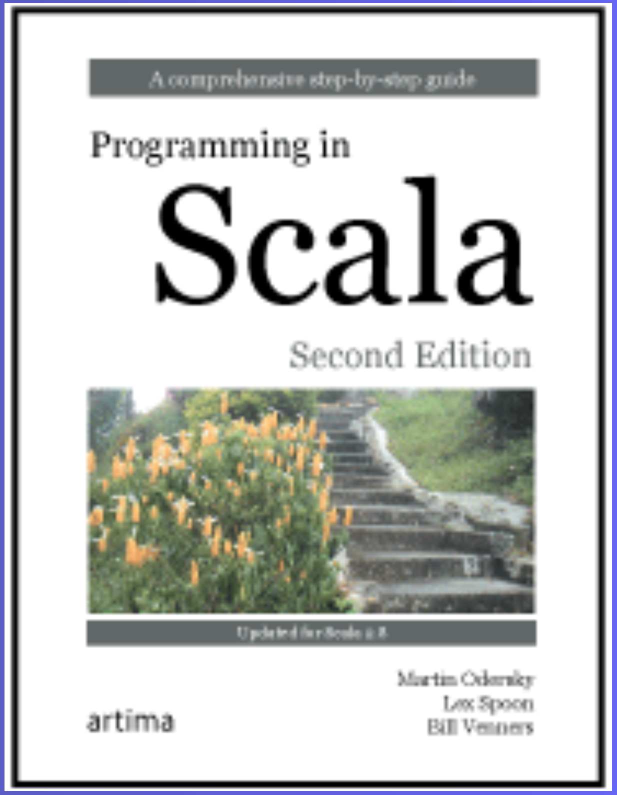
artima

Stairway to Scala - Flight 9

# Traits

Bill Venners
Dick Wall

www.artima.com



A comprehensive step-by-step guide

Programming in
Scala

Second Edition

Updated for Scala 2.8

Martin Odersky
Lex Spoon
Bill Venners

artima

# Flight 9 goal

Traits are Scala's solution to enabling the power of multiple inheritance without many of the inherent problems. They rock!

# About traits

- Java classes have single super-class, multiple interfaces.
- Scala still has a single super-class, but the interfaces have been extended to traits.
- Traits can have method implementations and fields.
- You can mix traits together in a class.
- One superclass: no diamond inheritance problem.
- Java already has cases of this, e.g. Serializable.

# Defining a trait

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
```

- Defines a trait called Philosophical
- Creates one concrete method: philosophize
- Does not specify superclass, so super is AnyRef

# Using a trait

```
class Frog extends Philosophical {
    override def toString = "green"
}

scala> val frog = new Frog
frog: Frog = green
scala> frog.philosophize()
I consume memory, therefore I am!

scala> val phil: Philosophical = frog
phil: Philosophical = green
scala> phil.philosophize()
I consume memory, therefore I am!
```

# Using traits in addition to a superclass

```
class Animal

class Frog extends Animal with Philosophical {
    override def toString = "green"
}

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
    override def toString = "green"
}
```

# Overriding methods/fields from traits

```
class Animal

class Frog extends Animal with Philosophical {
    override def toString = "green"
    override def philosophize() {
        println("It ain't easy being "+ toString +"!")
    }
}

scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green
scala> phrog.philosophize()
It ain't easy being green!
```

# Differences between traits and classes

- Traits can have fields and state, but not constructor params

```
class Point(x: Int, y: Int)   // Fine

trait Point(x: Int, y: Int)     // Does not compile
```

- Traits resolve calls to super dynamically at runtime
- Classes resolve calls to super statically at compile time

# Thin vs. rich interfaces

- Thin (sparse) interfaces easier for implementors
- Rich (extensive) interfaces better for clients (consumers)
- E.g. CharSequence interface vs. String Implementation

```
class Point(val x: Int, val y: Int)
trait Rectangular {
    def topLeft: Point
    def bottomRight: Point
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

# Free stuff!

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
        extends Rectangular {
    // other methods...
}

scala> val rect = new Rectangle(new Point(1, 1), new Point(10, 10))
rect: Rectangle = Rectangle@3536fd
scala> rect.left
res2: Int = 1
scala> rect.right
res3: Int = 10
scala> rect.width
res4: Int = 9
```

# The ordered trait

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
// ...
    def compare(that: Rational) =
        (this.numer * that.denom) - (that.numer * this.denom)
}

scala> val half = new Rational(1, 2)
scala> val third = new Rational(1, 3)
scala> half < third
res5: Boolean = false
scala> half > third
res6: Boolean = true
```

# An Int queue

- Let's say we define a basic queue of Ints:

```
abstract class IntQueue {
    def get(): Int
    def put(x: Int)
}

import scala.collection.mutable.ArrayBuffer
class BasicIntQueue extends IntQueue {
    private val buf = new ArrayBuffer[Int]
    def get() = buf.remove(0)
    def put(x: Int) { buf += x }
}
```

# Modifying the behavior with a trait

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2*x) }
}

scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue
scala> val queue = new MyQueue
scala> queue.put(10)
scala> queue.get()
res12: Int = 20
```

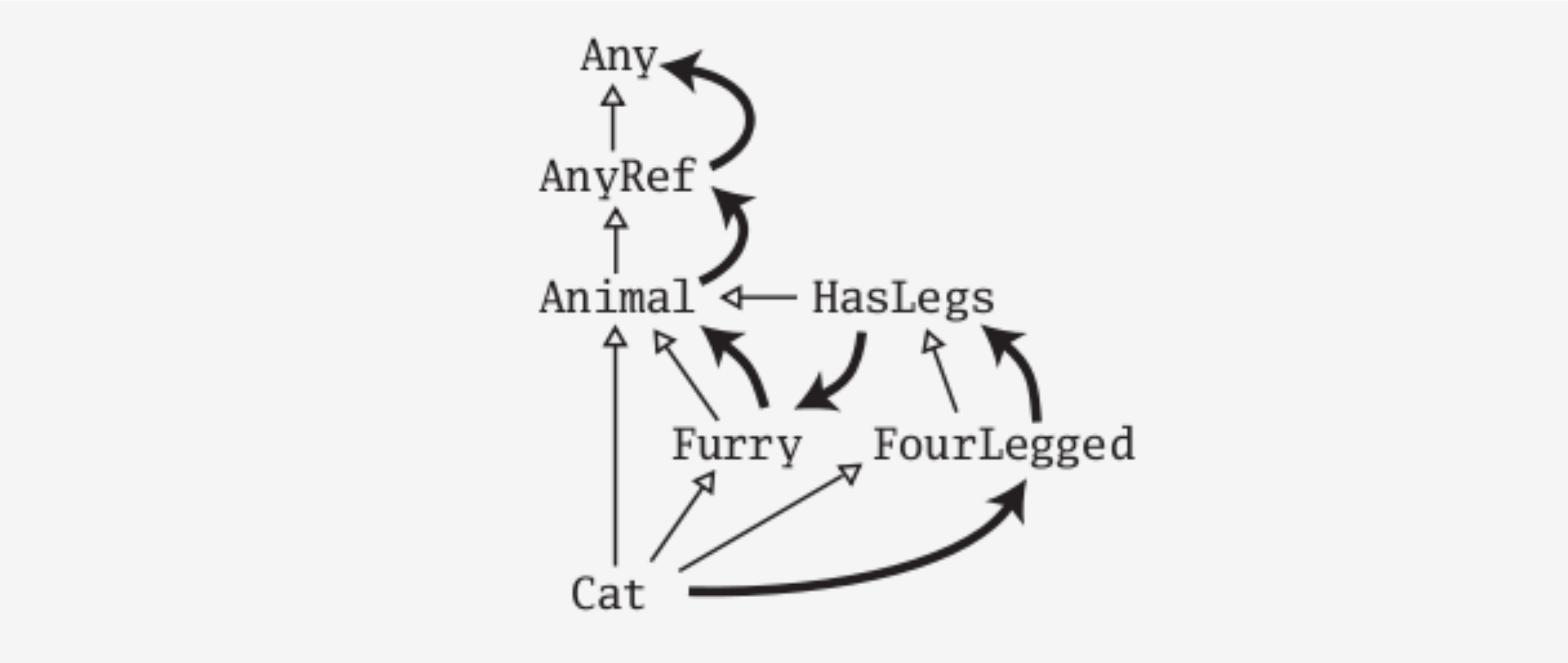
# Stackable modifications

```
trait Filtering extends IntQueue {
    abstract override def put(x: Int) { if (x >= 0) super.put(x) }
}
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) { super.put(x+1) }
}
val ifq = (new BasicIntQueue with Incrementing with Filtering)
scala> ifq.put(-1)
scala> ifq.put(0)
scala> ifq.put(1)

scala> ifq.get()
res28: Int = 1
scala> ifq.get()
res29: Int = 2
```

# Linearization

class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged



# To trait or not to trait

No firm rule, but consider these guidelines:

- Behavior will not be re-used -> concrete class
- Might be re-used in multiple, unrelated classes -> trait
- Inherit from it in Java code -> abstract class
- Distributed it in compiled form -> abstract class
- Efficiency is very important -> class (* don't early optimize)
- Still don't know? -> trait