Chapter 3

Type Information and Reflection

Java classes preserve a wealth of information about programmer intent. Rather than just containing a jumble of executable instructions, binary classes[1] also contain large amounts of *metadata*—data that describes the structure of the binary class. Most of this metadata is *type information* enumerating the base class, superinterfaces, fields, and methods of the class. Type information is used to make the dynamic linking of code more reliable by verifying at runtime that clients and servers share a common view of the classes they use to communicate.

The presence of type information also enables dynamic styles of programming. You can *introspect* against a binary class to discover its fields and methods at runtime. Using this information, you can write generic services to add capabilities to classes that have not even been written yet.

The binary class format is a simple data structure that you could parse to perform introspection yourself. Rather than going to this trouble, you can use the Java Reflection API instead. Reflection provides programmatic access to most of the metadata in the binary class format. It also provides not only the ability to introspect classes for metadata, but also the ability to dynamically access fields and methods. Reflective invocation is critical for writing generic object services. As of SDK version 1.3, reflection also includes the ability to manufacture classes called dynamic proxies at runtime. This chapter introduces

[1] [LY99] uses the term "class file" instead. This usage encourages the mistaken assumption that classes *must* live in files, and it will probably be replaced in a future edition of the spec. Throughout this book, I will use the more generic term "binary class."

the binary class format, the uses of metadata, the Reflection API, dynamic proxies, and custom metadata.

3.1 The Binary Class Format

The binary class format means different things to different people. To an application developer, the binary class is the compiled output of a Java class. Most of the time, you can treat the class format as a black box—a detail that is thankfully hidden by the compiler. The binary class is also the unit of executable code recognized by the virtual machine. Virtual machine developers see the binary class as a data structure that can be loaded, interpreted, and manipulated by virtual machines and by Java development tools. The binary class is also the unit of granularity for dynamic class loading. Authors of custom class loaders take this view and may use their knowledge of the binary class format to generate custom classes at runtime. But most importantly, the binary class is a well-defined format for conveying class code and class metadata.

Most of the existing literature on the binary class format targets compiler and virtual machine developers. For example, the virtual machine specification provides a wealth of detail about the exact format of a binary class, plus a specific explanation of extensions that can legally be added to that format. For a Java developer, such detail is overkill. However, hidden in that detail is information that the virtual machine uses to provide valuable services, such as security, versioning, type-safe runtime linkage, and runtime type information. The availability and quality of these services is of great concern to all Java developers. The remainder of Section 3.1 will describe the information in the binary class format, and how that information is used by the virtual machine. Subsequent sections show you how you can use this information from your own programs.

3.1.1 Binary Compatibility

A clear example of the power of class metadata is Java's enforcement of binary compatibility at runtime. Consider the `MadScientist` class and its client class `BMovie`, shown in Listing 3–1. If you compile the two classes and then execute the `BMovie` class, you will see that the `threaten` method executes as expected. Now, imagine that you decide to ship a modified version of MadScientist with the threaten method removed. What happens if an old version of BMovie tries to use this new version of MadScientist?

In a language that does not use metadata to link methods at runtime, the outcome is poorly defined. In this particular case, the old version of BMovie probably would link to the first method in the object. Since threaten is no longer part of the class, blowUpWorld is now the first method. This program error would literally be devastating to the caller.

Listing 3–1 The MadScientist Class

```
public class MadScientist {
  public void threaten() {
    System.out.println("I plan to blow up the world");
  }
  public void blowUpWorld() {
    throw new Error("The world is destroyed. Bwa ha ha ha!");
  }
}
public class BMovie {
  public static void main(String [] args) {
    MadScientist ms = new MadScientist();
    ms.threaten();
  }
}
```

As bad as this looks, an obvious failure is actually one of the *best* possible outcomes for version mismatches in a language without adequate metadata. Consider what might happen in a systems programming language, such as C++, that encodes assumptions about other modules as numeric locations or offsets. If these assumptions turn out to be incorrect at runtime, the resulting behavior is undefined. Instead of the desired behavior, some random method may be called, or some random class may be loaded. If the random method does not cause an immediate failure, the symptoms of this problem can be incredibly difficult to track down. Another possibility is that the code execution will transfer to some location in memory that is not a method at all. Hackers may exploit this situation to inject their own malicious code into a process.

Compare all the potential problems above with the actual behavior of the Java language. If you remove the `threaten` method, and recompile only the `MadScientist` class, you will see the following result:

```
>java BMovie
java.lang.NoSuchMethodError
   at BMovie.main(BMovie.java:4)
```

If a class makes a reference to a nonexistent or invalid entity in some other class, that reference will trigger some subclass of `IncompatibleClassChangeError`, such as the `NoSuchMethodError` shown above. All of these exception types indirectly extend `Error`, so they do not have to be checked and may occur at any time. Java assumes fallible programmers, incomplete compile-time knowledge, and partial installations of code that change over time. As a result, the language makes runtime metadata checks to ensure that references are resolved correctly. Systems languages, on the other hand, tend to assume expert programmers, complete compile-time knowledge, and full control of the installation processes. The code that results from these may load a little faster than Java code, but it will be unacceptably fragile in a distributed environment.

In the earlier example, the missing method `threaten` caused the new version of `MadScientist` to be incompatible with the original version of `BMovie`. This is an obvious example of incompatibility, but some other incompatibilities are a little less obvious. The exact rules for binary class compatibility are enumerated in [LY99], but you will rarely need to consult the rules at this level. The rules all support a single, common-sense objective: no mysterious failures. A reference either resolves to the exact thing the caller expects, or an error is thrown; “exactness” is limited by what the caller is looking for. Consider these examples:

- You cannot reference a class, method, or field that does not exist. For fields and methods, both names and types must match.
- You cannot reference a class, method, or field that is invisible to you, for example, a private method of some other class.
- Because private members are invisible to other classes anyway, changes to private members will *not* cause incompatibilities with other classes. A

similar argument holds for package-private members *if* you always update the entire package as a unit.

- You cannot instantiate an abstract class, invoke an abstract method, subclass a `final` class, or override a `final` method.
- Compatibility is in the eye of the beholder. If some class adds or removes methods that you never call anyway, you will not perceive any incompatibility when loading different versions of that class.

Another way to view all these rules is to remember that changes to invisible implementation details will never break binary compatibility, but changes to visible relationships between classes will.

3.1.1.1 Declared Exceptions and Binary Compatibility

One of the few oddities of binary compatibility is that you *can* refer to a method or constructor that declares checked exceptions that you do not expect. This is less strict than the corresponding compile-time rule, which states that the caller must handle all checked exceptions. Consider the versions of `Rocket` and `Client` shown in Listing 3–2. You can only compile `Client` against version 1 of the `Rocket` since the client does not handle the exception thrown by version 2. At runtime, a `Client` could successfully reference and use either version because exception types are not checked for binary compatibility.

This loophole in the binary compatibility rules may be surprising, but it does not compromise the primary objective of preventing inexplicable failures. Consider what happens if your `Client` encounters the second version of `Rocket`. If and when the `InadequateNationalInfrastructure` exception is thrown, your code will not be expecting it, and the thread will probably terminate. Even though this may be highly irritating, the behavior is clearly defined, and the stack trace makes it easy to detect the problem and add an appropriate handler.

Listing 3–2 Checked Exceptions Are Not Enforced by the VM.

```
public class Client {
  Rocket r = new Rocket();
}
public class Rocket { //version 1
  public Rocket() { … }
}
public class Rocket { //version 2
  public Rocket()
  throws InadequateNationalInfrastructure { … }
}
```

3.1.1.2 Some Incompatible Changes Cannot Be Detected

The Java compiler enforces the rules of binary compatibility at compile time, and the virtual machine enforces them again at runtime. The runtime enforcement of these rules goes a long way toward preventing the accidental use of the wrong class. However, these rules do not protect you from bad decisions when you are shipping a new version of a class. You can still find clever ways to write new versions of classes that explode when called by old clients.

Listing 3–3 shows an unsafe change to a class that Java cannot prevent. Clients of the original version of `Rocket` expect to simply call `launch`. The second version of `Rocket` changes the rules by adding a mandatory `preLaunchSafetyCheck`. This does not create any structural incompatibilities with the version 1 clients, who can still find all the methods that they expect to call. As a result, old versions of the client might launch new rockets without the necessary safety check. If you want to rely on the virtual machine to protect the new version of `Rocket` from old clients, then you must deliberately introduce an incompatibility that will break the linkage. For example, your new version could implement a new and different `Rocket2` interface.[2]

Listing 3–3 Some Legal Changes to a Class May Still Be Dangerous.

```
public interface Rocket {  //version 1
  public void launch();
}
public interface Rocket { //version 2
  public void mandatoryPreLaunchSafetyCheck();
  public void launch();
}
```

[2] Package reflection, discussed later in this chapter, provides another approach to preventing this problem.

3.1.2 Binary Class Metadata

[LY99] documents the exact format of a binary class. My purpose here is not to reproduce this information but to show what kinds of metadata the binary class includes. Figure 3–1 shows the relevant data structures that you can traverse in the binary class format. The constant pool is a shared data structure that contains elements, such as class constants, method names, and field names, that are referenced by index elsewhere in the class file. The other structures in the class file do not hold their own data; instead, they hold indexes into the constant pool. This keeps the overall size of the class file small by avoiding the repetition of similar data structures.

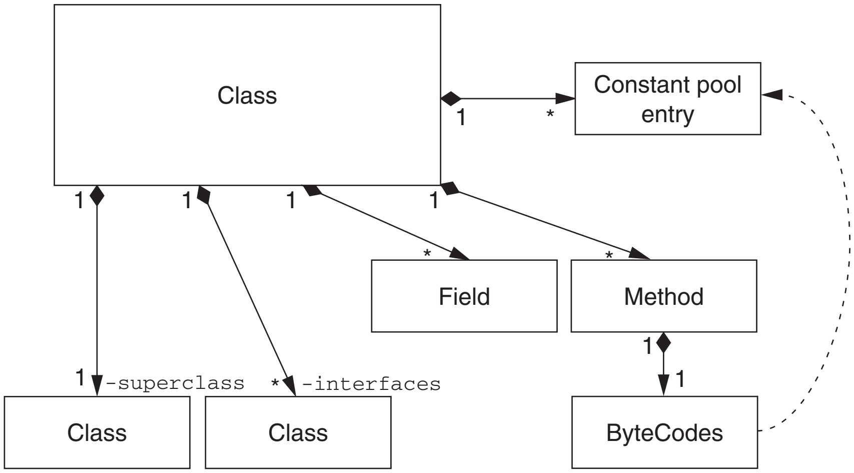


Figure 3–1 Metadata in the binary class format

The `-superclass` and `-interfaces` references contain indices into the constant pool. After a few levels of indirection, these indices eventually lead to the actual string names of the class's base class and superinterfaces. The use of actual string names makes it possible to verify *at runtime* that the class meets the contractual expectations of its clients.

Note that the class name format used by the virtual machine is different from the dotted notation used in Java code. The VM uses the "/" character as a package delimiter. Also, it often uses the "L" and ";" characters to delimit class names if the class name appears inside a stream where other types of data might also appear. So, the class `java.lang.String` will appear as either `java/lang/String` or `Ljava/lang/String;` in the class file's constant pool.

The fields and methods arrays also contain indices into the constant pool. Again, these constant pool entries lead to the actual string names of the referenced types, plus the string names of the methods and fields. If the referenced type is a primitive, the VM uses a special single-character string encoding for the type, as shown in Table 3–1. A method also contains a reference to the Java bytecodes that implement the method. Whenever these bytecodes refer to another class, they do so through a constant pool index that resolves to the string name of the referenced class. Throughout the virtual machine, types are referred to by their full, package qualified string names. Fields and methods are also referenced by their string names.

Table 3–1 Virtual Machine Type Names

Java Type	Virtual Machine Name
int	I
float	F
long	J
double	D
byte	B
boolean	Z
short	S
char	C
type[]	[*type*
package.SomeClass	L*package.SomeClass*;

3.1.2.1 Analyzing Classes with javap

The details of binary class data structures are of interest to VM writers, and they are covered in detail in the virtual machine specification [LY99]. Fortunately, there are a large number of tools that will display information from the binary class format in a human-friendly form. The `javap` tool that ships with the SDK is a simple class decompiler. Consider the simple `Echo1` class:

```
public class Echo1 {
  private static final String prefix = "You said: ";
  public static void main(String [] args) {
    System.out.println(prefix + args[0]);
  }
}
```

If you run `javap` on the compiled `Echo1` class, you will see output similar to Listing 3–4. As you can see, the class format contains the class names, the method names, and the parameter type names. The `javap` utility has a variety of more verbose options as well, including the –`c` flag to display the actual bytecodes that implement each method, shown in Listing 3–5. Without worrying about what specific bytecodes do, you can easily see that the bytecode instructions refer to classes, fields, and members by name. The `#10`, `#5`, `#1`, and `#8` in the output are the indices into the constant pool; `javap` helpfully resolves these indices so that you can see the actual strings being referenced.

Listing 3–4 Standard javap Output

```
>javap Echo
Compiled from Echo1.java
public class Echo1 extends java.lang.Object {
    public Echo1();
    public static void main(java.lang.String[]);
}
```

Listing 3–5 Javap Output with Bytecodes Included

```
>javap -c Echo1
  {output clipped for brevity}
Method void main(java.lang.String[])
   0 getstatic #10 <Field java.io.PrintStream out>
   3 new #5 <Class java.lang.StringBuffer>
   6 dup
   7 ldc #1 <String "You said: ">
   9 invokespecial #8 <Method
          java.lang.StringBuffer(java.lang.String)>
   etc…
```

3.1.3 From Binary Classes to Reflection

Java class binaries always contain metadata, including the string names for classes, fields, field types, methods, and method parameter types. This metadata is used implicitly to verify that cross-class references are compatible. Both metadata and the notion of class compatibility are built into the bones of the Java language, so there is no subterranean level where you can avoid their presence. By themselves, the binary compatibility checks provided by the virtual machine would be sufficient to justify the cost of creating, storing, and processing class metadata. In reality, these uses only scratch the surface. You can access the same metadata directly from within your Java programs using the Reflection API.

3.2 Reflection

The Java Reflection API presents a Java interface to the metadata contained in the binary class format. You can use reflection to dynamically discover the characteristics of a Java class: its base class, superinterfaces, method signatures, and field types. Better yet, you can use reflection to dynamically instantiate objects, invoke methods, and mutate fields. These features make it possible to write generic object services that do not rely on, or even have advance knowledge of, the specific classes they will be working on.

Reflection makes it straightforward to serialize an instance to a stream, generate relational database tables that correspond to a class's fields, or create a user interface that can manipulate instances of any arbitrary class. Reflection can also be used to automatically generate source or compiled code that forwards method calls for a set of interfaces to a generic handler. This feature is invaluable for adding layers of code for logging, auditing, or security. Reflection allows you to write service layers that do not require compile-time knowledge of the specific systems they will support.

Some examples of reflection in the core API include serialization and JavaBeans. Java can serialize class instances by writing their state into an opaque stream to be reloaded in some other time or place. Java serialization works even for classes that have not been written yet, because the serialization API uses reflection to access class fields.

At the bare minimum, a JavaBean is a serializable class with a default constructor. However, bean-aware tools can use reflection to discover the properties and events associated with a bean. This means that tools can deal with classes that they have never seen simply by reflecting against them.

Serialization and JavaBeans are powerful idioms, but they are still just idioms. Their underlying architecture is reflection. If you understand reflection you can develop your own idioms, more suited to your particular problem domain.

Most of the Reflection API lives in the `java.lang.reflect` package, but the central class in reflection is the `java.lang.Class` class. A `Class` represents a single binary class, loaded by a particular class loader, within the virtual machine. By using a `Class` instance as a starting point, you can discover all type information about a class. For example, you might want to know all of a class's superclasses and superinterfaces. The `ListBaseTypes` class shown in Listing 3–6 uses the `Class` methods `getInterfaces` and `getSuperclass` to return a class's superinterfaces and superclass, and then it follows these recursively back to the beginning, which is defined to be `java.lang.Object` for classes and `null` for interfaces. Sample output for `ListBaseTypes` is shown in Listing 3–7.

Listing 3–6 The ListBaseTypes Class

```
public class ListBaseTypes {
  public static void main(String [] args) throws Exception
  {
    Class cls = Class.forName(args[0]);
    System.out.println("Base types for " + args[0]);
    listBaseTypes(cls, "");
  }
  public static void listBaseTypes(Class cls, String pref) {
    if (cls == Object.class) return;
    Class[] itfs = cls.getInterfaces();
    for (int n=0; n<itfs.length; n++) {
      System.out.println(pref + "implements " + itfs[n]);
      listBaseTypes(itfs[n], pref+"\t");
    }
    Class base = cls.getSuperclass();
    if (base == null) return;
    System.out.println(pref + "extends " + base);
    listBaseTypes(base, pref+"\t");
  }
}
```

Listing 3–7 Sample Output from ListBaseTypes

```
Base types for java.io.ObjectOutputStream
implements interface java.io.ObjectOutput
   implements interface java.io.DataOutput
implements interface java.io.ObjectStreamConstants
extends class java.io.OutputStream
   extends class java.lang.Object
```

3.2.1 Reflecting on Fields

A more interesting use of reflection is to discover the fields of a class. Fields are represented by the `java.lang.reflect.Field` class. As Listing 3–8 shows, the `Field` class contains all of the information from the original source code declaration of a field: the field's name, type, and modifiers. The `Class` class provides several methods for retrieving a class's fields, shown in Listing 3–9.

Listing 3–8 Type Information in the Field Class

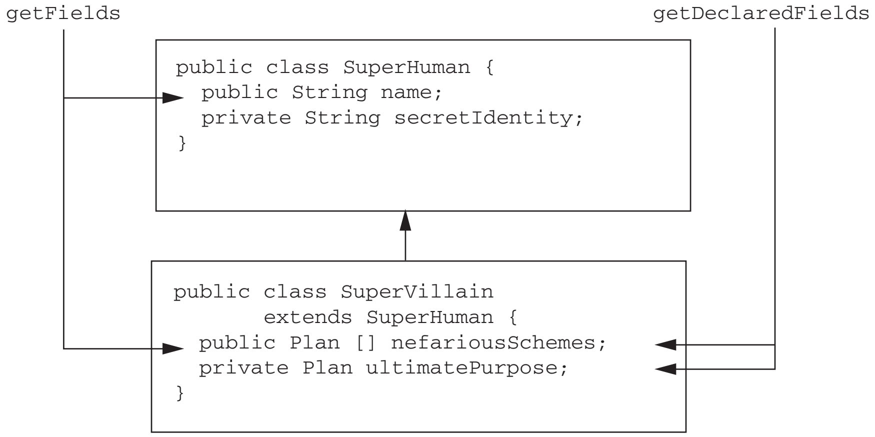
```
package java.lang.reflect;
public class Field {
  public String getName();
  public int getModifiers();
  public Class getType();
}//remainder omitted for clarity
```

Listing 3–9 Class Methods for Accessing Fields

```
package java.lang;
public class Class {
  public Field[] getDeclaredFields();
  public Field[] getFields();
  public Field getDeclaredField(String name)
    throws NoSuchFieldException;
  public Field getField(String name)
    throws NoSuchFieldException;
  //remainder omitted for clarity
}
```

3.2.2 The Difference between get and getDeclared

The `Field` methods of `Class` exhibit a pattern that occurs throughout the Reflection API: an accessor method named `getXXX`, plus a similar accessor named `getDeclaredXXX`. These method forms differ in two ways. The getXXX methods return only public members of a class, but they will return members from a class and all of its base classes. The getDeclaredXXX methods will return all members of a class, regardless of protection modifier; however, they will *not* recurse to base classes. There is no compelling reason for or against this convention; you simply must memorize it. The combination of getFields and getDeclaredFields is not sufficient to access nonpublic base class fields. As shown in Figure 3–2, in order to list all fields of a class and its base classes, you need to use the getDeclaredField form and then also recurse to base classes using getSuperclass.



```
public class SuperHuman {
  public String name;
  private String secretIdentity;
}
```

```
public class SuperVillain
       extends SuperHuman {
  public Plan [] nefariousSchemes;
  private Plan ultimatePurpose;
}
```

Figure 3–2 The difference between get and getDeclared

If you run ListMostFields (Listing 3–10) against String, you will see a list of nine fields (on the Java 2 SDK1.3, anyway). This surprisingly large list is due to the fact that the field APIs return static fields as well as instance fields. In order to distinguish static from instance fields, or to discover any other field modifiers, you must call getModifiers.

Listing 3–10 ListMostFields

```
import java.lang.reflect.*;
/**
 * lists all fields except superinterface fields
 * and fields from Object
 */
```

```
public class ListMostFields {
  public static void main(String [] args) throws Exception
  {
    Class cls = Class.forName(args[0]);
    System.out.println("Fields for " + args[0]);
    listMostFields(cls);
  }
  public static void listMostFields(Class cls) {
    if (cls == Object.class) return;
    Field[] fields = cls.getDeclaredFields();
    for (int n=0; n<fields.length; n++) {
      System.out.println(fields[n]);
    }
    Class base = cls.getSuperclass();
    if (base == null) return;
    System.out.println("extends " + base);
    listMostFields(base);
  }
}
```

The `getModifiers` call is one of the few places where Java thoroughly shows its nuts-and-bolts C language heritage. Instead of returning a `Modifiers` object, `getModifiers` returns an `int` that is a collection of bit flags. There actually is a `java.lang.reflect.Modifiers` class, but all it does is provide constants and static methods to interpret the bit flags. So, you can determine whether a `Field` is static by calling a static method on the `Modifiers` class but not through an instance of `Field` itself, as shown here:

```
//this is correct
Modifiers.isStatic(f.getModifiers());

//These are intuitive possibilities, but will not compile!
f.isStatic();
f.getModifiers().hasStatic();
```

3.2.3 Type Errors Occur at Runtime

The `getField` and `getDeclaredFields` of `Class` request a particular `Field` by name. These APIs introduce another wrinkle that recurs throughout reflection—the possibility that any call may fail at runtime. With “normal” nonreflective code, you get substantial feedback at compile time. If you reference a field that does not exist, the compiler will simply refuse to compile the class. With reflection, you are querying for the existence of a field at runtime, and you must be prepared for the possibility of failure at runtime. The `getField` and `getDeclaredField` methods remind you of this possibility by throwing the checked exception `NoSuchFieldException`. This is the blessing and curse of reflective programming. You can use reflection to refer to classes that do not even exist at compile time, but you must remember that the compiler is powerless to protect you from misusing such classes.

Listing 3–11 Type Information in the Method and Constructor Classes

```
//these methods appear in both
//java.lang.reflect.Method and
//java.lang.reflect.Constructor
public String getName();
public int getModifiers();
public Class getReturnType();
public Class[] getParameterTypes();
public Class[] getExceptionTypes();
```

3.2.4 Reflecting on Methods

You can use reflection to access methods in a fashion very similar to accessing fields. Methods are represented by the `java.lang.reflect.Method` class, which preserves the information from the method signature in the original Java source code file, as shown in Listing 3–11. Finding `Method` objects is slightly more complex than finding `Field` objects because `Method` names can be overloaded. The `Class` APIs for `Methods`, shown in Listing 3–12, require that you specify argument types in addition to the method name. In addition to argument types, the `Method` class also lets you access the return type and checked exceptions thrown by a method.

Listing 3–12 Class Methods for Accessing Methods and Constructors

```
//java.lang.Class methods for accessing Methods
//versions that take a name throw NoSuchMethodException
public Method getDeclaredMethod(String name, Class[]
                                parameterTypes);
public Method getMethod(String name,
                        Class[] parameterTypes);
```

```
public Method[] getDeclaredMethods();
public Method[] getMethods();
//methods for accessing Constructors
public Constructor getDeclaredConstructor(Class[]
                   parameterTypes);
public Constructor getMethod(Class[] parameterTypes);
public Constructor[] getDeclaredConstructors();
public Constructor[] getConstructors();
```

The rules for `Methods` are exactly the same as for `Fields`—the `getDeclared` versions of the APIs ignore protection modifiers, while the `get` versions return public members for the class and its base classes. The `Constructor` APIs work almost exactly like the `Method` APIs, but since the name of a constructor is implicit, the name parameter is absent from the `Constructor`-related methods.

3.3 Reflective Invocation

The Reflection API's ability to report the fields, methods, and constructors of a class is a very useful feature for the authors of Java development environments. For example, Java IDE wizards can automatically generate empty implementations of an interface for you to fill in. Java reflection makes this simple, whereas with many other languages, the development environment must build its own custom metadata representation to provide these services. While reflective reporting is a nifty parlor trick, the real power of reflection lies elsewhere, in the invocation APIs. With reflective invocation, you can access or change the value of a field and you can even invoke a method without any compile-time knowledge of the classes involved.

The invocation portion of the Reflection API uses the `Field`, `Method`, and `Constructor` classes to query and modify the state of Java classes and instances at runtime. This ability can be used to emulate function pointers, which do not exist in Java. Invocation can also be used as a substitute for inheritance. Instead of casting an object to a known interface type, you simply reflect against the class to see if it implements a particular method. Reflective invocation is also critical in crossing class loader boundaries. If you have a reference to a type that is not visible to your class loader delegation, you can use reflection to access that type's fields, constructors, and methods.

The reflective invocation APIs share several common elements. Because they are invoked on instances of `Field`, `Method`, or `Constructor`, they do not have an implicit `this` reference to the actual type being modified, and they must specify the `this` reference as an explicit parameter. Because reflective invocation APIs do not know the parameter types in advance, parameters are specified as `Object` or arrays of `Object`. Similarly, return types are always of type `Object`. Finally, reflective invocation cannot know in advance what checked exceptions might be thrown by a method, so all exceptions are caught by the APIs and rethrown as some wrapper type. The key reflective invocation APIs are summarized in Listing 3–13.

Listing 3–13 Key Reflective Invocation APIs

```
//all invocation APIs may throw
//IllegalArgumentException, IllegalAccessException

//from java.lang.reflect.Field
public Object get(Object this);
public void set(Object this, Object value);

//from java.lang.reflect.Method
public Object invoke(Object this, Object[] args)
       throws InvocationTargetException

//from java.lang.reflect.Constructor
public Object newInstance(Object[] args)
       throws InstantiationException,
              InvocationTargetException
```

3.3.1 A Reflective Launcher

As an example of a situation in which the Reflection API is essential, consider the `RunFromURL` class shown in Listing 3–14. `RunFromURL` is very similar to the Java launcher except that it loads a class from a URL passed on the command line instead of from the classpath. The meat of the code is the `run` method, which uses `getMethod` to find a method with the signature `main(String[] args)` and then invokes that method with a `null` reference for `this` (since the method is assumed to be static)[3] and an array of arguments from the command line. `RunFromURL` can execute the main method of any arbitrary class. Moreover, reflection is the only way this can be accomplished. `RunFromURL` cannot directly reference the class to be loaded. If `RunFromURL` referenced the class directly, implicit class loading would cause the class to be loaded from the classpath, which would defeat `RunFromURL`'s ability to dynamically load classes from a location chosen at runtime.

Listing 3–14 RunFromURL

```
//imports, error checking removed for brevity
public class RunFromURL {
  public static void run(String url, String clsName,
                         String[] args) throws Exception
  {
    URLClassLoader ucl = new URLClassLoader(new URL[]
                       {new URL(url)});
    Class cls = ucl.loadClass(clsName);
    Class argClass = String[].class;
    Method mainMeth = cls.getMethod("main",
                      new Class[]{argClass});
    mainMeth.invoke(null, new Object[]{args});
  }
  public static void main(String [] args) throws Exception
  {
    int argCount = args.length-2;
    String[] newArgs = new String[args.length-2];
    for (int n=0; n<argCount; n++) {
      newArgs[n] = args[n+2];
    }
    run(args[0], args[1], newArgs);
  }
}
```

3.3.2 Wrapping Primitive Types

The syntax shown so far works fine for object types, but what about methods and fields that utilize primitive types? Java includes a set of primitive types (`short`, `long`, `byte`, `boolean`, `double`, `float`, `int`, and `char`) that can be

[3] A production version should use the `Modifiers` class to make sure the method actually is static.

represented directly in memory without the overhead of extending Object. These types pose a problem for reflection because the invocation APIs have generic signatures based on Object and Object[]. To enable the use of primitive types in reflective invocation, each primitive type in Java has a corresponding immutable class in the java.lang package. If you want to use reflection to call a method that takes a primitive type, you must use an instance of one of the wrapper classes instead. So for a hypothetical method

```
int add(int n1, int n2);
```

you would use the following reflective syntax:

```
//to reflectively add ints n1 and n2
Object[] args = new Object[]{new Integer(n1),
                             new Integer(n2)};
Integer temp = (Integer) addMethod.invoke(args);
int result = temp.intValue();
```

Before calling add, you must “box” the int parameters as Integers. Then, you must take the result of the call, cast it to Integer, and then “unbox” it to an int. This is inconvenient but necessary given the dual nature of numeric types in Java.

A related issue is the process of requesting a method with primitive types in its signature. Given that reflective invocation must use Integer wherever int was originally specified, how can getMethod distinguish between the two signatures shown in Listing 3–15? Even though both of these methods would be reflectively invoked with the same argument types, there is a way to distinguish between them to getMethod and related methods. Integer, like any other Java class, can be represented by its corresponding pseudo-literal, in this case Integer.class. For reflection purposes, the primitive types, such as int, have a special Class representation that is the static TYPE constant of the corresponding wrapper class. So, you should use Integer.class to request an Integer but Integer.TYPE to request an int, as Listing 3–15 demonstrates.

Listing 3–15 Finding Methods That Use Primitive Types

```
//two methods differing only in wrapper vs. primitive
public Integer add(Integer n1, Integer n2);
public int add(int n1, int n2);
```

```
//to access the wrapper version
Class[] wcls = new Class[]{Integer.class,Integer.class};
Method methWrapper = cls.getMethod("add", wcls);

//to access the primitive version
Class[] pcls = new Class[]{Integer.TYPE,Integer.TYPE};
Method primitiveWrapper = cls.getMethod("add", pcls);
```

3.3.3 Bypassing Language Access Rules

The rules for dealing with primitive types are arbitrary but raise no interesting design questions. A more challenging design issue is deciding how to enforce language access rules during reflective invocation. In normal Java programming, member access is controlled by the `public`, `private`, and `protected` keywords, plus the implied package-private setting if no keyword is specified. If you attempt to access a member whose access is restricted, compilation will fail. With reflective invocation, there is no way to know the protection modifier of a method at compile time. At runtime, the virtual machine can use class metadata to determine the protection modifier, and it can possibly prevent the operation being attempted.

If a piece of code attempts to use reflection to access a member that it would not have been allowed to access from compiled Java code, should the operation be allowed to proceed? The initial intuition is "no!" There are at least two good reasons to enforce the language protection rules during reflective invocation. First, reflection should not act as a back door to compromise encapsulation. It is both a design flaw and a security risk to open private members to code outside of a class. Second, reflection should obey the common-sense rule of least surprise. If the language works a certain way during normal method invocation, reflective invocation should mimic that behavior to the extent possible.

3.3.3.1 Using setAccessible

Despite these arguments in favor of limiting reflection, reflective code sometimes needs to bypass Java's protection modifiers in order to implement low-level subsystems that provide services for arbitrary Java classes. For example, one use of the Reflection API is for generic persistence services for instances. A generic persistence layer that did not persist the entire state of an object, including private members, would not be very generic. However, the design of the Reflection API also addresses the concerns raised previously about blithely accessing private data. The rules for language-level access checks provide something for everybody:

1. By default, reflective invocation does *not* bypass language access checks. If you attempt to access a private, package-private, or protected member that you could not access normally, reflection will fail with an `IllegalAccessException`. This preserves the rule of least surprise.
2. If you want to bypass language-level access checks, you can request this ability by calling `setAccessible(true)` on an `AccessibleObject` (see Listing 3–16). The reflection classes `Field`, `Method`, and `Constructor` all extend `AccessibleObject`. This allows a service like an XML serializer to gain access to all of an object's state.

Listing 3–16 AccessibleObject

```
// from java.lang.reflect.AccessibleObject
package java.lang.reflect;
public class AccessibleObject {
  //request permission to bypass access checks
  public void setAccessible(boolean flag)
              throws SecurityException;
  //bypass access checks for several items in one shot
  public static void setAccessible(AccessibleObject[]
                     arr, boolean flag);
  //remainder omitted for clarity
}
```

You can control which code is allowed to call `setAccessible` by installing a `SecurityManager` for a process.[4] The `setAccessible` method provides a convenient chokepoint for a security check, and the default security implementation will prevent application code from calling `setAccessible`, while allowing system code to do so.

To see these rules in action, consider a `Reporter`'s attempts to discover a `Superhero`'s secret identity, shown in Listing 3–17. The `Reporter` wants to

[4] See [Gon99] for a detailed treatment of Java 2 Platform Security.

retrieve the `Superhero`'s `secretIdentity` field via reflection, thus bypassing the fact that the field is private. If you execute this code as is, it will fail with an `IllegalAccessException`. In order to bypass the language check, add a call to `secret.setAccessible(true)` before the call to `secret.get(s)`. With this call in place, reflection will bypass the language access check and return the private field. However, if you turn security on with the `java.security.manager` flag, the call to `setAccessible` will fail with an `AccessControlException` as it does here:

```
>java –cp classes –Djava.security.manager Reporter
java.security.AccessControlException: access denied
(java.lang.reflect.ReflectPermission suppressAccessChecks)
```

Listing 3–17 Misusing Reflection to Access a Private Field

```
public class Superhero {
public class Superhero {
  public final String name;
  private final String secretIdentity;
  public Superhero(String name, String secretIdentity) {
    this.name = name;
    this.secretIdentity = secretIdentity;
  }
}
public class Reporter {
  public static void main(String [] args)
         throws Exception {
    Superhero s = new Superhero("ReflectionMan",
                                "Brian Maso");
    hackIdentity(s);
  }
  public static void hackIdentity(Superhero s)
                     throws Exception {
    Field secret = Superhero.class.
                   getDeclaredField("secretIdentity");
    System.out.println("Identity is " + secret.get(s));
  }
}
```

To call `setAccessible` when security is enabled, you must have the `suppressAccessChecks` permission. By default, code that is in the core API or the extensions directory will have the `suppressAccessChecks` permission and be able to perform services such as serializing an object's private state. Application code loaded from the classpath or via a `URLClassLoader` will not have this permission, and therefore, it will be unable to inadvertently manipulate an object's implementation details.

3.3.3.2 Reflective Modification of Final Fields

Given that reflection sometimes has a legitimate reason to modify `private` fields, it is also logical to consider whether reflection might need to modify `final` fields. Before you look at how Java actually handles this today, consider the problem. Imagine a `Person` class that wanted to have some immutable fields:

```
public class Person {
  private final String name;
  private final String socialSecurityNo;
  //etc...
}
```

Now consider what would happen if you wanted to write a generic mechanism to instantiate `Person` from a row in a database. There are three approaches that might work:

1. You could use reflection to instantiate a `Person` and then assign its fields. However, since some of the fields are final, this would require that the reflection APIs be designed with the ability to modify final fields. Many people argue that this ability is counter to the very definition of `final`.
2. You could use a native method to instantiate a `Person` and then assign its fields. The Java Native Interface (JNI) includes APIs to do all sorts of things that would be illegal in Java, including instantiating objects without running their constructors and modifying final fields. The major downside to this approach is that you have to develop and deploy native code.
3. You could require that the `Person` class provide an "all-fields constructor" that would simply pass through an initial value for every single field in the object. Generic services, such as the hypothetical database code, would use reflection to invoke this special constructor. The constructor probably would not be used during normal operation, and in fact, you could mark it `private` to guarantee this. This approach has neither of the disadvantages of

the first two. Because constructors are allowed to set final fields, reflection does not need the ability to subvert final semantics. Also, no native code is required. The problem with this approach is that it requires the author of a class to explicitly write a special constructor in order to use serialization. The first two approaches simply leverage metadata and require no assistance from the classes that want generic services.

None of these solutions is ideal, and the Java language is still evolving in this area. Prior to SDK version 1.3, the first approach was used. The `setAccessible` method gave permission to modify `final` fields, and APIs such as serialization leveraged reflection. This led to some very confusing situations, as demonstrated in Listing 3–18.

Listing 3–18 Reflectively Modifying final Fields Is a Bad Idea.

```
import java.lang.reflect.*;
public class LimitedInt {
  public static void main(String[] args) throws Exception
  {
    System.out.println("Integer.MAX_VALUE=" +
                       Integer.MAX_VALUE);
    Field mv = Integer.class.getField("MAX_VALUE");
    mv.setAccessible(true);
    mv.set(null, new Integer(42));
    System.out.println("Retest: MAX_VALUE=" +
                       Integer.MAX_VALUE);
    System.out.println("Reflective: MAX_VALUE=" +
                       mv.get(null));
  }
}
```

Prior to SDK 1.3, `setAccessible` worked for `final` fields, and it was actually possible to change `Integer.MAX_VALUE`! This was made doubly confusing because Java compilers typically optimize `static final` primitive type declarations by inlining any access. So, even though you could change the value of `MAX_VALUE` seen by reflection, you could not change the value of `MAX_VALUE` where that constant was referenced in a class. On SDK 1.2, the code in Listing 3–18 prints the following:

```
Integer.MAX_VALUE=2147483647
Retest: MAX_VALUE=2147483647
Reflective: MAX_VALUE=42
```

SDK 1.3 removed reflection's ability to change `final` fields. In the case of `static final` fields, this was definitely a step in the right direction. However, this also broke the approach to serialization that leveraged reflection to populate instance fields for classes like `Person`. In order to get around this new limitation, services like serialization switched to the second approach detailed earlier, using native methods to populate fields. This is not likely to be the final word on the subject. The proposed changes to Java's memory model would remove even the ability of native code to modify `final` fields (see [JMM] for details). If the new model is adopted, then the third approach might become necessary; this would require class authors to provide a special constructor for serializable classes that have `final` fields.

3.3.4 Exceptions Caused by Reflective Invocation

As mentioned above, attempts to bypass language restrictions without calling `setAccessible(true)` will generate an `IllegalAccessException`. `IllegalAccessException` is one of several exceptions that reflective invocation throws to indicate a problem that normal code would catch at compile time. Another example is `IllegalArgumentException`, used to indicate that a method argument is invalid. This will occur if one of the `Objects` in the array passed to `invoke` cannot be converted to the correct type for the underlying method. If you try to use reflection to instantiate an abstract class, `Class` will throw an `InstantiationException`.

Taken together, these three exception types are the reflection equivalent of the compiler's enforcement of language rules. While `setAccessible` can bypass the access rules, there is no way to get around the situations that cause an `IllegalArgumentException` or an `InstantiationException`. You must always pass arguments of the correct type, and you can never instantiate abstract classes.

Another situation where reflection may throw an exception is if the underlying method or constructor throws an exception. If this occurs, one of three things will happen:

1. If a method throws an unchecked exception, such as an `Error` or `RuntimeException`, then this exception will be passed unmodified through the reflection layer to the caller. This behavior is no different from ordinary method invocation, where unchecked exceptions do not require an explicit catch block.
2. A method may throw a declared checked exception. This is a problem for reflection because checked exceptions must be declared in a method signature, and reflection does not know a method's signature at compile time. The reflection APIs catch checked exceptions and convert them into a checked type `InvocationTargetException` that is declared in the signature of `Method.invoke`. This situation occurs frequently and leads to a special idiom on the part of the caller, as shown in Listing 3–19. `InvocationTargetException` is a reflection-provided wrapper class that can access the original exception via `getTargetException`. A client is typically more interested in the original exception, which indicates what went wrong, than it is in the `InvocationTargetException`, which only indicates that reflection was involved. As the sample shows, reflective clients will frequently catch `InvocationTargetException`, extract the original exception, cast it to some type expected by the caller, and rethrow.
3. A method may throw a checked type that it did *not* declare. If this happens, `invoke` will convert the exception into the unchecked type `UndeclaredThrowableException`. This situation is extremely rare since the compiler will prevent a class from compiling if it tries to throw a checked exception that it did not declare. `UndeclaredThrowableException` indicates either a version mismatch caused by a partial recompile, or a corrupted binary class.

Listing 3–19 Extracting the "Real" Exception

```
//invokes some I/O method unknown at compile time
public void reflectiveIO(Method m, Object this,
            String fileName) throws IOException
{
  try {
    m.invoke(this, new Object[]{filename});
```

```
      }
      catch (InvocationTargetException ite) {
        throw (IOException) ite.getTargetException();
      }
    }
```

3.4 Dynamic Proxies

Dynamic proxies, available since SDK 1.3, are the exact opposite of reflective invocation. With reflective invocation, a client uses a generic API to call methods (on a server class) that are not known at compile time. With dynamic proxies, a server uses a generic API to implement methods on a server class that is manufactured at runtime to meet a client's specification. Dynamic proxies are chameleons that take the shape of any set of interfaces desired by the client. When combined with reflective invocation, dynamic proxies can implement *generic interceptors.* An *interceptor* is a piece of code that sits between a client and a server and adds an additional service, such as transaction enlistment, auditing, security checking, or parameter marshalling. A *generic interceptor* requires no compile-time knowledge of the client or server APIs being intercepted.

3.4.1 Delegation instead of Implementation Inheritance

Generic interception makes it possible to use an object-oriented style based not just on inheritance, but also on delegation. Consider an entity class that accesses employee information from a database. If you wanted to use inheritance to layer in transaction enlistment, auditing, and security checks, you would need to use multiple inheritance. After pointing out that multiple implementation inheritance does not exist in Java, you might also object to the fact that this design would require eight different concrete subclasses of `Employee`, one each for every possible service being turned on or off, as shown in Figure 3–3. In general, adding another service doubles the number of concrete subclasses you need.

With delegation, each new service adds only one class, and classes are simply chained together to provide the exact mix of services needed, as shown in Figure 3–4.

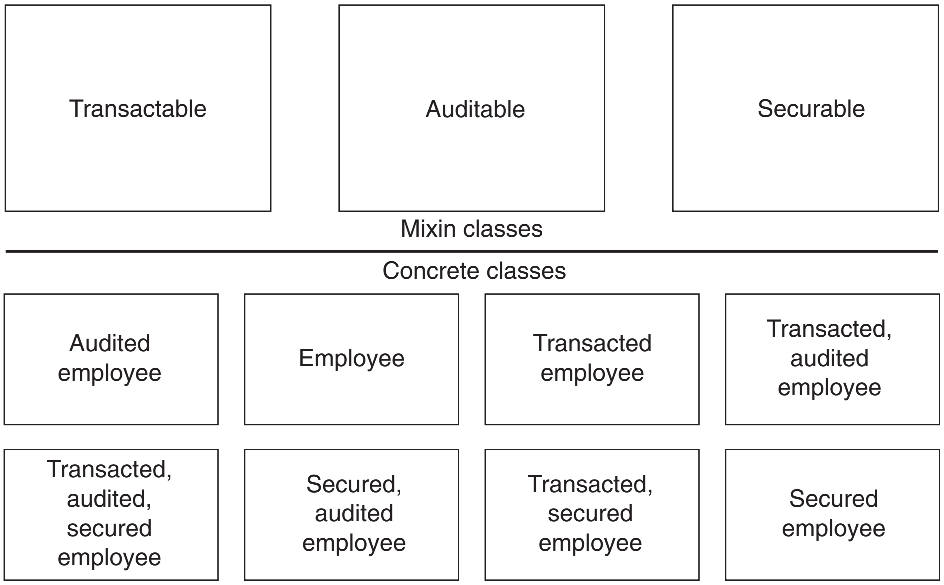

Transactable
Auditable
Securable
Mixin classes
Concrete classes
Audited employee
Employee
Transacted employee
Transacted, audited employee
Transacted, audited, secured employee
Secured, audited employee
Transacted, secured employee
Secured employee

Figure 3–3 Implementation inheritance causes class proliferation.

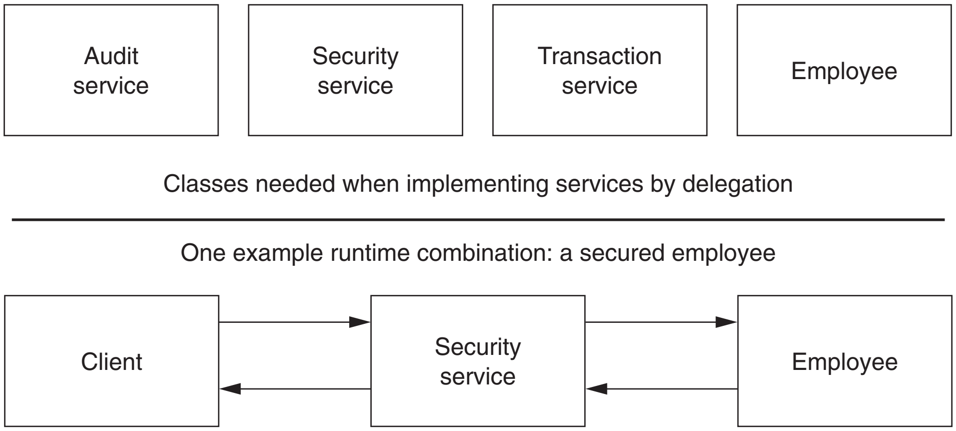

Audit service
Security service
Transaction service
Employee
Classes needed when implementing services by delegation
One example runtime combination: a secured employee
Client
Security service
Employee

Figure 3–4 Delegation requires only one class per service.

3.4.2 Dynamic Proxies Make Delegation Generic

Historically, the problem with the delegation model was how to make each service generic. For example, an `Audit` class would need to implement `EmployeeItf` when it was dealing with an `Employee` entity, but it would need to implement InventoryItf when it was dealing with the Inventory entity. Also, the service classes would need to implement interfaces that had not even been designed yet. Dynamic proxies neatly solve this problem by allowing the transaction class to manufacture an implementation of whatever interface the client expects at runtime.

To manufacture a dynamic proxy, you need only call Proxy.newProxyInstance, passing in an implementation of the InvocationHandler interface. Summaries of these classes appear in Listing 3–20. The newProxyInstance method manufactures a new binary class in memory. This special class implements all the interfaces passed in the itfs array by forwarding every single method to an instance of InvocationHandler. Then, the new class is loaded into the virtual machine by the class loader specified by ldr, and it is used to construct a proxy instance that forwards to handler.

Listing 3–20 Key Elements of Proxy and InvocationHandler

```
package java.lang.reflect;
public class Proxy {
  static Object newProxyInstance(ClassLoader ldr,
          Class[] itfs, InvocationHandler handler)
                throws IllegalArgumentException;
  //remainder omitted for clarity
}
public interface InvocationHandler {
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable;
}
```

3.4.3 Implementing InvocationHandler

Dynamic proxies allow you to implement a single invoke method on the invocation handler and then use it to service any interface you choose at runtime. For example, consider an InvocationHandler that logs method calls, shown in Listing 3–21. This LoggingHandler class provides a trivial implementation of any interface that simply logs method calls as they are made. DemoLogging demonstrates using the LoggingHandler to log calls to DataOutput. One possible use for LoggingHandler is during development, when you need to stub out an interface that you have not yet implemented.

Notice that the toString method is treated specially. In addition to any interface methods, dynamic proxies always forward the Object methods toString, hashCode, and equals to the handler. In this case, the proxy's toString method is invoked by the handler's call to System.out.println. If the toString method were not special-cased, the call to toString would trigger the invoke method of LoggingHandler, which triggers another call to toString, and so on, recursing until the stack overflowed.

Listing 3–21 LoggingHandler

```
import java.lang.reflect.*;
public class LoggingHandler implements InvocationHandler {
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable
{
    if (method.getName().equals("toString")) {
      return super.toString();
    }
    System.out.println("Method " + method +
                       " called on " + proxy);
    return null;
  }
}
import java.lang.reflect.*;
import java.io.*;
public class DemoLogging {
  public static void main(String [] args)
    throws IOException
  {
    ClassLoader cl = DemoLogging.class.getClassLoader();
    DataOutput d = (DataOutput) Proxy.newProxyInstance(cl,
                                new Class[] {DataOutput.class},
                                new LoggingHandler());
    d.writeChar('a');
    d.writeUTF("stitch in time");
  }
}
```

3.4.4 Implementing a Forwarding Handler

Although "standalone" dynamic proxies such as LoggingHandler are useful, they suffer from a major limitation in dealing with return values. Because they are totally generic, InvocationHandlers have no idea how to generate a legitimate return value for a method call. LoggingHandler finesses this issue by always returning null, which the generated proxy class will coerce to the return type of the interface method. In Listing 3–21, the DataOutput methods happen to return void, so the generated proxy simply ignores the return from LoggingHandler's invoke method. This coincidence will not hold up in more complex cases, those in which methods might return any type, and the compile-time type might need to be further constrained at runtime in accordance with the documented semantics of the method. In order to reasonably mimic any arbitrary interface, a dynamic proxy will either need to know the semantics of the interface, or it will need to forward the method call to some other object that does. Since the raison d'être of a proxy is to be generic, knowing the specifics on an interface is not an option. Instead, most dynamic proxies are used to forward calls to other objects.

The strength of dynamic proxies is method call forwarding. A dynamic proxy can intercept a method call, examine or modify the parameters, pass the call to some other object, examine or modify the result, and return that result to the caller. When correctly configured, dynamic proxies work transparently without the knowledge of either the client or server code.

Figure 3–5 shows how a dynamic proxy enables generic services. A generic service implements only one method, InvocationHandler.invoke, and forwards the call using only one method, Method.invoke. Without changing, or even reading, any existing implementation code, you can insert a dynamic proxy between two objects to inject some additional service.

3.4.5 The InvocationHandler as Generic Service

To appreciate the power and simplicity of this model for reuse, imagine the following scenario: A large server application has been ported to Java and continues to access legacy code through a bridge that presents the legacy code as a set of Java interfaces. Unfortunately, the legacy code was written in a pointer-based language and experiences occasional memory corruption. The specific symptom is that methods sometimes return the java.util.Date

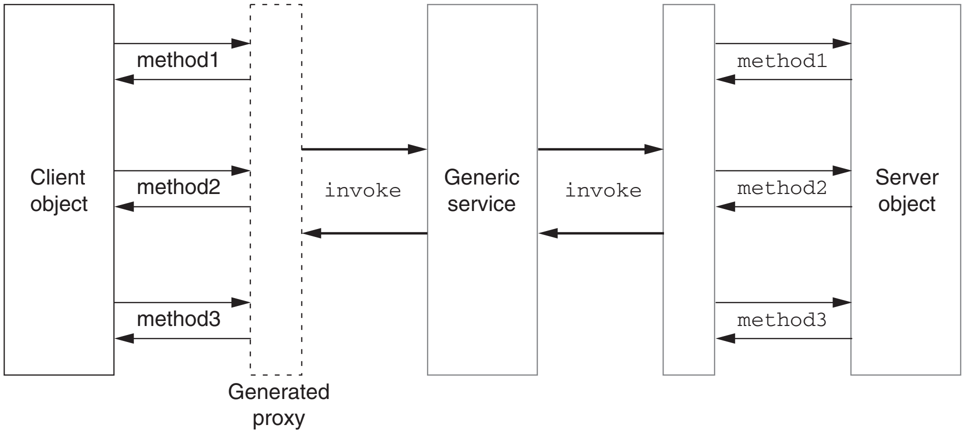


Figure 3–5 Dynamic proxies enable generic services.

"Thu Dec 25 07:42:41 EST 1969."[5] Your task is to guarantee that this bug does not introduce corrupt data into the application.

With dynamic proxies, you can add an interceptor that traps all attempts to return the offending value, as is shown in Listing 3–22. Here, a `TrappingHandler` forwards all method calls to its `delegate`. If the `delegate` functions unexceptionally and returns an object, the handler checks to see if it is a `Date` instance indicative of the memory corruption bug. If it is, then the handler might throw an error, as the example shows, or it might take whatever other action may be necessary.

Listing 3–22 TrappingHandler

```
import java.lang.reflect.*;
import java.util.*;
public class TrappingHandler implements InvocationHandler {
  //BAD_DATE is "Thu Dec 25 07:42:41 EST 1969"
  //This value corresponds to an error code on some systems
  public static final Date BAD_DATE = new Date(-559038737);
  private Object delegate;
  public TrappingHandler(Object delegate) {
    this.delegate = delegate;
  }
```

[5] Bonus points will be awarded for figuring out why this particular date value is likely to indicate memory corruption.

```
  public Object invoke(Object proxy, Method method,
                 Object[] args) throws Throwable {
    Object result = null;
    try {
      result = method.invoke(delegate, args);
    } catch (InvocationTargetException e) {
      throw e.getTargetException();
    }
    if (result instanceof Date) {
      Date d = (Date) result;
      if (d.equals(BAD_DATE)) {
       throw new Error("Corrupted date " + d);
      }
    }
    return result;
  }
}
```

3.4.6 Handling Exceptions in an InvocationHandler

It is important to code carefully against the possibility that the `delegate` will itself throw an exception. Because the handler is generic, you have no compile-time knowledge of what particular exceptions the `delegate` might throw. However, since the `delegate`'s methods are invoked via reflection, any exception will be wrapped by an `InvocationTargetException`. You should not let this exception percolate out of the dynamic proxy code.

The dynamic proxy provides some help here. Since a proxy wants to be transparent to the client, it will only permit exceptions that the client expects. To enforce this, a proxy compares any thrown exception to the list of checked exceptions for the current method. If an exception is not in the list, a proxy will wrap it in an `UndeclaredThrowableException`, which is a `RuntimeException` subclass that signals a programmer error.

To complete the illusion of transparency, your `InvocationHandler` must not give the proxy any reason to throw an `UndeclaredThrowable Exception`. Therefore, the canonical implementation of a forwarding proxy includes a try/catch block that catches the `InvocationTargetException` and then extracts and throws the underlying exception, which is the one the client expects. Refer back to Listing 3–22, which demonstrates this.

3.4.7 Either Client or Server Can Install a Proxy

Either the client or the server code can wrap suspicious objects with the `TrappingHandler` before using them. In the `TestTrappingHandler` example (Listing 3–23), the client wraps an instance of the Test interface. If either `getGoodDateValue` or `getBadDateValue` of the Test interface return a bad date, the handler will protect the client by throwing an exception. More importantly, the same `TrappingHandler` can be used throughout a system to protect any number of different interfaces and implementation classes.

You could make the proxy code transparent to both client and server by using an object factory to hide the details of connecting client and server code.

Listing 3–23 Testing a TrappingHandler

```
//Test.java
import java.util.*;
interface Test {
  Date getGoodDateValue();
  Date getBadDateValue();
}

//TestTrappingHandler.java
import java.lang.reflect.*;
import java.util.*;

public class TestTrappingHandler implements Test {
  public static void main(String [] args) {
    TestTrappingHandler t = new TestTrappingHandler();
    System.out.println("Testing unwrapped object.\n" +
                       "This should permit date value " +
                       TrappingHandler.BAD_DATE);
    executeTests(t);
    Test wrap = (Test)Proxy.newProxyInstance(
    TestTrappingHandler.class.getClassLoader(),
    new Class[]{Test.class}, new TrappingHandler(t));
    System.out.println("Testing wrapped object.\n" +
                       "This should reject date value " +
                       TrappingHandler.BAD_DATE);
    executeTests(wrap);
  }
  public Date getGoodDateValue() {
    return new Date();
  }
  public Date getBadDateValue() {
    return TrappingHandler.BAD_DATE;
  }
  public static void executeTests(Test t) {
    System.out.println(t.getGoodDateValue());
    System.out.println(t.getBadDateValue());
  }
}
```

3.4.8 Advantages of Dynamic Proxies

Dynamic proxies do not provide any service that you could not provide yourself by hand-coding a custom delegator class for every different interface. The advantage of dynamic proxies over hand-rolled delegators is twofold. First, you do not have to write dynamic proxies. Second, they can be generated on-the-fly at runtime to handle new interfaces as they appear. Because dynamic proxies are generic, they tend to be used for services that do not rely on any specific knowledge of the interface or method being forwarded:

- Parameter validation or modification, where a parameter value is of interest regardless of where it appears, as in the example above
- Security checks, some of which can be made based on the identity of the user or the source of the code, not the particular method being called
- Propagation of “implicit” or “context” parameters, such as the transaction ID, which is automatically handled by an EJB container and does not appear in any interface declaration
- Auditing, tracing, or debugging of method calls
- Marshalling a Java method call to some other process, machine, or language

Like any generic tool, dynamic proxies may be inefficient compared to a solution hand-tuned to a particular problem. Situations in which performance considerations may rule out the use of dynamic proxies are discussed in §3.5.

3.5 Reflection Performance

Reflective invocation is slower than direct method invocation. Similarly, using dynamic proxies as a generic forwarding device results in slower code than manually crafting interface-specific delegation code does. When is the performance penalty acceptable? Programmers who are used to dynamic, interpreted runtime environments are already accustomed to dynamic programming styles and see uses for reflection everywhere. On the other hand, programmers who are used to the performance of a strongly typed, compiled language often laugh at the slow speed of reflection and reject it outright.

The truth lies somewhere in between. Reflection is best suited for writing "glue"—code that sits at the boundaries between disparate classes, packages, and subsystems. If your task description includes words such as "adapter" or "decorator," then reflection may be a good fit. But reflection is not well suited for code on the critical path in an application. If your task vocabulary tends more toward "inner loop" or "heavy recursion," avoid reflection. This section presents some order-of-magnitude estimates of reflection performance, and then it gives some specific examples of where it should and should not be used.

Before directing your attention to some performance numbers, I must begin with some major caveats about their use. Evaluating a system's performance is a tricky task. Many factors impact the execution speed of even a simple Java application: virtual machine, processor speed, available memory, memory speed, OS, OS version, and other applications that are also running.

Because of these complexities, measurement is essential. Do not rely on intuition. Programmers' intuitions about performance are reliable only to within about six orders of magnitude. If you need better precision than that, you must measure. In Java, even measurement can be tricky. The virtual machine introduces yet another layer of indirection between you and the Platonic ones and zeroes. As of SDK version 1.3, the virtual machine (HotSpot) is adaptive over time, so short-run tests can grossly misrepresent long-run behaviors.

Fortunately, there is some value even in very rough measurements. For the purpose of deciding where reflection might fit into a program design, it is sufficient to have some order-of-magnitude, relative measurements. The test harness used to make the following measurements is the `Timer` class from the com.develop.benchmark package, which is included in the sample code for this book [Hal01].

Table 3–2 Rough Estimate of Reflection Performance

Operation Tested	Time (Nearest Power of 10 nsec)
Increment integer field	10^0
Invoke a virtual method	10^1
Invoke through a manual delegate	10^1–10^2
Reflective method invocation*	10^{4-}–10^5
Invoke through a proxy delegate	10^4–10^5
Reflectively increment integer field	10^4–10^5
RMI call on a local machine	10^5
Open, write, close a file	10^6
Light travels 3000km	10^7

*Reflective method invocation actually involves three steps: getting a Method object with a call to getMethod, boxing the arguments into an array of Object, and calling invoke. Many reflection-based systems make only a single call to getMethod followed by many calls to invoke, so it may not always be appropriate to include the getMethod overhead in the per-call timings. In my tests, the costs with getMethod included were closer to 10^5 nsec, and the costs just for boxing the arguments and calling the method were closer to 10^4 nsec.

Table 3–2 lists order-of-magnitude comparisons for reflective and nonreflective tasks. These tests were made on the HotSpot 1.3 client virtual machine, with JIT enabled, on a Pentium2-450 running Windows NT 4.0 server. However, the conclusions I plan to draw are very limited, and they should hold for a wide variety of virtual machines and hardware. First, notice that direct access to a field and direct invocation of a virtual method are very fast, in the ones and tens of nanoseconds respectively. This is getting reasonably close to a single operation per processor tick since the P450's clock ticks in a little over two nanoseconds. When you switch to reflective field access, reflective method invocation, or invocation through a dynamic proxy, you will notice that they all impose a stiff penalty, taking in the tens of microseconds to execute. This is the statistic that causes systems programmers to scoff and abandon reflection.

When you look a little further down the table, you will see some reasons for a more optimistic assessment. Though reflection may be slow compared to direct access to class members, it is quite fast compared to many other common programming tasks. A simple, cross-process method invocation on a local machine is an order of magnitude slower than a reflective invocation. Opening, writing, and closing a file is another order of magnitude slower than that. Hundreds of reflective operations could transpire in the time it takes light to travel 3,000km, which in turn, is likely faster than the time in which network packets can travel between your computer and some location on the Internet. A single online transaction involves several of these expensive operations: invoking methods cross-process, reading and writing files, and routing packets around the Internet. A single reflective method call would have a negligible impact on an Internet-based transaction.

The performance numbers could vary wildly on different virtual machines, and they are not intended to guide optimization decisions other than at the broadest level. What they do show, however, is that reflective access is not onerous if it is used sparingly in an application that is also doing interesting work. Moreover, reflection performance should improve substantially in the 1.4 SDK release. If you can easily make your design work without resorting to reflection, then do so. But do not be alarmed about performance if reflection appears to be the most convenient glue between subsystems in your application.

3.6 Package Reflection

Most of the Reflection API deals with class-level metadata. However, version information is another form of metadata, and it plays a critical role as code evolves; it makes sure that client code accesses only compatible versions of needed classes. Version metadata is also useful when you need to know the exact version of software that caused a problem. In Java, version information is typically tracked at the level of a package, and it is stored at the level of the JAR file. This is reasonable because the next smaller unit, the class file, is typically too small to be the standard unit of versioning or deployment. Version information is embedded in JAR files, added to the virtual machine by a class loader, and accessed programmatically via the `java.lang.Package` class.

3.6.1 Setting Package Metadata

In order to use the version metadata currently provided by Java, you must deploy your code as a JAR file instead of as separate class files. Adding version information to a JAR file is a simple matter of adding name/value pairs to the manifest file. For example, Listing 3–24 specifies all the possible version fields for a hypothetical `com.develop.hello` package.

Listing 3–24 Manifest Entries for Package Versioning

```
Manifest-version: 1.0
Name: com/develop/hello/
Specification-Title: Hello World
Specification-Version: 1.0.0
Specification-Vendor: DevelopMentor
Implementation-Title: com.develop.hello
Implementation-Version: build1
Implementation-Vendor: DevelopMentor
```

The manifest information is added to the JAR file with the `–m` switch. Assuming the file above is named hellov1.mf, you might create a JAR file with the command

```
jar -cmf hellov1.mf hellov1.jar com/develop/hello/Main.class
```

When a `URLClassLoader` loads classes from a JAR file that includes version information, it registers the version information with the virtual machine by calling the `definePackage` method defined by the `ClassLoader` class, shown in Listing 3–25. This method takes the name of the package, the six well-known version information strings, and the `sealBase` (more on sealing in §3.6.3).

Listing 3–25 The definePackage Method

```
package java.lang;
public class ClassLoader {
protected Package definePackage(
                  String name, String specTitle,
                  String specVersion, String specVendor,
                  String implTitle, String implVersion,
                  String implVendor, URL sealBase);
//other methods omitted for clarity
}
```

3.6.2 Accessing Package Metadata

The virtual machine makes package metadata available to code at runtime through the `Package` class. Included in the `Package` class are accessor methods for the six version info strings, plus two methods for looking up the packages known to the virtual machine, as shown in Listing 3–26.

Listing 3–26 Querying Package Metadata

```
package java.lang;
public class Package {
  public String getSpecificationTitle();
  public String getSpecificationVersion();
  public String getSpecificationVendor();
  public String getImplementationTitle();
  public String getImplementationVersion();
  public String getImplementationVendor();
  public static Package getPackage(String packageName);
  public static Package[] getPackages();
  public Boolean isCompatibleWith(String desired);
  //additional methods omitted for clarity
}
```

The version strings provide a minimal infrastructure you can use as a starting point when you are developing versioning semantics for your applications. All six well-known version strings can be set to any arbitrary String value; none has any semantics that are enforced by the current version of the SDK.

One of the six version strings provides a documented semantic. If you set the `specVersion` value to a dotted string, such as 1.0.2, you can use the `isCompatibleWith` API to see if a package is compatible with a particular version number. The specification version should be a dotted number such as 1.0.5, and the compatibility check uses a simple definition of compatibility, in which higher-numbered versions are always compatible with lower numbered ones. Although this usage of `specVersion` is documented, it is not enforced by the platform. For the other version strings, no semantics are even documented, and you can define any semantic that is convenient for you.

3.6.3 Sealing Packages

JAR files also support the process of *sealing* a package. When you seal a package, you guarantee that all the code from that package must come from the same location. This is valuable for versioning and security because it allows you to guarantee that your packages are not polluted by invalid or malicious versions of any classes. To seal a package, you add a `Sealed: true` pair to the metadata, somewhere after a package's `Name` field and before the next blank line, like so:

```
Name: com/develop/hello/
Sealed: true
```

You can also seal all packages in a JAR file by adding the `Sealed` entry to the main section of the manifest. For package sealing to take effect, you must use a class loader that honors the metadata in the JAR file, such as your good friend `URLClassLoader`.

Sealing all of your packages is a *very good idea.* Consider the following, all-too-common scenario. Version 2.0 of an application modifies several classes from version 1.0 while it also adds some new classes. Unfortunately, JAR files for both version 1.0 and version 2.0 are on the classpath. As a result, the virtual machine loads a mix of version 1.0 and version 2.0 classes, a sure recipe for trouble and confusion. If either version had sealed its packages, this configuration problem would have triggered an easily diagnosed error. Unless you specifically want to load package code from more than one JAR, always seal all application packages.

3.6.4 Weaknesses of the Versioning Mechanism

There are several problems with the versioning mechanism as it exists today. First, though the class loaders provided with the core API correctly propagate version information as described above, they do not automatically reject incompatible versions of a package, nor do they seek out compatible ones. Second, the `URLClassLoader` and subclasses do not load package information until *after* the first class in a particular package is loaded, so you have to load at least one class in a package before you can find out whether the package version actually meets your needs. Third, the text format for the manifest is not adequately validated; so for example, a spelling error in the name of a version field silently obliterates that field's information. The 1.4 version of Java is slated to have a built-in XML parser, which would permit a more structured manifest format. For the other problems, you will have to wait a while longer or address them in your own code. (§5.5.1 presents a more robust approach to versioning based on custom class loaders.)

3.7 Custom Metadata

The type information stored in a Java class file is very thorough, as far as it goes. When you first move to Java from a nonreflective programming environment, the new possibilities seem limitless. Knowing the names and types of all methods and fields makes it easy to implement all sorts of runtime services for your Java objects: XML views, object/relational mappings, generic user interfaces, and on and on. Nevertheless, it is possible to imagine wanting even *more* metadata.

Consider the hypothetical `LaunchVehicle` interface shown in Listing 3–27. As a human reader, you can infer several important details about how to use this interface. For example, you know to use liters when you `addFuel`. From your knowledge of the problem domain, you know that you should always `countdown` before you `launch`. These are important, contractual elements of the interface, but they do not have a standard language representation and are not part of the class metadata. You cannot count on clients always getting these details correct. Even if you could, some other important details of the design are *not* obvious to the reader. What units are to be used when calling `thrust`? Is it acceptable to `addFuel` during the `countdown`? This example illustrates the need for two kinds of metadata not available to the Java language: the correct units for numeric arguments, and tables of state transitions allowed by an interface. If these metadata elements were added to Java, the virtual machine could enforce the rules for you, eliminating two more sources of program errors.

Listing 3–27 The LaunchVehicle Interface

```
public interface LaunchVehicle {
  public void addFuel(int liters);
  public void countdown(int seconds);
  public void launch(int thrust);
  //etc.
}
```

It would be unreasonable to expect a Java virtual machine to support every possible flavor of useful class metadata. Virtual machines would need to be far more complex than they are today, and classes would carry around enormous amounts of metadata not useful to their problem domain. It was good design to limit the scope of the virtual machine's responsibilities; the line had to be drawn somewhere.

Fortunately, the virtual machine specification offers a hook for customization by permitting the addition of custom *attributes* to the class file format. Attributes can be any binary data, and they are housed in a data structure called an `attribute_info`. The `attribute_info` structure, shown in Listing 3–28, contains a constant pool index to a string that names the attribute, plus an opaque array of bytes containing the attribute's data. You can attach attributes to classes, methods, fields, or even to the bytecodes that implement a method.

Listing 3–28 The attribute_info Structure

```
//pseudocode from the JVM spec
attribute_info {
  u2 attribute_name_index;  //reference to constant pool
  u4 attribute_length;
  u1 info[attribute_length]; //custom data
}
```

Figure 3–6 is an expanded view of the binary class format diagram, originally presented in Figure 3–1, with custom attributes shown below the solid line. The virtual machine spec already defines some standard attributes for its own use. The bytecodes that implement a method are stored as an attribute, which can in turn have custom subattributes. The standard attributes defined by the specification are listed in Table 3–3.

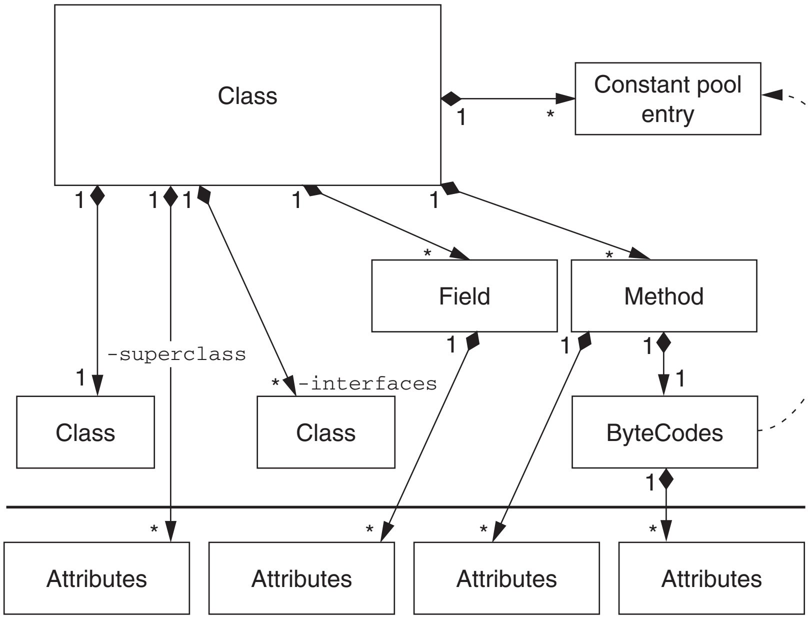


Figure 3–6 Custom metadata in the binary class format

Table 3–3 Attributes Defined by the Virtual Machine Specification

Attribute Name	Attibute Purpose
Code	Holds bytecodes that implement a method
ConstantValue	Holds value used to initialize a constant field
Exceptions	Lists checked exceptions a method may throw
InnerClasses	Links nested classes and outer classes
Synthetic	Marks methods not present in original source file
SourceFile	Holds name of original source file
LineNumberTable	Maps bytecode offsets to source line numbers
LocalVariableTable	Maps variables to source variable names
Deprecated	Marks deprecated class, field, or method

The Code, ConstantValue, and Exceptions attributes contribute to the documented semantics of class files and must be understood by conformant virtual machine implementations. The InnerClass and Synthetic attributes contribute to the semantics of the core API libraries and will therefore also be understood by compliant virtual machines.

The remaining standard attributes provide useful information, but they are not essential for the correct functioning of the language. For example, the SourceFile, LineNumberTable, and LocalVariableTable attributes enable source-level debuggers. Because these attributes are optional, they can be silently ignored by virtual machines that do not understand them. Any custom attributes that you develop must also be optional.

If a custom attribute were necessary for a class to function correctly, then a standard virtual machine would be unable to load a class that relied on the custom attribute. Such a custom attribute would encourage developers to write nonportable Java classes, defeating the write-once, run-anywhere nature of the language. It is legal to write custom tools that manipulate or even mandate custom attributes, but virtual machine implementations must silently ignore attributes that they do not recognize. The rules limit custom attributes to *optional* data that adds value when it is recognized by the virtual machine but does not break functionality when it is not. Even when they are operating within this constraint, custom attributes have many uses. They can store domain-specific metadata, debugging information, profiling information, documentation, and JIT optimization hints or hints that support interoperation with other programming environments.

Standard Java compilers will not emit your custom attributes, and the Reflection API provides no support for accessing them. If you want to define and use custom attributes, you will need to develop a development tool that injects attributes, a custom class loader that tracks attributes as classes are loaded, and extensions to reflection that can access these attributes at runtime.

Custom attributes are not in wide use today because they receive only limited support from the standard Java tools. Java compilers typically do not emit custom attributes, which is not surprising since there is no Java syntax for defining them. Programmers who want to add custom attributes must write their own tools to crack the class file format and inject attributes. Such tools are easy to write, and many are freely available, but none have the blessing of being standardized.

You are also on your own for extracting custom attributes. The Reflection API currently does not define methods for extracting custom attributes, although such methods are easy to design and add. Listing 3–29 shows part of the Java Class File Editor (JCFE). JCFE is an open source library developed by the author for manipulating custom attributes; see [JCFE] for details. The `ClassEx` augments the standard `java.lang.Class` class to include access to custom attributes, and the `Attribute` base class (not shown here) can be used as a subclass for different custom attributes that you design.

Listing 3–29 JCFE Methods for Custom Attributes

```
package com.develop.reflect;
public class ClassEx {
  public void addAttribute(Attribute ca);
  public Attribute[] getAttributes(String name);
  public Attribute[] getAttributes();
}
```

In order to access custom attributes at runtime, you need a custom class loader to extract them. A class loader sees the class file as a byte array during `findClass`, which provides a hook for manually parsing the class file and remembering any custom metadata. §5.5 develops a full example of this technique, using custom attributes and a custom class loader to automatically locate the correct version of a Java class.

Because the virtual machine must already parse the binary class format, parsing the class file a second time in a custom loader is inefficient. In an ideal world, access to custom attributes belongs in the platform, not in custom loaders. Hopefully a future version of Java will move custom attribute support into the core API.

3.8 Onward

Java reflection preserves name and type information in the compiled class format. This is an evolution from many pointer-based programming languages, which make compile-time assumptions about name and type that might fail to be true at runtime. The availability of class metadata makes the dynamic linking of code more reliable.

Most of the metadata in the class file is also exposed programmatically, via the Reflection API. Reflection includes both discovery and invocation. Reflective invocation can be used to write generic services that adapt themselves to the types they discover at runtime. Dynamic proxies extend the notion of a generic service further by allowing service classes to be manufactured at runtime. While reflective access is noticeably slower than direct access, it is fast enough to be used as the glue code between component, process, or network boundaries.

3.9 Resources

For more information on the binary class format, you should go straight to the source, and read the most recent version of the Java Virtual Machine Specification, which is [LY99] at the time of this writing. Although the virtual machine spec is most valuable for virtual machine developers, it is an invaluable read for all Java programmers.

To the author's knowledge, there are no books providing a treatment of the Reflection API that is complementary to the material presented here. There are, however, several tools that read and manipulate the binary class format, including [JCFE] and [CFParse].