XML and JSON

Serialization and Extractors

Agenda

XML Data Example:

```xml
<library>
    <book>
      <title>Catcher in the Rye</title>
      <author>J.D. Salinger</author>
      <year>1951</year>
      <pages>224</pages>
    </book>
    <journal>
      <title>National Geographic</title>
      <issue>170</issue>
      <month>12</month>
      <year>2005</year>
    </journal>
    <dvd>
      <title>This is Spinal Tap</title>
      <director>Rob Reiner</director>
      <year>1984</year>
      <length>82</length>
    </dvd>
</library>
```

Isn't that pretty?

XML Support in Scala?

- Now in a Separate Module

```
libraryDependencies += "org.scala-lang.modules" %% "scala-xml" % "1.0.6"
```

- Although XML syntax is still enabled without this, you need it for file support, x-path, etc.

- Loading an XML file:

```
import java.io.File

val xmlFile = new File(
  getClass.getClassLoader.getResource("LibraryItems.xml").toURI
)

import scala.xml.XML

val books = XML.loadFile(xmlFile)
```

X-Path in Scala

- X-Path uses notation like `outer / child / field` and `outer // field` to select specific fields or do deep searches
- Since // denotes comments in Scala, we use \ and \\ instead

```
books \ "book" // scala.xml.NodeSeq = <book>...

books \ "title" // Empty (no match)

books \\ "title" // scala.xml.NodeSeq = <title>Catcher in the Rye</title>...
```

To get just the text within the \\ lists, use `.text`, e.g.:

```
for (book <- books \ "book") {
  println((book \ "title").text)
}

// Catcher in the Rye
// Daemon
// Lila
```

XML Attributes vs Elements

Another common form of XML:

```xml
<library>
    <item type="book" title="Catcher in the Rye"
          author="J.D. Salinger" year="1951" pages="224"/>
    <item type="journal" title="National Geographic"
          issue="170" month="12" year="2005"/>
    <item type="dvd" title="This is Spinal Tap"
          director="Rob Reiner" year="1984" length="82"/>
</library>
```

Attribute form. Usage in Scala x-path is similar but using `@`:

```scala
val booksAttrs = XML.loadFile(xmlFileAttr)

booksAttrs \\ "item"

for (b <- booksAttrs \\ "item") {
  println((b \ "@type").text + ": " + (b \ "@title").text)
}
```

XML Literals

XML Literals can be placed directly into Scala code:

```
val bookXml = <book>
    <title>I ah to Thai</title>
    <author>E.R. Das</author>
    <year>1992</year>
    <pages>10</pages>
  </book>
```

They can also be used to pattern match:

```
bookXml match {
  case <book>{contents @ _*}</book> =>
    println(s"""It's a book: ${contents.text}""")
  case _ =>
    println("Not a book")
}
```

XML Serialization

Serializing data out to XML is trivial once you know that Scala code in {}s can be mixed with XML elements:

```
case class Book(title: String, author: String, year: Int, pages: Int)

def bookToXML(book: Book) = {
  import book._
  <book>
    <title>{title}</title>
    <author>{author}</author>
    <year>{year}</year>
    <pages>{pages}</pages>
  </book>
}

val bookList = Seq(
  Book("Catcher in the Rye", "J.D. Salinger", 1951, 224),
  Book("I Ah to Thai", "E.R. Das", 1992, 10)
)

println(<library>{bookList.map(bookToXML)}</library>)
```

XML Serialization

Likewise, deserialization is just a case of applying x-path, .text and parsing functions:

```
def xmlToBook(xml: scala.xml.Elem): Book = {
  val title = (xml \ "title").text
  val author = (xml \ "author").text
  val year = (xml \ "year").text.toInt
  val pages = (xml \ "pages").text.toInt
  Book(title, author, year, pages)
}

val bookXml = <book>
    <title>I ah to Thai</title>
    <author>E.R. Das</author>
    <year>1992</year>
    <pages>10</pages>
  </book>

xmlToBook(bookXml)
```

XML Type Class Solution

- The above will work, but we are used to better in Scala
- For a more idiomatic Scala solution, http://scalaxb.org/ is a mature type class solution
- Or for the experience you could roll your own (then throw it away and use this instead :-))

JSON in Scala

- Unlike XML, there is no standard JSON feature or library in Scala
- Instead it is perhaps the single most third-party-implemented library in Scala
- We had to pick one for this course
- In the past it would have been Spray.JSON
- But since that is largely defunct, we went with Play JSON instead

Play JSON

```
libraryDependencies += "com.typesafe.play" %% "play-json" % "2.6.6"
```

```
import play.api.libs.json._

val fred = Json.parse("""{"name": "fred", "age": 20}""")

fred \ "name"
fred \ "age"

(fred \ "name").asOpt[String]  // res2: Option[String] = Some(fred)
(fred \ "age").asOpt[Int]  // res3: Option[Int] = Some(20)
(fred \ "age").asOpt[String]  // res4: Option[String] = None
(fred \ "rage").asOpt[Int]   // res5: Option[Int] = None
```

X-path syntax like XML, but better, type-class based parsing.

Play Create JSON

```scala
val title = "I Ah to Thai"
val author = "E.R. Das"
val pages = 10
val year = 1992

val book = JsObject(Map(
  "title" -> JsString(title),
  "author" -> JsString(author),
  "pages" -> JsNumber(pages),
  "year" -> JsNumber(year)
))

// alternatively:
Json.parse(
  s"""{ "title": "$title", "author": "$author", "pages": $pages, "year": $year}"""
)
```

Both have the same result, though in the first case you will get a `JsObject` which is narrower than the `JsValue` returned by the parse.

Play JSON Printing and Standard Definitions

```
Json.prettyPrint(book) // indented and formatted
book.toString   // compact printed
```

Standard type-class definitions are included

```
val nums = List(1,2,3,4,5)
Json.toJson(nums) // play.api.libs.json.JsValue = [1,2,3,4,5]

val words = List("nitwit", "wibble", "floobah")
Json.toJson(words) // play.api.libs.json.JsValue = ["nitwit","wibble","floobah"]

Json.toJson(Map(
  "hi" -> 1,
  "yo" -> 2
)) // play.api.libs.json.JsValue = {"hi":1,"yo":2}
```

Compile Time Type Checked

E.g. the following will not work:

```
Json.toJson(Map(
  "hi" -> 1,
  "yo" -> "two"
))

// No Json serializer found for type scala.collection.immutable.Map[String,Any].
// Try to implement an implicit Writes or Format for this type.
// Json.toJson(Map(
//              ^
```

And no, you should not implement a type-class for Any :-)

Custom Serializers

Simple, self provided:

```scala
case class Person(first: String, last: String, age: Int)

object Person {
  implicit val personFormat: Format[Person] = new Format[Person] {
    def reads(json: JsValue): JsResult[Person] = {
      val first = (json \ "first").as[String]
      val last = (json \ "last").as[String]
      val age = (json \ "age").as[Int]
      JsSuccess(Person(first, last, age))
    }

    def writes(o: Person): JsValue = {
      val itemsMap = Map(
        "first" -> JsString(o.first),
        "last" -> JsString(o.last),
        "age" -> JsNumber(o.age)
      )
      JsObject(itemsMap)
    }
  }
}
```

Play JSON Functional Syntax

```
import play.api.libs.functional.syntax._

case class Car(make: String, model: String, year: Int)

object Car {
  implicit val carFormat: Format[Car] = (
    (JsPath \ "make").format[String] and
      (JsPath \ "model").format[String] and
      (JsPath \ "year").format[Int]
    )(Car.apply, unlift(Car.unapply))
}
```

Must bring in the import or you will get confusing compile errors.

Or... Use Macros

```scala
case class Car(make: String, model: String, year: Int)

object Car {
  // or, the macro way...
  implicit val carFormat: Format[Car] = Json.format[Car]
}
```

- Very easy, but two down-sides:
  - You can no longer customize the names used for the fields
  - You have to use macros (and that can make debugging harder)

Using the Serializers

```scala
val p1 = Person("Harry", "Potter", 22)
val p2 = Person("Sally", "James", 23)

val people = Map(
  "harry" -> p1,
  "sally" -> p2
)

val peopleStr = Json.prettyPrint(Json.toJson(people))
val items = Json.parse(peopleStr).as[Map[String, Person]]
```

and

```scala
val cars = Seq(
  Car("Ford", "Mustang", 1965),
  Car("Honda", "S2000", 2002)
)

val carsStr = Json.prettyPrint(Json.toJson(cars))
val carsAgain = Json.parse(carsStr).as[Seq[Car]]
val carsJson = Json.parse(carsStr)
for (make <- carsJson \\ "make") println(make.as[String])
```

Extractors

When we make a case class we can pattern match on it:

```
case class Person(name: String, age: Int)

val p1 = Person("Gloria", 39)

p1 match {
  case Person(name, 39) => s"name is 39"
}
```

This is because one of the methods generated by Scala on a case class is called `unapply` and it works like this:

```
Person.unapply(p1)  // Option[(String, Int)] = Some((Gloria,39))
```

Knowing this, we can write our own custom extractors for anything we like.

Custom URL Extractor

```scala
import scala.util.Try

val webUrl = "http://www.escalatesoft.com/training/index.html"

object WebURL {
  def unapply(s: String): Option[(String, String, String)] = Try {
    val split1 = s.indexOf("://")
    val protocol = s.substring(0, split1)
    val rest = s.substring(split1 + 3)
    val split2 = rest.indexOf("/")
    val server = rest.substring(0, split2)
    val loc = rest.substring(split2 + 1)
    (protocol, server, loc)
  }.toOption

  def apply(fields: (String, String, String)): String = {
    val (proto, svr, loc) = fields
    s"$proto://$svr/$loc"
  }
}
```

Note the symmetry between `apply` and `unapply` - this is good practice.

Pattern Matching Using the Extractor

```scala
WebURL.unapply(webUrl) // Option[(String, String, String)] =
// Some((http,www.escalatesoft.com,training/index.html))

WebURL.apply("http", "www.escalatesoft.com", "training/index.html")
// String = http://www.escalatesoft.com/training/index.html

def splitUrl(s: String) = s match {
  case WebURL(proto, svr, loc) =>
    println(proto)
    println(svr)
    println(loc)
  case _ => println("no match")
}

splitUrl(webUrl)

splitUrl("https://www.google.com/images")
```

unapplySeq

Suppose we also want to include repeating segments of location at the end (delimited by /)

```scala
object WebURLSeq {
  def unapplySeq(s: String): Option[(String, String, Seq[String])] = Try {
    val split1 = s.split("://")
    val protocol = split1(0)
    val rest = split1(1).split("/").toList
    (protocol, rest.head, rest.tail)
  }.toOption

  def apply(fields: (String, String, Seq[String])): String = {
    val (proto, svr, locs) = fields
    s"""$proto://$svr/${locs.mkString("/")}"""
  }
}

val (protocol, server, locations) = WebURLSeq.unapplySeq(webUrl).get
// protocol: String = http
// server: String = www.escalatesoft.com
// locations: Seq[String] = List(training, index.html)
```

Pattern Matching an unapplySeq

The last item is repeating, possibly empty. Use expansion operator...

```
def splitUrlSeq(s: String) = s match {
  case WebURLSeq(proto, svr, loc @ _*) =>
    println(proto)
    println(svr)
    println(loc)
  case _ => println("no match")
}

splitUrlSeq(webUrl)

splitUrlSeq("https://www.google.com/images")
```

Regex Extractors

We can also use a regular expression to pattern match directly as well:

```
val URLString = """^([^:]+.)://([^/]+)/(.+)?$""".r
```

`.r` creates a Regex automatically from the String.

In use

```
val URLString(proto, site, rest) = "https://www.google.com/images"
// proto: String = https
// site: String = www.google.com
// rest: String = images
```

But beware!

```
val URLString(proto2, site2, rest2) = "http://www.something.com/"
// proto2: String = http
// site2: String = www.something.com
// rest2: String = null   // oh noooooooooooo!
```