

Your Scala code on steroids with Type Classes

dgk42

Learning new stuff...



Learning new stuff...

- Scala
 - Odersky et al's MOOCs
 - Twitter Scala School
- Haskell
 - Learn You a Haskell for Great Good! (LYaHfGG)
 - Haskell Wikibook

Basic Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism

...And some little problems...

Joe Armstrong (Erlang):

"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle!"

Polymorphism VS. Inheritance

- Polymorphism lets you call methods of a class without knowing the exact type of the class
- Inheritance lets derived classes share interfaces and code of their base classes

Polymorphism

- Subtype polymorphism

 - Notion of substitutability

 - OOP: polymorphism using inheritance

- Parametric polymorphism

 - Java / Scala: with generics

- Ad-hoc polymorphism

 - Haskell / Scala: with type classes, F-bounded polymorphism (explicitly in Scala)

Some type classes pros

- Everything resolved at compile-time
- Type safety
- Plenty of room for compiler optimizations
- A form of retroactive supertyping that avoids structural types (their implementation needs reflection)
- Cleaner and more straight-forward code

Some type classes cons

- Closer to the FP paradigm than the OOP counterpart (although FP and OOP are orthogonal)

"Everything is an object"???

Must read: https://wiki.haskell.org/OOP_vs_type_classes

- "Traditional programmers" are not familiar with them
- In Scala: implicits all over the place!
- In Scala: chains of type classes are tricky!

A trivial example



DOUBLE FACEPALM

FOR WHEN ONE FACEPALM DOESN'T CUT IT

A trivial example

- We want to express approximate equality for a given type
- Approximate equality for numbers

A number is approximately equal to another number iff the former is in a specified range of the latter

- The code is located at:

<https://github.com/dgk42/ScalaTypeClassesExample>

Examples for $\text{range} == 0.001$

$42 \sim 42$ is true $|42 - 42| == 0 < 0.001$

$42 \sim 69105$ is false $|42 - 69105| == 69063 > 0.001$

$5.01 \sim 5.02$ is false $|5.01 - 5.02| == 0.01 > 0.001$

$5.00005 \sim 5.0008$ is true

$|5.00005 - 5.0008| == 0.00075 < 0.001$

$5.1 + 2.8j \sim 5.1001 + 2.805j$ is false

$\text{sqrt}((5.1 - 5.1001)^2 + (2.8 - 2.805)^2) \sim 0.005 > 0.001$

Without type classes

```
trait WithDistanceG[T] {  
  def v: T  
  def distance(that: T): Double  
}  
  
trait ApproxEqualG[T] extends WithDistanceG[T] {  
  def approxEqual(that: ApproxEqualG[T]): Boolean = distance(that.v) < 0.001  
  
  def == = approxEqual(_)  
  def ==# = { that: ApproxEqualG[T] =>  
    !approxEqual(that)  
  }  
}
```

With type classes

```
trait WithDistanceTC[T] {  
  def distance(t1: T, t2: T): Double  
}
```

```
trait ApproxEqualTC[T] {  
  def approxEqual(t1: T, t2: T)(implicit ev: WithDistanceTC[T]): Boolean =  
    (ev distance (t1, t2)) < 0.001  
}
```

With type classes (cont'd)

```
trait ApproxEqualTCSyntax[T] {  
  def =~=(t: T): Boolean  
  def ==#(t: T): Boolean  
}  
  
object ApproxEqualTCSyntax {  
  implicit def approxEqualTCSyntax[T](t1: T) (  
    implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T]): ApproxEqualTCSyntax[T] = {  
  
    new ApproxEqualTCSyntax[T] {  
      def =~=(t2: T): Boolean = ev2.approxEqual (t1, t2)  
      def ==#(t2: T): Boolean = !(t1 =~= t2)  
    }  
  }  
}
```

Let's implement approx. equality for Ints

Without type classes

```
class IntApproxEqualG(val v: Int) extends ApproxEqualG[Int] {  
  def distance(that: Int): Double = math.abs(v - that).toDouble  
}
```

With type classes

```
object IntWithDistanceTC {  
  implicit val intHasDistance = new WithDistanceTC[Int] {  
    def distance(t1: Int, t2: Int): Double = math.abs(t1 - t2).toDouble  
  }  
}
```

```
object IntApproxEqualTC {  
  implicit val intIsApproxEqual = new ApproxEqualTC[Int] {}  
}
```

```
// reference: trait ApproxEqualTC[T] {  
//   def approxEqual(t1: T, t2: T)(implicit ev: WithDistanceTC[T]): Boolean =  
//     (ev distance (t1, t2)) < 0.001  
// }
```

Usage example

Usage example

Without type classes

```
object GExample extends App {  
  val t11 = new IntApproxEqualG(42)  
  val t12 = new IntApproxEqualG(42)  
  println(t11 == t12)  
}
```

With type classes

```
object TCExample extends App {  
  import IntWithDistanceTC._  
  import IntApproxEqualTC._  
  import ApproxEqualTCSyntax._  
  
  println(42 ~= 42)  
}
```

Notice that with type classes

- We operate on the `Int` type directly (we don't need a wrapper class)
- We don't need a `WithDistanceTC` implementation (although we provide one).

This will eventually be provided at the call site (that is, `TCExample` in our case)

Notice that with type classes

- `IntWithDistanceTC`: We employ a form closer to the FP paradigm (distance with 2 arguments - compare with `IntApproxEqualG`)
- The singleton objects hold the implicit values.
We don't spawn new objects all the time

Let's repeat this for a class of our own

```
// A trivial 2D vector implementation.  
case class Vec2D(x: Double, y: Double) {  
  def euclideanDistance(that: Vec2D): Double = {  
    val dX = x - that.x  
    val dY = y - that.y  
    math.sqrt(dX * dX + dY * dY)  
  }  
}
```

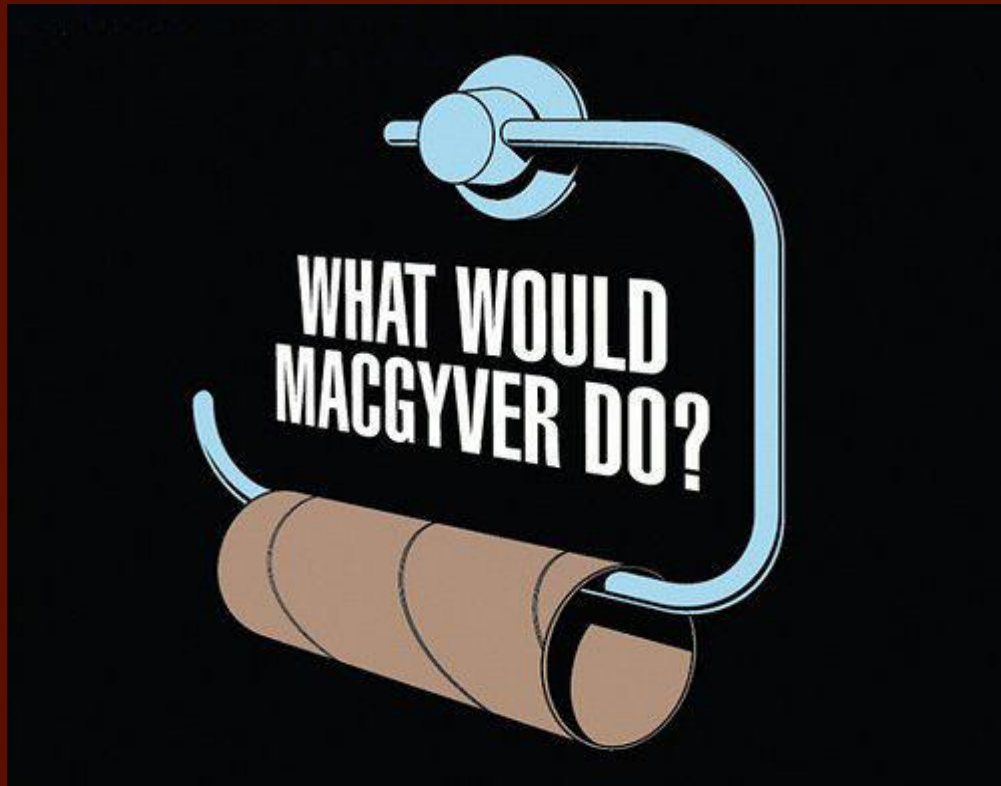

Let's repeat this for a class of our own (cont'd)

```
def manhattanDistance(that: Vec2D): Double = {  
    val dX = math.abs(x - that.x)  
    val dY = math.abs(y - that.y)  
    dX + dY  
}  
}
```

Important

We want to be able to choose the distance metric that will be used for the approximate equality attribute at the call site.

Important



Without type classes

```
class Vec2D1ApproxEqualG(val v: Vec2D) extends ApproxEqualG[Vec2D] {  
  def distance(that: Vec2D): Double = v.euclideanDistance(that)  
}
```

```
class Vec2D2ApproxEqualG(val v: Vec2D) extends ApproxEqualG[Vec2D] {  
  def distance(that: Vec2D): Double = v.manhattanDistance(that)  
}
```

With type classes

```
object Vec2DWithDistanceTC {  
  implicit val vec2DHasDistance = new WithDistanceTC[Vec2D] {  
    def distance(t1: Vec2D, t2: Vec2D): Double = t1.euclideanDistance(t2)  
  }  
}
```

```
object AnotherVec2DWithDistanceTC {  
  implicit val anotherVec2DHasDistance = new WithDistanceTC[Vec2D] {  
    def distance(t1: Vec2D, t2: Vec2D): Double = t1.manhattanDistance(t2)  
  }  
}
```

With type classes (cont'd)

```
object Vec2DApproxEqualTC {  
  implicit val vec2DIsApproxEqual = new ApproxEqualTC[Vec2D] {}  
}  
  
// reference: trait ApproxEqualTC[T] {  
//   def approxEqual(t1: T, t2: T)(implicit ev: WithDistanceTC[T]): Boolean =  
//     (ev distance (t1, t2)) < 0.001  
// }
```

With type classes

Notice that the `Vec2DApproxEqualTC`'s implicit val doesn't know anything about any distance metric.

Conclusion

- A natural way of expressing properties in domain entities
- "Pimp my lib" pattern
- Superior pattern to F-bounded polymorphism
- Scalaz is full of these!
- We love Haskell \Leftrightarrow We love Scalaz

BONUS STAGE



BONUS STAGE

Property testing with ScalaCheck and ScalaTest

BONUS STAGE

```
trait ApproxEqualTCLaws[T] {  
  implicit val ev1: WithDistanceTC[T]  
  implicit val ev2: ApproxEqualTC[T]  
  
  def commutative(t1: T, t2: T): Boolean =  
    (ev2 approxEqual (t1, t2)) == (ev2 approxEqual (t2, t1))  
  def reflexive(t: T): Boolean = ev2 approxEqual (t, t)  
  def transitive(t1: T, t2: T, t3: T): Boolean =  
    conditional(  
      (ev2 approxEqual (t1, t2)) && (ev2 approxEqual (t2, t3)),  
      (ev2 approxEqual (t1, t3))  
    )  
}
```

BONUS STAGE

```
object ApproxEqualTCLawProperties {  
  def approxEqualLaw[T](implicit e1: WithDistanceTC[T], e2: ApproxEqualTC[T]) =  
    new ApproxEqualTCLaws[T] {  
      val ev1 = e1  
      val ev2 = e2  
    }  
}
```

BONUS STAGE

```
def commutativity[T] (
  implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T], ev3: Arbitrary[T]): Prop =
{
  forall (approxEqualLaw.commutative _)
}

def reflexivity[T] (
  implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T], ev3: Arbitrary[T]): Prop =
{
  forall (approxEqualLaw.reflexive _)
}
```

BONUS STAGE

```
// WATCH OUT for this one! It bites!
def transitivity[T](
  implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T], ev3: Arbitrary[T]): Prop =
{
  forall (approxEqualLaw.transitive _)
}
```

BONUS STAGE

```
def laws[T] (
  implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T], ev3: Arbitrary[T]):
Properties = {

  new Properties("approxEqualTC") {
    property("commutativity") = commutativity[T]
    property("reflexivity") = reflexivity[T]
    property("transitivity") = transitivity[T]
  }
}
```

BONUS STAGE

```
trait CheckTCLaws extends FunSuite with Checkers {  
  def checkLaws[T] (  
    implicit ev1: WithDistanceTC[T], ev2: ApproxEqualTC[T], ev3: Arbitrary[T],  
    ev4: ClassTag[T]): Unit = {  
  
    ApproxEqualTCLawProperties.laws[T].properties foreach {  
      case (name, law) =>  
        test(s"$name for ${ev4.toString}") {  
          check(law)  
        }  
    }  
  }  
}
```


BONUS STAGE

```
class ApproxTCTest extends CheckTCLaws {  
  import IntWithDistanceTC._  
  import IntApproxEqualTC._  
  import DoubleWithDistanceTC._  
  import DoubleApproxEqualTC._  
  
  checkLaws[Int]  
  checkLaws[Double]  
}
```

Thank you! Your turn

<https://www.youtube.com/watch?v=vP1QNz-9tf0>

Q & A