Hierarchy, Types and Options

Scala's Core Hierarchy, Type Calculus, Optionals and More.

Agenda

1. Top Classes
2. Organization From the Top
3. Bottom Classes
4. Basic Type Calculus
5. Scala Primitives and Implicit Conversions
6. Value Classes and Extension Methods
7. Nil, Null, Nothing, None
8. Option Type
9. Equals and Hashcode
10. Product Types

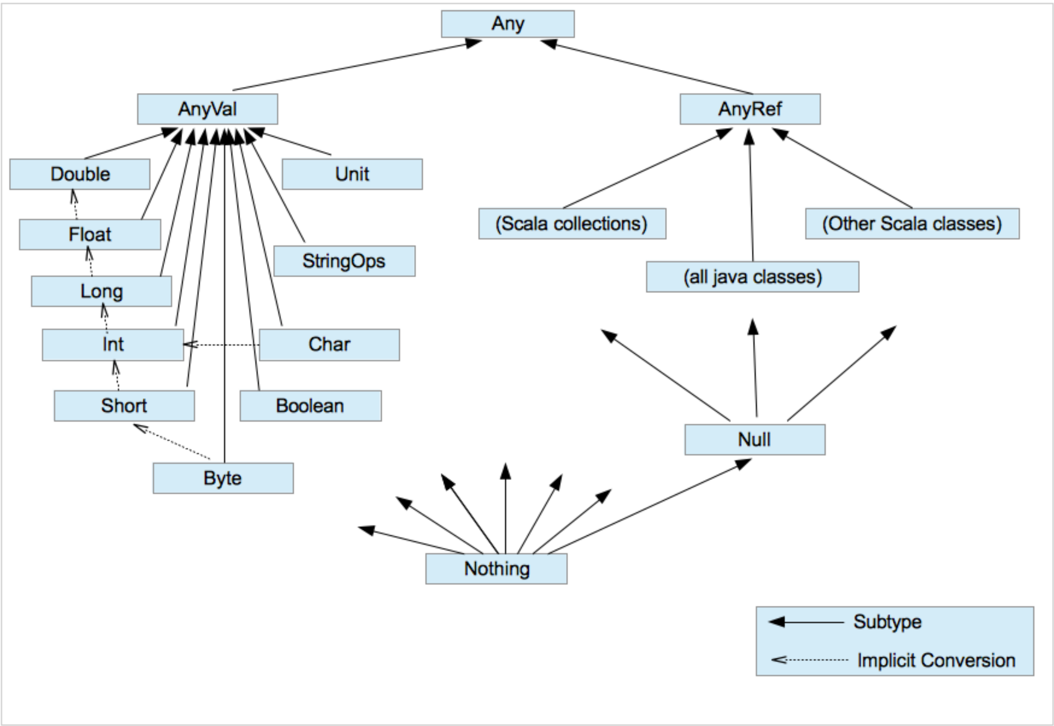
Top Classes

- OO Languages typically have an object at the top of the type hierarchy
  - e.g. Object in Java
- Scala has 3 such Top Classes:
  - `Any` is the absolute top of the hierarchy, everything in Scala is an `Any` (including instances)
  - `AnyVal` is under `Any`, but above all of the "primitive" types (+ `Unit`)
  - `AnyRef` is equivalent to Java's `Object`, it is above all user defined class types
- `AnyRef`s can be `new`ed to make instances
- `object`s are also `AnyRef`s
- `Any` has methods: `toString`, `equals`, `hashCode`, `==`, `##`, `!=`
- `AnyRef` has methods: `isInstanceOf[T]`, `asInstanceOf[T]`, `eq`, `ne`, `synchronized`, `wait`, `notify`, `notifyAll`

Top Types Example

```
val s: String = "hello"

val sa: Any = s          // OK
val sar: AnyRef = s      // OK
val sav: AnyVal = s      // Compile Error

val i: Int = 10

val ia: Any = i          // OK
val iav: AnyVal = i      // OK
val iar: AnyRef = i      // Does not compile

ia.isInstanceOf[Int]     // true
ia.asInstanceOf[Int]     // Int: 10
ia.asInstanceOf[String]// Class cast exception

sa.isInstanceOf[String]// true
sa.asInstanceOf[String]// String: hello
sa.asInstanceOf[Int]     // Class cast exception
```

Organization From the Top



- The "primitives" and classes fan out from the top `Any` types
- They also fan back in to `Null` and `Nothing` types, we'll look at those next

Bottom Classes

- `null` is a concept familiar in many other languages
- It can be assigned to any reference type in those languages to denote the absence of a reference
- "It's as though there is a `Null` type that is a sub-class of all other types, with a single `null` instance" -- Paraphrased from the Java Language Spec
- In Scala this is literally the case, `Null` is a type, it is a sub-class of the whole set of `AnyRefs`, and there is a single instance `null` of that type
- There is also a `Nothing` type which is the sub-type of everything in the Scala type system. There is **no instance** of Nothing, nor can there be one. It exists solely to complete the type system.

Null and Nothing

```
val s1: String = "hello"

s1.charAt(1)  // Char: e

val s2: String = null

s2.charAt(1)  // Null pointer exception!

s1.isInstanceOf[String]  // true
s2.isInstanceOf[String]  // false
```

- For `isInstanceOf` checks, `null` will always return `false`
- How can `Nothing` be useful?

```
val emptyList = List.empty  // List[Nothing]

1 :: emptyList              // List[Int] = List(1)
"hello" :: emptyList        // List[String] = List(hello)
```

Even More Nothing

- Is `Nothing` used for anything other than bottom type parameters?

```
def fail(msg: String): Nothing =
  throw new IllegalStateException(msg)
```

- A method with a `Nothing` return type must throw an exception

- Why is `Nothing` useful?
    - It completes the type system…

Scala Type Calculus (Simplified)

- if type `A` is a subtype of `AnyRef`, `A + Null` is `A` (because `Null` is a sub-type of all `AnyRefs`, so `A` becomes the LUB (Least Upper Bounds)
- For **any** type `A` in Scala, `A + Nothing` is `A` (because `Nothing` is a sub-type of everything so the same reasoning as above applies for the LUB)
- If `B` and `C` both sub-class `A` then `B + C` is `A`, that being the LUB (the first place the type-hierarchy coverges for `B` and `C` as you go up through the super-classes)
- E.g. practical examples (using an `if` expression with two return types):

```
val flag = true  // could be false...
if (flag) 1.0 else ()  // Double + Unit = AnyVal
if (flag) 1.0  // implicit Unit, Double + Unit = AnyVal
if (flag) "hi" // implicit Unit, String + Unit = Any
if (flag) "Hello" else null // String + Null = String
if (flag) 2.0 else null // Double + Null = Any
def fail(msg: String): Nothing =
  throw new IllegalStateException(msg)
if (flag) 2.0 else fail("not 2.0") // Double + Nothing = Double
if (flag) "yo" else fail("no yo") // String + Nothing = String
```

Scala Type Inference Tricks

- Left to its own devices, Scala will often infer more than you need:

```
trait Fruit
case class Apple(name: String) extends Fruit
case class Orange(name: String) extends Fruit

if (true) Apple("Fiji") else Orange("Jaffa")
// Product with Serializable with Fruit = Apple(Fiji)

List(Apple("fiji"), Orange("Jaffa"))
// List[Product with Serializable with Fruit] = List(Apple(fiji), Orange(Jaffa))
```

- Since `case` classes extend both `Product` and `Serializable`
- You can explicitly type the result:

```
val result: Fruit = if (true) Apple("Fiji") else Orange("Jaffa")
```

- Or you can add `Product` and `Serializable` to the superclass:

```
trait Fruit extends Product with Serializable
```

Primitives and Implicit Conversions

- Most languages have a form of Implicit Conversion, often referred to as *type coercion*
- E.g. in Java you can call a method expecting a `long` with an `int`
- Type widening of primitives in Scala is achieved with `implicits`
- And you can use them (carefully) for your own purposes as well
- Unlike Java, Scala makes no distinction in written code between primitives and boxed types
- Also while methods can be invoked like `1.+(2)` behind the scenes Scala implements these efficiently on primitive types
- In addition to implicit widenings, and implicit adapters for Java types, there are also "Rich" wrappers for primitive types and other common classes (like Strings)

Rich Wrappers

```
val d1 = -1.5  // Double = -1.5
val d2 = 1.5   // Double = 1.5

d1.abs         // Double = 1.5
d1 min d2      // Double = -1.5
d1 max d2      // Double = 1.5

d1.floor       // Double = -2.0
d2.ceil        // Double = 2.0
d2.round       // Long = 2

val str = "hello"  // String = hello
str.reverse        // String = olleh
str.toSeq          // Seq[Char] = hello
str.slice(2, 4)    // String = ll
```

- Double methods above come from `RichDouble` brought in by `Predef`
- String methods come from `SeqLike` and `WrappedString` also via `Predef`

@specialized

- Autoboxing in Generics can introduce unwanted overhead
- `@specialized` can be used to generate templated specific versions for primitives:

```
def sumOf[@specialized(Int, Double, Long) T: Numeric](items: T*): T = {
  val numeric = implicitly[Numeric[T]]
  items.foldLeft(numeric.zero)(numeric.plus)
}
sumOf(1,2,3)
```

- In this case there will be one version of this method for `AnyRefs` + one each of `Int`, `Double`, and `Long`
- If you overuse this, you can end up with a type matrix explosion:

```
def pair[@specialized T, @specialized U](t: T, u: U): (T, U) = (t, u)
```

- Will produce 100 implementations of `pair` (9 primitives + `AnyRef`) * (9 primitives + `AnyRef`)

@specialized generation

```
$ javap Demo$.class
public final class Demo$ {
  public static Demo$ MODULE$;
  public static {};
  public <T, U> scala.Tuple2<T, U> pair(T, U);
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZZc$sp(boolean, bo
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZBc$sp(boolean, by
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZCc$sp(boolean, ch
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZDc$sp(boolean, do
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZFc$sp(boolean, fl
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZIc$sp(boolean, in
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZJc$sp(boolean, lo
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mZSc$sp(boolean, sh
  public scala.Tuple2<java.lang.Object, scala.runtime.BoxedUnit> pair$mZVc$sp(bool
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mBZc$sp(byte, boole
  public scala.Tuple2<java.lang.Object, java.lang.Object> pair$mBBc$sp(byte, byte)
...
```

- And it didn't even work - look at that return type

Extension Methods and Implicit Classes

- `RichDouble` and `WrappedString` introduce extension methods on existing classes
- We can do that too:

```
implicit class TimesDo(i: Int) {
  def times(fn: => Unit): Unit = {
    for (_ <- 1 to i) fn
  }
}

5 times {
  println("hello")
}
```

- `implicit classes` combine the class definition and an implicit conversion from the single type parameter to the class
- As such, because there is an `implicit def` included, implicit classes may only go inside another class, object or package object.

Value class (`extends AnyVal`)

- The implicit def also means that the `Int` is actually wrapped in a new instance to run `times`

- adding `extends AnyVal` avoids this wrapping when possible (and also makes the definition work *only* outside of an enclosing class):

```
implicit class TimesDo(val i: Int) extends AnyVal {
  def times(fn: => Unit): Unit = {
    for (_ <- 1 to i) fn
  }
}

5 times {
  println("hello")
}
```

- To extend `AnyVal` you must:
  - Have a single public parametric field
  - Have no other state in the class (because there may be no instance to hold the state)

- Behind the scenes, static methods are used

Nil, Null, Nothing, None

- Scala has several "negative" types:
- `Nil` is the empty List, and is also the terminator of every `List`
- `Null` and its single instance `null` is the absence of an `AnyRef` instance
- `Nothing` is a type without an instance, and implies an exception being thrown
- `None` is the absence of a `Some` in the `Option` type, and is a safer alternative to `null`

Option

- `null` is the absence of an instance. The compiler cannot help you:

```
val s1: String = "hello"   // compiles
val s2: String = null      // likewise

s1.length                  // compiles, gives 5
s2.length                  // compiles, NullPointerException
```

- Making an `Option` type means the compiler has your back:

```
val os1: Option[String] = Some("hello")
val os2: Option[String] = None

os1.length   // will not compile

os1.map(_.length) // Some(5)
os2.map(_.length) // None
```

- This takes a common runtime error, and moves it into a compile time concern

Working with `Option`

- Option is supported throughout the core libraries and third party APIs

```
val numWords = Map(1 -> "one", 2 -> "two", 3 -> "three")
numWords(1) // one
numWords(4) // NoSuchElementException
val word1 = numWords.get(1) // Some("one")
val word2 = numWords.get(4) // None
```

- You can pattern match:

```
word1 match {
  case Some(word) => word
  case None => "unknown"
} // one
```

- Use `getOrElse`:

```
word2.getOrElse("unknown") // "unknown"
```

Working with Option

- Or compose options with `for` expressions:

```
def fourthLetter(i: Int): Option[Char] = for {
  word <- numWords.get(i)
  char <- word.drop(4).headOption
} yield char
fourthLetter(1)  // None
fourthLetter(3)  // Some(e)
```

- Mixing Options and Collections:

```
def fourthLetters(nums: Seq[Int]): Seq[Char] = for {
  i <- nums
  word <- numWords.get(i).toSeq
  char <- word.drop(4).headOption.toSeq
} yield char
fourthLetters(List(1, 2, 3))  // List('e')
```

- In this case `toSeq` on the options is not required, but is recommended
- Collection following an option in a for expression needs `toSeq`

`equals` and `hashCode`

- In Scala, == is aliased to call `.equals` and is marked `final`
- The `equals` and `hashCode` contract are hard to get right and defaults are not useful

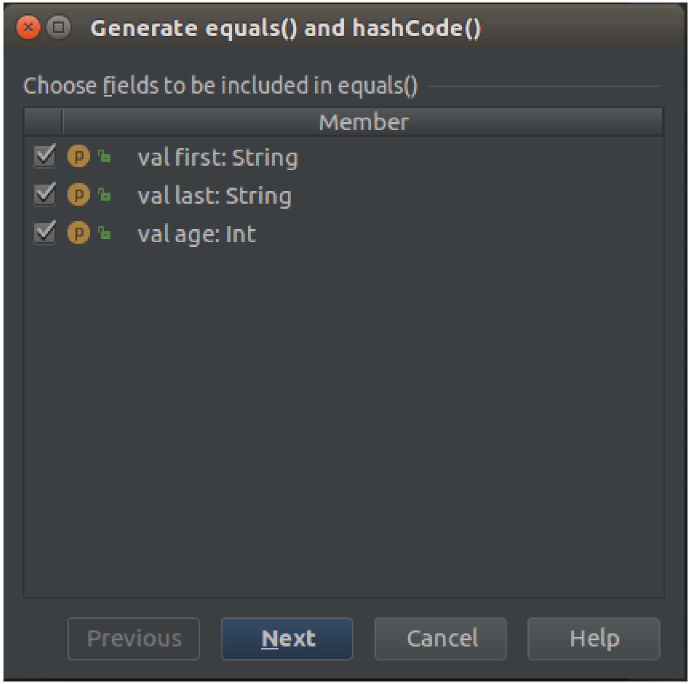
```scala
class Person(val first: String, val last: String, val age: Int)

val p1 = new Person("Wavey", "Davey", 25)
val p2 = new Person("Wavey", "Davey", 25)

p1 == p2  // false
p1.##     // 1466295063
p2.##     // 405671158
```

- So we will usually need to provide better equals/hashCode if we want to use these in collections, etc.

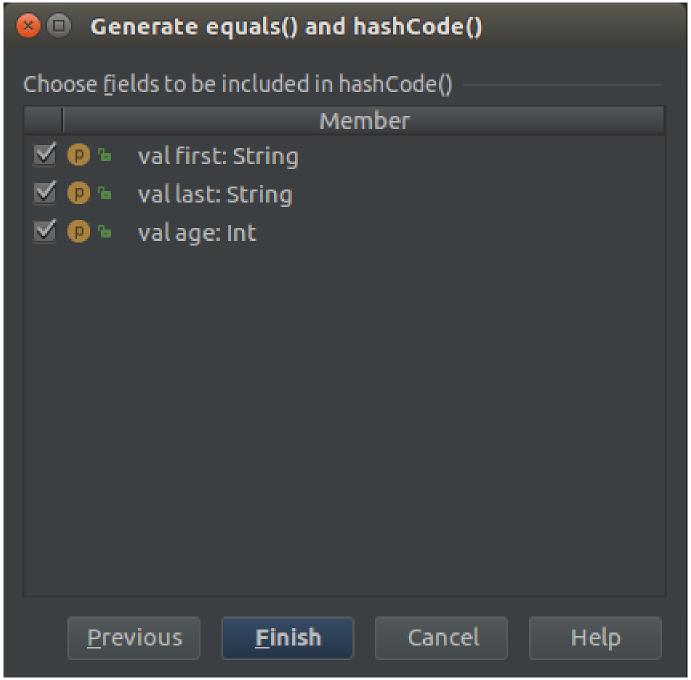
Option 1, Generate with IDEA

- Alt+insert, equals and hashCode, Select fields and Done

equals



hashCode



Option 2, follow this formula:

```
class Fruit(val name: String) {

  def canEqual(other: Any): Boolean = other.isInstanceOf[Fruit]

  override def equals(other: Any): Boolean = other match {
    case that: Fruit =>
      (that canEqual this) &&
        name == that.name
    case _ => false
  }

  override def hashCode(): Int = name.hashCode
}
```

- Simple case, no inheritance, only one field for `hashCode`

Continued... sub-classes

```scala
class Apple(val brand: String, val color: String) extends Fruit("apple") {

  override def canEqual(other: Any): Boolean = other.isInstanceOf[Apple]

  override def equals(other: Any): Boolean = other match {
    case that: Apple =>
      super.equals(that) &&
        (that canEqual this) &&
        brand == that.brand &&
        color == that.color
    case _ => false
  }

  override def hashCode(): Int = {
    41 * (
      41 * (
        41 + super.hashCode
      ) + brand.hashCode
    ) + color.hashCode
  }
}
```

- Remember `super` for both `equals` and `hashCode`

Option 3, just use `case` classes

```
case class Banana(brand: String, ripe: Boolean) extends Fruit("banana")

val b1 = Banana("Ffyfes", true)
val b2 = Banana("Ffyfes", true)

b1 == b2  // true
b1.##     // -1396355227
b2.##     // -1396355227
```

- But!
  - Superclass `equals` must still be correct if present (will not be used if omitted)
  - `case classes` cannot extend other `case classes`
  - Technically all `case classes` should be marked `final`

Product Types

- `case classes` and tuples are `Product` types
- Their equality is determined by their public state and (for case classes) their type

```
1, '2', "three") == (1, '2', "three")                // true
(1, 'a', "three") == (1, '2', "three")               // false

case class Triplet1(i: Int, ch: Char, str: String)

Triplet1(1, '2', "three") == Triplet1(1, '2', "three")   // true
Triplet1(1, '2', "three") == Triplet1(2, '2', "three")   // false
Triplet1(1, '2', "three") == (1, '2', "three")           // false

case class Triplet2(i: Int, ch: Char, str: String)

Triplet2(1, '2', "three") == Triplet2(1, '2', "three")   // true
Triplet2(1, '2', "three") == Triplet2(2, '2', "three")   // false
Triplet1(1, '2', "three") == Triplet2(1, '2', "three")   // false
Triplet2(1, '2', "three") == (1, '2', "three")           // false
```

Product Type Features

```
val triplet2 = Triplet2(1, '2', "three")

triplet2.productIterator.toList  // List[Any] = List(1, 2, three)
triplet2.productArity            // Int = 3
triplet2.productElement(2)       // Any = three
triplet2.productPrefix           // String = Triplet2

val tup3 = (1, '2', "three")

tup3.productIterator.toList      // List[Any] = List(1, 2, three)
tup3.productArity                // Int = 3
tup3.productElement(2)           // Any = three
tup3.productPrefix               // String = Tuple3
```

- Much more on `case classes` in a later module

Exercises for Module 8

- Find the `Module08` class and follow the instructions to make the tests pass