Application Development Framework
Application Express
Big Data
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
Database
.NET
Dynamic Scripting Languages
Embedded
Digital Experience
Enterprise Architecture
Enterprise Management
Identity & Security
Java
Linux
Mobile
Service-Oriented Architecture
Solaris
SQL & PL/SQL
Systems - All Articles
Virtualization

# Using Java Reflection

*By Glen McCluskey*

January 1998

Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.

The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.

One tangible use of reflection is in JavaBeans, where software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded.

## A Simple Example

To see how reflection works, consider this simple example:

```
import java.lang.reflect.*;

public class DumpMethods {
  public static void main(String args[])
  {
    try {
      Class c = Class.forName(args[0]);
      Method m[] = c.getDeclaredMethods();
      for (int i = 0; i < m.length; i++)
      System.out.println(m[i].toString());
    }
    catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

For an invocation of:

```
java DumpMethods java.util.Stack
```

the output is:

```
public java.lang.Object java.util.Stack.push(
  java.lang.Object)
public synchronized
  java.lang.Object java.util.Stack.pop()
public synchronized
  java.lang.Object java.util.Stack.peek()
public boolean java.util.Stack.empty()
public synchronized
  int java.util.Stack.search(java.lang.Object)
```

That is, the method names of class `java.util.Stack` are listed, along with their fully qualified parameter and return types.

This program loads the specified class using `class.forName`, and then calls `getDeclaredMethods` to retrieve the list of methods defined in the class. `java.lang.reflect.Method` is a class representing a single class method.

## Setting Up to Use Reflection

The reflection classes, such as `Method`, are found in java.lang.reflect. There are three steps that must be followed to use these classes. The first step is to obtain a `java.lang.Class` object for the class that you want to manipulate.`java.lang.Class` is used to represent classes and interfaces in a running Java program.

One way of obtaining a Class object is to say:

```
Class c = Class.forName("java.lang.String");
```

to get the Class object for `String`. Another approach is to use:

```
Class c = int.class;
```

or

```
Class c = Integer.TYPE;
```

to obtain Class information on fundamental types. The latter approach accesses the predefined `TYPE` field of the wrapper (such as `Integer`) for the fundamental type.
The second step is to call a method such as `getDeclaredMethods`, to get a list of all the methods declared by the class.

Once this information is in hand, then the third step is to use the reflection API to manipulate the information. For example, the sequence:

```
Class c = Class.forName("java.lang.String");    Method m[] = c.getDeclaredMethods();    System.out.println(m[0].toString());
```

will display a textual representation of the first method declared in `String`.
In the examples below, the three steps are combined to present self contained illustrations of how to tackle specific applications using reflection.

## Simulating the `instanceof` Operator

Once Class information is in hand, often the next step is to ask basic questions about the Class object. For example, the`Class.isInstance` method can be used to simulate the `instanceof` operator:

```
class A {}

public class instance1 {
  public static void main(String args[])
  {
    try {
      Class cls = Class.forName("A");
```

```
        boolean b1
          = cls.isInstance(new Integer(37));
        System.out.println(b1);
        boolean b2 = cls.isInstance(new A());
        System.out.println(b2);
      }
      catch (Throwable e) {
        System.err.println(e);
      }
    }
  }
```

In this example, a Class object for A is created, and then class instance objects are checked to see whether they are instances of A. Integer(37) is not, but new A() is.

### Finding Out About Methods of a Class

One of the most valuable and basic uses of reflection is to find out what methods are defined within a class. To do this the following code can be used:

```
  import java.lang.reflect.*;

  public class method1 {
    private int f1(
     Object p, int x) throws NullPointerException
    {
      if (p == null)
        throw new NullPointerException();
      return x;
    }

    public static void main(String args[])
    {
      try {
        Class cls = Class.forName("method1");

        Method methlist[]
          = cls.getDeclaredMethods();
        for (int i = 0; i < methlist.length;
          i++) {
          Method m = methlist[i];
          System.out.println("name
            = " + m.getName());
          System.out.println("decl class = " +
                    m.getDeclaringClass());
          Class pvec[] = m.getParameterTypes();
          for (int j = 0; j < pvec.length; j++)
            System.out.println("
              param #" + j + " " + pvec[j]);
          Class evec[] = m.getExceptionTypes();
          for (int j = 0; j < evec.length; j++)
            System.out.println("exc #" + j
              + " " + evec[j]);
          System.out.println("return type = " +
                        m.getReturnType());
          System.out.println("-----");
        }
      }
      catch (Throwable e) {
        System.err.println(e);
      }
    }
  }
```

The program first gets the Class description for method1, and then calls getDeclaredMethods to retrieve a list of Method objects, one for each method defined in the class. These include public, protected, package, and private methods. If you use getMethods in the program instead of getDeclaredMethods, you can also obtain information for inherited methods.
Once a list of the Method objects has been obtained, it's simply a matter of displaying the information on parameter types, exception types, and the return type for each method. Each of these types, whether they are fundamental or class types, is in turn represented by a Class descriptor.

The output of the program is:

```
  name = f1
    decl class = class method1
    param #0 class java.lang.Object
    param #1 int
    exc #0 class java.lang.NullPointerException
    return type = int
    -----
    name = main
    decl class = class method1
    param #0 class [Ljava.lang.String;
    return type = void
    -----
```

### Obtaining Information About Constructors

A similar approach is used to find out about the constructors of a class. For example:

```
  import java.lang.reflect.*;

  public class constructor1 {
    public constructor1()
    {
    }

    protected constructor1(int i, double d)
    {
    }

    public static void main(String args[])
    {
      try {
        Class cls = Class.forName("constructor1");

        Constructor ctorlist[]
          = cls.getDeclaredConstructors();
```

```
      for (int i = 0; i < ctorlist.length; i++) {
          Constructor ct = ctorlist[i];
          System.out.println("name
            = " + ct.getName());
          System.out.println("decl class = " +
                    ct.getDeclaringClass());
          Class pvec[] = ct.getParameterTypes();
          for (int j = 0; j < pvec.length; j++)
            System.out.println("param #"
              + j + " " + pvec[j]);
          Class evec[] = ct.getExceptionTypes();
          for (int j = 0; j < evec.length; j++)
            System.out.println(
              "exc #" + j + " " + evec[j]);
          System.out.println("-----");
        }
       }
       catch (Throwable e) {
         System.err.println(e);
       }
    }
  }
```

There is no return-type information retrieved in this example, because constructors don't really have a true return type.
When this program is run, the output is:

```
name = constructor1
decl class = class constructor1
-----
name = constructor1
decl class = class constructor1
param #0 int
param #1 double
-----
```

### Finding Out About Class Fields

It's also possible to find out which data fields are defined in a class. To do this, the following code can be used:

```
import java.lang.reflect.*;

public class field1 {
  private double d;
  public static final int i = 37;
  String s = "testing";

  public static void main(String args[])
  {
    try {
      Class cls = Class.forName("field1");

      Field fieldlist[]
        = cls.getDeclaredFields();
      for (int i
        = 0; i < fieldlist.length; i++) {
        Field fld = fieldlist[i];
        System.out.println("name
          = " + fld.getName());
        System.out.println("decl class = " +
                  fld.getDeclaringClass());
        System.out.println("type
          = " + fld.getType());
        int mod = fld.getModifiers();
        System.out.println("modifiers = " +
                Modifier.toString(mod));
        System.out.println("-----");
      }
    }
    catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

This example is similar to the previous ones. One new feature is the use of Modifier. This is a reflection class that represents the modifiers found on a field member, for example "private int". The modifiers themselves are represented by an integer, and Modifier.toString is used to return a string representation in the "official" declaration order (such as "static" before "final"). The output of the program is:

```
name = d
decl class = class field1
type = double
modifiers = private
-----
name = i
decl class = class field1
type = int
modifiers = public static final
-----
name = s
decl class = class field1
type = class java.lang.String
modifiers =
-----
```

As with methods, it's possible to obtain information about just the fields declared in a class (getDeclaredFields), or to also get information about fields defined in superclasses (getFields).

### Invoking Methods by Name

So far the examples that have been presented all relate to obtaining class information. But it's also possible to use reflection in other ways, for example to invoke a method of a specified name.

To see how this works, consider the following example:

```
import java.lang.reflect.*;

public class method2 {
```

```
  public int add(int a, int b)
  {
    return a + b;
  }

  public static void main(String args[])
  {
    try {
      Class cls = Class.forName("method2");
      Class partypes[] = new Class[2];
       partypes[0] = Integer.TYPE;
       partypes[1] = Integer.TYPE;
       Method meth = cls.getMethod(
         "add", partypes);
       method2 methobj = new method2();
       Object arglist[] = new Object[2];
       arglist[0] = new Integer(37);
       arglist[1] = new Integer(47);
       Object retobj
         = meth.invoke(methobj, arglist);
       Integer retval = (Integer)retobj;
       System.out.println(retval.intValue());
    }
    catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

Suppose that a program wants to invoke the add method, but doesn't know this until execution time. That is, the name of the method is specified during execution (this might be done by a JavaBeans development environment, for example). The above program shows a way of doing this.
getMethod is used to find a method in the class that has two integer parameter types and that has the appropriate name. Once this method has been found and captured into a Method object, it is invoked upon an object instance of the appropriate type. To invoke a method, a parameter list must be constructed, with the fundamental integer values 37 and 47 wrapped in Integer objects. The return value (84) is also wrapped in an Integer object.

### Creating New Objects

There is no equivalent to method invocation for constructors, because invoking a constructor is equivalent to creating a new object (to be the most precise, creating a new object involves both memory allocation and object construction). So the nearest equivalent to the previous example is to say:

```
import java.lang.reflect.*;

public class constructor2 {
  public constructor2()
  {
  }

  public constructor2(int a, int b)
  {
    System.out.println(
      "a = " + a + " b = " + b);
  }

  public static void main(String args[])
  {
    try {
      Class cls = Class.forName("constructor2");
      Class partypes[] = new Class[2];
       partypes[0] = Integer.TYPE;
       partypes[1] = Integer.TYPE;
       Constructor ct
         = cls.getConstructor(partypes);
       Object arglist[] = new Object[2];
       arglist[0] = new Integer(37);
       arglist[1] = new Integer(47);
       Object retobj = ct.newInstance(arglist);
    }
    catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

which finds a constructor that handles the specified parameter types and invokes it, to create a new instance of the object. The value of this approach is that it's purely dynamic, with constructor lookup and invocation at execution time, rather than at compilation time.

### Changing Values of Fields

Another use of reflection is to change the values of data fields in objects. The value of this is again derived from the dynamic nature of reflection, where a field can be looked up by name in an executing program and then have its value changed. This is illustrated by the following example:

```
import java.lang.reflect.*;

public class field2 {
  public double d;

  public static void main(String args[])
  {
    try {
      Class cls = Class.forName("field2");
      Field fld = cls.getField("d");
      field2 f2obj = new field2();
      System.out.println("d = " + f2obj.d);
      fld.setDouble(f2obj, 12.34);
      System.out.println("d = " + f2obj.d);
    }
    catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

In this example, the d field has its value set to 12.34.

### Using Arrays

One final use of reflection is in creating and manipulating arrays. Arrays in the Java language are a specialized type of class, and an array reference can be assigned to an Object reference.

To see how arrays work, consider the following example:

```
import java.lang.reflect.*;

public class array1 {
   public static void main(String args[])
   {
      try {
         Class cls = Class.forName(
           "java.lang.String");
         Object arr = Array.newInstance(cls, 10);
         Array.set(arr, 5, "this is a test");
         String s = (String)Array.get(arr, 5);
         System.out.println(s);
      }
      catch (Throwable e) {
         System.err.println(e);
      }
   }
}
```

This example creates a 10-long array of Strings, and then sets location 5 in the array to a string value. The value is retrieved and displayed.
A more complex manipulation of arrays is illustrated by the following code:

```
import java.lang.reflect.*;

public class array2 {
   public static void main(String args[])
   {
      int dims[] = new int[]{5, 10, 15};
      Object arr
        = Array.newInstance(Integer.TYPE, dims);

      Object arrobj = Array.get(arr, 3);
      Class cls =
        arrobj.getClass().getComponentType();
      System.out.println(cls);
      arrobj = Array.get(arrobj, 5);
      Array.setInt(arrobj, 10, 37);

      int arrcast[][][] = (int[][][])arr;
      System.out.println(arrcast[3][5][10]);
   }
}
```

This example creates a 5 x 10 x 15 array of ints, and then proceeds to set location [3][5][10] in the array to the value 37. Note here that a multi-dimensional array is actually an array of arrays, so that, for example, after the first Array.get, the result in arrobj is a 10 x 15 array. This is peeled back once again to obtain a 15-long array, and the 10th slot in that array is set using Array.setInt.
Note that the type of array that is created is dynamic, and does not have to be known at compile time.

### Summary

Java reflection is useful because it supports dynamic retrieval of information about classes and data structures by name, and allows for their manipulation within an executing Java program. This feature is extremely powerful and has no equivalent in other conventional languages such as C, C++, Fortran, or Pascal.

*Glen McCluskey has focused on programming languages since 1988. He consults in the areas of Java and C++ performance, testing, and technical documentation.*