



Type Classes in Scala and Haskell

© 2018 Hermann Hueck



Table of Contents

- Scala type classes
- A type class and its instances
- Example: type class `Printable[A]`
- Better Design
- Where to store the instances?
- Benefit of type classes
- Type classes in Haskell

Example: List.sorted + List.sum

- `class List[+A] {`
 - `...`
 - `def sorted[B >: A](implicit ord: math.Ordering[B]): List[A]`
 - `def sum[B >: A](implicit num: Numeric[B]): B`
 - `...`
- `}`

Some Type Classes (Scala)

- `scala.math.Ordering[T]`
- `scala.math.Numeric[T]`
- JSON Serialization (in play-json etc.)
- `cats.{Show, Monoid, Functor, Monad ...}`
- etc.

How to use the Type Class Pattern

- Define a type class - a trait with at least one type parameter.

```
trait Printable[A] { ... }
```

- For each type to support the type class define a type class instance. Each instance replaces the type parameter *A* by a concrete type (*Int*, *Cat*, etc.).

```
implicit val intPrintable Printable[Int] = ...
```

```
implicit val catPrintable Printable[Cat] = ...
```

- Provide a generic user interface with an implicit type class parameter.

```
def myPrint[A](value: A)(implicit p: Printable[A]) = ...
```



Define a type class

```
trait Printable[A] {  
  def format(value: A): String  
}
```

Define type class instances (1)

```
implicit val intPrintable: Printable[Int] = new Printable[Int] {  
  override def format(value: Int): String =  
    "How many cats? " + value.toString  
}
```

```
implicit val datePrintable: Printable[Date] = new Printable[Date] {  
  override def format(value: Date): String =  
    "Date of meeting: " + value.toString  
}
```



Use the type class instance (1)

```
def myPrint[A](value: A)(implicit printable: Printable[A]): Unit =  
    println(printable.format(value))
```

```
myPrint(2)
```

```
myPrint(new Date)
```


Define type class instances (2)

```
final case class Cat(name: String, age: Int, color: String)

object Cat {
  implicit val catPrintable: Printable[Cat] = new Printable[Cat] {
    override def format(cat: Cat): String = {
      val name  = Printable.format(cat.name)
      val age   = Printable.format(cat.age)
      val color = Printable.format(cat.color)
      s"$name is a $age year-old $color cat."
    }
  }
}
```

Use the type class instance (2)

```
def myPrint[A](value: A)(implicit printable: Printable[A]): Unit =  
    println(printable.format(value))
```

```
val mizzi = Cat("Mizzi", 1, "black")
```

```
val garfield = Cat("Garfield", 38, "ginger and black")
```

```
myPrint(mizzi)
```

```
myPrint(garfield)
```



Better Design

- Move the print method into a singleton object (e.g. the companion object of the type class).
- Use extension methods (= type enrichment) by defining an implicit class. (The implicit class must be parameterized with the same type as the type class.)

Better Design (1)

- Move the print method into a singleton object (e.g. the companion object of the type class).

```
object Printable {  
  def format[A](value: A)(implicit printable: Printable[A]): String =  
    printable.format(value)  
  def print[A](value: A)(implicit printable: Printable[A]): Unit =  
    println(printable.format(value))  
}  
  
Printable.print(mizzi)
```

Better Design (2)

- Use extension methods (= type enrichment) by defining an implicit class. (The implicit class must be parameterized with the same type as the type class.)

```
implicit class PrintableOps[A](value: A) {  
    def format(implicit printable: Printable[A]): String =  
        printable.format(value)  
    def print(implicit printable: Printable[A]) = println(format)  
}  
mizzi.print
```



Where to keep the type class instances?

- Type class instances for standard types (`String`, `Int`, `Date` etc.) should be stored in the same package as the type class itself.
- Type class instances for your own types, i.e. domain classes (`Cat`, `Person`, `Customer`, `Order`, `Invoice` etc.) should be stored in the same package as the respective domain class.



Benefit of type classes

- The type class (`Printable`) and the domain class (`Cat`) are completely decoupled.
- You can extend and enrich not only your own types but also sealed types from libraries which you do not own.
- You do not need inheritance to extend existing library classes.



Type class `cats.Show`

- No need to implement the `Printable` type class
- `Cats` already has such a type class: `cats.Show`



Type classes in Cats

- Cats provides most of its core functionality as **type classes**: `cats.{Show, Eq, Ord, Num, Monoid, Functor, Monad, Applicative, Foldable}` and many more.
- See <https://typelevel.org/cats/typeclasses.html>

Type classes in Haskell

- Define a type class.

```
class Printable a where ...
```

- For each type that should support the type class.
(This enriches each type with the methods of the type class.)

```
instance Printable Int where ...  
instance Printable Cat where ...
```

- Use the type class methods for the types that have an instance. No extra user interface needs to be provided (like in Scala).

Define a type class

```
class Printable a where
```

```
  format :: a -> String
```

```
  pprntt :: a -> IO ()
```

```
  pprntt x = putStrLn $ format x
```

Define type class instances (1)

`instance` Printable Int where

`format = show`

`instance` Printable UTCTime where

`format time = "The exact date is: "`

`++ formatTime defaultTimeLocale "%F, %T (%Z)" time`

Define type class instances (2)

```
data Cat = Cat  
  { name :: String  
    , age  :: Int  
    , color :: String  
  }
```

instance Printable Cat where

```
format cat = "Cat {name=" ++ name cat  
            ++ ", age=" ++ show (age cat) ++ ", color=" ++ color cat ++ "}"
```



Use the type class methods with the instance types.

```
putStrLn $ format $ utcTime 2018 3 8 16 38 19
```

```
pprintt $ utcTime 2018 3 8 16 38 19
```

```
let mizzi = Cat "Mizzi" 1 "black"
```

```
putStrLn $ format mizzi
```

```
pprintt mizzi
```



Type class Show

- No need to implement the `Printable` type class
- Haskell already has a type class `Show` in the Prelude

Type classes in Haskell

- Many type classes are available in the Haskell Prelude
- Haskell provides its own kosmos of type classes in Base, most of them available in the Prelude:
`Show, Eq, Ord, Num, Integral, Fractional, Monoid, Functor, Applicative, Monad, Foldable` **etc.**



Comparison

- Haskell has its own type class syntax (key words **class** and **instance**).
- Scala uses implicits to provide type classes.
- In Scala (using `implicit val ...`) you need to create an object for each type class instance.
- No object creation in Haskell.
- No implicit hocus-pocus in Haskell.



Resources

- Source code and slides –
<https://github.com/hermannhueck/typeclasses>
- „Scala with Cats“ by Noel Welsh and Dave Gurnell
– <https://gumroad.com/discover?query=scala+cats>
- „Haskell Programming from first principles“ by
Christopher Allen and Julie Moronuki –
<https://gumroad.com/discover?query=allan+haskell>



Thank you!

Q & A