Sample Chapter

JAVA Reflection IN ACTION

Ira R. Forman
Nate Forman

MANNING

Java Reflection in Action

by Ira R. Forman

and

Nate Forman

Sample Chapter 1

contents

1 A few basics

In this chapter

- Reflection basics
- Class fundamentals
- Using methods reflectively

We are often faced with problems that could be solved simply and elegantly with reflection. Without it, our solutions are messy, cumbersome, and fragile. Consider the following scenarios:

- Your project manager is committed to a pluggable framework, knowing that the system needs to accept new components even after it is built and deployed. You set up some interfaces and prepare a mechanism for patching your JAR, but you know that this will not completely satisfy the need for pluggability.
- After months of developing a client-side application, marketing tells you that using a different remote mechanism will increase sales. Although switching is a good business decision, you now must reimplement all of your remote interfaces.
- The public API to your module needs to accept calls only from specific packages to keep outsiders from misusing your module. You add a parameter to each of the API calls that will hold the package name of the calling class. But, now legitimate users must change their calls, and unwelcome code can fake a package name.

These scenarios illustrate, in turn, modularity, remote access, and security—and do not seem to have much in common. But they do: each one contains a change in requirements that can be satisfied only by making decisions and modifying code based upon the structure of the program.

Reimplementing interfaces, patching JAR files, and modifying method calls are all tedious and mechanical tasks. So mechanical, in fact, that you could write an algorithm that describes the necessary steps:

1. Examine the program for its structure or data.
2. Make decisions using the results of the examination.
3. Change the behavior, structure, or data of the program based upon the decisions.

While these steps may be familiar to you in your role as programmer, they are not tasks that you would imagine a program doing. As a result, you assume that adapting code must be accomplished by a person sitting at a keyboard instead of by a program running on a computer. Learning reflection allows you to get beyond this assumption and make your program do this adaptation for you. Consider the following simple example:

```
public class HelloWorld {
    public void printName() {
        System.out.println(this.getClass().getName());
    }
}
```

The line

```
(new HelloWorld()).printName();
```

sends the string HelloWorld to standard out. Now let `x` be an instance of `HelloWorld` or one of its subclasses. The line

```
x.printName();
```

sends the string naming the class to standard out.

This small example is more dramatic than it seems—it contains each of the steps previously mentioned. The `printName` method examines the object for its class (`this.getClass()`). In doing so, the decision of what to print is made by delegating to the object's class. The method acts on this decision by printing the returned name. Without being overridden, the `printName` method behaves differently for each subclass than it does for `HelloWorld`. The `printName` method is flexible; it adapts to the class that inherits it, causing the change in behavior. As we build our examples in scope and complexity, we will show you many more ways to attain flexibility using reflection.

1.1 Reflection's value proposition

Reflection is the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds.

To perform this self-examination, a program needs to have a representation of itself. This information we call **metadata**. In an object-oriented world, metadata is organized into objects, called **metaobjects**. The runtime self-examination of the metaobjects is called **introspection**.

As we saw in the small example above, the introspection step is followed by behavior change. In general, there are three techniques that a reflection API can use to facilitate behavior change: direct metaobject modification, operations for using metadata (such as dynamic method invocation), and **intercession**, in which code is permitted to intercede in various phases of program execution. Java supplies a rich set of operations for using metadata and just a few important intercession capabilities. In addition, Java avoids many complications by not allowing direct metaobject modification.

These features give reflection the power to make your software flexible. Applications programmed with reflection adapt more easily to changing requirements. Reflective components are more likely to be reused flawlessly in other applications. These benefits are available in your current Java development kit.

Reflection is powerful, but it is not magical. You must master the subject in order to make your software flexible. It's not enough to just learn the concepts and the use of the API. You must also be able to distinguish between situations when reflection is absolutely required from those when it may be used advantageously from those when it should be shunned. The examples in this book will help you acquire this skill. In addition, by the time you reach the end, you will understand the three issues that have thus far impeded the broad use of reflection:

- security
- code complexity
- runtime performance

You will learn that the concern over security was misguided. Java is so well crafted and its reflection API so carefully constrained that security is controlled simply. By learning when to use reflection and when not to, you will avoid unnecessarily complex code that can often be the result of amateurish use of reflection. In addition, you will learn to evaluate the performance of your designs, thereby ensuring the resulting code satisfies its performance requirements.

This introduction describes reflection, but scarcely reveals its value. Software maintenance costs run three to four or more times development costs. The software marketplace is increasing its demand for flexibility. Knowing how to produce flexible code increases your value in the marketplace. Reflection—introspection followed by behavior change—is the path to flexible software. The promise of reflection is great and its time has come. Let's begin.

1.2 Enter George the programmer

George is a programmer at Wildlife Components, a leading animal simulation software company. In his daily work, George faces many challenges such as the ones previously mentioned. Throughout this book, we will follow George as he discovers the benefits of implementing reflective solutions.

For one project, George is working on a team that is implementing a user interface. George's team uses several standard Java visual components, others that are developed in house, a few that are open source, and still others that have been licensed from third parties. All of these components are integrated to form the user interface for the team's application.

Each of these components provides a `setColor` method that takes a `java.awt.Color` parameter. However, the hierarchies are set up such that the only common base class for all of them is `java.lang.Object`. These components cannot be referenced using a common type that supports this `setColor` method.

This situation presents a problem for George's team. They just want to call `setColor` regardless of a component's concrete type. The lack of a common type that declares `setColor` means more work for the team. In case this scenario seems contrived, we invite you to explore the JDK API and see the number of classes that support the same method but implement no common interface.

1.2.1 Choosing reflection

Given a component, the team's code must accomplish two steps:

1. Discover a `setColor` method supported by the component.
2. Call that `setColor` method with the desired color.

There are many alternatives for accomplishing these steps manually. Let's examine the results of each of these.

If George's team controlled all of the source code, the components could be refactored to implement a common interface that declares `setColor`. Then, each component could be referenced by that interface type and `setColor` could be invoked without knowing the concrete type. However, the team does not control the standard Java components or third-party components. Even if they changed the open source components, the open source project might not accept the change, leaving the team with additional maintenance.

Alternatively, the team could implement an adapter for each component. Each such adapter could implement a common interface and delegate the `setColor` call to the concrete component. However, because of the large number of component classes that the team is using, the solution would cause an explosion in the number of classes to maintain. In addition, because of the large number of component instances, this solution would cause an explosion of the number of objects in the system at runtime. These trade-offs make implementing an adapter an undesirable option.

Using `instanceof` and casting to discover concrete types at runtime is another alternative, but it leaves several maintenance problems for George's team. First, the code would become bloated with conditionals and casts, making it difficult to read and understand. Second, the code would become coupled with each concrete type. This coupling would make it more difficult for the team to add, remove, or change components. These problems make `instanceof` and casting an unfavorable alternative.

Each of these alternatives involves program changes that adjust or discover the type of a component. George understands that it is only necessary to find a `setColor` method and call it. Having studied a little reflection, he understands how to query an object's class for a method at runtime. Once it is found, he knows that a method can also be invoked using reflection. Reflection is uniquely suited to solving this problem because it does not over-constrain the solution with type information.

1.2.2 Programming a reflective solution

To solve his team's problem, George writes the static utility method `setObjectColor` in listing 1.1. George's team can pass a visual component to this utility method along with a color. This method finds the `setColor` method supported by the object's class and calls it with the color as an argument.

Listing 1.1 George's `setObjectColor` code

```
public static void setObjectColor( Object obj, Color color )   {
    Class cls = obj.getClass();                                  (1) Query object for its class

    try {
        Method method = cls.getMethod( "setColor",               (2) Query class object for setColor method
                                       new Class[] {Color.class} );
        method.invoke( obj, new Object[] {color} );              (3) Call resulting method on target obj
    }

    catch (NoSuchMethodException ex) {                           (4) Class of obj does not support setColor method
        throw new IllegalArgumentException(
                    cls.getName()
                    + " does not support method setColor(Color)" );
    }

    catch (IllegalAccessException ex) {                          (5) Invoker cannot call setColor method
        throw new IllegalArgumentException(
                    "Insufficient access permissions to call"
                    + "setColor(:Color) in class " + cls.getName());
    }

    catch (InvocationTargetException ex) {                       (6) setColor method throws an exception
        throw new RuntimeException(ex);
     }
}
```

This utility method satisfies the team's goal of being able to set a component's color without knowing its concrete type. The method accomplishes its goals without invading the source code of any of the components. It also avoids source code bloating, memory bloating, and unnecessary coupling. George has implemented an extremely flexible and effective solution.

Two lines in listing 1.1 use reflection to examine the structure of the parameter `obj`:

❶ This line of code queries the object for its class.

❷ This line queries the class for a `setColor` method that takes a `Color` argument.

In combination, these two lines accomplish the first task of finding a `setColor` method to call.

These queries are each a form of **introspection**, a term for reflective features that allow a program to examine itself. We say that `setObjectColor` *introspects* on its parameter, `obj`. There is a corresponding form of introspection for each feature of a class. We will examine each of these forms of introspection over the next few chapters.

One line in listing 1.1 actually affects the behavior of the program:

❸ *This line calls the resulting method on `obj`, passing it the color*—This reflective method call can also be referred to as dynamic invocation. **Dynamic invocation** is a feature that enables a program to call a method on an object at runtime without specifying which method at compile time.

In the example, George does not know which `setColor` method to call when writing the code because he does not know the type of the `obj` parameter. George's program discovers which `setColor` method is available at runtime through introspection. Dynamic invocation enables George's program to act upon the information gained through introspection and make the solution work. Other reflective mechanisms for affecting program behavior will be covered throughout the rest of the book.

Not every class supports a `setColor` method. With a static call to `setColor`, the compiler reports an error if the object's class does not support `setColor`. When using introspection, it is not known until runtime whether or not a `setColor` method is supported:

❹ *The class of `obj` does not support a `setColor` method*—It is important for introspective code to handle this exceptional case. George has been guaranteed by his team that each visual component supports `setColor`. If that method

is not supported by the type of the `obj` parameter, his utility method has been passed an illegal argument. He handles this by having `setObjectColor` throw an `IllegalArgumentException`.

The `setObjectColor` utility method may not have access to nonpublic `setColor` methods. In addition, during the dynamic invocation, the `setColor` method may throw an exception:

❺ The class containing listing 1.1 does not have access privileges to call a `protected`, `package`, or `private` visibility `setColor` method.

❻ The invoked `setColor` method throws an exception.

It is important for methods using dynamic invocation to handle these cases properly. For simplicity's sake, the code in listing 1.1 handles these exceptions by wrapping them in runtime exceptions. For production code, of course, this would be wrapped in an exception that the team agrees on and declared in the utility method's `throws` clause.

All of this runtime processing also takes more time than casts and static invocation. The method calls for introspection are not necessary if the information is known at compile time. Dynamic invocation introduces latency by resolving which method to call and checking access at runtime rather than at compile time. Chapter 9 discusses analysis techniques for balancing performance trade-offs with the tremendous flexibility benefits that reflection can give you.

The rest of this chapter focuses on the concepts necessary to fully understand listing 1.1. We examine, in detail, the classes that George uses to make it work. We also discuss the elements supported by Java that allow George such a flexible solution.

1.3 Examining running programs

Reflection is a program's ability to examine and change its behavior and structure at runtime. The scenarios previously mentioned have already implied that reflection gives programmers some pretty impressive benefits. Let's take a closer look at what reflective abilities mean for the structure of Java.

Think of introspection as looking at yourself in a mirror. The mirror provides you with a representation of yourself—your *reflection*—to examine. Examining yourself in a mirror gives you all sorts of useful information, such as what shirt goes with your brown pants or whether you have something green stuck in your teeth. That information can be invaluable in adjusting the structure of your wardrobe and hygiene.

A mirror can also tell you things about your behavior. You can examine whether a smile looks sincere or whether a gesture looks too exaggerated. This information can be critical to understanding how to adjust your behavior to make the right impression on other people.

Similarly, in order to introspect, a program must have access to a representation of itself. This self-representation is the most important structural element of a reflective system. By examining its self-representation, a program can obtain the right information about its structure and behavior to make important decisions.

Listing 1.1 uses instances of `Class` and `Method` to find the appropriate `setColor` method to invoke. These objects are part of Java's self-representation. We refer to objects that are part of a program's self-representation as **metaobjects**. *Meta* is a prefix that usually means *about* or *beyond*. In this case, metaobjects are objects that hold information about the program.

`Class` and `Method` are classes whose instances represent the program. We refer to these as *classes of metaobjects* or *metaobject classes*. Metaobject classes are most of what make up Java's reflection API.

We refer to **objects** that are used to accomplish the main purposes of an application as **base-level objects**. In the `setObjectColor` example above, the application that calls George's method as well as the objects passed to it as parameters are base-level objects. We refer to the nonreflective parts of a program as the **base program**.

Metaobjects represent parts of the running application, and, therefore, may describe the base program. Figure 1.1 shows the `instanceof` relationship between base-level objects and the objects that represent their classes. The diagramming convention used for figure 1.1 is the Unified Modeling Language (UML). For readers unfamiliar with UML, we will describe the conventions briefly in section 1.7. For the moment, it is important to understand that the figure can be read as "*fido, a base-level object, is an instance of Dog, a class object on the metalevel.*"

Metaobjects are a convenient self-representation for reflective programming. Imagine the difficulty that George would have in accomplishing his task if he had tried to use the source code or the bytecodes as a representation. He would have to parse the program to even begin examining the class for its methods. Instead, Java metaobjects provide all of the information he needs without additional parsing.

Metaobjects often also provide ways of changing program structure, behavior, or data. In our example, George uses dynamic invocation to call a method that he finds through introspection. Other reflective abilities that make changes include reflective construction, dynamic loading, and intercepting method calls. This book shows how to use these mechanisms and others to solve common but difficult software problems.

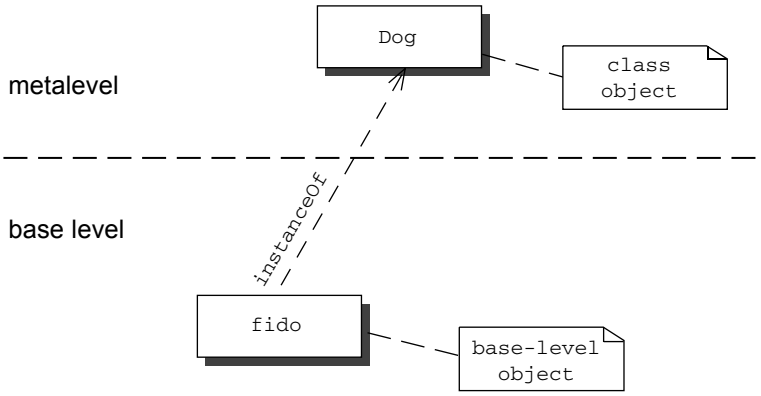

Figure 1.1 `Dog` **is a class object, a metaobject that represents the class Dog. The object** `fido` **is an instance of** `Dog` **operating within the application. The** `instanceof` **relationship, represented in this diagram by a dependency, connects objects on the base level to an object that represents their class on the metalevel.**

1.4 Finding a method at runtime

At the beginning of our example, George's `setObjectColor` method is passed a parameter `obj` of type `Object`. The method cannot do any introspection until it knows the class of that parameter. Therefore, its first step is to query for the parameter's class:

```
Class cls = obj.getClass();
```

The `getClass` method is used to access an object's class at runtime. The `getClass` method is often used to begin reflective programming because many reflective tasks require objects representing classes. The `getClass` method is introduced by `java.lang.Object`, so any object in Java can be queried for its class[1].

The `getClass` method returns an instance of `java.lang.Class`. Instances of `Class` are the metaobjects that Java uses to represent the classes that make up a program. Throughout this book, we use the term **class object** to mean an instance of `java.lang.Class`. Class objects are the most important kind of metaobject because all Java programs consist solely of classes.

[1] The `getClass` method is final. This keeps Java programmers from fooling reflective programs. If it were not final, a programmer could override `getClass` to return the wrong class.

Class objects provide programming metadata about a class's fields, methods, constructors, and nested classes. Class objects also provide information about the inheritance hierarchy and provide access to reflective facilities. For this chapter, we will concentrate on the use of `Class` in listing 1.1 and related fundamentals.

Once the `setObjectColor` method has discovered the class of its parameter, it queries that class for the method it wants to call:

```
Method method = cls.getMethod("setColor", new Class[] {Color.class});
```

The first parameter to this query is a `String` containing the desired method's name, in this case, `setColor`. The second parameter is an array of class objects that identify the types of the method's parameters. In this case, we want a method that accepts one parameter of type `Color`, so we pass `getMethod` an array of one element containing the class object for `Color`.

Notice that the assignment does not use `getClass` to provide the class object for Color. The `getClass` method is useful for obtaining the class for an object reference, but when we know only the name of the class, we need another way. **Class literals** are Java's way to specify a class object statically. Syntactically, any class name followed by `.class` evaluates to a class object. In the example, George knows that `setObjectColor` always wants a method that takes one `Color` argument. He specifies this using `Color.class`.

`Class` has other methods for introspecting about methods. The signatures and return types for these methods are shown in table 1.1. As in the previous example, the queries use an array of `Class` to indicate the types of the parameters. In

Table 1.1 The methods defined by `Class` for method query

Method	Description
Method **getMethod**(String name, Class[] parameterTypes)	Returns a `Method` object that represents a public method (either declared or inherited) of the target `Class` object with the signature specified by the second parameters
Method[] **getMethods**()	Returns an array of `Method` objects that represent all of the public methods (either declared or inherited) supported by the target `Class` object
Method **getDeclaredMethod**(String name, Class[] parameterTypes)	Returns a `Method` object that represents a declared method of the target `Class` object with the signature specified by the second parameters
Method[] **getDeclaredMethods**()	Returns an array of `Method` objects that represent all of the methods declared by the target `Class` object

querying for a parameterless method, it is legal to supply `null`, which is treated the same as a zero-length array.

As their names indicate, `getDeclaredMethod` and `getDeclaredMethods` return method objects for methods explicitly declared by a class. The set of declared methods does not include methods that the class inherits. However, these two queries do return methods of all visibilities—public, protected, package, and private.

The queries `getMethod` and `getMethods` return method objects for a class's public methods. The set of methods covered by these two includes both methods declared by the class and those it inherits from superclasses. However, these queries return only a class's public methods.

A programmer querying a class using `getDeclaredMethod` might accidentally specify a method that the class does not declare. In this case, the query fails with a `NoSuchMethodException`. The same exception is thrown when `getMethod` fails to find a method among a class's public methods.

In the example, George needs to find a method, and he does so using one of the methods from table 1.1. Once retrieved, these method objects are used to access information about methods and even call them. We discuss method objects in detail later in this chapter, but first let's take a closer look at how class objects are used with the methods from table 1.1.

1.5 Representing types with class objects

The discussion of the methods from table 1.1 indicates that Java reflection uses instances of `Class` to represent types. For example, `getMethod` from listing 1.1 uses an array of `Class` to indicate the types of the parameters of the desired method. This seems fine for methods that take objects as parameters, but what about types not created by a class declaration?

Consider listing 1.2, which shows a fragment of `java.util.Vector`. One method has an interface type as a parameter, another an array, and the third a primitive. To program effectively with reflection, you must know how to introspect on classes such as `Vector` that have methods with such parameters.

Listing 1.2 A fragment of `java.util.Vector`

```
public class Vector ... {
   public synchronized boolean    addAll( Collection c ) ...
   public synchronized void       copyInto( Object[] anArray ) ...
   public synchronized Object     get( int index ) ...
}
```

Table 1.2 Methods defined by `Class` that deal with type representation

Method	Description
String **getName**()	Returns the fully qualified name of the target `Class` object
Class **getComponentType**()	If the target object is a `Class` object for an array, returns the `Class` object representing the component type
boolean **isArray**()	Returns `true` if and only if the target `Class` object represents an array
boolean **isInterface**()	Returns `true` if and only if the target `Class` object represents an interface
boolean **isPrimitive**()	Returns `true` if and only if the target `Class` object represents a primitive type or `void`

Java represents primitive, array, and interface types by introducing class objects to represent them. These class objects cannot do everything that many other class objects can. For instance, you cannot create a new instance of a primitive or interface. However, such class objects are necessary for performing introspection. Table 1.2 shows the methods of `Class` that support type representation.

The rest of this section explains in greater detail how Java represents primitive, interface, and array types using class objects. By the end of this section, you should know how to use methods such as `getMethod` to introspect on `Vector.class` for the methods shown in listing 1.2.

1.5.1 Representing primitive types

Although primitives are not objects at all, Java uses class objects to represent all eight primitive types. These class objects can be indicated using a class literal when calling methods such as those in table 1.1. For example, to specify type `int`, use `int.class`. Querying the `Vector` class for its `get` method can be accomplished with

```
Method m = Vector.class.getMethod("get", new Class[] {int.class});
```

A class object that represents a primitive type can be identified using `isPrimitive`.

The keyword `void` is not a type in Java; it is used to indicate a method that does not return a value. However, Java does have a class object to represent `void`. The `isPrimitive` method returns true for `void.class`. In section 1.6, we cover introspection on methods. When introspecting for the return type of a method, `void.class` is used to indicate that a method returns no value.

1.5.2 Representing interfaces

Java also introduces a class object to represent each declared interface. These class objects can be used to indicate parameters of interface type. The `addAll`

method of `Vector` takes an implementation of the `Collection` interface as an argument. Querying the `Vector` class for its `addAll` method can be written as

```
Method m = Vector.class.getMethod( "addAll",
                                   new Class[] {Collection.class} );
```

A class object that represents an interface may be queried for the methods and constants supported by that interface. The `isInterface` method of `Class` can be used to identify class objects that represent interfaces.

1.5.3 Representing array types

Java arrays are objects, but their classes are created by the JVM at runtime. A new class is created for each element type and dimension. Java array classes implement both `Cloneable` and `java.io.Serializable`.

Class literals for arrays are specified like any other class literal. For instance, to specify a parameter of a single-dimension `Object` array, use the class literal `Object[].class`. A query of the `Vector` class for its `copyInto` method is written as

```
Method m = Vector.class.getMethod( "copyInto", new Class[]{Object[].class} );
```

Class objects that represent arrays can be identified using the `isArray` method of `Class`. The component type for an array class can be obtained using `getComponentType`. Java treats multidimensional arrays like nested single-dimension arrays. Therefore, the line

```
int[][].class.getComponentType()
```

evaluates to `int[].class`. Note the distinction between component type and element type. For the array type `int[][]`, the component type is `int[]` while the element type is `int`.

Not all Java methods take non-interface, non-array object parameters like `setColor` from our George example. In many cases, it is important to introspect for methods such as the `Vector` methods of listing 1.2. Now that you understand how to introspect for any Java method, let's examine what can be done once a method is retrieved.

1.6 Understanding method objects

Most of the examples over the last few sections have used the identifier `Method` but not explained it. `Method` is the type of the result of all of the method queries in table 1.1. George uses this class in listing 1.1 to invoke `setColor`. From this context, it should be no surprise that `java.lang.reflect.Method` is the class of the

Table 1.3 Methods defined by `Method`

Method	Description
Class **getDeclaringClass**()	Returns the `Class` object that declared the method represented by this `Method` object
Class[] **getExceptionTypes**()	Returns an array of `Class` objects representing the types of the exceptions declared to be thrown by the method represented by this `Method` object
int **getModifiers**()	Returns the modifiers for the method represented by this `Method` object encoded as an `int`
String **getName**()	Returns the name of the method represented by this `Method` object
Class[] **getParameterTypes**()	Returns an array of `Class` objects representing the formal parameters in the order in which they were declared
Class **getReturnType**()	Returns the `Class` object representing the type returned by the method represented by this `Method` object
Object **invoke**(Object obj, Object[] args)	Invokes the method represented by this `Method` object on the specified object with the arguments specified in the `Object` array

metaobjects that represent methods. Table 1.3 shows some of the methods supported by the metaobject class `Method`.

Each `Method` object provides information about a method including its name, parameter types, return type, and exceptions. A `Method` object also provides the ability to call the method that it represents. For our example, we are most interested in the ability to call methods, so the rest of this section focuses on the `invoke` method.

1.6.1 Using dynamic invocation

Dynamic invocation enables a program to call a method on an object at runtime without specifying which method at compile time. In section 1.2, George does not know which `setColor` method to call when he writes the program. His program relies upon introspection to examine the class of a parameter, `obj`, at runtime to find the right method. As a result of the introspection, the `Method` representing `setColor` is stored in the variable `method`.

Following the introspection in listing 1.1, `setColor` is invoked dynamically with this line:

```
method.invoke(obj, new Object[] {color});
```

where the variable `color` holds a value of type `Color`. This line uses the `invoke` method to call the `setColor` method found previously using introspection. The `setColor` method is invoked on `obj` and is passed the value of `color` as a parameter.

The first parameter to `invoke` is the target of the method call, or the `Object` on which to invoke the method. George passes in `obj` because he wants to call `setColor` (the method represented by `method`) on `obj`. However, if `setColor` is declared `static` by the class of `obj`, the first parameter is ignored because static methods do not need invocation targets. For a static method, `null` can be supplied as the first argument to `invoke` without causing an exception.

The second parameter to `invoke`, `args`, is an `Object` array. The `invoke` method passes the elements of this array to the dynamically invoked method as actual parameters. For a method with no parameters, the second parameter may be either a zero-length array or `null`.

1.6.2 Using primitives with dynamic invocation

The second parameter to `invoke` is an array of `Object`, and the return value is also an `Object`. Of course, many methods in Java take primitive values as parameters and also return primitives. It is important to understand how to use primitives with the `invoke` method.

If the type of a parameter is a primitive, `invoke` expects the corresponding `args` array element to be a wrapper object containing the argument. For example, when invoking a method with an `int` parameter, wrap the `int` argument in a `java.lang.Integer` and pass it into the `args` array. The `invoke` method unwraps the argument before it passes it to the actual code for the method being invoked.

The `invoke` method handles primitive return types by wrapping them before they are returned. Thus, when invoking a method with an `int` return type, the program receives an object of type `Integer` in return. If the method being invoked is declared with a `void` return, `invoke` returns the value `null`.

So, primitives need to be wrapped when passed into a dynamic invocation and unwrapped when received as a return value. For clarity, consider the following dynamic call to `hashCode` method on our `obj` variable from the example.

```
Method method = obj.getClass().getMethod("hashCode", null);
int code = ((Integer) method.invoke(obj, null)).intValue();
```

The first line introspects for the method `hashCode` with no arguments. This query does not fail because that method is declared by `Object`. The `hashCode` method returns an `int`. The second line invokes `hashCode` dynamically and stores the return value in the variable `code`. Notice that the return value comes back wrapped

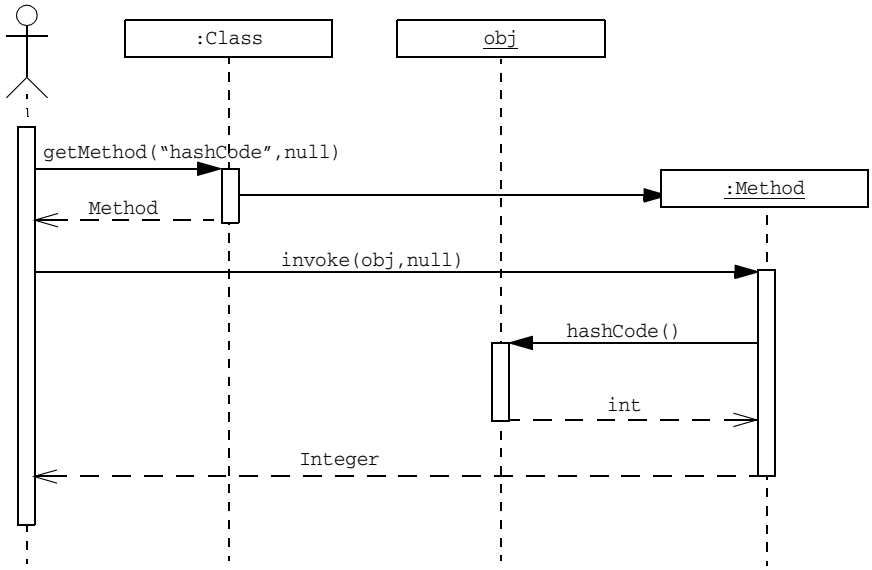

Figure 1.2 Sequence diagram illustrating the use of `getMethod` and `invoke`. The return arrows are labeled with the type of the value that is returned. Note that the call to `invoke` wraps the `int` return value in an `Integer` object.

in an `Integer`, and it is cast and unwrapped. The above snippet of code is illustrated in the sequence diagram in figure 1.2.

1.6.3 Avoiding invocation pitfalls

At one point, George thinks, "*If I have a `Method` representing `setColor`, why do I need to introspect for it every time? I'll just cache the first one that comes along and optimize out the rest of the queries.*" When he tries this, he gets an `IllegalArgumentException` from `invoke` on many of the subsequent calls. The exception message means that the *method was invoked on an object that is not an instance of the declaring class.*

George's optimization fails because it assumes that all methods with the same signature represent the same method. This is not the case. In Java, each method is identified by both its signature and its declaring class.

Let's take a closer look at this failure. Figure 1.3 shows the classes `Animal` and `Shape`, which both declare a `setColor` method with the same signature. These two `setColor` methods are not the same method in Java because they do not have the same declaring class.

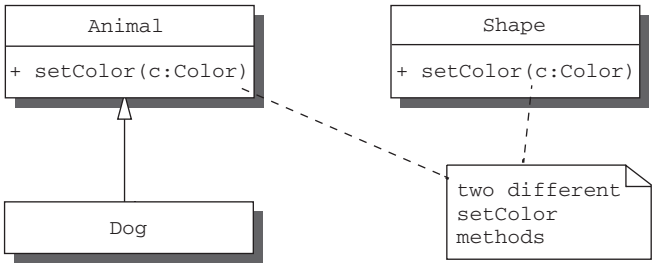

Figure 1.3 A Unified Modeling Language (UML) class diagram. `Dog` is a subclass of `Animal`. `Animal` and `Shape` both declare a `setColor` method of the same signature. The Java language considers the two `setColor` methods shown to be different methods. However, the `setColor` method for `Dog` is the same method as the one for `Animal`.

Another class, `Dog`, extends `Animal` and inherits its `setColor` method. The `setColor` method for `Dog` is the same as the `setColor` method for `Animal` because Dog inherits `setColor` from `Animal`. The `setColor` method for `Dog` is not the same method as the one for `Shape`. Therefore, when dealing with this situation, it is usually simplest to introspect for a `Method` each time instead of caching.

Several other exceptions can occur when calling `invoke`. If the class calling `invoke` does not have appropriate access privileges for the method, `invoke` throws an `IllegalAccessException`. For example, this exception can occur when attempting to invoke a private method from outside its declaring class.

`IllegalArgumentException` can be thrown by `invoke` under several circumstances. Supplying an invocation target whose class does not support the method being invoked causes an `IllegalArgumentException`. Supplying an `args` array of incorrect length or with entries of the wrong type also causes an `IllegalArgumentException`. If any exception is thrown by the method being invoked, that exception is wrapped in an `InvocationTargetException` and then thrown.

Dynamic invocation is a truly important feature in Java reflection. Without it, each method call must be hard-coded at compile time, denying programmers the flexibility of doing what George does in listing 1.1. In later chapters, we return to dynamic invocation for more advanced applications and expose other powerful ways to use information gained through introspection.

1.7 Diagramming for reflection

Throughout this book, we use the Unified Modeling Language (UML) for diagrams like figure 1.4. Those familiar with UML will probably notice that figure 1.4 combines UML class and object diagrams. Reflection represents all of the class diagram entities at runtime using metaobjects. Therefore, combining class and object diagrams is useful for clearly communicating reflective designs.

UML diagrams typically include only classes or only non-class objects. Modeling reflection calls for combining the two and using the `instanceOf` dependency to connect an object with its instantiating class. UML defines the `instanceOf` dependency with same meaning as the Java `instanceof` operator. However, this book uses the `instanceOf` dependency only to show that an object is a direct instance of a class. For clarity, we partition figure 1.4 into its base level and metalevel, although that partition is not standard UML. For more detail on UML, see appendix C.

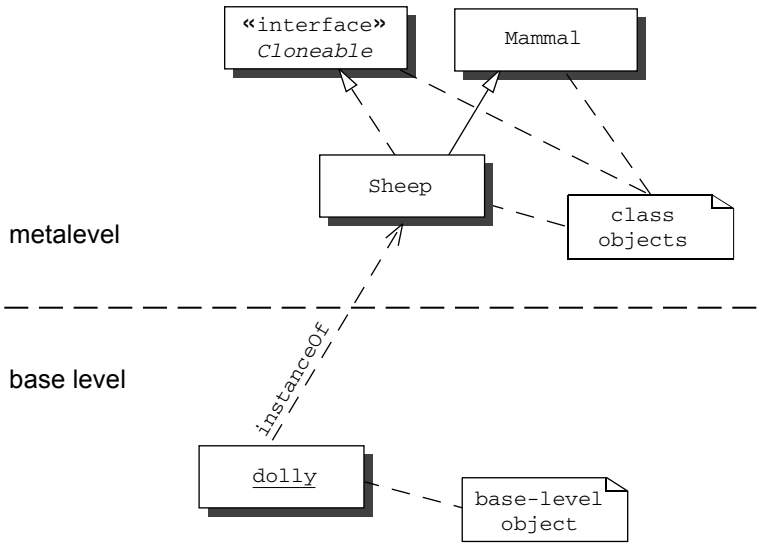

Figure 1.4 This is a Unified Modeling Language (UML) diagram describing Dolly the cloned sheep. The diagram shows an object, `dolly`, which is an instance of the class `Sheep`. It describes `Sheep` as a `Mammal` that implements `Cloneable`. The important thing to notice about this diagram is that it includes both objects and classes, as is necessary for describing reflective systems.

1.8 Navigating the inheritance hierarchy

After George's team has been using `setObjectColor` from listing 1.1 for a while, one of his team members, Martha, runs into a problem. Martha tells George that `setObjectColor` is not seeing a `setColor` method inherited by her component. After exploring the inheritance hierarchy, George and Martha discover that the inherited method is `protected`, and so it is not found by the line

```
Method method = cls.getMethod("setColor", new Class[] {Color.class});
```

George decides that he needs a method that introspects over methods of all visibilities, declared or inherited. Looking back at the methods from table 1.1, George notices that there is no method that does this, so he decides to write his own. Listing 1.3 shows the source code for `getSupportedMethod`, a method that George has written to accomplish that query. George has placed `getSupportedMethod` in his own convenience facility called `Mopex`. This is one of many useful methods that George has put in `Mopex`, and throughout this book, we explain and make use of them.

Listing 1.3 Code for `Mopex.getSupportedMethod`

```
public static Method getSupportedMethod( Class cls,
                                         String name,
                                         Class[] paramTypes)
    throws NoSuchMethodException
{
    if (cls == null) {
        throw new NoSuchMethodException();
    }
    try {
        return cls.getDeclaredMethod( name, paramTypes );
    }
    catch (NoSuchMethodException ex) {
      return getSupportedMethod( cls.getSuperclass(), name, paramTypes );
    }

}
```

The `getSupportedMethod` method is a recursive method that traverses the inheritance hierarchy looking for a method with the correct signature using `getDeclaredMethod`. It uses the line

```
return getSupportedMethod( cls.getSuperclass(), name, paramTypes );
```

to accomplish this traversal. The `getSuperclass` method returns the class object representing the class that its target extends. If there is no extends clause, `getSuperclass` returns the class object for `Object`. If `cls` represents `Object`, `getSuperclass` returns `null`, and `getSupportedMethod` throws a `NoSuchMethodException` on the next call.

Now that George has implemented `getSupportedMethod`, which performs the introspection that he wants, he can change `setObjectColor` to use this new functionality. Listing 1.4 shows this update to `setObjectColor`.

Listing 1.4 `setObjectColor` updated to use `getSupportedMethod`

```
public static void setObjectColor( Object obj, Color color ) {
    Class cls = obj.getClass();
    try {
        Method method = Mopex.getSupportedMethod( cls,
                                                  "setColor",
                                                  new Class[]{Color.class}
                                                  );
        method.invoke( obj, new Object[] {color} );
    }
    catch (NoSuchMethodException ex) {
        throw new IllegalArgumentException(
                              cls.getName() + " does not support"
                              + "method setColor(:Color)");
    }
    catch (IllegalAccessException ex) {
        throw new IllegalArgumentException(
                              "Insufficient access permissions to call"
                              + "setColor(:Color) in class "
                              + cls.getName());
    }
    catch (InvocationTargetException ex) {
        throw new RuntimeException(ex);
    }
}
```

This update allows `setObjectColor` to retrieve metaobjects for private, package, and protected methods that are not retrieved by `getMethod`. However, this update does not guarantee permission to invoke the method. If `setObjectColor` does not have access to Martha's inherited method, an `IllegalAccessException` is thrown instead of a `NoSuchMethodException`.

George has just observed one way that reflection can save him effort. Before the reflective enhancement, he and Martha needed to explore the inheritance

hierarchy to diagnose Martha's problem. George's enhancement traverses the inheritance hierarchy and reports the problem, saving them the trouble. In chapter 2, we discuss bypassing visibility checks using reflection. For now, let's continue to discuss the tools that make George and Martha's enhancement possible.

1.8.1 Introspecting the inheritance hierarchy

As shown in the previous section, runtime access to information about the inheritance hierarchy can prevent extra work. Getting the superclass of a class is only one of the operations that Java reflection provides for working with the inheritance hierarchy. Table 1.4 shows the signatures and return types for the methods of `Class` for dealing with inheritance and interface implementation.

Table 1.4 Methods of `Class` that deal with inheritance

Method	Description
Class[] **getInterfaces**()	Returns an array of `Class` objects that represent the direct superinterfaces of the target `Class` object
Class **getSuperclass**()	Returns the `Class` object representing the direct superclass of the target `Class` object or `null` if the target represents `Object`, an interface, a primitive type, or void
boolean **isAssignableFrom**(Class cls)	Returns `true` if and only if the class or interface represented by the target `Class` object is either the same as or a superclass of or a superinterface of the specified `Class` parameter
boolean **isInstance**(Object obj)	Returns `true` if and only if the specified `Object` is assignment-compatible with the object represented by the target `Class` object

The `getInterfaces` method returns class objects that represent interfaces. When called on a class object that represents a class, `getInterfaces` returns class objects for interfaces specified in the `implements` clause of that class's declaration. When called on a class object that represents an interface, `getInterfaces` returns class objects specified in the `extends` clause of that interface's declaration.

Note the method names `getInterfaces` and `getSuperclass` are slightly inconsistent with terminology defined by the *Java Language Specification*. A **direct superclass** is the one named in the extends clause of a class declaration. A class `X` is a **superclass** of a class `Y` if there is a sequence of one or more direct superclass links from `Y` to `X`. There is a corresponding pair of definitions for **direct superinterface** and **superinterface**. Consequently, `getSuperclass` returns the direct superclass and `getInterfaces` returns the direct superinterfaces.

To get all of the methods of a class, a program must walk the inheritance hierarchy. Luckily, this walk is not necessary to query whether a class object represents a subtype of another class object. This query can be accomplished using the `isAssignableFrom` method. The name `isAssignableFrom` tends to be confusing. It helps to think of

```
X.isAssignableFrom(Y)
```

as "an `X` field *can be assigned* a value from a `Y` field." For example, the following lines evaluate to true:

```
Object.class.isAssignableFrom(String.class)

java.util.List.class.isAssignableFrom(java.util.Vector.class)

double.class.isAssignableFrom(double.class)
```

The line below, however, evaluates to false:

```
Object.class.isAssignableFrom(double.class)
```

The `isInstance` method is Java reflection's dynamic version of `instanceof`. If the target class object represents a class, `isInstance` returns `true` if its argument is an instance of that class or any subclass of that class. If the target class object represents an interface, `isInstance` returns `true` if its argument's class implements that interface or any subinterface of that interface.

1.8.2 Exposing some surprises

In the Java reflection API, there are some relationships that may be surprising upon first glance. Discussing these relationships now prepares us for encountering them later in the book and in reflective programming in general. Being prepared in this manner allows for better reflective programming.

The `isInstance` method can be used to show a very interesting fact about the arrangement of the classes in the Java reflection API. The line

```
Class.class.isInstance(Class.class)
```

evaluates to `true`. This means that the class object for `Class` is an instance of itself, yielding the circular `instanceOf` dependency of figure 1.5. `Class` is an example of a **metaclass**, which is a term used to describe classes whose instances are classes. `Class` is Java's only metaclass.

In Java, all objects have an instantiating class, and all classes are objects. Without the circular dependency, the system must support an infinite tower of class objects, each one an instance of the one above it. Instead, Java uses this circularity to solve this problem.

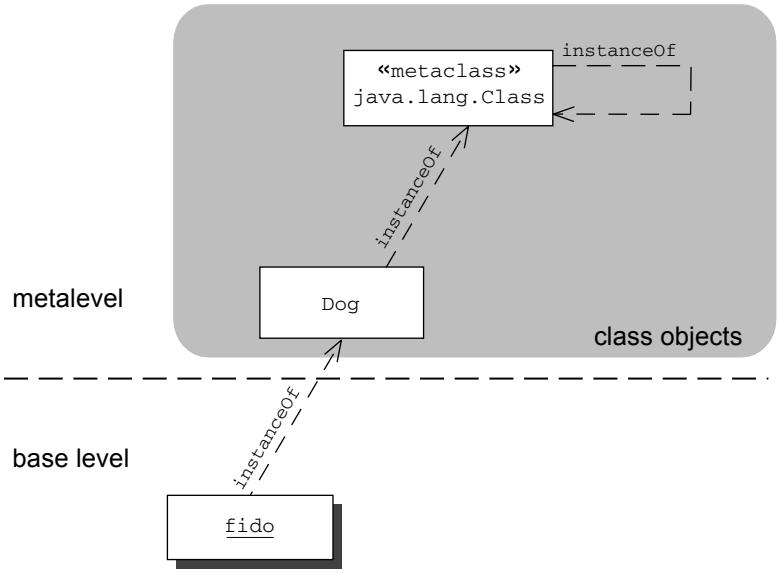

Figure 1.5 The object `fido` is an instance of the `Dog` class. `Dog` is an instance of the class `Class`. `Class` is also an instance of `Class`. `Class` is a metaclass because it is a class whose instances are classes.

The circularity presented in figure 1.5 makes people uncomfortable because we instinctively mistrust circular definitions. However, as programmers, we are familiar with other kinds of circular definitions. For example, consider recursion. A method that uses recursion is defined in terms of itself; that is, it has a circular definition. When used properly, recursion works just fine. Similarly, there are constraints on the definition of `java.lang.Class` that make this circularity work just fine.

For more information about this circularity, see *Putting Metaclasses to Work* [33]. *Putting Metaclasses to Work* is an advanced book on reflection and metaobject protocols written by one of the authors of this book. It is a good resource for readers who are interested in the theoretical and conceptual basis for reflection.

1.8.3 Another reflective circularity

Adding inheritance to our previous diagram yields the arrangement in figure 1.6. Inheritance adds more circularity to the picture. `Object` is an instance `Class`, which can be validated because the following line returns `true`:

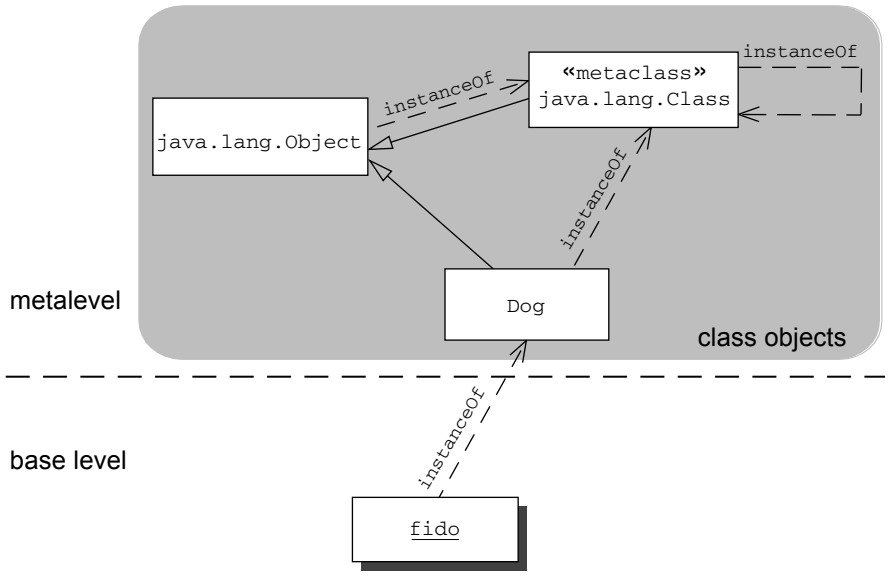

Figure 1.6 **`Object` is the top of the Java inheritance hierarchy, so classes of metaobjects, including `Class`, are subclasses of `Object`. This means that the methods of `Object` are part of the reflection API. All Java classes are instances of its only metaclass, `Class`. These two conditions create a cycle in the diagram.**

```
Class.class.isInstance(Object.class)
```

`Class` is also a subclass of `Object`, validated by

```
Object.class.isAssignableFrom(Class.class)
```

which also returns `true`. Conceptually, we already know these facts because in Java, each object has one instantiating class, and all classes are kinds of objects. However, it is comforting that the reflective model is consistent with our previous understanding of the language.

The new circularity implies additional constraints on the definitions of `Object` and `Class`. These constraints are satisfied when the Java Virtual Machine loads the `java.lang` package. Again, a full explanation of the constraints may be found in *Putting Metaclasses to Work* [33].

Figure 1.6 also illustrates why `Object` is considered part of the reflection API. All metaobjects extend `Object`, and so they inherit its methods. Therefore, each of those methods can be used in reflective programming.

1.9 Summary

Reflection allows programs to examine themselves and make changes to their structure and behavior at runtime. Even a simple use of reflection allows programmers to write code that does things that a programmer would normally do. These simple uses include getting the class of an object, examining the methods of a class, calling a method discovered at runtime, and exploring the inheritance hierarchy.

The metaobject classes `Class` and `Method` represent the classes and methods of running programs. Other metaobjects represent the other parts of the program such as fields, the call stack, and the loader. `Class` has additional methods to support these other metaobjects. Querying information from these metaobjects is called introspection.

Metaobjects also provide the ability to make changes to the structure and behavior of the program. Using dynamic invocation, a `Method` metaobject can be commanded to invoke the method that it represents. Reflection provides several other ways to affect the behavior and structure of a program such as reflective access, modification, construction, and dynamic loading.

There are several patterns for using reflection to solve problems. A reflective solution often starts with querying information about the running program from metaobjects. After gathering information using introspection, a reflective program uses that information to make changes in the behavior of the program.

Each new metaobject class allows us to grow our examples in scope and value. These examples reveal lessons that we have learned and techniques that we have applied. Each one follows the same basic pattern of gathering information with introspection and then using the information to change the program in some way.

JAVA

JAVA Reflection IN ACTION

Ira R. Forman and Nate Forman

Imagine programs that are able to adapt—with no intervention by you—to changes in their environment. With Java reflection you can create just such programs. Reflection is the ability of a running program to look at itself and its environment, and to change what it does depending on what it finds. This inbuilt feature of the Java language lets you sidestep a significant source of your maintenance woes: the "hard-coding" between your core application and its various components.

Java Reflection in Action shows you that reflection isn't hard to do. It starts from the basics and carefully builds a complete understanding of the subject. It introduces you to the reflective way of thinking. And it tackles useful and common development tasks, in each case showing you the best-practice reflective solutions that replace the usual "hard-coded" ones. You will learn the right way to use reflection to build flexible applications so you can nimbly respond to your customers' future needs. Master reflection and you'll add a versatile and powerful tool to your developer's toolbox.

What's Inside

- Practical introduction to reflective programming
- Examples from diverse areas of software engineering
- How to design flexible applications
- When to use reflection—and when not to
- Performance analysis

Dr. Ira Forman is a computer scientist at IBM. He has worked on reflection since the early 1990s when he developed IBM's SOM Metaclass Framework. **Nate Forman** works for Ticom Geomatics where he uses reflection to solve day-to-day problems. Ira and Nate are father and son. They both live in Austin, Texas.

"Even occasional users [of reflection] will immediately adopt the book's patterns and idioms to solve common problems."

—DOUG LEA
SUNY Oswego, author of
CONCURRENT PROGRAMMING IN JAVA

"... guide[s] you through one compelling example after another, each one illustrating reflection's power while avoiding its pitfalls."

—JOHN VLISSIDES
IBM, coauthor of
DESIGN PATTERNS

www.manning.com/forman

Authors respond to reader questions


Ebook edition available

ISBN 1-932394-18-4

$44.95 US/$67.95 Canada