Reflection

A Dip in the Reflection Pond

"Do you ever wonder if the guy in the puddle is real, and you're just a reflection of him?" — Bill Watterson

- Reflection/introspection is the ability of a programming language to examine itself. Java code can examine an object and find out the methods, fields, constructors and their attributes (within limits imposed by the Security Manager).
- Reflection is commonly used by programs that require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.
- Reflection is useful in situations where you need to work with objects that you don't know about in advance.
- Reflection is also a key strategy for metaprogramming.
- Reflection is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it.

Advantages of Reflection

- **Extensibility Features**: An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Class Browsers and Visual Development Environments**: A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- **Debuggers and Test Tools**: Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Drawbacks of Reflection

- **Performance Overhead**: Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Reflective operations should be avoided in sections of code that are called frequently in performance-sensitive applications.
- **Security Restrictions**: Reflection requires runtime permission, which may not be present when running under a security manager.
- **Exposure of Internals**: Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Reflection Classes

- The main reflection package is `java.lang.reflect`. The `Type` interface represents all types that can exist in a Java program. All concrete types are represented by an instance of the class `java.lang.Class`. The classes `Constructor`, `Method` and `Field` represent constructors, methods and fields. There is also a `java.lang.Package` class for package meta-data.
- The `Class` class is the starting point for reflection. It also provides a tool to manipulate classes, primarily to create objects of types whose names are specified by strings, and for loading classes using specialized techniques such as across the network.

The Class class

- The getClass() method of Object returns a reference to the Class object that produced the object instance.

```
String myString = "Moo!";
Class strClass = myString.getClass();
System.out.println(strClass.getName());
```

- For a particular class, we can get the class reference statically.

```
Class strClass = String.class;
```

- We can produce a new instance of a class using the reference to the Class.

```
Class strClass = String.class;
try { String s2 = (String)strClass.newInstance(); }
catch (InstantiationException e) {...} //for an  abstract class
catch (IllegalAccessException e) {...} //cannot access the constructor
```

- We can also look for a Class reference by name.

```
try { Class sneakersClass = Class.forName("Sneakers");
} catch (ClassNotFoundException e) {...}
```

"Bewilderment increases in the presence of the mirrors."
— Tarjei Vesaas, The Boat in the Evening

Example 1: SelfReflect

```
public class SelfReflect {
    public static void main(String [] args) {
        Class<Color> type = Color.class;

        Method [] methods = type.getMethods();
        for (int i=0; i< methods.length; i++)
            System.out.println( methods[i]);

        Field [] fields = type.getFields();
        for (int i=0; i< fields.length; i++)
            System.out.println( fields[i]);
    }
}
```

Example 2: ReflectOnAnother

```
public class ReflectOnAnother {
    public static void main(String [] args) {
        if (args.length == 0) {
            System.err.println("Usage: java ReflectOnAnother <class name>");
            System.exit(1);
        }
        Class<?> type = null;
        try {
            type = Class.forName(args[0]);
        } catch (ClassNotFoundException e) {
            System.err.println(e);
        }
        Method [] methods = type.getMethods();
        for (Method m: methods)
            if (Modifier.isPublic(m.getModifiers()))
                System.out.println("  " + m);

        Field [] fields = type.getFields();
        for (Field f: fields)
            if (Modifier.isPublic(f.getModifiers()))
                System.out.println("  " + f);
    }
}
```

Reflection versus Method Overriding

- Would reflection determine the right version of a overridden method that will be used?
- Suppose we have the following classes:

  `Animal.java`
  `Cat.java`
  `Calico.java`

  `Animal` has a `makeNoise()` method.
  `Cat` extends `Animal` and overrides the `makeNoise()` method. `Calico` extends `Cat`, but does not override any method.
- What would `ReflectOnAnother` program do with each of the above classes?

Example 3: TraceAncestry

```
public class TraceAncestry {
  public static void main(String[] args) {
      if (args.length == 0) {
          System.err.println("Usage: java TraceAncestry <class name>");
          System.exit(1);
      }
      Class<?> type = null;
      try {
          type = Class.forName(args[0]);
      } catch (ClassNotFoundException e) {
          System.err.println(e);
      }
      System.out.println("class " + type.getSimpleName());
      Class<?> superclass = type;
      do {
          superclass = superclass.getSuperclass();
          if (superclass == null) break;
          System.out.println("extends " + superclass.getCanonicalName());
      } while (!superclass.getCanonicalName().equals("java.lang.Object"));
  }
}
```

Reflection on reflection

With reflection one could write an interpreter in Java that could access the full Java API, create objects, invoke methods, modify variables, and do all the other things that a Java program can do at compile-time, while it is running!

Example 4: Dynamic Invocation of static method

```
// invoke a static method without any arguments
public class invoke {
 public static void main (String [] args) {
     try {
         Class c = Class.forName(args[0]);
         Method m = c.getMethod(args[1]);
         Object ret = m.invoke(null);
         System.out.println("Invoked static method: "+ args[1] +
               " of class: " + args[0] + " with no args\n Results: "+ret);
     } catch (ClassNotFoundException e1) {
         // cannot find the class
     } catch (NoSuchMethodException e2) {
         // that method doesn't exist
     } catch (IllegalAccessException e3) {
         // don't have permission to invoke that method
     } catch (InvocationTargetException e4) {
        // an exception occurred while invoking that method
        System.out.println("Method threw an: "+e4.getTargetException());
     }
 }
}
```

Example 5: Dynamic Invocation of a non-static method

```
public class InvokeDynamic
{
    public static void main (String [] args) throws Throwable
    {
        String str = "java.lang.reflect";
        Throwable failure = null;
        try {
            Method indexM = String.class.getMethod("indexOf",
                    String.class, int.class);
            System.out.println(indexM.invoke(str, ".", 8));
        } catch (NoSuchMethodException e) {
            failure = e;
        } catch (IllegalAccessException e) {
            failure = e;
        } catch (InvocationTargetException e) {
            failure = e;
        }
        if (failure != null)
            throw failure;
    }
}
```

Recommended Exercises

- Create an `Interpret` program that creates an object of a requested type and allows the user to examine and modify fields of that object.
- Modify your `Interpret` program to invoke methods on the object. You should properly display any values returned or exceptions thrown.
- Modify your `Interpret` program further to let users invoke constructors of an arbitrary class, displaying any exceptions. If a construction is successful, let users invoke methods on the returned object.

From Chapter 16 of the *The Java Programming Language, 4th Ed.*