Updated: 2021-03-31

6.5
Java-Scala Interoperability

Interoperating Java with Scala

- The ClassLoader is language-agnostic:
  - Java classes are Scala classes and *vice versa*. The JVM (classloader) doesn't know the difference.

- So, what's the problem?
  - One type of problem arises when you want to extend a Java class with a Scala class, or *vice versa*. This can be done in many situations without problems, but I don't believe there's really any compelling reason to do so.
  - Another issue arises when you have a Java *List* and a Scala *List*. These are not the same class: *java.util.List* and *scala.collection.immutable.List*. Therefore, they cannot be used as if they were. But Scala provides a simple way to convert between the corresponding collection types (see following slides).
  - And then there's the fact that you can pass one of the standard monadic wrappers, e.g. *Try*, back to a Java method, but it won't be able to access it in a monadic way* (not really a major issue).

*** actually, this may no longer be true with Java 11**

Interoperating Java with Scala (2)

- Best practices:
  - Define cross-language APIs as much as possible using compatible types:
    - Simple, unwrapped, scalar types (*String, Int/Integer, Double*, etc.)—automatically converted;
    - “our own” data Structures;
- Tend to call Scala code from Java but not the other way around:
  - Provide additional Java-centric signatures which call their corresponding Scala methods;
  - Don’t return *Try[X]*: instead return *Option[X]* after logging the exception in the Scala method;
  - But if you return *Try[Boolean]* from Scala, it will not be converted to *Try<java.lang.Boolean>* in Java. You can only wrap explicit classes (that are the same).
    - For instance, a wrapped *Unit* gets converted to a wrapped *BoxedUnit*.
  - When it is necessary to reference a collection in a method, I recommend doing the conversion in Scala code and providing a Java-specific API for that method (in addition to the Scala API).

Interoperating Java with Scala (3)

- Best practices, continued:
  - When defining a Scala method with a function parameter, use only simple functions:
    - non-curried functions: i.e. those that correspond to Java8's function types.
  - Don't define a tuple as the return type from a Scala method:
    - instead, define a case class in your Scala code and return that so that it can be easily referenced on the Java side.
  - Avoid defining parameters with default values (Java can't omit the argument):
    - Instead, just create a method signature without the default value.
    - It is possible to use the *object.method$default$Number()* mechanism but that is extremely inelegant!
  - If your Scala method expects an *Option[T]*, then pass values as either *Option.apply(t)* or *Option.empty()*;
    - Alternatively, use *Optional<T>* in your signature.

Example of passing in *Optional<T>*

- If you pass in *Optional<T>*, then you will need to convert to *Option[T]*:
  - You can do this by code:
    - ```
      def toJavaOptional[A](maybeA: Option[A]): Optional[A] = maybeA match {
        case Some(a) => Optional.of(a)
        case _ => Optional.empty()
      }
      ```
    - ```
      def toScalaOption[A](maybeA: Optional[A]): Option[A] = if (maybeA.isPresent) Some(maybeA.get)
      else None
      ```
  - Or you can add a dependency (sbt):

    `"org.scala-lang.modules" %% "scala-java8-compat" % "0.9.0” (sbt)`
    - **Or (maven):**

      ```
      <dependency>
          <groupId>org.scala-lang.modules</groupId>
          <artifactId>scala-java8-compat_2.11</artifactId>
          <version>0.9.0</version>
      </dependency>
      ```
    - **And then, in your code:**

      ```
      import scala.compat.java8.OptionConverters._
      javaOptional.asScala
      ```
    - Or

      ```
      scalaOption.asJava
      ```

Interoperating Java with Scala: CollectionConverters

- Here is the definitive list of implicit conversions:
  - https://www.scala-lang.org/api/2.13.4/scala/jdk/CollectionConverters$.html
  - Import the converters and add *asScala* or *asJava* where appropriate.
  - Note that *Seq* in Scala is a trait, not a class. You cannot instantiate *Seq* like *List*.
  - However, in Scala, *Seq(1,2,3)* gets desugared into *List(1,2,3)* so it seems like it's a class with its own constructor or *apply* method.
  - Thus if you have a *java.util.List<A> list* and need a *Seq[A]*, just import the converters and write *list.asScala*.

Example: collections

- Scala:

```
object Collections {

def show[A](xs: Iterable[A]): Unit = xs foreach println

def showJava[A](xs: java.util.Collection[A]): Unit = {
  import collection.JavaConverters._
  show(xs.asScala)
  }
}
```

- Java:

```
public class CollectionsJ {

public static void main(String[] args) {
    Collection<String> strings = new ArrayList<>();
    strings.add("Curriculum");
    strings.add("Associates");
    Collections.showJava(strings);
}
}
```

Interoperating Java with Scala: Collections (2)

- Here's a neat trick that allows you to construct Java lists*:
  - It uses the same ability you use to construct Scala lists:
  - *Collections.scala*

```
        import collection.JavaConverters._
        import scala.annotation._
@varargs def createJavaList[A](xs: A*): java.util.List[A] = xs.asJava
```

  - *CollectionsJ.java*

```
        List<String> ca = Collections.createJavaList("Curriculum", "Associates");
Collections.showJava(ca);
```

  - The type of *ca* is a *Wrappers$SeqWrapper* which may not be what you want. But you can always get, say, an *ArrayList* like so:

```
        new ArrayList<>(Collections.createJavaList("Curriculum", "Associates"));
```

*** For some reason, the Java designers forgot to give us th**

Example: dealing with *Try[Unit]* in Java*:

- Scala code:

```
      object Trial {
  def trial(b: Boolean): Try[Unit] = if (b) Success() else Failure(new
Exception("b was false"))
}
```

- Java code (version 1):

```
   public static void main(String[] args) {
     Try<BoxedUnit> good = Trial.trial(true);
if (good.isSuccess()) System.out.println("good is OK");

Try<BoxedUnit> bad = Trial.trial(false);
if (bad.isFailure()) System.out.println("bad is OK");
String msg = bad.failed().get().getLocalizedMessage();
System.out.println("failure message: "+msg);
   }
```

*** If you feel you really have to**

Example: dealing with *Try[Unit]* in Java* (2):

- Scala code:

```
object Trial {
  def trial(b: Boolean): Try[Unit] = if (b) Success() else Failure(new Exception("b was false"))
}
```

- Java code (version 2):

```
public static void main(String[] args) {
    Try<BoxedUnit> tried = Trial.trial(false);
    tried.transform(TrialJ::processUnit, TrialJ::handleException);
}
private static Try<BoxedUnit> processUnit(BoxedUnit x) {
    System.out.println("OK");
    return new scala.util.Success<>( x );
}

private static Try<BoxedUnit> handleException(Throwable t) {
    System.out.println("Exception thrown: "+t.getLocalizedMessage());
    BoxedUnit x = BoxedUnit.UNIT;
    return new scala.util.Success<>( x );
}
```

*** If you feel you really have to**

Example:

- Java code fragment:

```
        // Get the datasets.
private Iterable<SourceDataSet> datasets = CleverDataSets.allSetsJava();
// Set up the flow.
private Flow<Progenitor, Account> flow = CleverAccountFlow.createFromJava(datasets, batchSize,
persister, akkaSystemName);
```

- Scala code fragments

```
        lazy val allSetsJava: java.lang.Iterable[SourceDataSet] = {
  import scala.collection.JavaConverters._
  import scala.language.implicitConversions
  allSets.asJava
}

def createFromJava(datasets: java.lang.Iterable[SourceDataSet], batchSize: Int, persister: Persister,
akkaSystemName: String): CleverAccountFlow = new CleverAccountFlow(datasets.asScala, batchSize,
persister, akkaSystemName)
```