https://github.com/benhutchison/ProgrammingWithTypeClasses.git

# PROGRAMMING WITH TYPECLASSES IN SCALA

Ben Hutchison

Lambdajam 2013

realestate.com.au
Australia's No.1 property site

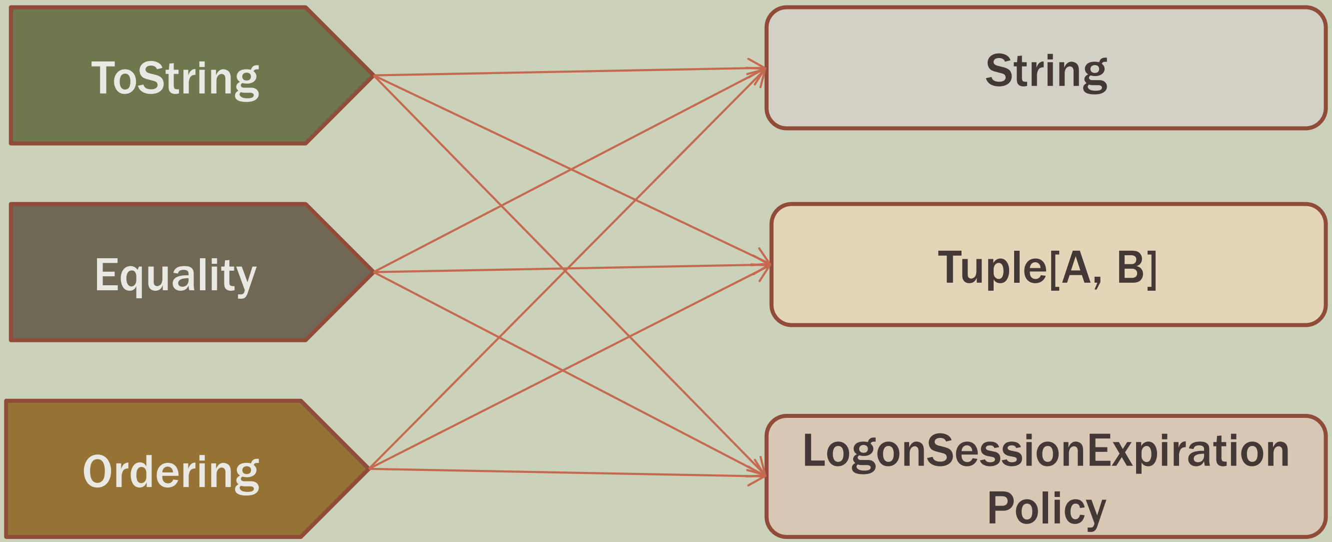
This talk was supported by my employer
careers.realestate.com.au

# HISTORY AND MOTIVATION OF TYPECLASSES

Part 1

# THE PROBLEM: *DATA ABSTRACTION*

- We want an operation have a different implementation when used in the context of different types of data
  - The operation has the same “meaning” in all contexts, but an implementation appropriate for each type of data

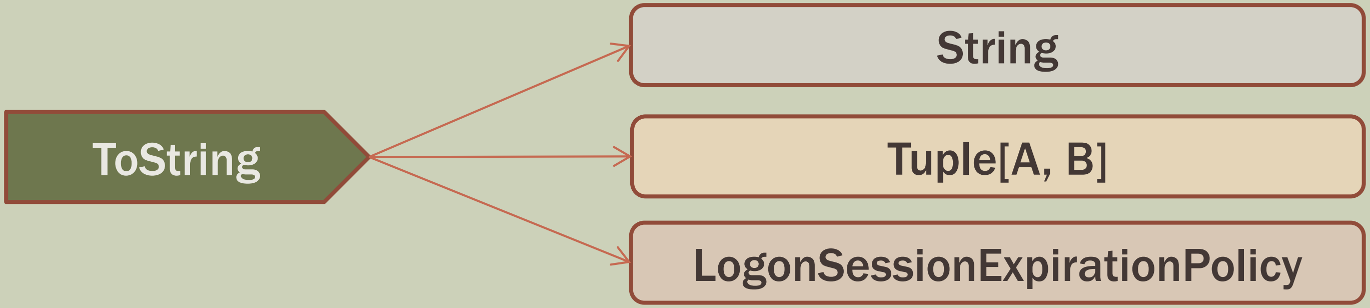


# AD HOC POLYMORPHISM

*“**Ad-hoc polymorphism** occurs when a function is defined over several different types, **acting in a different way** for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in 3*3) and multiplication of floating point values (as in 3.14*3.14).”*

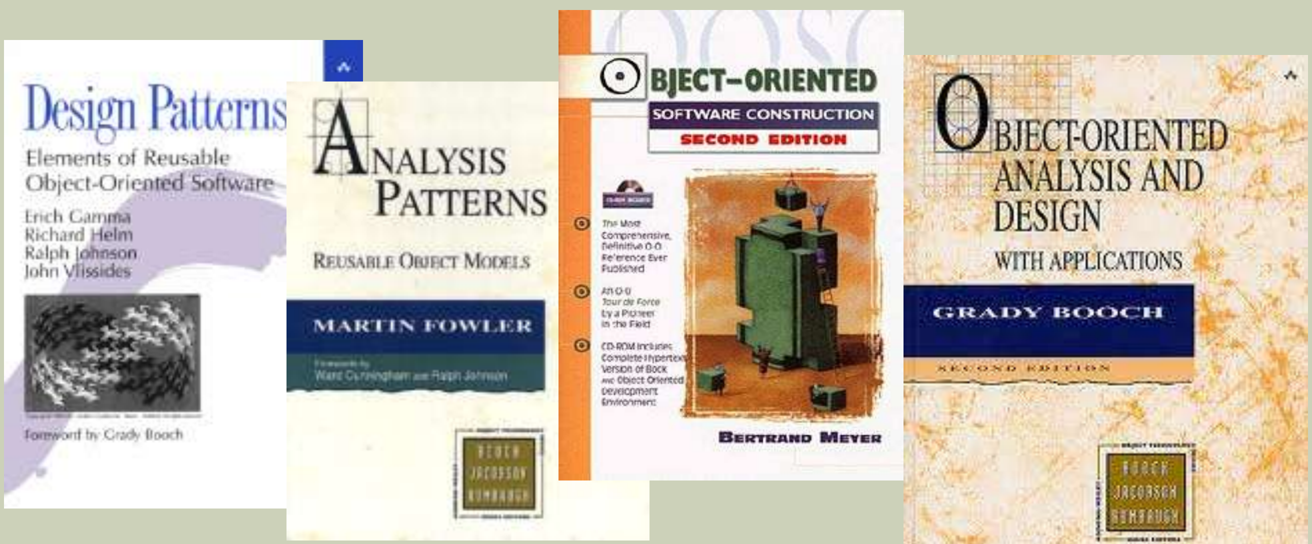
Philip Wadler and Stephen Blott, 1988

- Unfortunately named
  - “ad hoc” has negative connotation
- Sometimes known by other names:
  - *Overloading, Object-oriented polymorphism*



# AD HOC POLYMORPHISM

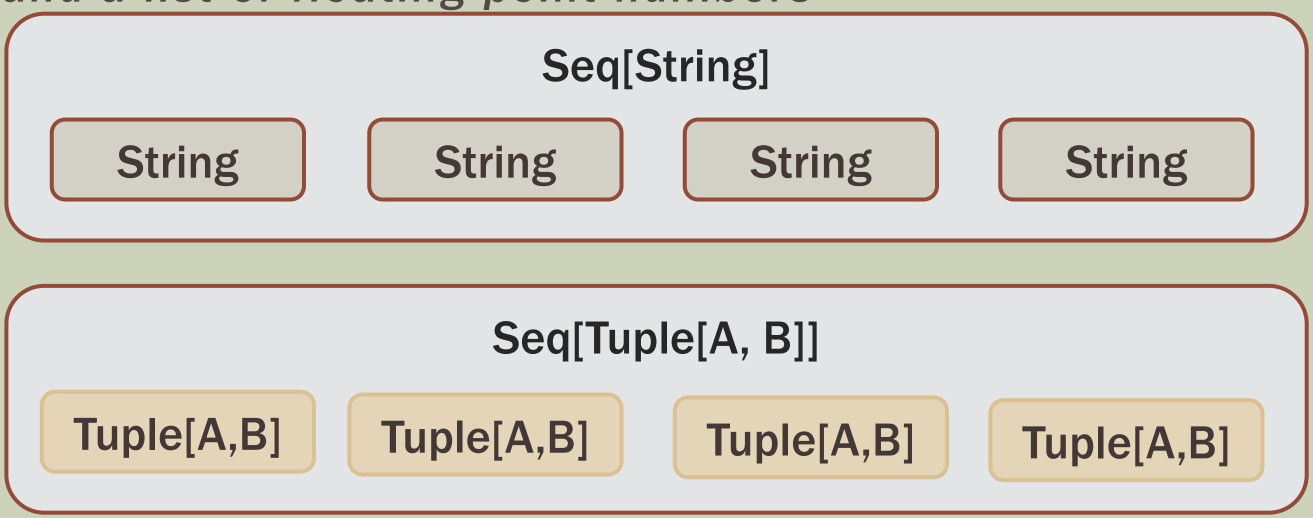
- Ad hoc polymorphism is a cornerstone of the object-oriented paradigm
- Much advocacy for object-oriented design is in fact advocacy for data abstraction generally, via ad hoc polymorphism
  - Object-orientation was the first type of ad hoc polymorphism to become popular, so the two became equated.



# PARAMETRIC POLYMORPHISM

- The other distinct kind of polymorphism is *Parametric Polymorphism* (“generics”):

*“**Parametric polymorphism** occurs when a function is defined over a range of types, **acting in the same way** for each type. A typical example is the length function, which acts in the same way on a list of integers and a list of floating point numbers”*



# DATA ABSTRACTION: A UNIVERSAL GOOD

- Data Abstraction is like *Peace* and *Love*
  - One of the few qualities in software engineering that almost everyone agrees is a *Good Thing*.
  - However, there's diversity in *how to achieve it*

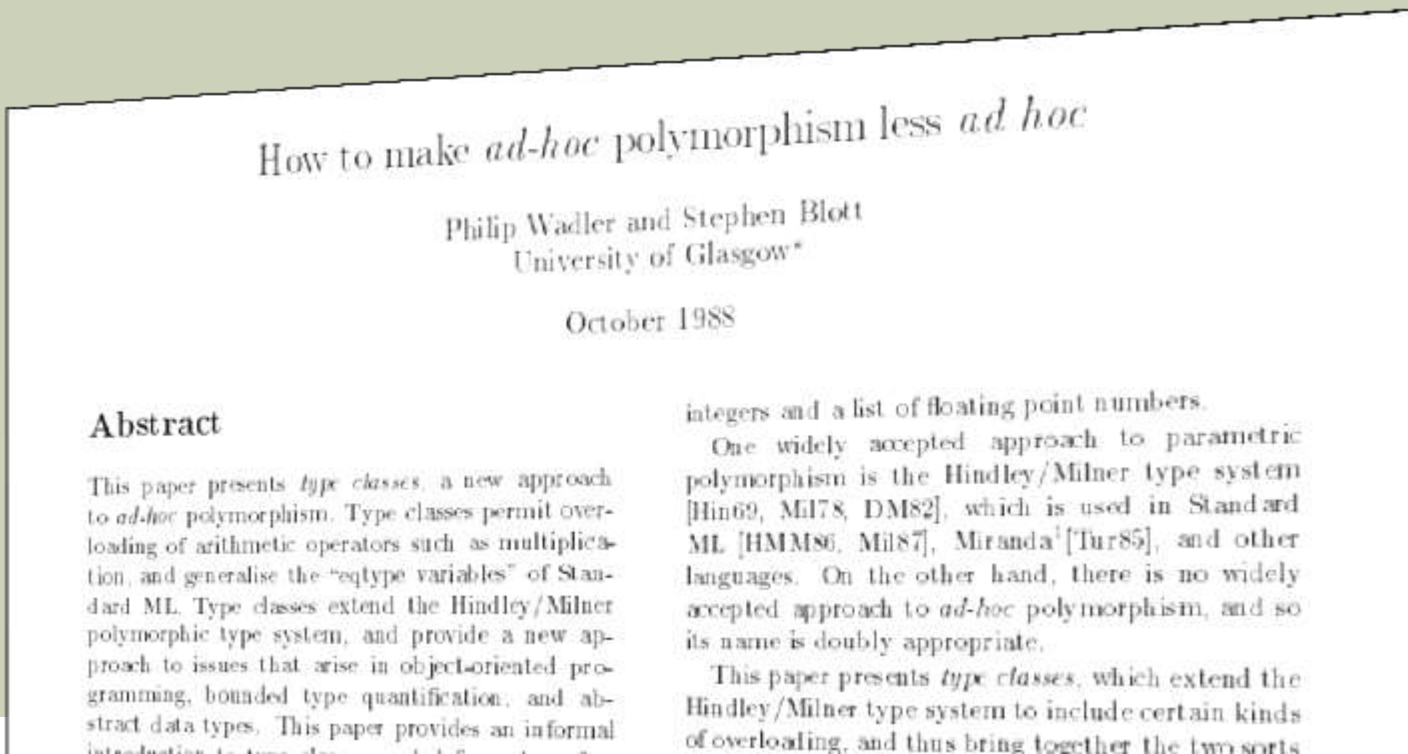
**Object Orientation**
Java
C++
Ruby
.C#
Python
Scala
PHP
Groovy

**Multi-methods and Protocols**
Clojure

**Typeclasses**
Haskell
Scala
Coq
Agda

# TROUBLE IN PARADISE

- The paper that proposed typeclasses begins by outlining the poor support for data abstraction in functional programming at that time
  - E.g. ML, Miranda
  - Contrasted with Object-orientation: C++, Smalltalk, Objective-C



How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

**Abstract**

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal

integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1] [Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

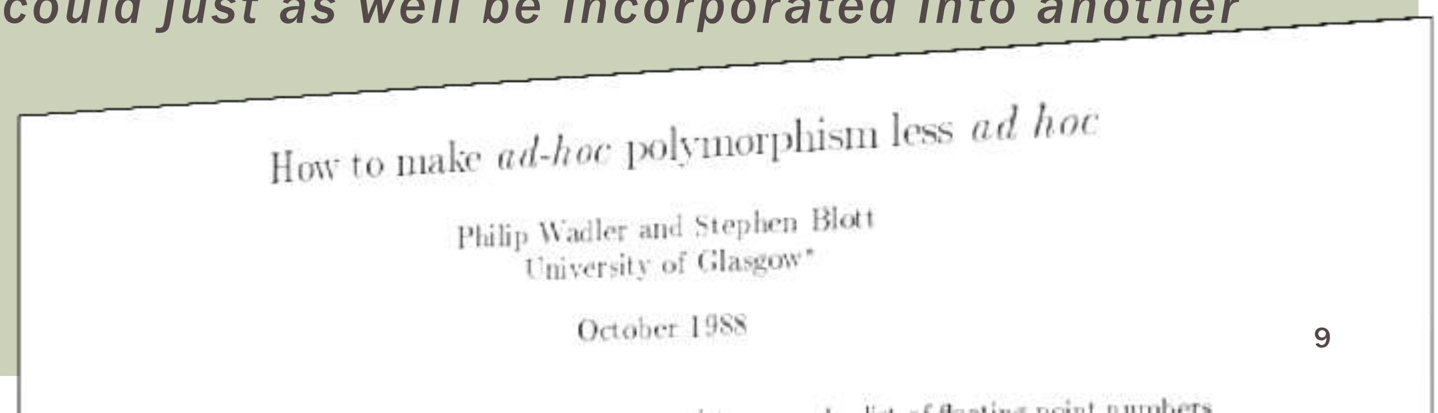
This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts

# TROUBLE IN PARADISE

*"One of the goals of the Haskell committee was to adopt 'off the shelf' solutions to problems wherever possible. We were a little surprised to realise that arithmetic and equality [i.e. data abstraction] were areas where no standard solution was available!*

*Type classes were developed as an attempt to find a better solution to these problems; the solution was judged successful enough to be included in the Haskell design.*

*However, type classes should be judged independently of Haskell; they could just as well be incorporated into another language"*

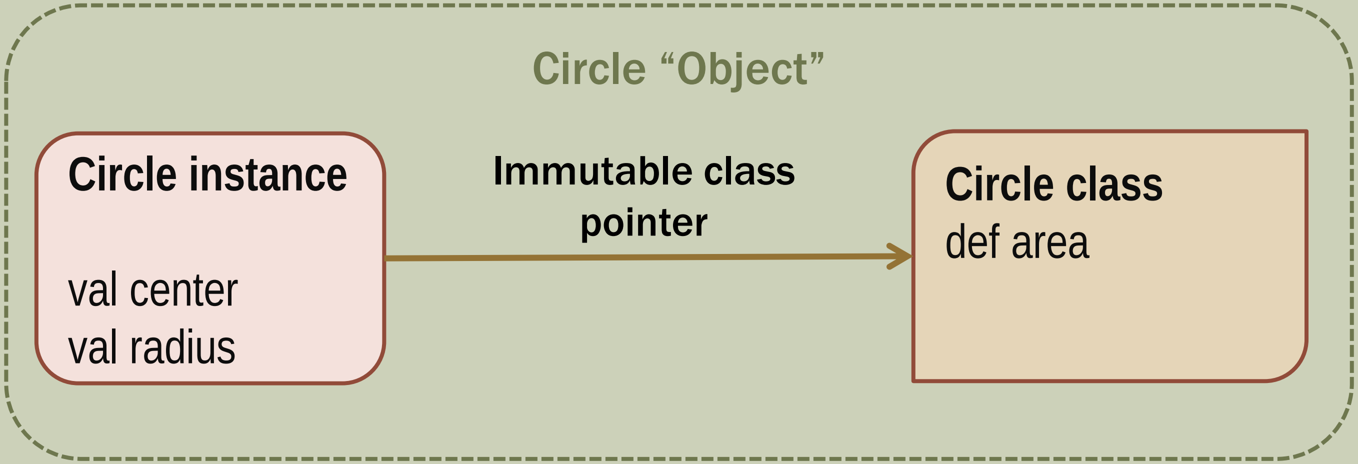

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

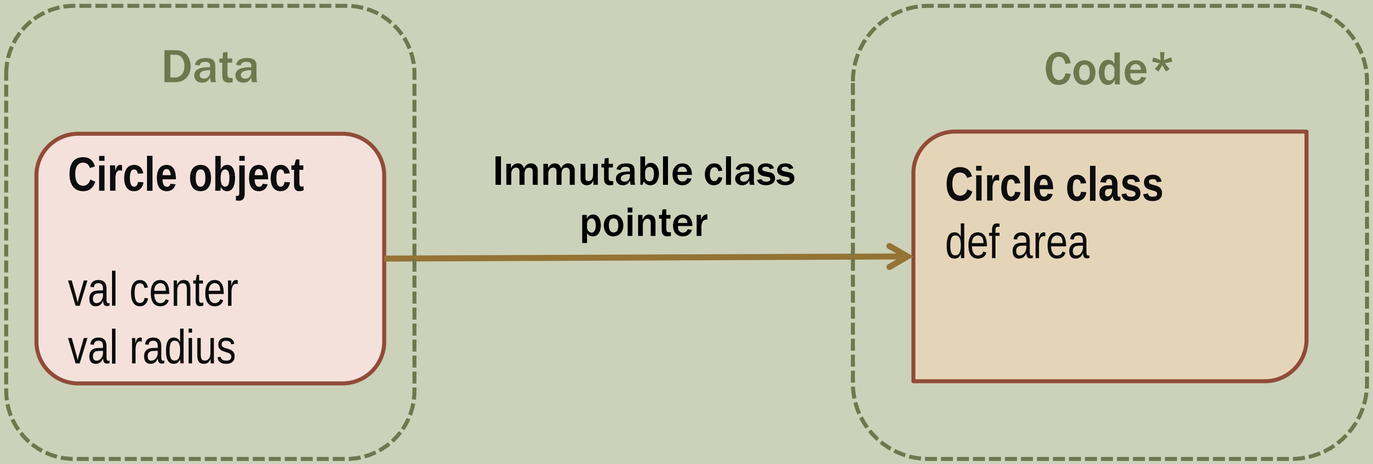
# OBJECT ORIENTATION

```
class Shape {def area: Double}
class Circle(val center: Point, value radius: Double) extends Shape {
        def area: Double = radius * radius * Pi
}
val circle: Shape = new Circle(Point(2.0, 0.0), 2.0)
println(circle.area)
```



# OBJECT ORIENTATION

```
class Shape {def area: Double}
class Circle(val center: Point, value radius: Double) extends Shape {
        def area: Double = radius * radius * Pi
}
val circle: Shape = new Circle(Point(2.0, 0.0), 2.0)
println(circle.area)
```

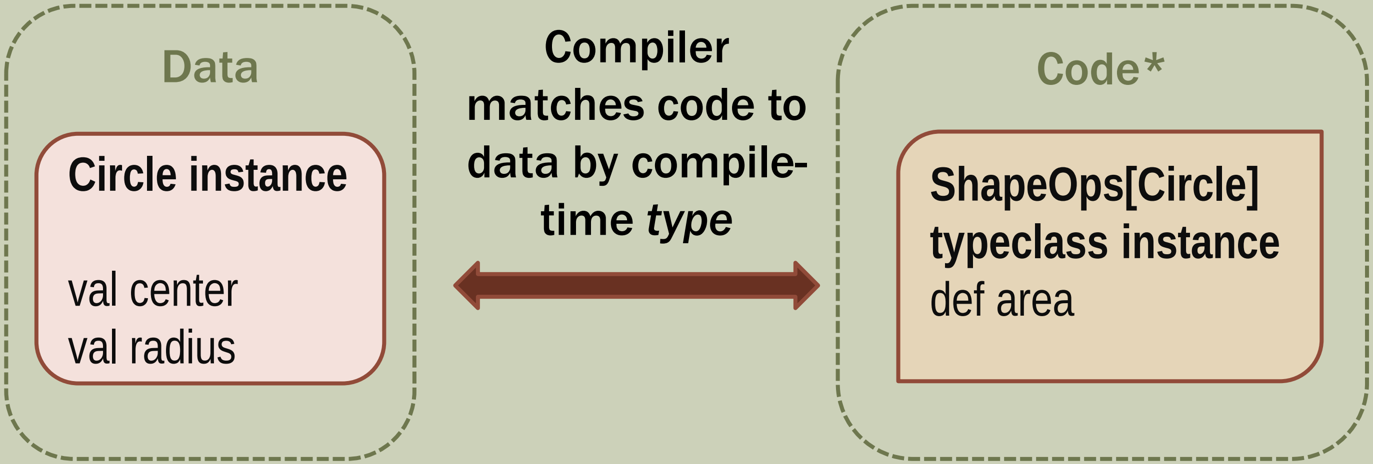





* Also called "Dictionary", or "V-table"

# TYPECLASSES

```
case class Circle(center: Point, radius: Double)
trait ShapeOps[A] {
        def area(a: A): Double
}
implicit val circleOps = new ShapeOps[Circle] {
        def area(c: Circle) = c.radius * c.radius * Pi
}
```

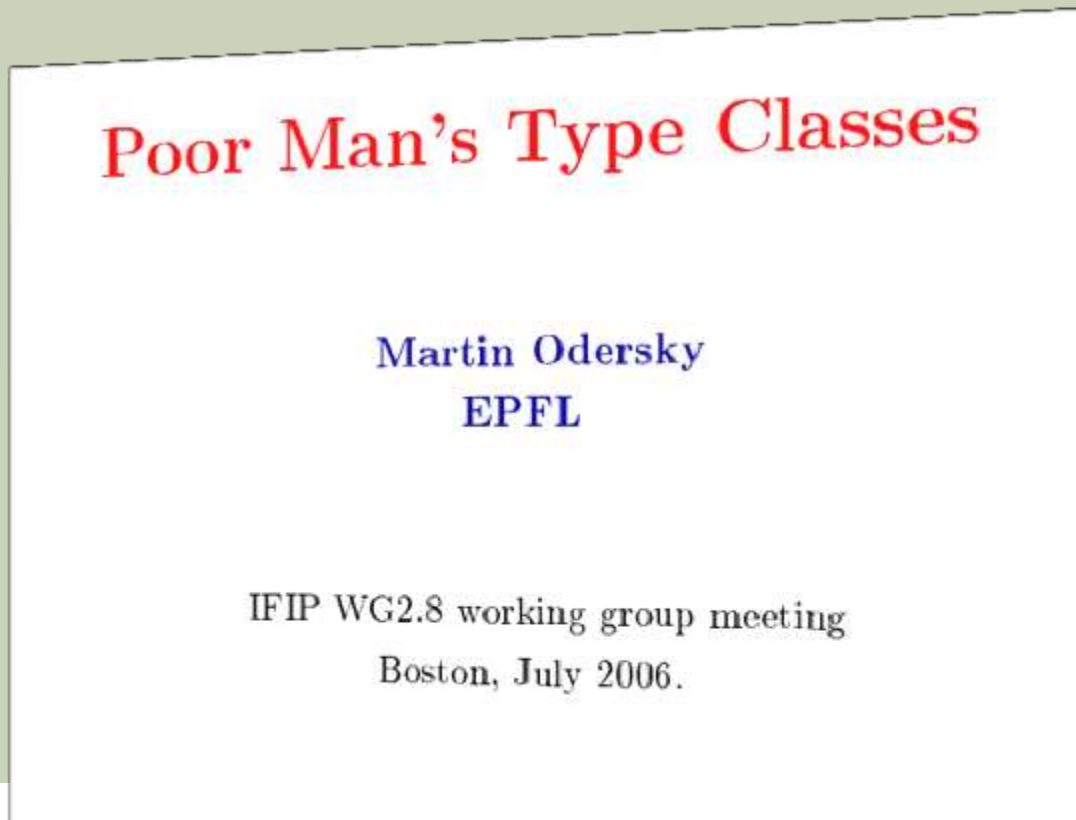





* Also called “Dictionary”, or “V-table”

# POOR MAN'S TYPE CLASSES

- Scala was not initially *designed* to support typeclasses
  - Intention was a purely object-oriented language with functional features
- Scala 1.0 in 2003
- First mention of typeclasses was 2006
- Re-purposed the power of *implicit parameters*



# INCREASING ADOPTION

- While OO remains Scala's "native mode", typeclasses have seen steady adoption in the community

# FORWARD TO 2013

- Scala 2.8 added Context Bounds, the first typeclass-focused language extension

```
def cubed[A: Numeric](a: A) = a * a * a
```

- Typeclasses seeping into mainstream Scala
  - Core Collections API
  - Play framework
  - Typeclass-based libraries: Scalaz, Shapeless, Spire
- Little coverage in books however
  - Still slightly un-orthodox

# CODE EXAMPLES:

# EQUIVALENCE
# ORDERING
# NUMBERS

Part 2

# IMPLICIT MECHANICS

Part 3

# IMPLICITS: PARAMETERS & CONVERSIONS

- The Scala compiler can do two things *implicitly* (automatically)
- *Pass implicit parameters*, matching by type
  - Typeclass support is built on this feature
- Convert one type into another by invoking an *implicit conversion*
  - Attempted in places where the code won't typecheck
  - Not especially relevant to typeclasses

# IMPLICIT SOURCES AND RECEIVERS

- Implicit keyword used in 2 situations
  - Define parameter lists that ***receive*** implicit values
  - Define sources that ***provide*** implicit values

- There are two ways to receive implicit values
  1. Define a implicit parameter list in a method signature

     ```
     def cubed[A](a: A)(implicit n: Numeric[A]) = a * a * a
     ```

  2. Add Context Bound(s) to a type parameter passed to a method or constructor. This is syntactic sugar for 1. above

     ```
     def cubed[A: Numeric](a: A) = a * a * a
     ```

# IMPLICIT SOURCES AND RECEIVERS

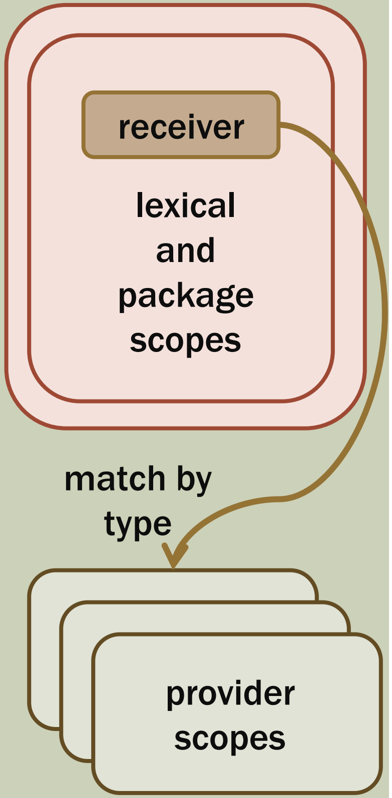
- Any and every implicit val or def (including implicit parameters) is a source of implicit values

- When the compiler finds an open receiver, it searches two implicit scope levels
  - First *enclosing*, then *provider* scope (see next slide)
  - Implicit values can always be passed explicitly (like a regular value) in which case no search is performed
  - Only unambiguous matches allowed in a scope level

- Common for implicit defs to themselves receive implicit parameters
  - Recursion during implicit search

# IMPLICIT SCOPES

- 1. Enclosing scope
  - Lexical (block) scopes
    - Matches sources *declared and imported* in enclosing scopes with equal priority
    - Includes all classes, methods or objects enclosing the receiver
  - Includes Package objects of the receivers package and all (chained) enclosing packages
    - Make implicit values or conversions available to all code in a package tree
- 2. Provider scope
  - Companion objects (if any) for all *parts* of the type are searched
  - Example: List[String] has parts (List, String)
  - Provide global default implicit values



# CODE EXAMPLES:

# IMPLICIT SCOPE SUBTYPING & VARIANCE

Part 4

# TYPECLASSES: PROS AND CONS

Part 5

# PRO: MULTIPLE IMPLEMENTATIONS

- It’s not uncommon to have multiple implementations of the same typeclass for the same data type, that all make sense
  - We saw this with both the Equivalence and Ordering examples
- Another one: valid instances of Semigroup[Int] include Addition, Multiplication, Max, Min, Average
  - Semigroups represent some ‘reduction’ over values
- Because implicits can be locally scoped, it’s possible to use different instances of a typeclass in different parts of a codebase

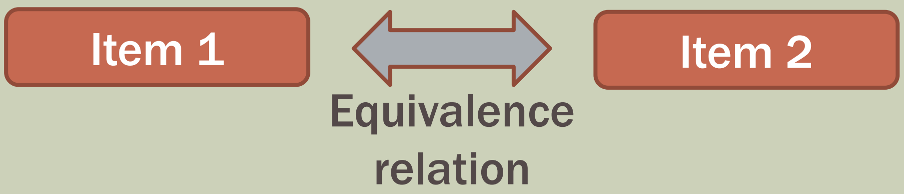
# PRO: MULTIPLE IMPLEMENTATIONS

- In contrast, the object-orientated does not support implementing the same interface more than once

- For Comparison, this limitation is acute
  - Object-oriented libraries define typeclass-like interface for Comparison
  - Java
    - java.util.Comparable (inherited ordering)
    - java.util.Comparator (external ordering)
  - C#
    - IComparable
    - Comparer<T>

# PRO: SYMMETRY

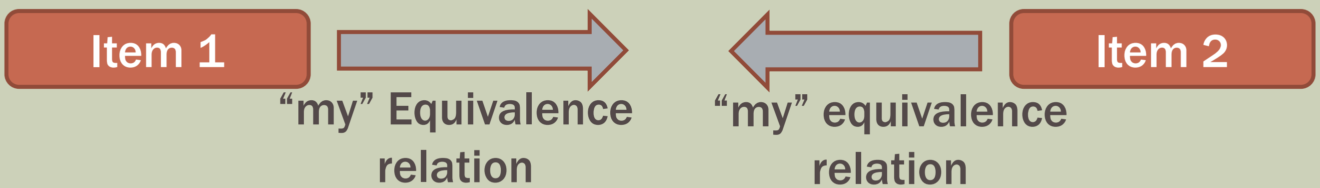
- If we say two data values are equivalent, or equal, the relation applies symmetrically to both
  - The equivalence is not “in the opinion” of one or other of the items




- Typeclasses model such relations naturally, type-safely and efficiently

# PRO: SYMMETRY

- The object-oriented alternative has an equivalence relation attached to *each* data item
  - At best, it's just inefficient to maintain two pointers to one thing
  - Worse, makes it very possible the equivalence relation is asymmetric





artima developer

Articles | News | Weblogs | Buzz | Books | Forums

Leading-Edge Java | Discuss | Print | Email

Sponsored Link • Flex 4 Fun - Get the PDF PrePrint eBook, only $23.00

How to Write an Equality Method in Java

by Martin Odersky, Lex Spoon, and Bill Venners

June 1, 2009

**Summary**

This article describes a technique for overriding the equals method that preserves the contract of equals even when subclassses of concrete classes add new fields.

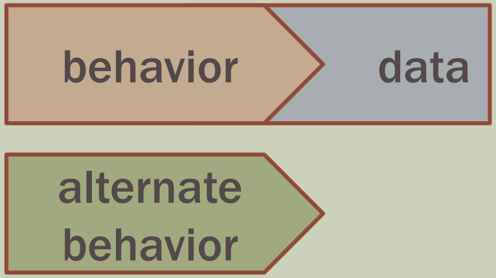
In Item 8 of *Effective Java*[1], Josh Bloch describes the difficulty of preserving the equals contract when subclassing as a "fundamental problem of equivalence relations in object-oriented languages." Bloch writes:

*There is no way to extend an instantiable class and add a value component while preserving the equals contract, unless you are willing to forgo the benefits of object-oriented abstraction.*

Chapter 28 of *Programming in Scala* shows an approach that allows subclasses to extend an instantiable class, add a value component, and nevertheless preserve the equals contract. Although in the technique is described in the book in the context of defining Scala classes, it is also applicable to classes defined in Java. In this article, we present the technique using text adapted from the relevant section of *Programming in Scala*, but with the code examples translated from Scala into Java.

# PRO: SEPERABLE BEHAVIOR AND DATA

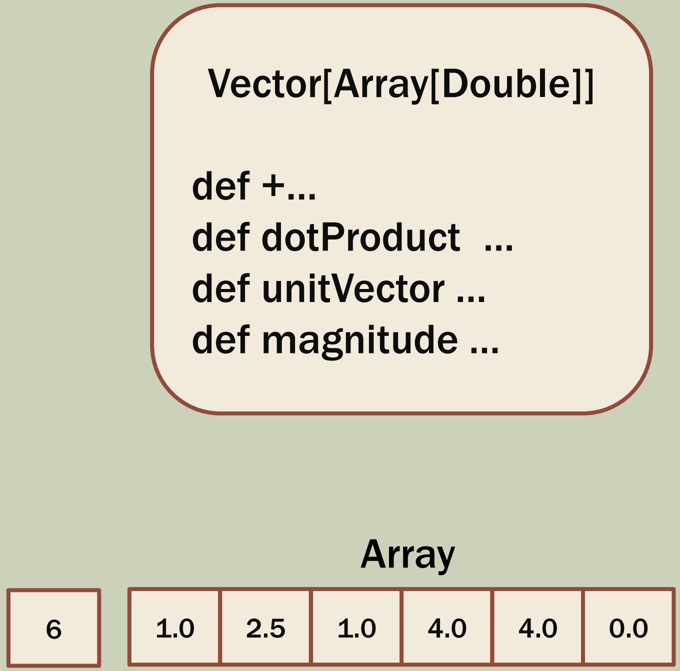
- Typeclass: data and behavior are coupled but *seperable*
  - Can substitute alternate behavior that works against that data
  - "Extend from outside"




- Object-oriented: data and behavior are coupled & *inseperable* from the outside
  - Only extend or modify behavior from the inside, by subclassing
  - "Do you control the code?"
  - Subclassing often limited in practice, eg arrays, primitives in Java

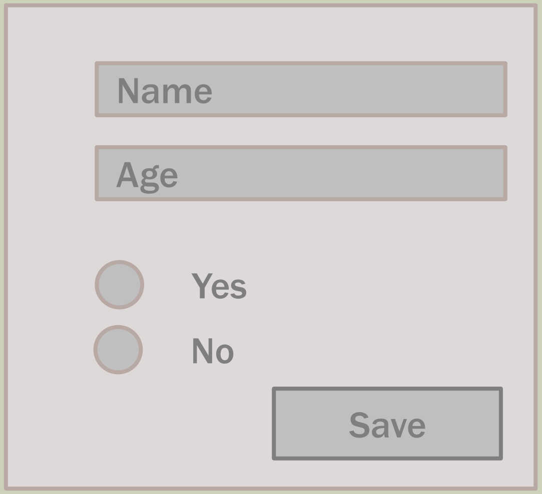
# PRO: SEPERABLE BEHAVIOR AND DATA

- On the JVM, separable code and data makes it easier to have
  - Good abstractions, and..
  - Efficiency
- Example: Efficient mathematical vectors
  - Data: Array[Double]
  - Behavior: Vectortypeclass
- Typeclasses require no change to, nor inheritance from, the data being operated over.
  - Do need way to ‘tag’ an Array[Double] as having Vector type
  - Array[Double] with Tag[Vector]



# CON: HETEROGENEOUS COLLECTIONS

- Typeclass dispatch based upon the static type of data
  - When we 'forget' the types of data, typeclasses can't be used
  - One common situation is where we want to store different types of objects together
- Example
  - Different types of UI widgets all placed within a container
  - All share some common interface
    - eg Swing Components



# CON: HETEROGENEOUS COLLECTIONS

- Q: How can typeclasses support heterogeneous collections?
- A: Collection of pairs, each
  - An item data of some type X that gets ‘forgotten’
  - A typeclass instance that operates on the type X
    - eg Draw[X]
  - Type X can be different for each items in the collection
  - But the collection can support draw operations through the Draw ‘interface’
- See code example

# CON: HETEROGENEOUS COLLECTIONS

- To make heterogeneous collections work, we
  - Fused Data and Behavior together
  - Hid the data representation type
  - Interacted through an interface
- We reinvented object-orientation!
  - “Pay as you go model”
- Hidden types are called *existential* types
  - Seemingly esoteric, and with a scary name, they are actually the basis of object-orientation

# CON: NOT SCALA'S 'NATIVE MODE'

- Typeclass support was not 'designed into' Scala
  - Something that turned out to be possible with implicit parameters
  - Odersky: 'Poor Man's Typeclasses'

- More syntactic overhead than OO
  - Not only
    - Typeclass
    - Instance definitions
  - ..but also
    - Enrichment implicit class to add typeclass methods
    - Extra Type annotations often needed

# CON: NOT SCALA'S 'NATIVE MODE'

- Typeclass methods can clash with inherited methods
  - Inherited methods always win
- Less tool support
  - Scaladoc
  - IDEs

# CODE EXAMPLES:

# JSON SERIALIZATION & DE-SERIALISATION

Part 6

# HIGHER-KINDED TYPES

# MULTI-PARAMETER TYPECLASSES

Part 7

# HIGHER-KINDED TYPES

- A higher-kinded type that has type-parameters yet to be specified
  - For example Seq, as distinct from eg Seq[Int] which is a regular type
  - Also known as a *Type Constructor,* in analogy to value constructors; it takes type parameters and yields a type

- Many useful typeclasses operate over higher-kinded types
  - Functor, Applicative, Monad

```
trait Functor[F[_]] {
  def fmap[A, B](a: F[A], f: A => B): F[B]
}
```

- Note how an fmap implementation "cannot care" about specific types A and B since it's parametric in them

# MULTI-PARAMETER TYPECLASSES

- It's typical for a typeclass to take a single type parameter. That type is what it provides ad hoc polymorphism over
  - E.g. typeclass Eq[A] provides polymorphism over different 'A's
- However, it's possible for type-classes to have 2 or 3 type parameters
  - These define a relation between types
- Two parameter typeclasses are rare in practice
  - Convert[A, B] – know's how to convert As  into Bs
  - ArrayStore[A, E] – Stores elements E into an array of type A
- Curiously, 3-parameter typeclasses are more common & useful
  - Used in Scala Collections API: CanBuildFrom[A, B, C]
  - Topic of a 30-min talk I'm giving tomorrow

# THANK YOU

Questions..