using a parameter name on a different position in the subclass is problematic:

```
trait A {
  def select(one: Int, two: Int): Int
}

object B extends A {
  def select(two: Int, one: Int) = one
}
```

warning: overriding method uses parameter name “second” on a different position.

Compilation part 1

- For every default argument, a method computing that default is generated

```
def f[T](a: Int = 1, b: Int)(c: T = a+1)

def f$default$1[T]: Int = 1
def f$default$3[T](a: Int, b: Int): Int =
  a+1
```

Compilation part 2

- When using defaults, local values are created for all arguments

```
def f[T](a: Int = 1, b: Int)(c: T = a+1)
```

```
f(b = compute-int)()

{
  val x$2 = compute-int
  val x$1 = f$default$1[Int]
  val x$3 = f$default$3[Int](x$1, x$2)
  f[Int](x$1, x$2)(x$3)
}
```

Consequences

- When overloading, only one of the methods is allowed to have defaults

```
def open(url: URL,
         codec: Codec = Codec.URLDefault)
def open(filename: String,
         codec: Codec = Codec.FileDefault)

// def open$default$2 = ???
```

Conclusion

- Default arguments integrate smoothly with dynamic lookup, inheritance and overriding
- Generic parameters can have defaults
- Functional datatype update without language support
- Everything is implemented and available in Scala 2.8.0.RC1