



Type Classes in Scala and Haskell

© 2018 Hermann Hueck



Table of Contents

- Recap: Scala Implicits
- Recap: Scala implicit views
- Scala type classes
- A type class and its instances
- Example: type class `Printable[A]`
- Better Design
- Where to store the instances?
- Type classes without import tax
- Benefit of type classes
- Type classes in Haskell

Recap: Implicit

- Implicit declarations

```
implicit val x: X = ...  
implicit def func: X = ...  
implicit object X extends MyTrait { ... }
```

- Implicit parameters

```
def method(implicit x: X): R = ...
```

- Implicit classes

```
implicit class Y(x: X) { ... }
```

- Implicit conversions (a.k.a. implicit views)

```
implicit def aToB(a: A): B = ...
```



Recap: Implicit parameter resolution

- If you do not give an argument to an implicit parameter, the compiler tries to provide one for you.
- Eligible are all implicit values that are visible at the point of call.
- If there is more than one eligible candidate, the most specific one is chosen.
- If there is no unique most specific candidate, an ambiguity error is reported.

Recap: Implicit parameter resolution

- The details of implicit parameter resolution are very complex and not covered here. There are 2 main resolution steps.
- Step 1: Implicit parameters are first looked up in “**local scope**”: local declarations, imports (explicit or wildcard), inheritance (base classes or traits), local package object.
- Step 2: If implicit parameters cannot be resolved from the local scope the compiler searches the “**implicit scope**”: package object of the implicit parameter type and the package object of their type parameters (if any).
- Local scope always takes precedence over implicit scope.
- There are other precedence rules within implicit scope (specificity, inheritance) which are not covered here.

Recap: Implicit views

- Define an implicit class
- The class must have a single parameter of the type in question.
- Define extension methods inside the class.
- This technique is also called the “*Pimp up my library*” pattern, some times also known as “*static monkey patching*”.

Pimpin' - Implicit views (1)

```
// Implicit view/conversion: Int => EnrichedInt  
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
```

Pimpin' - Implicit views (2)

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

// Implicit view/conversion: Cat => PimpedInt
implicit class PimpedCat(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")

val desc = mizzi.description
println(desc)

mizzi.describe()
```


Pimpin' - Implicit views (3)

```
// Implicit view/conversion: List => PimpedList  
implicit class PimpedList[A](xs: List[A]) {  
  def zipWith[B, C](ys: List[B])(f: (A, B) => C): List[C] =  
    xs zip ys 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)
```

Pimpin' - Implicit views (4)

```
class PimpedList[A](xs: List[A]) {  
  def zipWith[B, C](ys: List[B])(f: (A, B) => C): List[C] =  
    xs zip ys map { case (x, y) => f(x, y) }  
}
```

```
// Implicit view/conversion: List => PimpedList  
implicit def pimpList[A](xs: List[A]): PimpedList[A] =  
  new PimpedList[A](xs)
```

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

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



Implicit views – 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 conversion method or an implicit class that takes a parameter of the type in question.
- If the implicit class implements the method in question, it creates an instance, passes the parameter and invokes the method.
- Otherwise the compiler will bail out.

Type Classes

- Type classes are a fundamental concept in Scala and Haskell.
- Haskell provides specific keywords for type classes.
- Scala implements type classes based on implicits.
- A type class classifies a set of types by their common properties.
- E.g. the Scala type class `Numeric` (Haskell: `Num`) defines the arithmetic operations (as methods) which are common to all numeric types such as `Int`, `Long`, `Float`, `Double`, `BigInteger`, `BigDecimal` etc.

Examples: List methods

```
class List[+A] {  
  // ...  
  def sum[B >: A](implicit num: Numeric[B]): B  
  def sorted[B >: A](implicit ord: math.Ordering[B]): List[A]  
  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 ...}`
- 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 define a type class instance. Each instance replaces the type parameter `A` by a concrete type (`Int`, `Date`, `Cat`, `Option[A]`, 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
}

// generic type class instance for Option[A] (must be a def)
// requires an implicit Printable[A]
//
implicit def optionPrintable[A]
  (implicit pA: Printable[A]): Printable[Option[A]] = ???
```

Use the type class instance (3)

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

myPrint(2)
myPrint(new Date)

myPrint(Option(2))
myPrint(Option(mizzi))
```

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]
    (implicit pA: Printable[A]): Printable[Option[A]] =
    new Printable[Option[A]] {

        override def stringify(optA: Option[A]): String =
            optA.map(pA.stringify)
                .map(s => s"Option($s)")
                .getOrElse("None")
    }
```

Use the type class instance (3)

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

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

```
myPrint(Option(garfield))  
myPrint(Option.empty[Cat])
```

```
myPrint(List(mizzi, garfield))  
myPrint(List.empty[Cat])
```



Better Design

- Move `stringify` and `pprint` methods into an object (e.g. the companion object or package object of the type class).
- With a pimp (implicit class) type class methods can be used just like intrinsic methods of the respective type.
- The implicit class constructor must have a (typically generic) parameter.
- The implicit class methods take an implicit type class parameter.

Better Design (1)

- Move the interface object methods (`stringify` and `pprint`) into a singleton object (e.g. the companion object or package object of the type class).

```
// 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 pprint[A](value: A)(implicit p: Printable[A]): Unit =
    println(stringify(value))
}
```

```
- - - -
```

```
package user.code
```

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

```
Printable.pprint(mizzi)
```

Better Design (2)

- Use a pimp by defining a generic implicit class. (The constructor has one parameter of the generic type. Methods take a type class instance as implicit parameter.)

```
// interface syntax methods defined by a pimp (= implicit view)
//
implicit class PrintableOps[A](value: A) {
  def stringify(implicit p: Printable[A]): String = p.stringify(value)
  def pprint(implicit p: Printable[A]): Unit = println(stringify)
}

val stringMizzi = mizzi.stringify
mizzi.pprint
```


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 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 of the domain class).

Only one import

- The library should provide the type class instances in a **non-intrusive** way so that user code easily can override them.
- 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. They are visible in the **implicit scope**.
- Interface object methods and implicit class can be moved to the library's package object. By the above import they become visible in **local scope**. (Can be improved, see below.)
- If needed the user can provide his own instances in local scope: local declaration, import (explicit or wildcard), inheritance (base class or trait), package object.
- Local scope precedes implicit scope.

Only one import - code

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

```
object Printable {  
  implicit val intPrintable: Printable[Int] = ???  
  implicit val datePrintable: Printable[Date] = ???  
  implicit def optionPrintable[A: Printable]: Printable[Option[A]] = ???  
}
```

```
package path.to
```

```
package object libPrintable {
```

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

```
  implicit class PrintableOps[A](value: A) {  
    def stringify(implicit p: Printable[A]): String = p.stringify(value)  
    def pprint(implicit p: Printable[A]): Unit = println(stringify)  
  }  
}
```

```
package userpkg
```

```
import path.to.libPrintable._ // only one import needed
```

```
2.pprint  
new Date().pprint  
mizzi.pprint
```

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 pprint(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 pprint(): 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 pprint[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 pprint[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)  
  
  . . .  
}
```

No more imports

- Move interface object methods and the pimp into a trait (e.g. `PrintableUtils`).
- The local package object (user code) extends that trait and brings them into local scope.
`package object userpkg extends PrintableUtils`
- Interface object methods and pimp can easily be overridden in the local package object.
- This architecture is least intrusive and gives good flexibility to the library user.
- **No library imports** needed

No more imports - code

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

```
object Printable {  
  implicit val intPrintable: Printable[Int] = ???  
  implicit val datePrintable: Printable[Date] = ???  
  implicit def optionPrintable[A: Printable]: Printable[Option[A]] = ???  
}
```

```
trait PrintableUtils {  
  def stringify[A](value: A)(implicit p: Printable[A]): String = p.stringify(value)  
  def pprint(value: A)(implicit p: Printable[A]): Unit = println(stringify(value))  
  
  implicit class PrintableOps[A](value: A) {  
    def stringify(implicit p: Printable[A]): String = p.stringify(value)  
    def pprint(implicit p: Printable[A]): Unit = println(stringify)  
  }  
}
```

```
-- --  
package userpkg
```

```
import path.to.libPrintable.{Printable, PrintableUtils}
```

```
package object userpkg extends PrintableUtils {  
  // In the users package object base trait utilities and implicits can be overridden.  
  
  override def pprint[A: Printable](value: A): Unit = ???  
  
  implicit class MyPrintableOps[A: Printable](value: A) extends PrintableOps(value) {  
    override def pprint(implicit p: Printable[A]): Unit = ???  
  }  
}
```




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 classes or library classes (which is not possible if they are sealed).
- Less repetition and boilerplate, e.g. avoids repetitive passing of parameters and overloads.



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.
- (Not only) Implicits give Scala the reputation to be an arcane language.
- Resolution of implicits may slow down the compiler.



Type class `cats.Show`

- If you are using cats ...
- No need to implement the Printable type class
- Cats already provides such a type class:
`cats.Show`



Type classes in Cats

- Cats provides most of its 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 ...
```

- Create an instance 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 ...
```

- That's it. Just 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: signature
```

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

```
  -- pprint: signature + impl
```

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

```
  pprint 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 4 9 19 15 00
```

```
pprint $ utcTime 2018 4 9 19 15 01
```

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

```
putStrLn $ stringify mizzi
```

```
pprint 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, i.e. without extra import.
- Haskell provides its own kosmos of type classes in Base (the standard library), most of them available in the Prelude (like `scala.Predef`): `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.
- No objects, no inheritance in Haskell
- Haskell type classes are coherent. Haskell allows globally only one type class instance per type. => No ambiguity errors! No precedence rules necessary!

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: *"Implicits inspected and explained"* by Tim Soethout on Implicits at ScalaDays 2016, Berlin – <https://www.youtube.com/watch?v=UHQbj-9r8A>
- Talk: *"Don't fear the implicits"* by Daniel Westheide on Implicits and Type Classes at ScalaDays 2016, Berlin – <https://www.youtube.com/watch?v=1e9tcymPl7w>
- Talk: "Implicits without import tax" by Josh Suereth at North East Scala Symposium 2011 – <https://vimeo.com/20308847>

Resources (3)

- Blog post on the details of implicit parameter resolution – <http://ee3si9n.com/revisiting-implicits-without-import-tax>
- Implicits in the Scala 2.12 language specification – <https://scala-lang.org/files/archive/spec/2.12/07-implicits.html>
- Keynote: “*What to Leave Implicit*” by Martin Odersky at ScalaDays 2017, Chicago – <https://www.youtube.com/watch?v=Oij5V7LQJsA>



Thank you!

Q & A

<https://github.com/hermannhueck/typeclasses>