



Type Classes in Scala and Haskell

© 2018 Hermann Hueck



Table of Contents

- Scala extension methods
- 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



Scala Extension Methods

- Define an implicit class
- The class must have a single parameter of the type in question.
- Define extension methods inside the class.

Scala Extension methods

```
implicit class EnrichedInt(i: Int) {  
  def double: Int = 2 * i  
  def triple: Int = 3 * i  
  def square: Int = i * i  
  def cube: Int = i * i * i  
}
```

```
val double5: Int = 5.double // 10  
val triple5: Int = 5.triple // 15  
val squared5: Int = 5.square // 25  
val cubed5: Int = 5.cube // 125  
val doubledSquared5 = 5.double.square // 100
```

Scala Extension methods (2)

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

implicit class EnrichedCat(c: Cat) {
  def description: String =
    s"${c.name} is a ${c.age} year old ${c.color} colored cat."
  def describe(): Unit = println(c.description)
}

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

mizzi.describe()
```

Scala Extension methods (3)

```
implicit class EnrichedList[A](l1: List[A]) {  
    def zipWith[B, C](l2: List[B])(f: (A, B) => C): List[C] =  
        l1.zip(l2) map { case (x, y) => f(x, y) }  
}
```

```
val l1 = List(1, 2, 3)  
val l2 = List(10, 20, 30)
```

```
val result = l1.zipWith(l2)(_ + _)  
println(result) // --> List(11, 22, 33)
```



Extension Methods – How they work

- The compiler looks up a method for a class.
- If the class implements the method this one is used.
- If the class doesn't implement the method it looks for an implicit class that takes a parameter of the class in question.
- If the implicit class implements the method it takes that one.
- Otherwise the compiler is bailing out.

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  
  def map[B, That](f: (A) => B)  
    (implicit bf: CanBuildFrom[List[A], B, That]): That  
  // ...  
}
```


Some Type Classes (Scala)

- `scala.math.Ordering[T]`
- `scala.math.Numeric[T]`
- `scala.collection.generic.
CanBuildFrom[-From, -Elem, +To]`
- JSON Serialization (in play-json etc.)
- `cats.{Show, Monoid, Functor, Monad ...}`
- E.g. Akka and many other libraries
- 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

```
// the type class,  
// a trait with at least one type parameter  
//  
trait Printable[A] {  
  def stringify(value: A): String  
}
```

Define type class instances (1)

```
// type class instance for Int
//
implicit val intPrintable: Printable[Int] = new Printable[Int] {
  override def stringify(value: Int): String = value.toString
}

// type class instance for Date
//
implicit val datePrintable: Printable[Date] = new Printable[Date] {
  override def stringify(value: Date): String = value.toString
}
```

Use the type class instance (1)

```
// interface function for Printable
//
def myPrint[A](value: A)(implicit p: Printable[A]): Unit =
  println(p.stringify(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 stringify(cat: Cat): String = {
```

```
      val name  = Printable.stringify(cat.name)
```

```
      val age   = Printable.stringify(cat.age)
```

```
      val color = Printable.stringify(cat.color)
```

```
      s"$name is a $age year-old $color cat."
```

```
    }
```

```
  }
```

```
}
```

Generic type class instances

```
// a generic instance for Option[A] is a def with a type
// parameter A and an implicit Printable[A]. That means:
// if you can stringify an A, you also can stringify Option[A]
//
implicit def optionPrintable[A: Printable]
    :Printable[Option[A]] = (optA: Option[A]) =>
    optA
      .map(implicitly[Printable[A]].stringify)
      .map(s => s"Option($s)")
      .getOrElse("None")
```

Use the type class instance (2)

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

```
myPrint(mizzi)  
myPrint(garfield)  
myPrint(Option(garfield))
```




Better Design

- Move the print method into an object (e.g. the companion object of the type class).
- With extension methods type class methods can be used just like intrinsic methods of the respective type.
- Use extension methods (= type enrichment) by defining an implicit class. (The implicit class must have a parameter of the same type as the respective type class instance.)

Better Design (1)

Move the interface object methods (`stringify` and `print`) into a singleton object (e.g. the companion object of the type class). They will be found automatically without extra import statement in the user code.

```
// The type class companion object
//
object Printable {

  // interface object methods for the type class
  //
  def stringify[A](value: A)(implicit p: Printable[A]): String =
    p.stringify(value)
  def print[A](value: A)(implicit p: Printable[A]): Unit =
    println(stringify(value))
}

Printable.print(mizzi)
```

Better Design (2)

- Use extension methods (= type enrichment) by defining an implicit class. (The implicit class must have a parameter of the same type as the respective type class instance.)

```
// interface syntax methods as extension methods
//
implicit class PrintableOps[A](value: A) {
  def stringify(implicit p: Printable[A]): String = p.stringify(value)
  def print(implicit p: Printable[A]): Unit = println(stringify)
}
```

```
mizzi.print
```

Where to keep type class instances?

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

Implicits without import tax

- The library designer should provide the type class implementation in a way that user code needs only **one import statement**:
`import path.to.libPrintable._`
- Type class instances in the type class companion object (or in the type class package object) are found automatically without extra import.
- If the implicit class with the extension methods is located in the type class package object it is also found without extra import.

Implicits without import tax - code

```
package path.to.libPrintable
```

```
object Printable {
```

```
  def stringify[A](value: A)(implicit p: Printable[A]): String = p.stringify(value)
  def print(value: A)(implicit p: Printable[A]): Unit = println(stringify(value))
```

```
  implicit val intPrintable: Printable[Int] = ???
```

```
  implicit val datePrintable: Printable[Date] = ???
```

```
  implicit def optionPrintable[A: Printable]: Printable[Option[A]] = ???
```

```
}
```

```
- - - -
```

```
package path.to
```

```
package object libPrintable {
```

```
  implicit class PrintableOps[A](value: A) {
```

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

```
    def print(implicit p: Printable[A]): Unit = println(stringify)
```

```
  }
```

```
}
```

```
- - - -
```

```
package user.code
```

```
import path.to.libPrintable._
```

```
2.print
```

```
new Date().print
```

```
mizzi.print
```

Context Bound + implicitly (1)

```
// implicit class (without context bound)
//
implicit class PrintableOps[A](value: A) {
  def stringify(implicit p: Printable[A]): String = p.stringify(value)
  def print(implicit p: Printable[A]): Unit = println(stringify)
}
```

- - - -

```
// implicit class (with context bound and implicitly)
//
implicit class PrintableOps[A: Printable](value: A) {
  def stringify: String = implicitly[Printable[A]].stringify(value)
  def print(): Unit = println(stringify)
}
```

Context Bound + implicitly (2)

```
// interface object methods (without context bound)
//
def stringify[A](value: A)(implicit p: Printable[A]): String =
    p.stringify(value)
def print[A](value: A)(implicit p: Printable[A]): Unit =
    println(stringify(value))

- - - -

// interface object methods (with context bound and implicitly)
//
def stringify[A: Printable](value: A): String =
    implicitly[Printable[A]].stringify(value)
def print[A: Printable](value: A): Unit =
    println(stringify(value))
```


Method apply

```
object Printable {  
  
  def apply[A: Printable]: Printable[A] =  
    implicitly[Printable[A]]  
  
  def stringify[A: Printable](value: A): String =  
    Printable[A].stringify(value)  
  
  . . .  
}
```



Benefit of type classes

- The type class (`Printable`) and the type you create an instance for, e.g. a 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.



Downsides of type classes

- Type classes and implicits are hard to understand for the Scala newcomer (complicated rules of implicit resolution).
- Overuse of implicits often makes Scala code too difficult to read/understand.
- Implicits slow down the compiler.



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
```

```
    stringify :: a -> String
```

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

```
    pprintt x = putStrLn $ stringify x
```

Define type class instances (1)

`instance` Printable Int where

`stringify = show`

`instance` Printable UTCTime where

`stringify 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

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



Use the type class methods with the instance types.

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

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

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

```
putStrLn $ stringify 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 (1)

- Source code and slides – <https://github.com/hermannhueck/typeclasses>
- Book: „Scala with Cats“ by Noel Welsh and Dave Gurnell – <https://underscore.io/books/scala-with-cats>
- Book: „Haskell Programming from first principles“ by Christopher Allen and Julie Moronuki – <http://haskellbook.com>

Resources (2)

- Talk: Tim Soethout on Implicits at ScalaDays 2016 – <https://www.youtube.com/watch?v=UHQbj-9r8A>
- Talk: Daniel Westheide on Implicits and Type Classes at ScalaDays 2016 – <https://www.youtube.com/watch?v=1e9tcymPl7w>
- Talk: Josh Suereth at North East Scala Symposium 2011 – <https://vimeo.com/20308847>
- Blog post on the details of implicit parameter resolution – <http://ee3si9n.com/revisiting-implicits-without-import-tax>



Thank you!

Q & A