Implicits

Part 1

Implicit Parameters, Type-classes, Context Bounds, implicitly, Composability, Rules

Agenda

1. Simple implicit parameters
2. Explicit overrides
3. Type classes
4. Type classes with implicits
5. Context Bounds
6. The `implicitly` function
7. Composition (Just-In-Time implicit methods)
8. Implicit Rules and Restrictions
9. ClassTags and TypeTags

Simple Implicit Parameters

A very simple retrying logic example:

```
case class RetryParams(times: Int)

import scala.util.control.NonFatal

def retryCall[A](fn: => A, currentTry: Int = 0)(retryParams: RetryParams): A = {
    try fn
    catch {
        case NonFatal(_) if currentTry < retryParams.times =>
            retryCall(fn, currentTry + 1)(retryParams)
    }
}

def retry[A](fn: => A)(retryParams: RetryParams): A =
    retryCall(fn, 0)(retryParams)
```

- `fn` is a "by-name" function that is evaluated each time the name is referenced
- Any exception causes a re-try up to the limit in the `RetryParams`

Retrying In Action

```
var x = 0

def checkIt(): Int = {
    x = x + 1
    require (x > 4, "x not big enough")
    x
}

retry(checkIt())(RetryParams(3))
> java.lang.IllegalArgumentException: requirement failed:
>   x not big enough (requirement failed: x not big enough)

retry(checkIt())(RetryParams(5))
> res0: Int = 5
```

- It works, but we need to supply `RetryParams` on every call
- And likely we will want to use the same params on many calls

Implicit Parameters

- The last parameter list in a method can be marked as implicit

```
def retry[A](fn: => A)(implicit retryParams: RetryParams): A =
    retryCall(fn, 0)(retryParams)
```

- Now if we make an implicit RetryParams available:

```
implicit val defaultRetry = RetryParams(5)
```

- And Scala will use that one automatically

```
retry {
  checkIt()
}
```

- Note: here we also use curly braces (valid for single parameter lists) to make things look nice

Explicit Overrides

- Implicits are looked up by the Scala compiler when required
- If code will compile without an implicit, Scala will not look one up or apply it
- Therefore you can always override implicits with an explicit parameter

```
retry(checkIt())(RetryParams(2))
```

- Implicit parameter lists can have more than one parameter
- Only the final parameter list can be implicit
- If any implicits are explicitly provided, all of them must be provided
- `implicit` is only placed at the head of the parameter list, not on each parameter:

```
def method(item1: String, item2: Int)(implicit p1: Param1, p2: Param2, p3: Param3)
```

Other Notes on Implicit Parameters

- The type is what matters to the Scala compiler, not the name
- Although the name may be used to override or hide definitions
- There can be no ambiguity, it will not pick from two implicits with the same type

Type Classes

- A type-class is another name for a trait or abstract class with a single generic type parameter and abstract method(s) defined for that type, e.g. like our `CompareT` from earlier:

```
abstract class CompareT[T] {
  def isSmaller(i1: T, i2: T): Boolean
  def isLarger(i1: T, i2: T): Boolean
}
```

- The methods are different, since we will use it differently this time (they take two params now)
- We can then ask for it in a method that needs it using an implicit parameter (remember the generic type becomes part of the overall type in Scala)

```
def insertSort[T](xs: List[T])(implicit compare: CompareT[T]) = {
  // to use it
  compare.isSmaller(xs.head, xs.tail.head) // for example
}
```

Insertion Sort with Type Class

```
def genInsert[T](item: T, rest: List[T])(implicit cmp: CompareT[T]): List[T] =
  rest match {
    case Nil                                    => List(item)
    case head :: _ if cmp.isSmaller(item, head) => item :: rest
    case head :: tail                           => head :: genInsert(item, tail)
  }

def genSort[T](xs: List[T])(implicit cmp: CompareT[T]): List[T] = xs match {
  case Nil => Nil
  case head :: tail => genInsert(head, genSort(tail))
}
```

- Unlike bounds, type `T` can be any type regardless of inheritance
- But calls to the method will not compile unless the `cmp` parameter is provided, explicitly or implicitly
- Because we know nothing about `T`, we can't call methods on it, but we can call methods on the implicit parameter provided (and we know the type of that)
- Implicits really provide a secondary type system in Scala, orthogonal to the primary one

Providing Implicit Type Classes

```
case class Distance(meters: Int)  // note - no inheritance!
val dists = List(Distance(10), Distance(4), Distance(12))
genSort(dists)
> Main.scala:27: could not find implicit value
>   for parameter cmp: CompareT[Distance]
> genSort(dists)
>         ^
```

- We need that type-class

```
implicit val distCompare = new CompareT[Distance] {
    def isSmaller(i1: Distance, i2: Distance) = i1.meters < i2.meters
    def isLarger(i1: Distance, i2: Distance) = i1.meters > i2.meters
}

genSort(dists)
> res6: List[Distance] = List(Distance(4), Distance(10), Distance(12))
```

- we could also call as `genSort(dists)(distCompare)` but why would you?

Ad-hoc Polymorphism

- We can now call `genSort` for any type as long as we define `CompareT` regardless of inheritance
- Even for types we don't own (and couldn't add `CompareT` to the supertypes)
- So we can effectively add behavior to types *without* needing to extend other types
- This capability is known as ad-hoc polymorphism
- Scala is even more powerful in that you can choose which implicits will be imported/used

Implicit objects and methods

- It's not only `vals` that can be marked implicit, `object`, `class` and `def` can be implicit too
- Implicit classes will be covered later, but let's look at `implicit object` and `implicit def`
- The `CompareT` val definition could also be a `def`:

```
implicit def distCompare = new CompareT[Distance] {
    def isSmaller(i1: Distance, i2: Distance) = i1.meters < i2.meters
    def isLarger(i1: Distance, i2: Distance) = i1.meters > i2.meters
}
```

- Unlike vals, defs can take parameters themselves, this comes in useful later!
- Or we can use an `implicit object` (preferred in this case):

```
implicit object DistCompare extends CompareT[Distance] {
    def isSmaller(i1: Distance, i2: Distance) = i1.meters < i2.meters
    def isLarger(i1: Distance, i2: Distance) = i1.meters > i2.meters
}
```

Context Bounds

Let's look at `genSort`'s declaration again:

```
def genSort[T](xs: List[T])(implicit cmp: CompareT[T]): List[T] = xs match {
  case Nil => Nil
  case head :: tail => genInsert(head, genSort(tail))
}
```

- `cmp` is required, but never used in the method by name
- It simply must be there for further calls to `genSort` and `genInsert` which need the implicit
- Being marked as implicit in the parameter list, the `cmp` is also implicitly available in the body of the method for making these calls
- Scala has a syntactic shortcut for that, *Context Bounds*:

```
def genSort[T: CompareT](xs: List[T]): List[T] = ...
```

- This automatically adds the `(implicit anonName: CompareT[T])` parameter

implicitly

- Can we use context bounds where we still need to use the implicit?
- For this, we can use the `implicitly` function:

```
def genInsert[T: CompareT](item: T, rest: List[T]): List[T] = {
  val cmp = implicitly[CompareT[T]]
  rest match {
    case Nil                                    => List(item)
    case head :: _ if cmp.isSmaller(item, head) => item :: rest
    case head :: tail                           => head :: genInsert(item, tail)
  }
}
```

- We use `implicitly[CompareT[T]]` to look up the implicit we know must be there (from the context bounds)

```
@inline def implicitly[T](implicit theImplicit: T) = theImplicit
```

- This is the definition of implicitly, cool huh?

Composition of Type-Classes (Just-In-Time)

- Everyone eventually writes a JSON library (but boy do they make a good demo)

```
trait JSONWrite[T] {
    def toJsonString(item: T): String
}

def jsonify[T: JSONWrite](item: T): String =
    implicitly[JSONWrite[T]].toJsonString(item)
```

- Our implementation is deliberately very simple, goes straight to strings, and is mostly useless, but it shows how type classes can compose

jsonify a String

```
jsonify("hello")
> could not find implicit value for evidence parameter of type JSONWrite[String]

// so let's define one
implicit object StringJSONWrite extends JSONWrite[String] {
    def toJsonString(item: String) = s""""$item""""
}

jsonify("hello")
> res3: String = """
> "hello"
> """
```

And an Int:

```
implicit object IntJSONWrite extends JSONWrite[Int] {
    def toJsonString(item: Int) = item.toString
}

jsonify(7)
> res4_1: String = "7"
```

jsonify a List[T]

- How about if we want a generic List jsonify?
- We shouldn't have to write one for each type T we want to list out, and we don't
- Implicit `defs` !

```
implicit def listJSONWrite[T: JSONWrite] = new JSONWrite[List[T]] {
    def toJsonString(xs: List[T]): String = {
        xs.map(x => jsonify(x)).mkString("[", " ,", "]")
    }
}
```

- Wow!

```
jsonify(List(1,2,3))
jsonify(List("hello", "json", "composition"))

res6_0: String = "[1 ,2 ,3]"
res6_1: String = """
["hello" ,"json" ,"composition"]
"""
```

How about a Map?

```
implicit def mapJSONWrite[T: JSONWrite] = new JSONWrite[Map[String, T]] {
  def toJsonString(m: Map[String, T]): String = {
    val pairs = for ((k, v) <- m) yield
      s"${jsonify(k)}: ${jsonify(v)}"

    pairs.mkString("{\n  ", ",\n  ", "\n}")
  }
}
```

```
jsonify(Map("one" -> 1, "two" -> 2))
> res14: String = """
> {
>   "one": 1,
>   "two": 2
> }
> """
```

Map of Lists

```
jsonify(Map("names" -> List("fred", "harry"), "cars" -> List("mustang", "aston")))
> res15: String = """
> {
>   "names": ["fred" ,"harry"],
>   "cars": ["mustang" ,"aston"]
> }
> """
```

- This is still limited to a single map value type that must have a JSONWrite
- More sophisticated implementations work around this with intermediate types and other machinery
- See spray-json, upickle, play-json and numerous, multitudinous other implementations of JSON libraries for more details
- At the core of each of these is this same compositional behavior

Implicit Rules and Restrictions

- Only one parameter list can be implicit, and it must be the last parameter list
- The implicit parameter list starts with the `implicit` keyword
- All parameters in the list are then implicit
- Parameters may be supplied explicitly (but if so, all of them must be supplied)
- Implicit values, objects or defs must be imported directly into scope by name
- Scala will not apply an implicit if the code will compile without doing so

Class and Type Tags

In addition to type classes you define, or in the core libraries, there are a few that the Scala compiler can fill in automatically on demand: `ClassTag` and `TypeTag`

ClassTag

Inserts a `classOf` for a generic type when used as an implicit bound:

```
import scala.reflect._
def getClassTag[T: ClassTag](x: T): ClassTag[T] = classTag[T]
val intCT = getClassTag(10)
intCT.runtimeClass  // res0: Class[_$1] = int
getClassTag("hello").runtimeClass   // res1: Class[_$1] = class java.lang.String
```

ClassTag context bounds also allow creation of specific array types:

```
val intArr = intCT.newArray(5)  // intArr: Array[Int] = Array(0, 0, 0, 0, 0)
```

Which, in this case, creates an array of primitive ints of size 5, very handy.

ClassTags in Pattern Matches

`ClassTags` can help with certain type erasure problems in pattern matches:

```
def isA[T](x: Any): Boolean = x match {
  case _: T => true
  case _ => false
}
Warning: abstract type pattern T is unchecked since it is eliminated by erasure
    case _: T => true
isA[Int](7)        // true
isA[Int]("Hello")  // true!
```

But with an implicit ClassTag:

```
def isA[T: ClassTag](x: Any): Boolean = x match {
  case _: T => true
  case _ => false
}
// no warning
isA[Int](7)  // true
isA[Int]("Hello")  // false
```

ClassTag vs TypeTag

But ClassTag only provides class tagging (and pattern matching) for the top type still:

```
isA[Map[String, Int]](Map("hello" -> 2))     // true
isA[Map[String, Int]](Map("hello" -> "foo")) // true :-(
```

TypeTag provides most everything the Compiler knows about the type:

```
val ct = classTag[Map[String, List[Int]]]
// ct: scala.reflect.ClassTag[Map[String,List[Int]]] =
// scala.collection.immutable.Map

val tt = typeTag[Map[String, List[Int]]]
// tt: reflect.runtime.universe.TypeTag[Map[String,List[Int]]] =
// TypeTag[Map[String,scala.List[Int]]]
```

TypeTag in a Guard

Sadly, this is not a simple drop in fix for pattern matching, but it can be used with effort and guards

```
case class Tagged[A](value: A)(implicit val tag: TypeTag[A])

val taggedMap1 = Tagged(Map(1 -> "one", 2 -> "two"))
val taggedMap2 = Tagged(Map(1 -> 1, 2 -> 2))

def taggedIsA[A, B](x: Tagged[Map[A, B]]): Boolean = x.tag.tpe match {
  case t if t =:= typeOf[Map[Int, String]] => true
  case _ => false
}

taggedIsA(taggedMap1)  // true
taggedIsA(taggedMap2)  // false
```