Worksheet 02 — Scala version: Exercises on Scala Reflection

The Reflection API allows a Java program to inspect and manipulate itself; it comprises the `java.lang.Class` class and the `java.lang.reflect` package, which represents the members of a class with `Method`, `Constructor`, and `Field` objects. Scala can utilise the same API but this suffers from the same problems as the Java API, namely it looses the types of any generic types as Java erases this type information at compile time.

Scala provides an alternative mechanism which can retain the type information on generic types but the learning the Scala reflection API can be somewhat confusing. To help we will first discuss an example and then proceed to the worksheet questions.

Example

Let's start with a simple exercise to learn how to use the API.

How would we go about using the Scala reflection API to find out what methods are declared on a type?

First, you need to import the reflection runtime universe:

```
import scala.reflect.runtime.universe._
```

Most methods on the reflection api are centred around a `Type`

```
reflect.runtime.universe.Type
```

(We'll refer to `Type` as `universe.Type` from now on, to distinguish it from a normal type).

To get the `universe.Type` of a type, you can use the `typeOf` API method:

```
scala> typeOf[Option[_]]
res1: reflect.runtime.universe.Type = scala.Option[_]
```

Now that we have a `universe.Type` for our type, we can get the methods defined on it by using the `decls` method:

```
> res1.decls
res2: reflect.runtime.universe.MemberScope = SynchronizedOps(constructor Option, method isEmpty,
method isDefined, method get, method getOrElse, method orNull, method map, method fold,
method flatMap, method flatten, method filter, method filterNot, method nonEmpty,
method withFilter, class WithFilter, method contains, method exists, method forall,
method foreach, method collect, method orElse,
method iterator, method toList, method toRight, method toLeft)
```

You might notice that `decls` returns a `MemberScope`. WhatâĂŹs that? It's useful to realise that a `MemberScope` is a `Traversable`:

```
res2.is scala> res2.isInstanceOf[Traversable[_]]
res5: Boolean = true
```

You can use the methods available on any `Traversable` instance to process the `MemberScope`.

For instance, we could easily format the list of method declarations like so:

```
scala> res1.decls.mkString("\n")
res6: String =
def <init>: <?>
def isEmpty: <?>
def isDefined: <?>
def get: <?>
final def getOrElse[B <: <?>](default: <?>): B
final def orNull[A1 <: <?>](implicit ev: <?>): A1
final def map[B <: <?>](f: <?>): Option[B]
final def fold[B <: <?>](ifEmpty: <?>)(f: <?>): B
final def flatMap[B <: <?>](f: <?>): Option[B]
def flatten: <?>
final def filter(p: <?>): Option[A]
final def filterNot(p: <?>): Option[A]
final def nonEmpty: <?>
final def withFilter(p: <?>): Option.this.WithFilter
class WithFilter extends AnyRef
final def contains: <?>
final def exists(p: <?>): Boolean
final def forall(p: <?>): Boolean
final def foreach[U <: <?>](f: <?>): Unit
final def collect[B <: <?>](pf: <?>): Option[B]
final def orElse[B <: <?>](alternative: <?>): Option[B]
def iterator: <?>
def toList: <?>
final d...
```

Other useful methods

LetâĂŹs use the reflection API to figure out what other methods are available on `universe.Type`. We use the members method to list methods defined either directly or indirectly on `universe.Type`:

```
scala> typeOf[Type]
res7: reflect.runtime.universe.Type = scala.reflect.runtime.universe.Type

scala> res7.members.mkString("\n")
res8: String =
final def $asInstanceOf[T0](): T0
final def $isInstanceOf[T0](): Boolean
final def synchronized[T0](x$1: T0): T0
final def ##(): Int
final def !=(x$1: Any): Boolean
final def ==(x$1: Any): Boolean
final def ne(x$1: AnyRef): Boolean
final def eq(x$1: AnyRef): Boolean
final def notifyAll(): Unit
final def notify(): Unit
protected[package lang] def clone(): Object
```

```
final def getClass(): Class[_]
def hashCode(): Int
def toString(): String
def equals(x$1: Any): Boolean
final def wait(): Unit
final def wait(x$1: Long): Unit
final def wait(x$1: Long,x$2: Int): Unit
protected[package lang] def finalize(): Unit
final def asInstanceOf[T0]: T0
final def isInstanceOf[T0]: Boolean
def contains: <?>
def exists: <?>
def find: <?>
def foreach: <?>
def map: <?>
def substituteTypes: <?>
def...
```

What if you have an instance of a type and want to get a `universe.Type` for that? It looks like there is no built in method to do that. The recommended way is to write your own method for it:

```
scala> def getType[T: TypeTag](obj: T) = typeOf[T]
getType: [T](obj: T)(implicit evidence$1:
  reflect.runtime.universe.TypeTag[T])reflect.runtime.universe.Type
```

The Scala compiler will supply our `getType` method with an implicit for `TypeTag[T]`.

So what is a `TypeTag`?

> A `TypeTag[T]` encapsulates the runtime type representation of some type `T`. The main use case of `TypeTags` is to give access to erased types.

As with Java, Scala generic types which are present at compile time are erased at runtime (erasure). `TypeTags` are a way of having access to that lost compile time information at runtime.

With `getType` we can now extract the `universe.Type` of an instance:

```
scala> getType(List(1,2,3))
res9: reflect.runtime.universe.Type = List[Int]
```

Questions

1. Write a Scala program that reads the name of a class from the command line and emits the interface of the class syntax (trait or class, modifiers, constructors, methods, fields; no method bodies).

   Hint: You can still use the Java constructs so you can load a class whose name you know with `java.lang.Class.forName()`. The `java.lang.Class` class offers a rich interface that enables you to inspect the interface of any class.

   Apply this program to a set of classes and traits as test input. You may also use the program on itself.

2. Write a program that reads a class name and a list of arguments, and creates an object of that class where the read arguments are passed to the constructor.

   Hint: Treat arguments as strings. You can enumerate the constructors of the class/trait using the API. Choose a constructor with the appropriate parameter count. Then, find the parameter types. To create typed argument objects, call the appropriate constructors that take a string as their only argument.

   See the above example for a hint.

3. Normally it is up to the programmer to write a `toString` function for each class one creates. This exercise is about writing a general `toString` method once and for all. As part of the reflection API it is possible to find out which fields exist for a given object, and to get their values. The purpose of this exercise is to make a `toString` method that gives a printed representation of any object, in such a manner that all fields are printed, and references to other objects are handled as well.

   To solve this exercise, you will need to examine the API carefully.