

# Type Class: The Ultimate Ad Hoc

George Wilson

Data61/CSIRO

[george.wilson@data61.csiro.au](mailto:george.wilson@data61.csiro.au)

August 3, 2017



Type classes are a language feature

- ▶ Haskell
- ▶ Eta
- ▶ Purescript
- ▶ Clean

Type classes are a language feature

- ▶ Haskell
- ▶ Eta
- ▶ Purescript
- ▶ Clean

or sometimes a design pattern

- ▶ Scala
- ▶ OCaml

# Polymorphism

A type is *polymorphic* iff it can be applied at multiple types.

## Polymorphism is good

- ▶ less duplication
- ▶ more reuse
- ▶ fewer possible implementations

Broadly speaking there are two major forms of polymorphism:

- ▶ *parametric* polymorphism
- ▶ *ad-hoc* polymorphism

A type is parametrically polymorphic iff it has at least one *type parameter* which can be instantiated to *any type*.

```
reverse :: [a] -> [a]
```

```
id :: a -> a
```

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```



A type which is ad-hocly polymorphic can be instantiated to different types, and may behave differently at each type

# Interfaces

```
interface Equal<A> {  
    public boolean eq(A other);  
}
```

```
interface Equal<A> {  
    public boolean eq(A other);  
}  
  
class Person implements Equal<Person> {  
    public int age;  
    public String name;  
  
    public boolean eq(Person other) {  
        return this.age == other.age && this.name.equals(other.name);  
    }  
}
```

```
static <A extends Equal<A>> boolean elementOf(A a, List<A> list) {  
    for (A element : list) {  
        if (a.eq(element)) return true;  
    }  
    return false;  
}
```

```
static <A extends Equal<A>> boolean elementOf(A a, List<A> list) {  
    for (A element : list) {  
        if (a.eq(element)) return true;  
    }  
    return false;  
}
```

```
elementOf(me, functionalProgrammers);  
// true
```

```
package java.lang;
```

```
class String {  
    private char[] value;  
    // other definitions  
}
```

```
package java.lang;
```

```
class String implements Equal<String> {  
    private char[] value;  
    // other definitions  
}
```



```
class List<A> {  
    // implementation details
```

```
}
```

```
class List<A> implements Equal<List<A>> {  
    // implementation details  
  
}
```

```
class List<A> implements Equal<List<A>> {  
    // implementation details  
  
    public boolean eq(List<A> other) {  
        // implementation...  
    }  
}
```

```
class List<A> implements Equal<List<A>> {  
    // implementation details  
  
    public boolean eq(List<A> other) {  
        // implementation...  
        // ... but how do we compare A for equality?  
    }  
}
```

- ▶ Interface implementation can't be conditional
- ▶ We can only implement interfaces for types we control

# Type Classes

```
class Equal a where  
  eq :: a -> a -> Bool
```

```
class Equal a where  
  eq :: a -> a -> Bool
```

```
data Person = Person {  
  age :: Int  
, name :: String  
}
```



```
class Equal a where  
    eq :: a -> a -> Bool
```

```
data Person = Person {  
    age :: Int  
, name :: String  
}
```

```
instance Equal Person where  
    eq p1 p2 = eq (age p1) (age p2) && eq (name p1) (name p2)
```

```
elementOf :: Equal a => a -> [a] -> Bool
elementOf a list =
  case list of
    []      -> False
    (h:t)   -> eq a h || elementOf a t
```

Instances can be constrained

```
instance (Equal a) => Equal [a] where
  eq [] [] = True
  eq (x:xs) [] = False
  eq [] (y:ys) = False
  eq (x:xs) (y:ys) = eq x y && eq xs ys
```

Instances can be constrained

```
instance (Equal a) => Equal [a] where
  eq [] [] = True
  eq (x:xs) [] = False
  eq [] (y:ys) = False
  eq (x:xs) (y:ys) = eq x y && eq xs ys
```

We can add type class instances for types we didn't write

- ▶ Writing an instance for your type can unlock many functions
- ▶ Writing functions with typeclass constraints is easy
- ▶ You can write instances for types you did not write
- ▶ Instances can depend on other instances if necessary

talk about where instances are allowed to be  
talk about orphan instances

TODO show why type classes aren't perfectly flexible  
(no custom local instances, maybe show newtypes for sum and product or something)

Implicits



```
case class Person(age: Int, name: String)
```

```
case class Person(age: Int, name: String)
```

```
trait Equal[A] {  
  def eq(a: A, b: A): Boolean  
}
```

```
case class Person(age: Int, name: String)
```

```
trait Equal[A] {  
  def eq(a: A, b: A): Boolean  
}
```

```
implicit def equalPerson: Equal[Person] = new Equal[Person] {  
  def eq(a: Person, b: Person): Boolean =  
    a.age == b.age && a.name == b.name  
}
```

```
def elementOf[A] (a: A, list: List[A])  
    (implicit equalA: Equal[A]): Boolean = {  
  list match {  
    case Nil => false  
    case (h::t) => equal.eq(a, h) || elementOf(a, t)  
  }  
}
```

```
implicit def equalList(implicit equalA: Equal[A]): Equal[List[A]] =  
  new Equal[List[A]] {  
    def eq(a: List[A], b: List[A]): Boolean = {  
      (a,b) match {  
        case (Nil, Nil)      => true  
        case (x::xs, Nil)    => false  
        case (Nil, y::ys)    => false  
        case (x::xs, y::ys) => equalA.eq(x,y) || eq(xs,ys)  
      }  
    }  
  }
```

- ▶ We can define implicits for types we did not write
- ▶ We can write implicits that depend on implicits

```
sealed trait Ordering  
case object LT extends Ordering  
case object EQ extends Ordering  
case object GT extends Ordering
```

```
trait Order[A] {  
  def compare(a: A, b: A): Ordering  
}
```

```
sealed trait Ordering
case object LT extends Ordering
case object EQ extends Ordering
case object GT extends Ordering
```

```
trait Order[A] {
  def compare(a: A, b: A): Ordering
}
```

```
implicit def orderPerson: Order[Person] = new Order[Person] {
  def compare(a: Person, b: Person): Ordering =
    intOrder.compare(a.age, b.age) match {
      case LT => LT
      case EQ => stringOrder.compare(a.name, b.name)
      case GT => GT
    }
}
```



```
def sort[A](list: List[A])(implicit orderA: Order[A]): List[A] = {  
  // quicksort goes here  
}
```

```
sort (  
  List (  
    Person (30, "Robert")  
  ,   Person (20, "John")  
  ,   Person (40, "Alfred")  
  )  
)
```

```
sort (  
  List (  
    Person (30, "Robert")  
  ,   Person (20, "John")  
  ,   Person (40, "Alfred")  
  )  
)
```

==>

```
List (  
  Person (20, "John")  
,   Person (30, "Robert")  
,   Person (40, "Alfred")  
)
```

Then the boss says “I want those sorted by name”.

Then the boss says “I want those sorted by name”.

```
implicit def personOrderByName: Order[Person] = new Order[Person] {  
  def compare(a: Person, b: Person): Ordering =  
    stringOrder.compare(a.age, b.age)  
}
```

```
sort (  
  List (  
    Person (30, "Robert")  
  ,   Person (20, "John")  
  ,   Person (40, "Alfred")  
  )  
)
```

```
sort (  
  List (  
    Person (30, "Robert")  
  , Person (20, "John")  
  , Person (40, "Alfred")  
  )  
)
```

==>

```
List (  
  Person (40, "Alfred")  
  , Person (20, "John")  
  , Person (30, "Robert")  
)
```

talk about how impossible it is to understand understand which implicit wins

show Set example about incoherence doing nasty things quietly

talk about applying discipline in scala to get coherence