Reflection

In this section, we'll take a look at the Java Reflection API, supported by the classes in the `java.lang.reflect` package. As its name suggests, reflection is the ability for a class or object to examine itself. Reflection lets Java code look at an object (more precisely, the class of the object) and determine its structure. Within the limits imposed by the security manager, you can find out what constructors, methods, and fields a class has, as well as their attributes. You can even change the value of fields, dynamically invoke methods, and construct new objects, much as if Java had primitive pointers to variables and methods. And you can do all this on objects that your code has never even seen before. In Java 5.0, the Annotations API also has the ability to preserve metadata about source code in the compiled classes and we can retrieve this information with the Reflection API. We don't have room here to cover the Reflection API fully. As you might expect, the `reflect` package is complex and rich in details. But reflection has been designed so that you can do a lot with relatively little effort; 20% of the effort gives you 80% of the fun. The Reflection API is used by JavaBeans to determine the capabilities of objects at runtime. It's also used at a lower level by object serialization to tear apart and build objects for transport over streams or into persistent storage. Obviously, the power to pick apart objects and see their internals must be zealously guarded by the security manager. The general rule is that your code is not allowed to do anything with the Reflection API that it couldn't do with static (ordinary, compiled) Java code. In short, reflection is a powerful tool, but it isn't an automatic loophole. By default, an object can't use it to work with fields or methods that it wouldn't normally be able to access (for example, another object's private fields), although those privileges can be granted, as we'll discuss later. The three primary features of a class are its fields (variables), methods, and constructors. For purposes of describing and accessing an object, these three features are represented by separate classes in the Reflection API: `java.lang.reflect.Field`, `java.lang.reflect.Method`, and `java.lang.reflect.Constructor`. We can look up these members of a class through the `Class` object. The `Class` class provides two pairs of methods for getting at each type of feature. One pair allows access to a class's public features (including those inherited from its superclasses) while the other pair allows access to any public or nonpublic item declared directly within the class (but not features that are inherited), subject to security considerations. Some examples:

- `getFields( )` returns an array of `Field` objects representing all a class's public variables, including those it inherits.
- `geTDeclaredFields( )` returns an array representing all the variables declared in the class, regardless of their access modifiers, but not including inherited variables.
- For constructors, the distinction between "all constructors" and "declared constructors" is not meaningful (classes do not inherit constructors), so `getConstructors(`

`)` and `geTDeclaredConstructors( )` differ only in that the former returns public constructors while the latter returns all the class's constructors.

Each set of methods includes the methods for listing all the items at once (for example, `getFields( )`) and an additional method for looking up a particular item by name andfor methods and constructorsby signature (for example, `getField( )`, which takes the field name as an argument). The following listing shows the methods in the `Class` class:

Field [] getFields( );

Get all public variables, including inherited ones.

Field getField(String name);

Get the specified public variable, which may be inherited.

Field [] getDeclaredFields( );

Get all public and nonpublic variables declared in this class (not including those inherited from superclasses).

Field getDeclaredField(String name);

Get the specified variable, public or nonpublic, declared in this class (inherited variables not considered).

Method [] getMethods( );

Get all public methods, including inherited ones.

Method getMethod(String name, Class ... argumentTypes);

Get the specified public method whose arguments match the types listed in `argumentTypes`. The method may be inherited.

Method [] getDeclaredMethods( );

Get all public and nonpublic methods declared in this class (not including those inherited from superclasses).

Method getDeclaredMethod(String name, Class ... argumentTypes);

Get the specified method, public or nonpublic, whose arguments match the types listed in `argumentTypes`, and which is declared in this class (inherited methods not considered).

Constructor [] getConstructors( );

Get all public constructors of this class.

Constructor getConstructor(Class ... argumentTypes);

Get the specified public constructor of this class whose arguments match the types listed in `argumentTypes`.

Constructor [] getDeclaredConstructors( );

Get all public and nonpublic constructors of this class.

Constructor getDeclaredConstructor(Class ... argumentTypes);

Get the specified constructor, public or nonpublic, whose arguments match the types listed in `argumentTypes`.

Class [] getDeclaredClasses( );

Get all public and nonpublic inner classes declared within this class.

Constructor [] getInterfaces( );

Get all interfaces implemented by this class, in the order that they are declared.

As you can see, the four `getMethod( )` and `getConstructor( )` methods take advantage of the Java 5.0 variable-length argument lists to allow you to pass in the argument types. In older versions of Java, you have to pass an array of `Class` types in their place. We'll show an example later. As a quick example, we'll show how easy it is to list all the public methods of the `java.util.Calendar` class:

```
for ( Method method : Calendar.class.getMethods( ) )
System.out.println( method );
```

Here, we have used the `.class` notation to get a reference to the `Class` of `Calendar`. Remember the discussion of the `Class` class; the reflection methods don't belong to a particular instance of `Calendar` itself; they belong to the `java.lang.Class` object that describes the `Calendar` class. If we wanted to start from an instance of `Calendar` (or, say, an unknown object), we could have used the `getClass( )` method of the object instead:

```
Method [] methods = myUnknownObject.getClass( ).getMethods( );
```

MODIFIERS AND SECURITY

All of the types of members of a Java classfields, methods, constructors, and inner classeshave a method `getModifiers( )` that returns a set of flags indicating whether the member is private, protected, default level, or publicly accessible. You can test for these with the `java.lang.reflect.Modifier` class, like so:

```
Method method = Object.class.getDeclaredMethod( "clone" ); // no arguments
int perms = method.getModifiers( );
System.out.println( Modifier.isPublic( perms ) ); // false
System.out.println( Modifier.isProtected( perms ) ); // true
System.out.println( Modifier.isPrivate( perms ) ); // false
```

The `clone( )` method in `Object` is protected. Access to the Reflection API is governed by a security manager. A fully trusted app has access to all the previously discussed functionality; it can gain access to members of classes at the level of restriction normally granted code within its scope. It is, however, possible to grant special access to code so that it can use the Reflection API to gain access to private and protected members of other classes in a way that the Java language ordinarily disallows. The `Field`, `Method`, and `Constructor` classes all extend from a base class called `AccessibleObject`. The `AccessibleObject` class has one important method called `setAccessible( )`, which allows you to deactivate normal security when accessing that particular class member. That may sound too easy. It is indeed simple, but whether that method allows you to disable security or not is a function of the Java security manager and security policy. You can do this in a normal Java app running without any security policy, but not, for example, in an applet or other secure environment. For example, to be able to use the protected `clone( )` method of the `Object` class, all we have to do (given no contravening security manager) is:

```
Method method = Object.class.getDeclaredMethod( "clone" );
method.setAccessible( true );
```

ACCESSING FIELDS

The class `java.lang.reflect.Field` represents static variables and instance variables. `Field` has a full set of overloaded accessor methods for all the base types (for example, `getInt( )` and `setInt( )`, `getBoolean( )` and `setBoolean( )`) and `get(`

) and set( ) methods for accessing fields that are reference types. Let's consider this class:

```
class BankAccount {
public int balance;
}
```

With the Reflection API, we can read and modify the value of the public integer field balance:

```
BankAccount myBankAccount = ...;
...
try {
Field balanceField = BankAccount.class.getField("balance");
// read it
int mybalance = balanceField.getInt( myBankAccount );
// change it
balanceField.setInt( myBankAccount, 42 );
} catch ( NoSuchFieldException e ) {
... // there is no "balance" field in this class
} catch ( IllegalAccessException e2) {
... // we don't have permission to access the field
}
```

In this example, we are assuming that we already know the structure of a BankAccount object. In general, we could gather that information from the object itself. All the data access methods of Field take a reference to the particular object instance that we want to access. In this example, the getField( ) method returns a Field object that represents the balance of the BankAccount class; this object doesn't refer to any specific BankAccount. Therefore, to read or modify any specific BankAccount, we call getInt( ) and setInt( ) with a reference to myBankAccount, which is the particular object instance that contains the field we want to work with. For a static field, we'd use the value null here. An exception occurs if we try to access a field that doesn't exist, or if we don't have the proper permission to read or write to the field. If we make balance a private field, we can still look up the Field object that describes it, but we won't be able to read or write its value. Therefore, we aren't doing anything that we couldn't have done with static code at compile time; as long as balance is a public member of a class that we can access, we can write code to read and modify its value. What's important is that we're accessing balance at runtime, and we could just as easily use this technique to examine the balance field in a class that was dynamically loaded or that we just discovered by iterating through the class's fields with the geTDeclaredFields( ) method.

ACCESSING METHODS

The class `java.lang.reflect.Method` represents a static or instance method. Subject to the normal security rules, a `Method` object's `invoke( )` method can be used to call the underlying object's method with specified arguments. Yes, Java does have something like a method pointer! As an example, we'll write a Java app called `Invoke` that takes as command-line arguments the name of a Java class and the name of a method to invoke. For simplicity, we'll assume that the method is static and takes no arguments (quite a limitation):

```
//file: Invoke.java
import java.lang.reflect.*;
class Invoke {
public static void main( String [] args ) {
try {
Class clas = Class.forName( args[0] );
Method method = clas.getMethod( args[1], new Class [] { } );
Object ret = method.invoke( null, null );
// Java 5.0 with varargs
// Method method = clas.getMethod( args[1] );
// Object ret = method.invoke( null );
System.out.println(
"Invoked static method: " + args[1]
+ " of class: " + args[0]
+ " with no args\nResults: " + ret );
} catch ( ClassNotFoundException e ) {
// Class.forName( ) can't find the class
} catch ( NoSuchMethodException e2 ) {
// that method doesn't exist
} catch ( IllegalAccessException e3 ) {
// we don't have permission to invoke that method
} catch ( InvocationTargetException e4 ) {
// an exception occurred while invoking that method
System.out.println(
"Method threw an: " + e4.getTargetException( ) );
}
}
}
```

We can run `invoke` to fetch the value of the system clock:

```
% java Invoke java.lang.System currentTimeMillis
Invoked static method: currentTimeMillis of class:
```

```
java.lang.System with no args
Results: 861129235818
```

Our first task is to look up the specified `Class` by name. To do so, we call the `forName( )` method with the name of the desired class (the first command-line argument). We then ask for the specified method by its name. `getMethod( )` has two arguments: the first is the method name (the second command-line argument), and the second is an array of `Class` objects that specifies the method's signature. (Remember that any method may be overloaded; you must specify the signature to make it clear which version you want.) Since our simple program calls only methods with no arguments, we create an anonymous empty array of `Class` objects. Had we wanted to invoke a method that takes arguments, we would have passed an array of the classes of their respective types, in the proper order. For primitive types, we would have used the standard wrappers (`Integer`, `Float`, `Boolean`, etc.) to hold the values. The classes of primitive types in Java are represented by special static `TYPE` fields of their respective wrappers; for example, use `Integer.TYPE` for the `Class` of an `int`. As shown in comments in the code, in Java 5.0, the `getMethod( )` and `invoke( )` methods now accept variable-length argument lists, which means that we can omit the arguments entirely and Java will make the empty array for us. Once we have the `Method` object, we call its `invoke( )` method. This calls our target method and returns the result as an `Object`. To do anything nontrivial with this object, you have to cast it to something more specific. Presumably, since we're calling the method, we know what kind of object to expect. But if we didn't, we could use the `Method getReturnType( )` method to get the `Class` of the return type at runtime. If the returned value is a primitive type such as `int` or `boolean`, it will be wrapped in the standard wrapper class for its type. If the method returns `void`, `invoke( )` returns a `java.lang.Void` object. This is the "wrapper" class that represents `void` return values. The first argument to `invoke( )` is the object on which we would like to invoke the method. If the method is static, there is no object, so we set the first argument to `null`. That's the case in our example. The second argument is an array of objects to be passed as arguments to the method. The types of these should match the types specified in the call to `getMethod( )`. Because we're calling a method with no arguments, we can pass `null` for the second argument to `invoke( )`. As with the return value, you must use wrapper classes for primitive argument types. The exceptions shown in the previous code occur if we can't find or don't have permission to access the method. Additionally, an `InvocationTargetException` occurs if the method being invoked throws some kind of exception itself. You can find what it threw by calling the `getTargetException( )` method of `InvocationTargetException`.

ACCESSING CONSTRUCTORS

The `java.lang.reflect.Constructor` class represents an object constructor in the same way that the `Method` class represents a method. You can use it, subject to the security manager, of course, to create a new instance of an object, even with constructors that require arguments. Recall that you can create instances of a class with `Class.newInstance( )`, but you cannot specify arguments with that method. This is the solution to that problem, if you really need to do it. Here, we'll create an instance of `java.util.Date`,[*] passing a string argument to the constructor:

> [*] This `Date` constructor is deprecated but will serve us for this example.

```
// Java 5.0 Code, using generic type and varargs
try {
Constructor<Date> cons =
Date.class.getConstructor( String.class );
Date date = cons.newInstance( "Jan 1, 2006" );
System.out.println( date );
} catch ( NoSuchMethodException e ) {
// getConstructor( ) couldn't find the constructor we described
} catch ( InstantiationException e2 ) {
// the class is abstract
} catch ( IllegalAccessException e3 ) {
// we don't have permission to create an instance
} catch ( InvocationTargetException e4 ) {
// the construct threw an exception
}
```

The story is much the same as with a method invocation; after all, a constructor is really no more than a method with some strange properties. We look up the appropriate constructor for our `Date` classthe one that takes a single `String` as its argumentby passing `getConstructor( )` the `String.class` type. Here, we are using the Java 5.0 variable argument syntax. If the constructor required more arguments, we would pass additional `Class`es representing the class of each argument. We can then invoke `newInstance( )`, passing it a corresponding argument object. Again, to pass primitive types, we would wrap them in their wrapper types first. Finally, we print the resulting object to a `Date`. Note that we've slipped in another strange construct using generics here. The `Constructor<Date>` type here simply allows us to specialize the `Constructor` for the `Date` type, alleviating the need to cast the result of the `newInstance( )` method, as before. The exceptions from the previous example apply here, too, along with `IllegalArgumentException` and `InstantiationException`. The latter is thrown if the class is `abstract` and, therefore, can't be instantiated.

WHAT ABOUT ARRAYS?

The Reflection API allows you to create and inspect arrays of base types using the `java.lang.reflect.Array` class. The process is very much the same as with the other classes, so we won't cover it in detail. The primary feature is a static method of `Array` called `newInstance( )`, which creates an array, allowing you to specify a base type and length. You can also use it to construct multidimensional array instances, by specifying an array of lengths (one for each dimension). For more information, look in your favorite Java language reference.

ACCESSING GENERIC TYPE INFORMATION

In , we'll discuss generics in Java 5.0. Generics is a major addition that adds new dimensions (literally) to the concept of types in the Java language. With the addition of generics, types are no longer simply one-to-one with Java classes and interfaces but can be parameterized on one or more types to create a new, generic type. To make matters more complicated, these new types do not actually generate new classes but instead are artifacts of the compiler. To keep the generic information, Java adds information to the compiled class files. In Java 5.0, the Reflection API has been beefed up to accommodate all of this, mainly through the addition of the new `java.lang.reflect.Type` class, which is capable of representing generic types. Covering this in detail is a bit outside the scope of this tutorial and since we won't even get to generics until , we won't devote much more space to this topic here. However, the following code snippets may guide you later if you return to the topic of accessing generic type information reflectively:

```
// public interface List<E> extends Collection<E> { ... }
TypeVariable [] tv = List.class.getTypeParameters( );
System.out.println( tv[0].getName( ) ); // "E"
```

This snippet gets the type parameter of the `java.util.List` class and prints its name:

```
class StringList extends ArrayList<String> { }
Type type = StringList.class.getGenericSuperclass( );
System.out.println( type ); //
// "java.util.ArrayList<java.lang.String>"
ParameterizedType pt = (ParameterizedType)type;
System.out.println( pt.getActualTypeArguments( )[0] ); //
// "class java.lang.String"
```

This second snippet gets the `Type` for a class that extends a generic type and then prints the actual type on which it was parameterized.

ACCESSING ANNOTATION DATA

Later in this chapter, we discuss annotations, a new feature in Java 5.0 that allows metadata to be added to Java classes, methods, and fields. Annotations can optionally be retained in the compiled Java classes and accessed through the Reflection API. This is one of several intended uses for annotations, allowing code at runtime to see the metadata and provide special services for the annotated code. For example, a property on a Java object might be annotated to indicate that it is expecting a container app to set its value or export it in some way. Covering this in detail is outside the scope of this tutorial; however, getting annotation data through the Reflection API in Java 5.0 is easy. Java classes, as well as `Method` and `Field` objects, have the following pairs of methods (and some other related ones):

```
public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
public Annotation[] getDeclaredAnnotations( )
```

These methods (the first is a generic method, as covered in)
return `java.lang.annotation.Annotation` type objects that represent the metadata.

DYNAMIC INTERFACE ADAPTERS

Ideally, Java reflection would allow us to do everything at runtime that we can do at compile time (without forcing us to generate and compile source into bytecode). But that is not entirely the case. Although we can dynamically load and create instances of objects at runtime using the `Class.forName( )` method, there is no general way to create new types of objectsfor which no class files preexiston the fly.
The `java.lang.reflect.Proxy` class, however, takes a step toward solving this problem by allowing the creation of adapter objects that implement arbitrary Java interfaces at runtime. The `Proxy` class is a factory that can generate an adapter class, implementing any interface (or interfaces) you want. When methods are invoked on the adapter class, they are delegated to a single method in a designated `InvocationHandler` object. You can use this to create dynamic implementations of any kind of interface at runtime and handle the method calls anywhere you want. For example, using a `Proxy`, you could log all of the method calls on an interface, delegating them to a "real" implementation afterward. This kind of dynamic behavior is important for tools that work with Java beans, which must register event listeners. (We'll mention this again in .) It's also useful for a wide range of problems. In the following snippet, we take an interface name and construct a proxy implementing the interface. It outputs a message whenever any of the interface's methods is invoked:

```
import java.lang.reflect.*;
InvocationHandler handler =
new InvocationHandler( ) {
Object invoke( Object proxy, Method method, Object[] args ) {
System.out.println(
"Method: "+ method.getName( ) +"( )"
+" of interface: "+ interfaceName
+ " invoked on proxy!"
);
return null;
}
};
Class clas = Class.forName( MyInterface );
MyInterface interfaceProxy =
(MyInterface)Proxy.newProxyInstance(
clas.getClassLoader( ), new Class[] { class }, handler );
// use MyInterface
myInterface.anyMethod( ); // Method: anyMethod( ) ... invoked on proxy!
```

The resulting object, `interfaceProxy`, is cast to the type of the interface we want. It will call our handler whenever any of its methods are invoked. First, we make an implementation of `InvocationHandler`. This is an object with an `invoke( )` method that takes as its argument the `Method` being called and an array of objects representing the arguments to the method call. We then fetch the class of the interface that we're going to implement using `Class.forName( )`. Finally, we ask the proxy to create an adapter for us, specifying the types of interfaces (you can specify more than one) that we want implemented and the handler to use. `invoke( )` is expected to return an object of the correct type for the method call. If it returns the wrong type, a special runtime exception is thrown. Any primitive types in the arguments or in the return value should be wrapped in the appropriate wrapper class. (The runtime system unwraps the return value, if necessary.)

WHAT IS REFLECTION GOOD FOR?

Reflection, although in some sense a "backdoor" feature of the Java language, is finding more and more important uses. In this chapter, we mentioned that reflection is used to access runtime annotations in Java 5.0. In , we'll see how reflection is used to dynamically discover capabilities and features of JavaBean objects. Those are pretty specialized appswhat can reflection do for us in everyday situations? We could use reflection to go about acting as if Java had dynamic method invocation and other useful capabilities; in , we'll see a dynamic adapter class that uses reflection to make calls for us. As a general coding practice however, dynamic method invocation is a bad idea. One of the primary features of Java (and what distinguishes it from some scripting languages) is its strong typing and safety. You abandon much of that when you take a dip in the reflecting pool. And although the performance of the Reflection API is very good, it is not precisely as fast as compiled method invocations in general. More appropriately, you can use reflection in situations where you need to work with objects that you can't know about in advance. Reflection puts Java on a higher plane of coding languages, opening up possibilities for new kinds of apps. As we've mentioned, it makes possible one use of Java annotations at runtime, allowing us to inspect classes, methods, and fields for metadata. Another important and growing use for reflection is integrating Java with scripting languages. With reflection, you can write language interpreters in Java that can access the full Java APIs, create objects, invoke methods, modify variables, and do all the other things a Java program can do at compile time, while the program is running. In fact, you could go so far as to reimplement the Java language in Java, allowing completely dynamic programs that can do all sorts of things. Sound crazy? Well, someone has already done thisone of the authors of this tutorial! We'll explain next.

The BeanShell Java scripting language

I can't resist inserting a plug here for BeanShellmy free, open source, lightweight Java scripting language. BeanShell is just what I alluded to in the previous sectiona Java app that uses the Reflection API to execute Java statements and expressions dynamically. You can use BeanShell interactively to quickly try out some of the examples in this tutorial or take it further to start using Java syntax as a scripting language in your apps. BeanShell exercises the Java Reflection API to its fullest and serves as a demonstration of how dynamic the Java runtime environment really is. You can find a copy of BeanShell on the DVD that accompanies this tutorial and the latest release and documentation at its own web site, http://www.beanshell.org. In recent years, BeanShell has become quite popular. It is now distributed with the Java Development Environment for Emacs (JDEE) and has been bundled with popular app environments including BEA's WebLogic server, NetBeans, and Sun's Java Studio IDE. See Appendix A for more information on getting started. I hope you find it both interesting and useful!