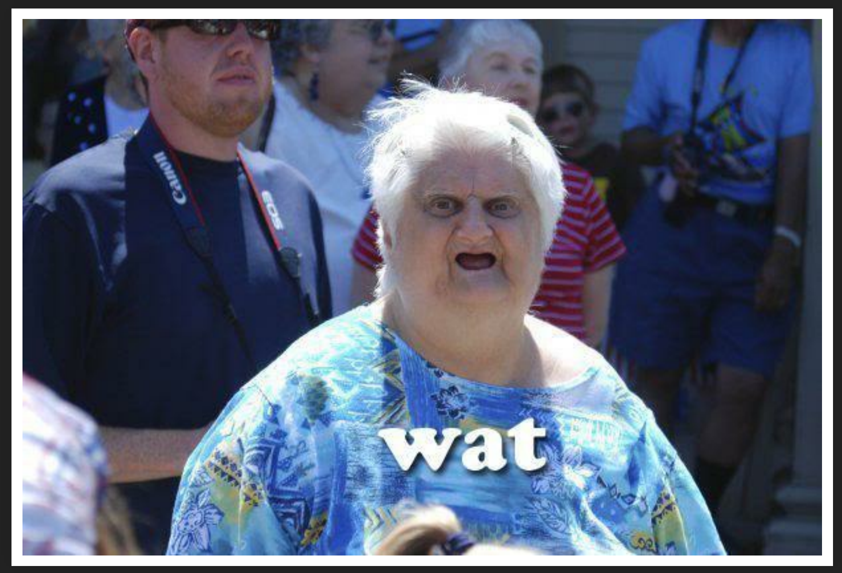
INTRODUCTION TO TYPE CLASSES

OBJECTIVE OF THIS TALK

Avoid



when presenting functional programming some day.

TYPE CLASS

A type class is a type system construct that supports ad hoc polymorphism.

POLYMORPHISM

> *Polymorphism is the provision of a single interface to entities of different types.*

- Subtyping: when a name denotes instances of many different classes related by some common superclass.
- Parametric polymorphism: when code is written without mention of any specific type and thus can be used transparently with any number of new types.
- Ad hoc polymorphism: when a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations.

SUBTYPING

```scala
trait Animal {
    def name: String
}

case class Fish(name: String) extends Animal

def sayHello(animal: Animal): String = s"Hello ${animal.name}"
```

```
scala> sayHello(Fish("Nemo"))
res2: String = Hello Nemo
```

PARAMETRIC POLYMORPHISM

Usually called "generics".

```
scala> def sayHello[A](a: A): String = s"Hello ${a.toString}"
sayHello: [A](a: A)String

scala> sayHello(Fish("Nemo"))
res3: String = Hello Fish(Nemo)
```

AD HOC POLYMORPHISM

Function overloading, not supported in Scala.

```
def add(x: String, y: String): String = x + y
def add(x: Double, y: Double): Double = x + y
```

AD HOC POLYMORPHISM

How is it done in Scala then? Type class

```
scala> def add[T](x: T, y: T)(implicit numeric: Numeric[T]):T = numeric.plus(x, y)
add: [T](x: T, y: T)(implicit numeric: Numeric[T])T

scala> add(3, 4)
res4: Int = 7

scala> add(3.0, 4.0)
res5: Double = 7.0
```

```
scala> def add[T : Numeric](x: T, y: T):T = implicitly[Numeric[T]].plus(x, y)
add: [T](x: T, y: T)(implicit evidence$1: Numeric[T])T

scala> add(3, 4)
res6: Int = 7

scala> add(3.0, 4.0)
res7: Double = 7.0
```

(ALMOST) A REAL WORLD EXAMPLE

```
scala> def sum(ns: List[Int]): Int = ns.fold(0)(_ + _)
sum: (ns: List[Int])Int

scala> def all(bs: List[Boolean]): Boolean = bs.fold(true)(_ && _)
all: (bs: List[Boolean])Boolean

scala> def concat[A](ss: List[List[A]]): List[A] = ss.fold(List.empty[A])(_ ::: _)
concat: [A](ss: List[List[A]])List[A]
```

Let's factorize the commonality. We'd like:

```
def genericSum[A](l: List[A]): A // for some adequate A
```

REQUIREMENTS

```
def genericSum[A](l: List[A]): A // for some adequate A
```

- represent *different* types that behave *similarly* (contract)
- can add capabilities to existing types (pimp my class pattern)
- statically checked by compiler

FIRST IDEA: SUBTYPING

Bad idea because:

- no reason to have a common supertype with additive capabilities for `Int`, `Boolean` and `List[A]`
- cannot rewrite `Int`!

```
trait Int extends Addable
```

SECOND IDEA: PARAMETRIC POLYMORPHISM (GENERICS)

```
def add[A](x: A, y: A): A = ??? // same code for Int, Boolean and List[A]
```

Impossible!

THIRD IDEA: AD HOC POLYMORPHISM

This is the one.

WRAPPER

```scala
trait Addable[A] {
  def add(b: Addable[A]): Addable[A]
  def get: A
}

case class AddableList[A](a: List[A]) extends Addable[List[A]] {
  def add(b: Addable[List[A]]): AddableList[A] = AddableList(a ++ b.get)
  def get: List[A] = a
}

def genericSum[A](l: List[Addable[A]]): A = l.reduce(_.add(_)).get
```

```scala
scala> genericSum(List(AddableList(List("foo", "bar")), AddableList(List("yep"))))
res10: List[String] = List(foo, bar, yep)
```

Cumbersome and unsafe (reduce fails when list is empty).

TYPE CLASS

```scala
trait Addable[A] { // type class (the contract)
  def add(a: A, b: A): A
  def zero: A
}

implicit val addableForInt = new Addable[Int] { // instance
  def add(a: Int, b: Int): Int = a + b
  def zero: Int = 0
}

implicit def addableForList[A] = new Addable[List[A]] { // instance
  def add(a: List[A], b: List[A]): List[A] = a ++ b
  def zero: List[A] = List.empty[A]
}

def genericSum[A](l: List[A])(implicit ev: Addable[A]): A = l.fold(ev.zero)(ev.add)
// ev for evidence
```

```
     | genericSum(List(List("foo", "bar"), List("yep")))
res15: List[String] = List(foo, bar, yep)

scala> genericSum(List(5, 3, 4))
res16: Int = 12
```

```
scala> genericSum(List(5L, 3L, 4L))
<console>:16: error: could not find implicit value for parameter ev: Addable[Long]
       genericSum(List(5L, 3L, 4L))
                 ^
```

PIMP MY CLASS PATTERN FOR FREE

```scala
implicit class AddableOps[A](a: A)(implicit ev: Addable[A]) {
  def add(b: A): A = ev.add(a, b)
}
```

```
scala> 4.add(3)
res18: Int = 7

scala> List("foo", "bar").add(List("yep"))
res19: List[String] = List(foo, bar, yep)
```

TYPE CLASS DEGRADES READABILITY

Code obfuscation:

```
def genericSum[F[_] : Foldable, A : Addable](l: F[A]): A =
    Foldable[F].foldLeft(l, Addable[A].zero)(Addable[A].add)
```

Not a real problem, it just takes some getting used to.

TYPE CLASS DEGRADES READABILITY #2

How do you know `List[A]` is `Addable`?

Look for the implicit instance. But:

- you have to track implicits in your code
- methods from pimp-my-class pattern don't appear in scaladoc
- you'd better use an IDE

Example with cats.

BOILERPLATE

```
trait Addable[A] { // type class (the contract)
  def add(a: A, b: A): A
  def zero: A
}

implicit val AddableForInt = new Addable[Int] { // instance
  def add(a: Int, b: Int): Int = a + b
  def zero: Int = 0
}

implicit def addableForList[A] = new Addable[List[A]] { // instance
  def add(a: List[A], b: List[A]): List[A] = a ++ b
  def zero: List[A] = List.empty[A]
}

implicit class AddableOps[A](a: A)(implicit ev: Addable[A]) { // boilerplate
  def add(b: A): A = ev.add(a, b)
}
```

SIMULACRUM

```
import simulacrum._

@typeclass trait Semigroup[A] {
  @op("|+|") def append(x: A, y: A): A
}
```

Generated code:

```
trait Semigroup[A] {
  def append(x: A, y: A): A
}

object Semigroup {
  def apply[A](implicit instance: Semigroup[A]): Semigroup[A] = instance

  trait Ops[A] {
    def typeClassInstance: Semigroup[A]
    def self: A
    def |+|(y: A): A = typeClassInstance.append(self, y)
  }

  trait ToSemigroupOps {
    implicit def toSemigroupOps[A](target: A)(implicit tc: Semigroup[A]): Ops[A] = new Op
      val self = target
      val typeClassInstance = tc
    }
  }
```

SIMULACRUM #2

```
implicit val semigroupInt: Semigroup[Int] = new Semigroup[Int] {
  def append(x: Int, y: Int) = x + y
}

import Semigroup.ops._
1 |+| 2 // 3
```

Simulacrum.

FUNCTIONAL PROGRAMMING LIBRARIES

Libraries	Method
scala.collection	F-bounded polymorphism
cats	Type class
scalaz	Type class

Good introduction to functional programming in Scala: herding cats.
More about typeclasses.

QUESTIONS