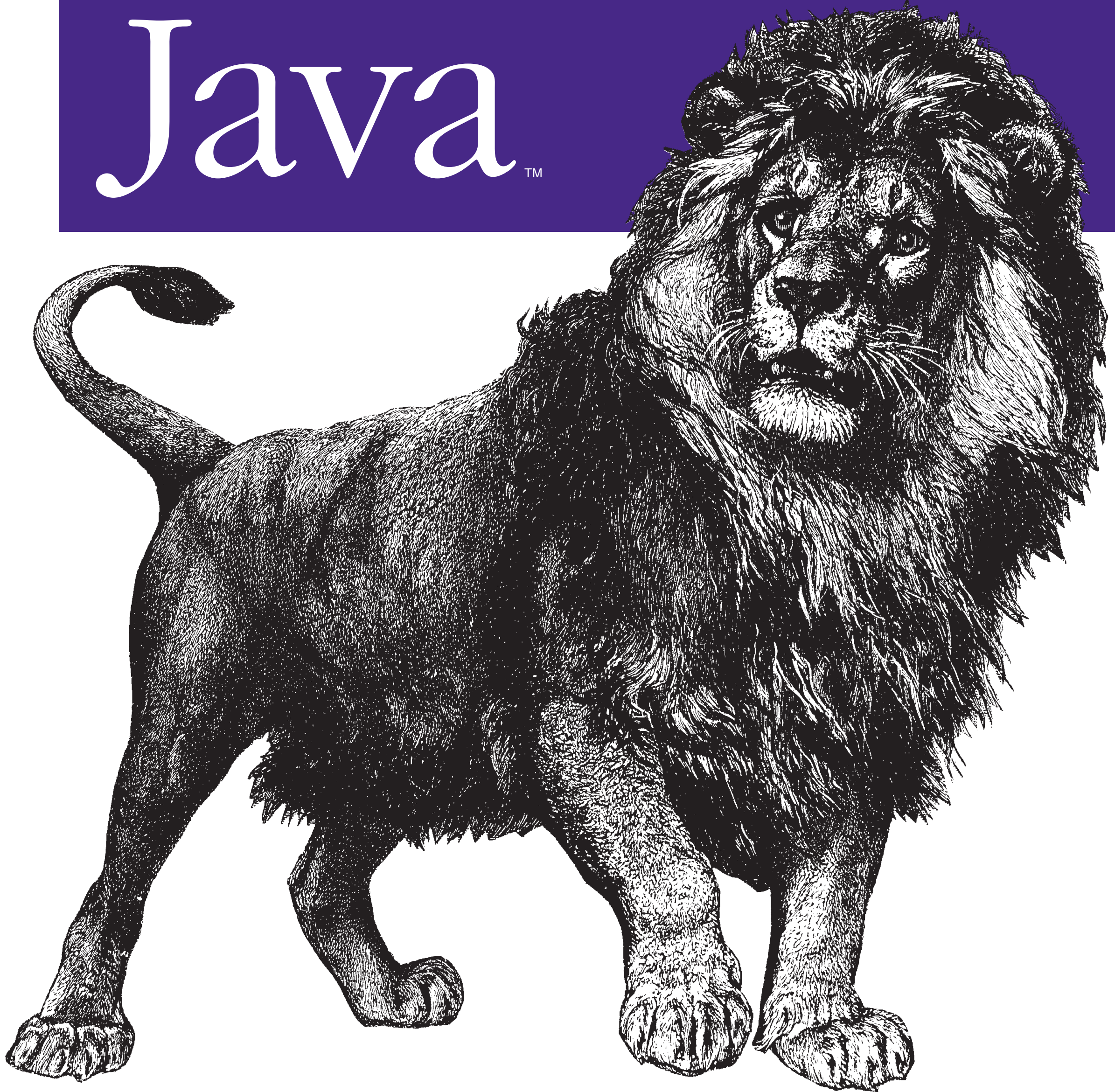
Secrets of the Java Masters

Hardcore Java™



O'REILLY®

Robert Simmons Jr.

CHAPTER 9
Practical Reflection

Reflection is one of the least understood aspects of Java, but also one of the most powerful. Reflection is used in almost every major production product on the market, yet it often has to be learned from developers who are already experts, and these developers are often few and far between.

The reason for using reflection is because of the difficulties in building complex projects in a short time. As your projects become more advanced, you may find that you have less time to finish them.

For example, consider the Online Only Bank data model from Chapter 8. It was relatively small and simple to understand. When designing a GUI to manipulate this data model, you may be tempted to fire up a GUI builder tool and start making forms for each of the objects. In the end, you would have as many forms as there are data model objects. In fact, this is how most GUIs are built today.

However, consider if you used the same technique on a data model with over 200 data model objects and thousands of relationships among the items. In this situation, the idea of using the GUI builder tool is much less appealing—you would need to budget months of man-hours to complete the project. Furthermore, whenever any of these objects changes, even slightly, you would need to go back and fix all of the panels of the affected object. The icing on this particularly bitter cake is that you would have to create new panels each time a new object is introduced into the system. Clearly, this is not an effective method of delivering software in a timely manner.

GUI builders certainly have their uses, such as to prototype applications. However, you should be reluctant to rely on them in production products. Since they must be very general to be useful, the code that drives them is often bulky and inefficient. This is a major reason why Java GUIs are considered slow.

Because of this complexity, the current rage in the Java programming industry is to turn everything into a web application. In fact, many applications that should be GUI-based have been turned into web applications for this very reason. However, even if you decide to go the web application route, the problem of building efficient GUIs remains. Now, instead of having to build GUI panels, you have to build JSP pages for each of your data model objects. Again, the 200 data model objects require much more time to develop than you can spare. Clearly, you need a solution to this problem, and reflection may provide that solution.

Reflection allows you to build tools rather than panels. In fact, you can use reflection to build panels that construct themselves on the fly according to the structure of the object they are displaying. Instead of designing 200 panels, you can design a tool that builds panels and works for any object. Despite the added complexity of the code introduced by using reflection, creating systems with reflection-based tools is significantly faster and much cheaper to maintain.

The Basics

Suppose you are walking down the street and a stranger approaches and hands you an object. What would you do? Naturally, you would look at whatever was handed to you. You inspect the item, and in an instant determine that the object is a piece of paper. Instantly, you know many things about this object: one can write on it; it can have messages such as advertisements, warnings, news, and other useful information printed on it; and you can even crush it into a ball and play basketball when your boss isn't looking.

Though this is a silly example, it does illustrate the basic idea of reflection, which allows you to conduct a similar inspection of Java objects and classes. Using a combination of reflection and introspection, you can determine the nature and possible function of an object that you didn't know about at compile time. Furthermore, you can use this information to execute methods or set field values on the object.

To see how reflection works, let's start with a class from your Online Only Bank data model. In Example 9-1, the `Person` class from the data model is displayed without any comments or content from its methods.

Example 9-1. A Person in a bank data model

```
package oreilly.hcj.bankdata;

public class Person extends MutableObject {
  public static final ObjectConstraint GENDER_CONSTRAINT =
    new ObjectConstraint("gender", false, Gender.class);

  public static final StringConstraint FIRST_NAME_CONSTRAINT =
    new StringConstraint("firstName", false, 20);
```

Example 9-1. A Person in a bank data model (continued)

```
  public static final StringConstraint LAST_NAME_CONSTRAINT =
    new StringConstraint("lastName", false, 40);

  public static final DateConstraint BIRTH_DATE_CONSTRAINT =
    new DateConstraint("birthDate", false, "01/01/1900",
                       "12/31/3000", Locale.US);

  public static final StringConstraint TAX_ID_CONSTRAINT =
    new StringConstraint("taxID", false, 40);

  private Date birthDate = Calendar.getInstance( )
                                 .getTime( );

  private Gender gender = Gender.MALE;

  /** The first name of the person. */
  private String firstName = "<<NEW PERSON>>";

  private String lastName = "<<NEW PERSON>>";

  private String taxID = new String( );

  public void setBirthDate(final Date birthDate) {  }

  public Date getBirthDate( ) {  }

  public void setFirstName(final String firstName) {  }

  public String getFirstName( ) {  }

  public void setGender(final Gender gender) {  }

  public Gender getGender( ) {  }

  public void setLastName(final String lastName) {  }

  public String getLastName( ) {  }

  public void setTaxID(final String taxID) {  }

  public String getTaxID( ) {  }

  public boolean equals(final Object obj) {  }

  public int hashCode( ) {  }
}
```

All of your Online Only Bank data model classes roughly look like this. They are all JavaBeans—that is, they conform to the naming standard that is set for JavaBeans. Since all constructed classes in Java ultimately inherit from `java.lang.Object` (see

Chapter 1), you can pass an instance of this class as an object to a method and then figure out which properties, methods, inner interfaces, etc., the class contains, as Example 9-2 demonstrates.

Example 9-2. Getting method information on an object

```
package oreilly.hcj.reflection;
import java.lang.reflect.Method;
import oreilly.hcj.bankdata.Person;
import oreilly.hcj.bankdata.Gender;

public class MethodInfoDemo {

  public static void printMethodInfo(final Object obj) {
    Class type = obj.getClass( );
    final Method[] methods = type.getMethods( );
    for (int idx = 0; idx < methods.length; idx++) {
      System.out.println(methods[idx]);
    }
  }

  /**
   * Demo method.
   *
   * @param args Command line arguments.
   */
  public static void main(final String[] args) {
    Person p = new Person( );
    p.setFirstName("Robert");
    p.setLastName("Simmons");
    p.setGender(Gender.MALE);
    p.setTaxID("123abc456");

    printMethodInfo(p);
  }
}
```

Inside `printMethodInfo( )`, inspect the object passed to you. After you determine the class of the object, examine contents of the class and print the string version of the methods:

```
>ant -Dexample=oreilly.hcj.reflection.MethodInfoDemo run_example
run_example:
     [java] public int oreilly.hcj.bankdata.Person.hashCode( )
     [java] public boolean oreilly.hcj.bankdata.Person.equals(java.lang.Object)
     [java] public void oreilly.hcj.bankdata.Person.setFirstName(java.lang.String)
     [java] public void oreilly.hcj.bankdata.Person.setLastName(java.lang.String)
     [java] public void oreilly.hcj.bankdata.Person.setGender
            (oreilly.hcj.bankdata.Gender)
     [java] public void oreilly.hcj.bankdata.Person.setTaxID(java.lang.String)
     [java] public void oreilly.hcj.bankdata.Person.setBirthDate(java.util.Date)
```

```
[java] public java.util.Date oreilly.hcj.bankdata.Person.getBirthDate( )
[java] public java.lang.String oreilly.hcj.bankdata.Person.getFirstName( )
[java] public oreilly.hcj.bankdata.Gender
       oreilly.hcj.bankdata.Person.getGender( )
[java] public java.lang.String oreilly.hcj.bankdata.Person.getLastName( )
[java] public java.lang.String oreilly.hcj.bankdata.Person.getTaxID( )
[java] public java.lang.String oreilly.hcj.datamodeling.MutableObject.toString( )
[java] public void oreilly.hcj.datamodeling.MutableObject.
       addPropertyChangeListener
       (java.lang.String,java.beans.PropertyChangeListener)
[java] public void oreilly.hcj.datamodeling.MutableObject.
       addPropertyChangeListener(java.beans.PropertyChangeListener)
[java] public void oreilly.hcj.datamodeling.MutableObject.
       removePropertyChangeListener
       (java.lang.String,java.beans.PropertyChangeListener)
[java] public void oreilly.hcj.datamodeling.MutableObject.
       removePropertyChangeListener(java.beans.PropertyChangeListener)
[java] public final native java.lang.Class java.lang.Object.getClass( )
[java] public final void java.lang.Object.wait(long,int) throws
       java.lang.InterruptedException
[java] public final void java.lang.Object.wait( ) throws
       java.lang.InterruptedException
[java] public final native void java.lang.Object.wait(long) throws
       java.lang.InterruptedException
[java] public final native void java.lang.Object.notify( )
[java] public final native void java.lang.Object.notifyAll( )
```

The `getMethods( )` call in `printMethodInfo( )` returned an array of `Method` objects from the `java.lang.reflect` package; the string version of these `Method` objects was then written to the console. The `getMethods( )` call extracted each of the public members of the `Person` class, including the members it inherited from other classes such as `MutableObject`. The emphasized lines contain the setters and getters for the `Person` class. The nice thing about `printMethodInfo( )` is that it will work for any object you pass to it.

If you want to get only the methods of the class without the inherited methods, you can use the `getDeclaredMethods( )` method defined on `Class`. However, `getDeclaredMethods( )` will also return private and protected members of the class.

In addition to printing out the `Method` objects, you can use them to execute the methods. This is demonstrated in Example 9-3 with a method that will set all string properties in a class as empty strings.

Example 9-3. Dynamically invoking methods

```
package oreilly.hcj.reflection;
public class MethodInfoDemo {
  public static void emptyStrings(final Object obj)
      throws IllegalAccessException, InvocationTargetException {
    final String PREFIX = "set";
```

Example 9-3. Dynamically invoking methods (continued)

```
    Method[] methods = obj.getClass( )
                          .getMethods( );
    for (int idx = 0; idx < methods.length; idx++) {
      if (methods[idx].getName( )
                      .startsWith(PREFIX)) {
        if (methods[idx].getParameterTypes( )[0] == String.class) {
          methods[idx].invoke(obj, new Object[] { new String( ) });
        }
      }
    }
  }
}
```

This method looks for all of the methods in a class that start with the word "set" and take a single `String` as a parameter. It then invokes these methods by passing an array of objects filled with only a single empty string to the method.

The nice thing about the last two methods in the example is that they will work for any object passed to them, not just the `Person` object. If you have complex code in this demonstration class, you could apply it to a wide variety of objects, just like a carpenter applies a saw to many kinds of wood.

Many pieces of professional software expand on this ability to implement exceedingly complex functionality. For example, the Eclipse IDE is based on a plug-in architecture and uses reflection to invoke plugins that the programmers of Eclipse have never seen.

Reflection and Greater Reflection

The tools available to a reflection programmer are not limited to the `java.lang.reflect` package, as you may expect. In fact, they are spread all over the core of the JDK. All the tools used in reflection are often referred to as *greater reflection*. In this section, you will address each of the relevant packages and the tools within those packages and assemble an arsenal of reflection tools that will improve your programming.

Although this book isn't intended as a reference manual explaining how to use the individual methods on the reflection tools, there are several examples in the `oreilly.hcj.reflection` package. These examples demonstrate many of the methods and basic techniques of reflection.

Package java.lang

This package contains the core functionality of reflection as well as the JDK itself. Since all classes ultimately inherit from `java.lang.Object`, the `java.lang` package plays a pivotal role in obtaining information about classes and objects.

Class java.lang.Class

This class contains information about the object's class, such as which fields and methods it contains, and which package it is in. You used it before to obtain information about the methods in a class. See Example 9-2.

Class java.lang.Object

From this class, you can get the `Class` instance for any object. Also, since the `Object` class is the basis for all constructed types in Java, it is used to pass parameters or get results from reflexively invoked methods and fields.

Class java.lang.Package

This class encapsulates the runtime manifestation of a package. However, `Package` is not a particularly useful class because you can't ask it for all of the classes in a package. I rarely use this class in my reflection programming.

Package java.lang.reflect

This package contains utilities for dynamic invocation of methods and for dynamic assignment and retrieval of fields. When you get catalogs of fields and methods from `Class`, they will be returned to you as instances of classes within `java.lang.reflect`. This package is one of the critical tools in building dynamic code, and it is used often in reflection code.

Class java.lang.reflect.Field

This class contains a description of a field and the means to access the field. With this class, you can dynamically set or retrieve field values.

Class java.lang.reflect.Method

This class contains a description of a method and the means to dynamically invoke it. You can use instances of this class to dynamically invoke methods. See Example 9-3.

Class java.lang.reflect.Modifier

This class is used to decode the bit field modifiers of a class, field, or method. Using `Modifier`, you can determine whether a field is `public`, `private`, `static`, `transient`, and so on. The modifiers are set by the native reflection code in the virtual machine that examines the actual Java byte code. These modifiers are read-only.

Class java.lang.reflect.Proxy

This class is used to create dynamic proxy classes, which are then used to create objects that implement specific interfaces. Usually, these classes are used to hide pieces of the implementation of a class from a user. For example, you could create a `Proxy` to the data object that presents an immutable interface to the object.

Class java.lang.reflect.AccessibleObject

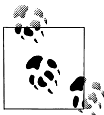
This is the base class from which `Method`, `Field`, and the other descriptors inherit. It allows you to bypass class access constraints such as "private," which allows you to execute methods and set fields that would normally be blocked by visibility specifiers. Serialization often uses this class because it must have access to private fields to serialize an object. In EJB, this class is off-limits.

Although this class can be useful, I caution against using it unless you have no other choice. Forcibly breaking the encapsulation on an object can lead to enormously confusing code that is difficult to debug.

Package java.beans

The `java.beans` package was designed to allow developers to build GUIs using visual tools. However, there are many classes within this package that can be used with reflection. This package has classes that provide convenient access to various parts of reflection that you would otherwise have to search for in several places.

Although this package contains some powerful tools, it is quite old. Most of the classes in this package were introduced long before the collections framework of Java Foundation Classes (JFC) were finalized. Therefore, this package deals with arrays and not with the superior collection classes in the `java.util` package.



This package has much more functionality than is covered in this chapter. Most of the JavaBeans programming paradigm is outside the scope of this book, so I don't delve into the world of customizers and other JavaBeans-specific concepts. Instead, I focus on the aspects of the `java.beans` package that are applicable to reflection as a whole and not just JavaBeans, which is one application of reflection.

Class java.beans.Introspector

This is a singleton class that can be used to obtain property information about a class declared with the JavaBeans naming specification. One important benefit of this class is that it caches the lookup of class information, which significantly speeds up the reflection process.

Although Introspector is useful, it does have some significant drawbacks. First, it does not detect polymorphic properties. Consider the following property:

```
public void setValue(final int value);
public int getValue( );
```

This property is declared using the JavaBeans naming specification. However, it can also be expressed using polymorphic notation:

```
public void value(final int value);
public int value( );
```

This notation uses the context of the method to determine whether the method is a getter or setter. The syntax is much cleaner since a field does not need to be capitalized. These properties are often used in products such as the Simple Widget Toolkit (SWT), which is used to build the Eclipse GUI.

To detect polymorphic properties, you can try to extend `Introspector`. However, if you do this, you encounter the second major disadvantage of this class. The `Introspector` class is implicitly private (see Chapter 2), which means that you cannot extend its functionality. Although it isn't declared as a `final` class, all of its constructors are private instead of protected, which means you can't extend it.

`Introspector` is a good example of why you should use protected methods instead of secured methods, and use private attributes instead of secured attributes. You never know when someone will want to extend your class, so you should give them as many options as possible. Therefore, make a method private only if there is an important reason why it shouldn't be available to subclasses.

Class java.beans.PropertyDescriptor

This class encapsulates the field, setter, and getter of a property all in one location. Essentially, it is a package where you get all of the information about a property in one place instead of having to look it up using `Class`.

Unfortunately, neither this class nor any of its subclasses support collection properties with `add` and `remove` methods. The `PropertyDescriptor` class is used to provide a convenient interface to properties of a class. Most reflection tools use this class often.

One problem with `PropertyDescriptor` is that it allows the user to set the read and write methods for a property manually. Although this appears to be a good idea, it can lead to some confusing results if multiple methods and classes are using the same property descriptor. For example, consider a GUI that views an object that is also being viewed in another panel showing a tree view of the object's structure. If one panel changes the accessor methods, the other panel would access the old method and could be corrupted. This is a good example of an object that should be immutable.

Class java.beans.IndexedPropertyDescriptor

This class is similar to PropertyDescriptor but has extra functionality to handle properties that are arrays (also referred to as indexed properties). Here is a simple example of an indexed property:

```
package oreilly.hcj.reflection;
public class SomeClass {
  private String[] values = new String[0];

  public void setValues(String[] values) {
    this.values = values;
  }

  public void setValues(final String value, final int index) {
    this.values[index] = value;
  }

  public String[] getValues( ) {
    return values;
  }

  public String getValues(final int index) {
    return values[index];
  }
}
```

In this case, the values property is an indexed property because it is an array and has methods to set or get individual values in the array.

The IndexedPropertyDescriptor class will allow you to access the get and set methods of the property and the indexed get and set methods of the property.

Classes MethodDescriptor and ParameterDescriptor

These classes encapsulate information on methods and their parameters. The MethodDescriptor class allows you to retrieve the information about a method in a class, and ParameterDescriptor allows you to get information about the method's parameters. These classes are useful for beans programming because they contain display information about the method and its parameters. However, for reflection, they don't provide any significant functionality above java.lang.reflect.Method. Therefore, these classes are rarely used in mainstream reflection programming.

Interface java.beans.BeanInfo

This class holds all of the information about a particular JavaBean. It provides a convenient place from which you can look up various descriptors and other information about the class. The Introspector is used to look up and cache BeanInfo instances.

However, the one major problem with the BeanInfo class is that it doesn't provide access to the property descriptors by name. Unfortunately, extending this class isn't possible because the Introspector is implicitly private. To get an Introspector to find your extended BeanInfo class, you would either have to define the objects manually on every class (which would defeat the point of reflection) or reimplement the Introspector from the ground up (which is not a small task).

Applying Reflection to MutableObject

Now that you have a thorough understanding of the tools used in reflection, it's time to put them to work to solve a practical problem.

In Chapter 8, we discussed how to implement a data model to encapsulate the data in a hypothetical bank. Most of the data model objects in this model are structurally the same. They differ only in the number, name, and types of properties they contain. Reflection can be used on this data model to implement several key features.

Reflecting on toString( )

In your Online Only Bank data model, there are many classes that encapsulate data, and each has several properties. Although you can compare them and obtain their hashCode( ), you cannot print them to the console. This is the job of the toString( ) method. However, the default toString( ) method defined on class Object will print only the hash code and type of object. This is not enough information to do any serious debugging with these classes. What you really need is for the toString( ) method to print the values of the properties in the mutable object.

One tactic you could implement is to reabstract the toString( ) method and make each class declare its own toString( ) method. This would force developers to implement the method on each class, which would solve the problem. However, the implementations of the toString( ) method on each class would be the same, except for the names of the properties and classes; also, the method would have to be modified manually if new properties are added or removed. It would be better if you could save the users of MutableObject all of this work by implementing a toString( ) method in the MutableObject class that would print the properties of the derived classes. You can do this with reflection (see Example 9-4).

Example 9-4. The toString() method of MutableObject

```
package oreilly.hcj.datamodeling;
public abstract class MutableObject implements Serializable {

public String toString( ) {
    try {
```

Example 9-4. The toString() method of MutableObject (continued)

```
final BeanInfo info = Introspector.getBeanInfo(this.getClass( ),
                                               Object.class);
final PropertyDescriptor[] props = info.getPropertyDescriptors( );
final StringBuffer buf = new StringBuffer(500);
Object value = null;
buf.append(getClass().getName( ));
buf.append("@");
buf.append(hashCode( ));
buf.append("={");
for (int idx = 0; idx < props.length; idx++) {
  if (idx != 0) {
    buf.append(", ");
  }
  buf.append(props[idx].getName( ));
  buf.append("=");
  if (props[idx].getReadMethod( ) != null) {
    value = props[idx].getReadMethod( )
                    .invoke(this, null);
    if (value instanceof MutableObject) {
      buf.append("@");
      buf.append(value.hashCode( ));
    } else if (value instanceof Collection) {
      buf.append("{");
      for (Iterator iter = ((Collection)value).iterator( );
           iter.hasNext( );) {
        Object element = iter.next( );
        if (element instanceof MutableObject) {
          buf.append("@");
          buf.append(element.hashCode( ));
        } else {
          buf.append(element.toString( ));
        }
      }
      buf.append("}");
    } else if (value instanceof Map) {
      buf.append("{");
      Map map = (Map)value;
      for (Iterator iter = map.keySet( )
                         .iterator(); iter.hasNext( );) {
        Object key = iter.next( );
        Object element = map.get(key);
        buf.append(key.toString( ));
        buf.append("=");
        if (element instanceof MutableObject) {
          buf.append("@");
          buf.append(element.hashCode( ));
        } else {
          buf.append(element.toString( ));
        }
      }
```

Example 9-4. The toString() method of MutableObject (continued)

```
          buf.append("}");
        } else {
          buf.append(value);
        }
      }
    }
    buf.append("}");
    return buf.toString( );
  } catch (Exception ex) {
    throw new RuntimeException(ex);
  }
} }
```

Although this method may seem long for such a simple task, it is a fraction of the cut-and-paste code that would be required to implement `toString( )` methods on all of the individual `MutableObject` types in a large data model. Furthermore, the reflexive `toString( )` method doesn't require maintenance if you decide to add a new property to your data model object. It automatically updates itself.

Most of the code in Example 9-4 is fairly mundane string construction. However, the emphasized portions show various uses of the reflection toolkit. First, ask the `Introspector` for the `BeanInfo` on the object and its superclasses up to, but not including, `Object`. Then get the `PropertyDescriptor` objects for that class, which will give you access to the properties. At this point, you can loop through the properties, but make sure you check for a `null read` method, which could happen if the property is write-only.

While looping through the properties, you also have to watch out for recursive reflection. If you used the `toString( )` method to print `MutableObjects` that are members of a `MutableObject`, then you could get caught in an endless loop. For example, if you used `toString( )` on `Account`, the `toString( )` method would be accessed on the `Customer` class; however, since `Customer` has a member holding its `Accounts`, you would recurse back into the `Account` object and end up in a loop. For this reason, you print only the `hashCode( )` of members that are `MutableObjects`. Additionally, since the collection and map properties may also contain `MutableObject` descendants, you have to iterate through them to make sure you don't get caught in a loop.

Once the `toString( )` method is compiled, it will work for any `MutableObject` descendant that you can dream up. Whether you have 4 data model objects or 40,000, you won't have to lift another finger to give your objects the ability to print their contents. Let's try it on some of your objects:

```
package oreilly.hcj.reflection;
public class DemoToStringUsage {

  public static void main(String[] args) {
    // Create some objects.
    Person p = new Person( );
```

```
    p.setFirstName("Robert");
    p.setLastName("Simmons");
    p.setGender(Gender.MALE);
    p.setTaxID("123abc456");

    Customer c = new Customer( );
    c.setPerson(p);
    c.setCustomerID(new Integer(414122));
    c.setEmail("foo@bar.com");

    SavingsAccount a = new SavingsAccount( );
    a.setCustomer(c);
    a.setBalance(new Float(2212.5f));
    a.setID(new Integer(412413789));
    a.setInterestRate(new Float(0.062f));

    Set accounts = new HashSet( );
    accounts.add(a);
    c.setAccounts(accounts);

    // Now print them.
    System.out.println(p);
    System.out.println(c);
    System.out.println(a);
  }
}
```

Using this code, let's examine the new toString( ) method. The resulting output of this code is shown here:

```
>ant -Dexample=oreilly.hcj.reflection.DemoToStringUsage run_example
run_example:
     [java] oreilly.hcj.bankdata.Person@-1989116069={birthDate=Sun Dec 07 18:04:11
CET 2003, firstName=Robert, gender=oreilly.hcj.bankdata.Gender.MALE,
lastName=Simmons, taxID=123abc456}
     [java] oreilly.hcj.bankdata.Customer@414122={accounts={@412413789},
applications={}, cardColor=java.awt.Color[r=0,g=0,b=255], customerID=414122,
email=foo@bar.com, passPhrase=null, payments={}, person=@-1989116069}
     [java] oreilly.hcj.bankdata.SavingsAccount@412413789={ID=412413789,
balance=2212.5, customer=@414122, interestRate=0.062, transactionList={}}
```

Each of the objects is printed correctly with each of their properties without any additional input from the developer of the data model.

Although this is a good demonstration of how reflection is used, it just scratches the surface of its capabilities.

Fetching Constraints

Since the constraints that you put on your data model objects are designed to be accessible to panels, JSP pages, and other classes using these objects, it would be convenient if there was a way to fetch the constraints from the model. Rather than

having the developer specify the constraints at compile time in a data structure, you can use reflection to fetch the constraints. Example 9-5 shows how you can build a map of the constraints on a `MutableObject` descendant. The emphasized lines use reflection.

Example 9-5. Fetching constraints on a MutableObject descendant

```
package oreilly.hcj.datamodeling;
public abstract class MutableObject implements Serializable {
  public static final Map buildConstraintMap(final Class dataType)
    throws IllegalAccessException {
    final int modifiers = Modifier.PUBLIC | Modifier.FINAL |
                          Modifier.STATIC;

    // --
    Map constraintMap = new HashMap( );
    final Field[] fields = dataType.getFields( );
    Object value = null;
    for (int idx = 0; idx < fields.length; idx++) {
      if ((fields[idx].getModifiers( ) & modifiers) == modifiers) {
        value = fields[idx].get(null);
        if (value instanceof ObjectConstraint) {
          constraintMap.put(((ObjectConstraint)value).getName( ), value);
        }
      }
    }
    return Collections.unmodifiableMap(constraintMap);
  }
}
```

Get the fields in the target class and then check each of the public static `final` fields to see whether they contain instances of the class `ObjectConstraint`. If they do, put the constraint in the map and key it with the name of the property it constrains (which you can get from the `ObjectConstraint` class). The `buildConstraintMap( )` method will give you references to the constraints on your objects, which you can use in your GUI or web page form validation.

However, there is one problem with the constraint mechanism. Although reflection is a powerful tool, the process of introspection (analyzing a class to determine its contents) takes up a lot of CPU time. If the users of `MutableObject` have to fetch the constraints in this manner every time they need them, your programs would run slowly. You need to cache your constraints in `MutableObject`. You can create a simple catalog to do this:

```
package oreilly.hcj.datamodeling;
public abstract class MutableObject implements Serializable {
  private static final Map CONSTRAINT_CACHE = new HashMap( );

  public static ObjectConstraint getConstraint(final Class dataType,
                                               final String name) {
```

```
    Map constraintMap = getConstraintMap(dataType);
    return (ObjectConstraint)constraintMap.get(name);
  }

  public static final Map getConstraintMap(final Class dataType)
    throws RuntimeException {
    try {
      Map constraintMap = (Map)CONSTRAINT_CACHE.get(dataType);
      if (constraintMap == null) {
        constraintMap = buildConstraintMap(dataType);
        CONSTRAINT_CACHE.put(dataType, constraintMap);
        return constraintMap;
      }
      Collections.unmodifiableMap(constraintMap)
    } catch (final IllegalAccessException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

This code will cache each of the constraint maps the first time they are fetched. Now all you have to do is change the visibility of `buildConstraintMap( )` to `protected` to force the users to call the caching methods:

```
package oreilly.hcj.reflection;
public class ConstraintMapDemo {
  public static final void main(final String[] args) {
    Map constraints = MutableObject.getConstraintMap(Customer.class);
    Iterator iter = constraints.values( )
                             .iterator( );
    ObjectConstraint constraint = null;
    while (iter.hasNext( )) {
      constraint = (ObjectConstraint)iter.next( );
      System.out.println("Property=" + constraint.getName( ) + " Type="
                         + constraint.getClass().getName( ));
    }

    constraint = MutableObject.getConstraint(SavingsAccount.class,
                                             "interestRate");
    System.out.println("\nSavingsAccount interestRate property");
    System.out.println("dataType = "
                       + ((NumericConstraint)constraint).getDataType( )
                                                         .getName( ));
    System.out.println("minValue = " +
                       ((NumericConstraint)constraint).getMinValue( ));
    System.out.println("maxValue = " +
                       ((NumericConstraint)constraint).getMaxValue( ));
  }
}
```

This results in the following output:

```
>ant -Dexample=oreilly.hcj.reflection.ConstraintMapDemo run_example
run_example:
```

```
[java] Property=accounts
       Type=oreilly.hcj.datamodeling.constraints.CollectionConstraint
[java] Property=cardColor
       Type=oreilly.hcj.datamodeling.constraints.ObjectConstraint
[java] Property=person
       Type=oreilly.hcj.datamodeling.constraints.ObjectConstraint
[java] Property=passPhrase
       Type=oreilly.hcj.datamodeling.constraints.StringConstraint
[java] Property=email Type=oreilly.hcj.datamodeling.constraints.StringConstraint
[java] Property=applications
       Type=oreilly.hcj.datamodeling.constraints.CollectionConstraint
[java] Property=payments
       Type=oreilly.hcj.datamodeling.constraints.CollectionConstraint
[java] Property=customerID
       Type=oreilly.hcj.datamodeling.constraints.NumericConstraint

[java] SavingsAccount interestRate property
[java] dataType = java.lang.Float
[java] minValue = 0.0
[java] maxValue = 1.0
```

As with `toString( )`, adding more data objects, properties, and constraints to your model will not require any more work. The `MutableObject` class adapts to your code.

Performance of Reflection

Our discussion of caching constraint maps leads to the question of reflection's performance. I am frequently asked how expensive reflection is in terms of CPU and memory.

The answer is that reflection can be cheap or expensive depending on how you use it. No tool in any programming language has zero cost. Example 9-6 shows how reflection performs.

Example 9-6. Measuring reflection's performance

```
package oreilly.hcj.reflection;
public class ReflexiveInvocation {
  /** Holds value of property value. */
  private String value = "some value";

  public ReflexiveInvocation( ) {
  }

  public static void main(final String[] args) {
    try {
      final int CALL_AMOUNT = 1000000;
      final ReflexiveInvocation ri = new ReflexiveInvocation( );
      int idx = 0;
```

Example 9-6. Measuring reflection's performance (continued)

```
      // Call the method without using reflection.
      long millis = System.currentTimeMillis( );

      for (idx = 0; idx < CALL_AMOUNT; idx++) {
        ri.getValue( );
      }

      System.out.println("Calling method " + CALL_AMOUNT
                         + " times programatically took "
                         + (System.currentTimeMillis( ) - millis)
                         + " millis");

      // Call while looking up the method at each iteration.
      Method md = null;
      millis = System.currentTimeMillis( );

      for (idx = 0; idx < CALL_AMOUNT; idx++) {
        md = ri.getClass( )
             .getMethod("getValue", null);
        md.invoke(ri, null);
      }

      System.out.println("Calling method " + CALL_AMOUNT
                         + " times reflexively with lookup took "
                         + (System.currentTimeMillis( ) - millis)
                         + " millis");

      // Call using a cache of the method.
      md = ri.getClass( )
           .getMethod("getValue", null);
      millis = System.currentTimeMillis( );

      for (idx = 0; idx < CALL_AMOUNT; idx++) {
        md.invoke(ri, null);
      }

      System.out.println("Calling method " + CALL_AMOUNT
                         + " times reflexively with cache took "
                         + (System.currentTimeMillis( ) - millis)
                         + " millis");
    } catch (final NoSuchMethodException ex) {
      throw new RuntimeException(ex);
    } catch (final InvocationTargetException ex) {
      throw new RuntimeException(ex);
    } catch (final IllegalAccessException ex) {
      throw new RuntimeException(ex);
    }
  }
```

Example 9-6. Measuring reflection's performance (continued)

```
  public String getValue( ) {
    return this.value;
  }
}
```

In the first part of this test, you measure the time it takes to call the getter of the class normally 1 million times. In the second phase, you measure the time it takes to call the getter 1 million times using a reflection-based invocation, in which the getter method is looked up each time the loop is iterated. Finally, you cache the lookup of the getter and invoke the getter method 1 million times using the cached `Method` object. The results of running this class are shown here:

```
>ant -Dexample=oreilly.hcj.reflection.ReflexiveInvocation run_example
run_example:
     [java] Calling method 1000000 times normally took 20 millis
     [java] Calling method 1000000 times reflexively with lookup took 5618 millis
     [java] Calling method 1000000 times reflexively with cache took 270 millis
```

Your times may vary depending on the capabilities of your computer, but generally, the ratios will be the same.

The phase during which the `Method` object was cached was significantly faster than the phase during which the `Method` object was looked up at each iteration. Naturally, both phases were slower than invoking the method normally.

This test teaches a couple of lessons. First, looking up the `Method` object is the most expensive part of reflection. If you cache the method, you can dramatically improve performance. Second, reflection is slower than invoking methods normally. However, this performance loss is compensated with the benefit of producing your products significantly faster and with fewer bugs. In most cases, the benefits of reflection are well worth the cost of 0.25 seconds over 1 million calls.

Reflection + JUnit = Stable Code

When designing architecture code, unit testing can be an important process. With unit testing, units of the code are tested in a controlled environment before they are incorporated into the entire system. Unit testing allows developers to catch bugs before the code is integrated with several other classes. Also, creating reusable unit tests with frameworks such as JUnit allows the programmer to perform a regression test when other features are implemented; they simply need to run the test to verify that nothing was broken. The most important and common question surrounding unit testing is how much of the code should be tested.

Ideally, you should test everything in the product. Unfortunately, this is often not possible because of deadlines and other pressures. You could easily spend more time writing tests than code. However, all architecture code should be tested because of the weight of the system that will be resting on that code.

You may believe that testing is not worth the effort; after all, these objects are only beans that hold data and not business processes. However, reflection allows you to have your cake and eat it too.

With reflection, you can test a data model object regardless of which properties it declares:

```
package oreilly.hcj.reflection;
public class TestMutableObject extends TestCase {
  private Class type = null;

  protected TestMutableObject(final Class type, final String testName) {
    super(testName);
    assert (type != null);
    assert (MutableObject.class.isAssignableFrom(type));
    this.type = type;
  }

  public static Test suite(final Class type) {
    if (type == null) {
      throw new NullPointerException("type");
    }
    if (!MutableObject.class.isAssignableFrom(type)) {
      throw new IllegalArgumentException("type");
    }

    TestSuite suite = new TestSuite(type.getName( ));
    suite.addTest(new TestMutableObject(type, "testConstraintsExist"));
    return suite;
  }

  public void testConstraintsExist( ) {
    try {
      final PropertyDescriptor[] props =
        Introspector.getBeanInfo(type, Object.class)
                    .getPropertyDescriptors( );

      for (int idx = 0; idx < props.length; idx++) {
        ObjectConstraint constraint =
          MutableObject.getConstraint(type, props[idx].getName( ));
        assertNotNull("Property " + props[idx].getName( )
                     + " does not have a constraint.", constraint);
      }
    } catch (final IntrospectionException ex) {
      throw new RuntimeException( );
    }
  }
}
```

This class is a standard JUnit test case with a twist. It does not know what kind of class it will be testing until the suite of test cases is created with the `suite( )` method. The user of the test case passes the class that he wants to test into the `suite( )` method, and the test is built for that particular class:

```
package oreilly.hcj.reflection;
public class BankDataTests {
  public static Test suite( ) {
    TestSuite suite = new TestSuite("Online-Only Bank Datamodel Tests");
    suite.addTest(TestMutableObject.suite(Account.class));
    suite.addTest(TestMutableObject.suite(AssetAccount.class));
    suite.addTest(TestMutableObject.suite(AutomaticPayment.class));
    suite.addTest(TestMutableObject.suite(BankOfficer.class));
    suite.addTest(TestMutableObject.suite(CreditCardAccount.class));
    suite.addTest(TestMutableObject.suite(Customer.class));
    suite.addTest(TestMutableObject.suite(Employee.class));
    suite.addTest(TestMutableObject.suite(ExternalTransaction.class));
    suite.addTest(TestMutableObject.suite(LiabilityAccount.class));
    suite.addTest(TestMutableObject.suite(LoanAccount.class));
    suite.addTest(TestMutableObject.suite(LoanApplication.class));
    suite.addTest(TestMutableObject.suite(OnlineCheckingAccount.class));
    suite.addTest(TestMutableObject.suite(Person.class));
    suite.addTest(TestMutableObject.suite(SavingsAccount.class));
    suite.addTest(TestMutableObject.suite(Transaction.class));
    return suite;
  }

  public static void main(final String[] args) {
    TestRunner.run(BankDataTests.class);
  }
}
```

You can run this test using the `ant` script:

```
>ant -Dexample=oreilly.hcj.reflection.BankDataTests run_example
```

This will bring up the JUnit test window and run each of the tests. On your healthy data model, the results will look like Figure 9-1.

However, if you comment out the constraint on the `balance` property of `Account`, your test case will acquire the defect in every class that inherits from `Account` (see Figure 9-2).

The `Account` class, as well as classes that inherit from `Account`, shows an error indicating that there isn't a constraint on the balance property.

Unit testing can be expanded to test almost all facets of your data model to make sure it conforms to the rules of your design specification. Also, you can use unit testing to write tests for other objects. With this technique, you can implement tests in one location and apply them to many classes.

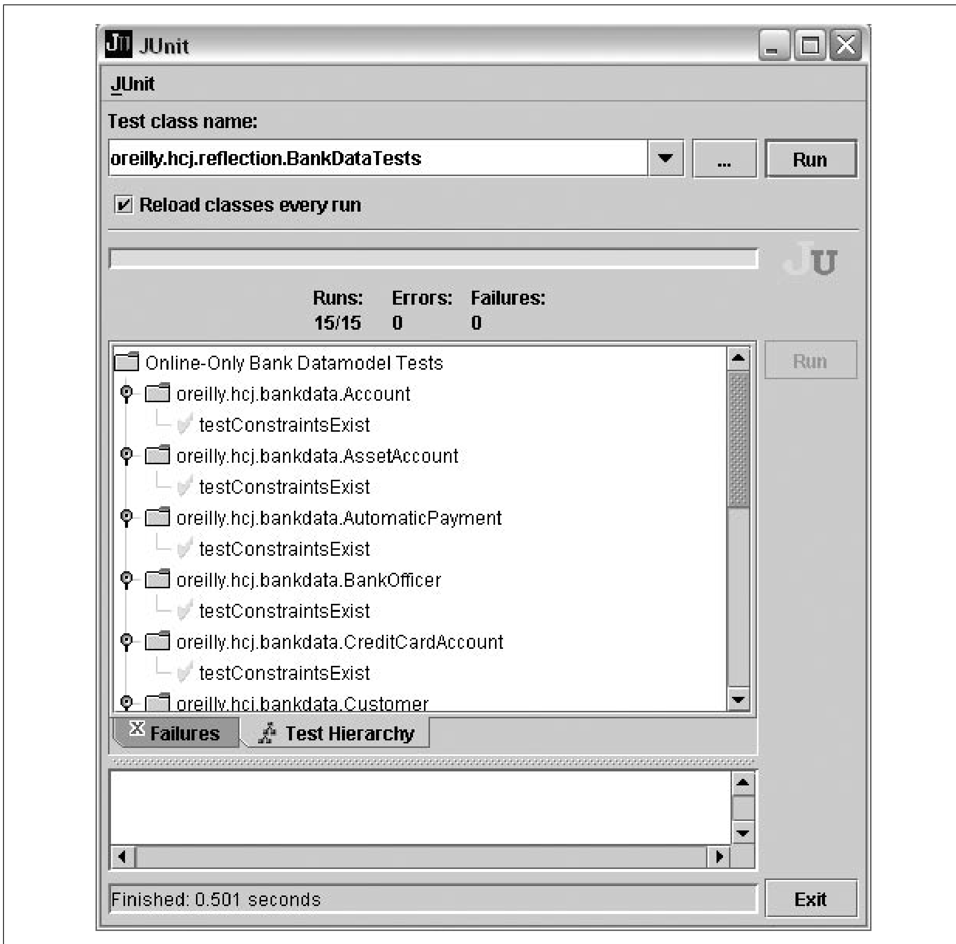


Figure 9-1. Testing the constraints in a healthy data model

Reflection can be used for a wide variety of other problems as well. This chapter has touched on only a fraction of the large number of things you can do with reflection. As you expand your reflection skills, you will find that you are designing more and more tools and copying less and less code. In the end, your code will be more solid and easier to maintain.

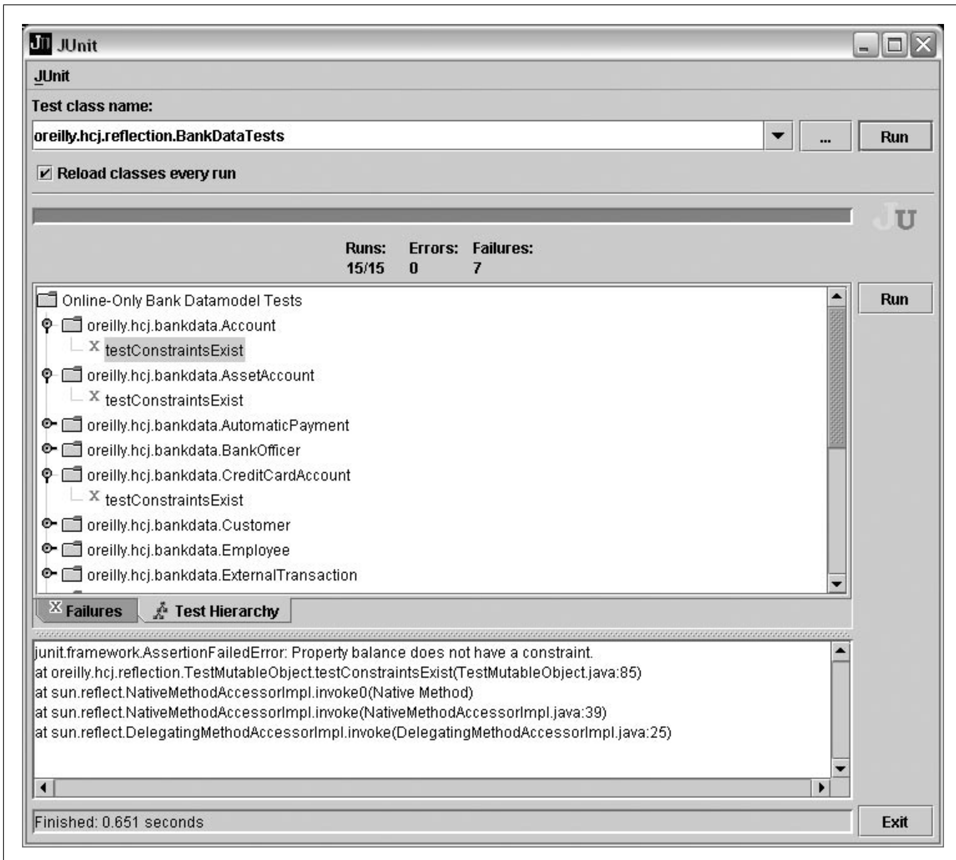


Figure 9-2. Testing an erroneous data model