



# **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 conversions (a.k.a. implicit views)

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

- Implicit classes

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



# 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**”: companion 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.

1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 2680, 26

- An implicit conversion is an implicit function which converts a value of type A to a value of type B.
- Use with caution: Can easily undermine type safety!!!

```
import scala.language.implicitConversions // suppress warnings
```

```
implicit def string2int(s: String): Int = Integer.parseInt(s)
```

```
val x: Int = "5"           // !! Assign a string to an int val
```

```
def useInt(x: Int): Unit = println("Int value: " + x)
useInt("42") // !! Pass a string to an int param
```

# 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 => PimpedCat
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)
```



# Mechanics of implicit views

- 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 ...}`
- There is hardly a Scala library not using type classes.

# 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 implement 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(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]
    (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)

```
val mizzi = Cat("Mizzi", 1, "black")
val garfield = Cat("Garfield", 38, "ginger and black")

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 userpkg

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
println(stringMizzi)

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 are found automatically without extra import. They are visible in the **implicit scope** (which can be overridden by local 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 = ???           // overrides default impl  
  
  implicit class MyPrintableOps[A: Printable](value: A) extends PrintableOps(value) {  
    override def pprint(implicit p: Printable[A]): Unit = ???       // overrides default impl  
  }  
}
```



# 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>

# Benefit of type classes

- Separation of abstractions: The type class (`Printable`) and the type you create an instance for, e.g. a domain class (`Cat`) are completely decoupled.
- Extensibility: You can extend and enrich not only your own types but also sealed types from libraries which you do not control.
- Composability: You do not need inheritance to extend existing classes or library classes (which is not possible if they are sealed).
- Overridability: You can override default instances (in companion object) with your own instances.
- Less repetition and boilerplate, e.g. avoids repetitive passing of parameters and overloads.
- Type safety is maintained.



# 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 (and some other features) give Scala the reputation to be an arcane language.
- Resolution of implicit parameters and conversions may slow down the compiler.

# 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: `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 needed!

# Resources (1)

- Source code and slides for this talk – <https://github.com/hermannhueck/typeclasses>
- Book: „*Scala with Cats*“ by Noel Welsh and Dave Gurnell – <https://underscore.io/books/scala-with-cats>
- Book: “*Scala in Depth*“ by Joshua D. Suereth – <https://www.manning.com/books/scala-in-depth>
- 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>
- Blog post: on type classes by Daniel Westheide – <http://danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html>
- Blog post: *"Pimp my Library"* by Martin Odersky, 2006 - <https://www.artima.com/weblogs/viewpost.jsp?thread=179766>

# Resources (3)

- Talk: “Implicits without import tax” by 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>
- 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>