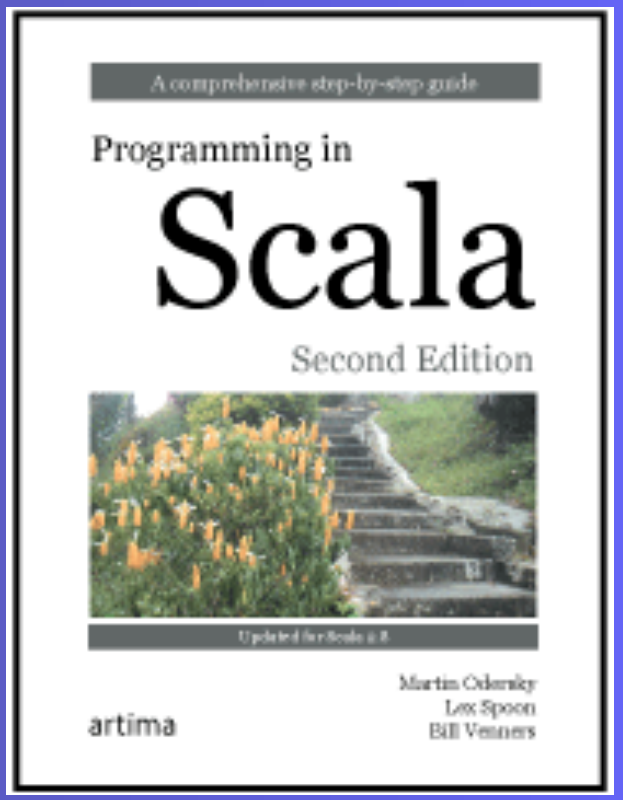
Stairway to Scala - Flight 8

Scala's hierarchy

Bill Venners
Dick Wall



www.artima.com

Flight 8 goal

Explore the Scala type heirarchy, learn about top and bottom types.

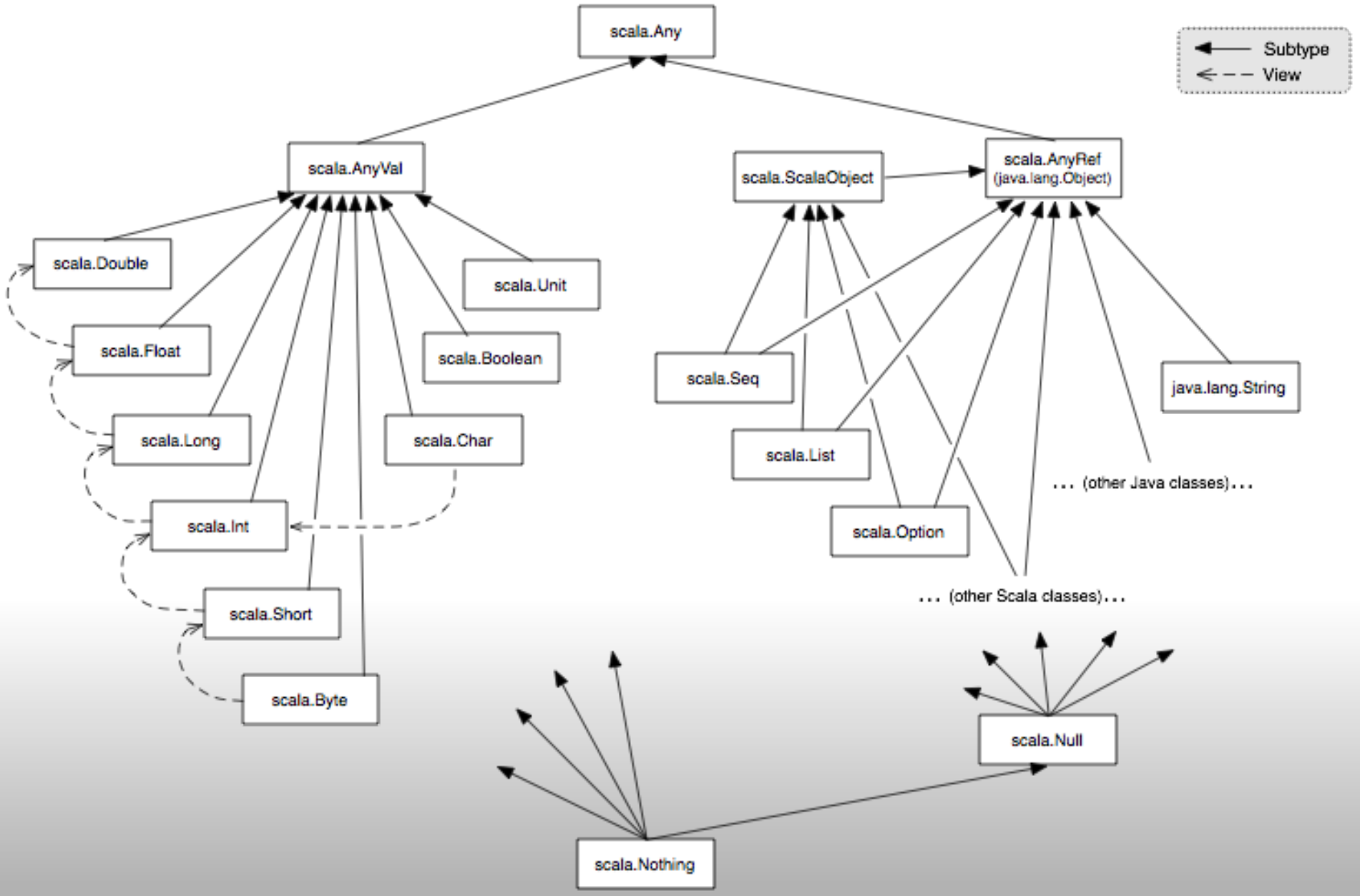
Scala top classes

- Java has a common superclass for all other classes: Object
- Scala has Any, which is a superclass for all other classes
- Methods defined on Any are universal methods and may be invoked on any object, e.g.:
  - toString
  - equals / == / !=
  - hashCode / ##
  - asInstanceOf[<Type>]
  - isInstanceOf[<Type>]

Any, AnyRef, AnyVal

- In addition to Any, there are also AnyRef and AnyVal
- AnyVal is the superclass for all Value classes
  - You might think of these as similar to Java primitives
  - e.g. Double, Int, Float, Long, Char, Short, Byte, Boolean and Unit
- AnyRef is the superclass for all Reference classes. Brings additional universal methods for these reference classes:
  - eq / ne (identity)
  - synchronized / wait / notify / notifyAll
- All Java classes subclass from AnyRef as well
- In fact, AnyRef is the true analog for java.lang.Object
- Scala classes also inherit from one other trait: ScalaObject

Scala class hierarchy diagram



Implicit conversions

- Derivatives of AnyVal can be implicitly converted to "wider" types. e.g. Char -> Int -> Long -> Float -> Double
- Other implicit conversions available, e.g. Int

```
scala> 42 max 43
res4: Int = 43
scala> 42 min 43
res5: Int = 42
scala> 1 until 5
res6: Range = Range(1, 2, 3, 4)
scala> 1 to 5
res7: Range.Inclusive = Range(1, 2, 3, 4, 5)
scala> 3.abs
res8: Int = 3
scala> (-3).abs
res9: Int = 3
```

Scala primitives

- Unlike Java, Scala does not differentiate between primitives and boxed classes (e.g. int vs Integer)
- Behind the scenes, it does use the Java box classes when necessary
- With Java boxed types, == tests reference equality.
  e.g. in Java

```
boolean isEqual(Integer x, Integer y) {
    return x == y;
}
System.out.println(isEqual(421, 421));
```

  will return false
- In Scala, == compares the primitive values, so will return true

== in Scala

- For value types, == is the natural (numeric or boolean) equality
- For reference types, == is aliased to .equals (defined in AnyRef and may not be overridden)
- You can still override .equals, and == will use that
- Use eq / ne on AnyRef derivatives for references
- Makes == less surprising, e.g.

```
scala> val x = "abcd".substring(2)
x: java.lang.String = cd
scala> val y = "abcd".substring(2)
y: java.lang.String = cd
scala> x == y
res12: Boolean = true
```

Bottom types

- Common subtypes - can be very useful
- Null is a subtype of any reference class (c.f. AnyRef)
- Null is the type of the null instance
- Nothing is a subtype of all classes (c.f. Any)
- Unlike Null, there is no instance for Nothing
- Honorable mention: Unit
- Equivalent to void in Java
- One instance of Unit: ()

```
scala> def un = ()
un: Unit
```

Bottom types in use

```
def error(message: String): Nothing =
    throw new RuntimeException(message)

def divide(x: Int, y: Int): Int =
    if (y != 0) x / y
    else error("can't divide by zero")
```

Equality recipe (part 1)

1. Define a canEqual method

```
def canEqual(other: Any): Boolean =
```

2. canEqual should yield true if the argument is an instance of the current class:

```
other.isInstanceOf[Rational] // in class Rational
```

Equality recipe (part 2)

3. In the equals method, be sure to declare the type of the passed object as Any:

```
override def equals(other: Any): Boolean =
```

Equality recipe (part 3)

4. Write the body of the equals as a single match expression on the passed object:

```
other match {
  // ...
}
```

5. The match expression should have two cases. The first case is a type pattern for the type of the current class:

```
case that: Rational => // for class Rational
```

Equality recipe (part 4)

6.In the case body, write an expression that logical-ands together the individual expressions that must be true for the object to be equal. If overriding, probably want:

```
super.equals(that) &&
```

If in a class that defines canEqual for the first time, invoke it on that, passing in this:

```
(that canEqual this) &&
```

And of course:

```
numer == that.numer &&
denom == that.denom
```

Equality recipe (part 5)

7. For the second case, use a wildcard that result in false:

```
case _ => false
```

Hashcode recipe

- Add 41 to first value, multiply sum by 41 and add second value, multiple that by 41 and add third value, ...

```
override def hashCode: Int =
  41 * (
    41 + numer
  ) + denom
```

Hashcode recipe

- If equals method invokes super.equals, start your hashCode with a super.hashCode:

```
override def hashCode: Int =
  41 * (
    41 * (
      super.hashCode
    ) + numer
  ) + denom
```