Helping Java + Scala Interact

- Implicits, “Pimp My Library” and various conversion helper tools simplify the work of interacting with Java.
- Scala and Java have their own completely different collection libraries.
- Some builtins ship with Scala to make this easier.

Helping Java + Scala Interact

- Implicits, “Pimp My Library” and various conversion helper tools simplify the work of interacting with Java.
- Scala and Java have their own completely different collection libraries.
- Some builtins ship with Scala to make this easier.

Helping Java + Scala Interact

- Implicits, “Pimp My Library” and various conversion helper tools simplify the work of interacting with Java.
- Scala and Java have their own completely different collection libraries.
- Some builtins ship with Scala to make this easier.

Interoperability in Scala 2.7.x

- Scala 2.7.x shipped with `scala.collection.jcl`.
- `scala.collection.jcl.Conversions` contained some implicit converters, but only to and from the wrapper versions - no support for “real” Scala collections.
- Neglected useful base interfaces like `Iterator` and `Iterable`
- @jorgeortiz85 provided `scala-javautils`, which used “Pimp My Library” to do a better job.

Interoperability in Scala 2.7.x

- Scala 2.7.x shipped with `scala.collection.jcl`.
- `scala.collection.jcl.Conversions` contained some implicit converters, but only to and from the wrapper versions - no support for "real" Scala collections.
- Neglected useful base interfaces like `Iterator` and `Iterable`
- @jorgeortiz85 provided `scala-javautils`, which used "Pimp My Library" to do a better job.

Interoperability in Scala 2.7.x

- Scala 2.7.x shipped with `scala.collection.jcl`.
- `scala.collection.jcl.Conversions` contained some implicit converters, but only to and from the wrapper versions - no support for “real” Scala collections.
- Neglected useful base interfaces like `Iterator` and `Iterable`
- @jorgeortiz85 provided `scala-javautils`, which used “Pimp My Library” to do a better job.

Interoperability in Scala 2.8.x

- Scala 2.8.x improves the interop game significantly.
- JCL is gone - focus has shifted to proper interoperability w/ built-in types.
- `scala.collection.jcl.Conversions` replaced by `scala.collection.JavaConversions` - provides implicit conversions to & from Scala & Java Collections.
- Includes support for the things missing in 2.7 (`Iterable`, `Iterator`, etc.)
- Great for places where the compiler can guess what you want (implicits); falls short in some cases (like BSON Encoding, as we found in Casbah)
- @jorgeortiz85 has updated `scala-javautils` for 2.8 with `scalaj-collection`
- Explicit `asJava` / `asScala` methods for conversions. Adds `foreach` method to Java collections.

Interoperability in Scala 2.8.x

- Scala 2.8.x improves the interop game significantly.
- JCL is gone - focus has shifted to proper interoperability w/ built-in types.
- `scala.collection.jcl.Conversions` replaced by `scala.collection.JavaConversions` - provides implicit conversions to & from Scala & Java Collections.
- Includes support for the things missing in 2.7 (`Iterable`, `Iterator`, etc.)
- Great for places where the compiler can guess what you want (implicits); falls short in some cases (like BSON Encoding, as we found in Casbah)
- @jorgeortiz85 has updated `scala-javautils` for 2.8 with `scalaj-collection`
- Explicit `asJava` / `asScala` methods for conversions. Adds `foreach` method to Java collections.

Interoperability in Scala 2.8.x

- Scala 2.8.x improves the interop game significantly.
- JCL is gone - focus has shifted to proper interoperability w/ built-in types.
- `scala.collection.jcl.Conversions` replaced by `scala.collection.JavaConversions` - provides implicit conversions to & from Scala & Java Collections.
- Includes support for the things missing in 2.7 (`Iterable`, `Iterator`, etc.)
- Great for places where the compiler can guess what you want (implicits); falls short in some cases (like BSON Encoding, as we found in Casbah)
- @jorgeortiz85 has updated `scala-javautils` for 2.8 with `scalaj-collection`
- Explicit `asJava` / `asScala` methods for conversions. Adds `foreach` method to Java collections.

Interoperability in Scala 2.8.x

- Scala 2.8.x improves the interop game significantly.
- JCL is gone - focus has shifted to proper interoperability w/ built-in types.
- `scala.collection.jcl.Conversions` replaced by `scala.collection.JavaConversions` - provides implicit conversions to & from Scala & Java Collections.
- Includes support for the things missing in 2.7 (`Iterable`, `Iterator`, etc.)
- Great for places where the compiler can guess what you want (implicits); falls short in some cases (like BSON Encoding, as we found in Casbah)
- @jorgeortiz85 has updated `scala-javautils` for 2.8 with `scalaj-collection`
- Explicit `asJava` / `asScala` methods for conversions. Adds `foreach` method to Java collections.

So WTF is an ‘Implicit’, anyway?

- Implicit Arguments
  - ‘Explicit’ arguments indicates a method argument you pass, well *explicitly*.
  - ‘Implicit’ indicates a method argument which is. . . *implied*. (But you can pass them explicitly too.)
  - Implicit arguments are passed in Scala as an additional argument list:

```
import com.mongodb._
import org.bson.types.ObjectId

def query(id: ObjectId)(implicit coll: DBCollection) = coll.findOne(id)

val conn = new Mongo()
val db = conn.getDB("test")
implicit val coll = db.getCollection("testData")

// coll is passed implicitly
query(new ObjectId())

// or we can override the argument
query(new ObjectId())(db.getCollection("testDataExplicit"))
```

- How does this differ from default arguments?

So WTF is an 'Implicit', anyway?

- Implicit Arguments
  - 'Explicit' arguments indicates a method argument you pass, well *explicitly*.
  - 'Implicit' indicates a method argument which is. . . *implied*. (But you can pass them explicitly too.)
  - Implicit arguments are passed in Scala as an additional argument list:

```
import com.mongodb._
import org.bson.types.ObjectId

def query(id: ObjectId)(implicit coll: DBCollection) = coll.findOne(id)

val conn = new Mongo()
val db = conn.getDB("test")
implicit val coll = db.getCollection("testData")

// coll is passed implicitly
query(new ObjectId())

// or we can override the argument
query(new ObjectId())(db.getCollection("testDataExplicit"))
```

  - How does this differ from default arguments?

So WTF is an 'Implicit', anyway?

- Implicit Methods/Conversions
  - If you try passing a type to a Scala method argument which doesn't match...

```
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212") // won't compile
```

  - A fast and loose example, but simple. Fails to compile.
  - But with implicit methods, we can provide a conversion path...

```
implicit def strToNum(x: String) = x.toInt
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212")
```

  - In a dynamic language, this may be called "monkey patching". Unlike Perl, Python, etc. Scala resolves implicits at compile time.

NOVUS

So WTF is an 'Implicit', anyway?

- Implicit Methods/Conversions
  - If you try passing a type to a Scala method argument which doesn't match. . .

```
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212") // won't compile
```

  - A fast and loose example, but simple. Fails to compile.
  - But with implicit methods, we can provide a conversion path. . .

```
implicit def strToNum(x: String) = x.toInt
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212")
```

  - In a dynamic language, this may be called "monkey patching". Unlike Perl, Python, etc. Scala resolves implicits at compile time.

So WTF is an ‘Implicit’, anyway?

- Implicit Methods/Conversions
  - If you try passing a type to a Scala method argument which doesn’t match. . .

```
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212") // won't compile
```

  - A fast and loose example, but simple. Fails to compile.
  - But with implicit methods, we can provide a conversion path. . .

```
implicit def strToNum(x: String) = x.toInt
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212")
```

  - In a dynamic language, this may be called “monkey patching”. Unlike Perl, Python, etc. Scala resolves implicits at compile time.

So WTF is an ‘Implicit’, anyway?

- Implicit Methods/Conversions
  - If you try passing a type to a Scala method argument which doesn’t match...

```
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212") // won't compile
```

  - A fast and loose example, but simple. Fails to compile.
  - But with implicit methods, we can provide a conversion path...

```
implicit def strToNum(x: String) = x.toInt
def printNumber(x: Int) = println(x)

printNumber(5)
printNumber("212")
```

  - In a dynamic language, this may be called “monkey patching”. Unlike Perl, Python, etc. Scala resolves implicits at compile time.

Pimp My Library

- Coined by Martin Odersky in a 2006 Blog post. Similar to C# extension methods, Ruby modules.
- Uses implicit conversions to tack on new methods at runtime.
- Either return a new "Rich_" or anonymous class. . .

```
import com.mongodb.gridfs.{GridFSInputFile => MongoGridFSInputFile}

class GridFSInputFile protected[mongodb](override val underlying:
      MongoGridFSInputFile) extends GridFSFile {
  def filename_=(name: String) = underlying.setFilename(name)
  def contentType_=(cT: String) = underlying.setContentType(cT)
}

object PimpMyMongo {
  implicit def mongoConnAsScala(conn: Mongo) = new {
    def asScala = new MongoConnection(conn)
  }

  implicit def enrichGridFSInput(in: MongoGridFSInputFile) =
    new GridFSInputFile(in)
}

import PimpMyMongo._
```

- A note: with regards to type boundaries, `[A <: SomeType]` won't allow implicitly converted values. You can whitelist them by using `[A <% SomeType]` instead.

Pimp My Library

- Coined by Martin Odersky in a 2006 Blog post. Similar to C# extension methods, Ruby modules.
- Uses implicit conversions to tack on new methods at runtime.
- Either return a new "Rich_" or anonymous class. . .

```
import com.mongodb.gridfs.{GridFSInputFile => MongoGridFSInputFile}

class GridFSInputFile protected[mongodb](override val underlying:
      MongoGridFSInputFile) extends GridFSFile {
  def filename_=(name: String) = underlying.setFilename(name)
  def contentType_=(cT: String) = underlying.setContentType(cT)
}

object PimpMyMongo {
  implicit def mongoConnAsScala(conn: Mongo) = new {
    def asScala = new MongoConnection(conn)
  }

  implicit def enrichGridFSInput(in: MongoGridFSInputFile) =
    new GridFSInputFile(in)
}

import PimpMyMongo._
```

- A note: with regards to type boundaries, `[A <: SomeType]` won't allow implicitly converted values. You can whitelist them by using `[A <% SomeType]` instead.

Pimp My Library

- Coined by Martin Odersky in a 2006 Blog post. Similar to C# extension methods, Ruby modules.
- Uses implicit conversions to tack on new methods at runtime.
- Either return a new “Rich_” or anonymous class. . .

```
import com.mongodb.gridfs.{GridFSInputFile => MongoGridFSInputFile}

class GridFSInputFile protected[mongodb](override val underlying:
      MongoGridFSInputFile) extends GridFSFile {
  def filename_=(name: String) = underlying.setFilename(name)
  def contentType_=(cT: String) = underlying.setContentType(cT)
}

object PimpMyMongo {
  implicit def mongoConnAsScala(conn: Mongo) = new {
    def asScala = new MongoConnection(conn)
  }

  implicit def enrichGridFSInput(in: MongoGridFSInputFile) =
    new GridFSInputFile(in)
}

import PimpMyMongo._
```

- A note: with regards to type boundaries, `[A <: SomeType]` won’t allow implicitly converted values. You can whitelist them by using `[A <% SomeType]` instead.