Trail: The Reflection API

by Dale Green

The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. You'll want to use the reflection API if you are writing development tools such as debuggers, class browsers, and GUI builders. With the reflection API you can:

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

First, a note of caution. Don't use the reflection API when other tools more natural to the Java programming language would suffice. For example, if you are in the habit of using function pointers in another language, you might be tempted to use the `Method` objects of the reflection API in the same way. Resist the temptation! Your program will be easier to debug and maintain if you don't use `Method` objects. Instead, you should define an interface, and then implement it in the classes that perform the needed action.

Other trails use the term "member variable" instead of "field." The two terms are synonymous. Because the `Field` class is part of the reflection API, this trail uses the term "field."

This trail uses a task-oriented approach to the reflection API. Each lesson includes a set of related tasks, and every task is explained, step by step, with a sample program. The lessons are as follows:

Examining Classes explains how to determine the class of an object, and how to get information about classes and interfaces.

Manipulating Objects shows you how to instantiate classes, get or set field values, and invoke methods. With the reflection API, you can perform these tasks even if the names of the classes, fields, and methods are unknown until runtime.

Working with Arrays describes the APIs used to create and to modify arrays whose names are not known until runtime.

Summary of Classes lists the classes that comprise the reflection API, and provides links to the appppropriate API documentation.

Trail: The Reflection API: Table of Contents

Trail: The Reflection API

Lesson: Examining Classes

If you are writing a class browser, you need a way to get information about classes at runtime. For example, you might want to display the names of the class fields, methods, and constructors. Or, you might want to show which interfaces are implemented by a class. To get this information you need to get the `Class` object that reflects the class.

For each class, the Java Runtime Environment (JRE) maintains an immutable `Class` object that contains information about the class. A `Class` object represents, or reflects, the class. With the reflection API, you can invoke methods on a `Class` object which return `Constructor`, `Method`, and `Field` objects. You can use these objects to get information about the corresponding constructors, methods, and fields defined in the class.

`Class` objects also represent interfaces. You invoke `Class` methods to find out about an interface's modifiers, methods, and public constants. Not all of the `Class` methods are appropriate when a `Class` object reflects an interface. For example, it doesn't make sense to invoke `getConstructors` when the `Class` object represents an interface. The section Examining Interfaces explains which `Class` methods you may use to get information about interfaces.

Retrieving Class Objects

First things first. Before you can find out anything about a class, you must first retrieve its corresponding `Class` object.

Getting the Class Name

It's easy to find out the name of a `Class` object. All you have to do is invoke the `getName` method.

Discovering Class Modifiers

This section shows you the methods you need to call to find out what modifiers a particular class has.

Finding Superclasses

In this section you'll learn how to retrieve all of the `Class` objects for the ancestors of a given class.

Identifying the Interfaces Implemented by a Class

If you want to find out what interfaces a class implements, then check out this section.

Examining Interfaces

In this section you'll learn how to tell if a `Class` object represents an interface or a class. You'll also get some tips on how to get more information about an interface.

Identifying Class Fields

This section shows you how to discover what fields belong to a class, and how to find out more about these fields by accessing `Field` objects.

Discovering Class Constructors

This section, which introduces the `Constructor` class, explains how to get information about a class's contructors.

Obtaining Method Information

To find out about a class's methods, you need to retrieve the corresponding `Method` objects. This section shows you how to do this.

Trail: The Reflection API
Lesson: Examining Classes

Retrieving Class Objects

You can retrieve a `Class` object in several ways:

- If an instance of the class is available, you can invoke `Object.getClass`. The `getClass` method is useful when you want to examine an object but you don't know its class. The following line of code gets the `Class` object for an object named `mystery`:

```
Class c = mystery.getClass();
```

- If you want to retrieve the `Class` object for the superclass that another `Class` object reflects, invoke the `getSuperclass` method. In the following example, `getSuperclass` returns the `Class` object associated with the the `TextComponent` class, because `TextComponent` is the superclass of `TextField`:

```
TextField t = new TextField();
Class c = t.getClass();
Class s = c.getSuperclass();
```

- If you know the name of the class at compile time, you can retrieve its `Class` object by appending `.class` to its name. In the next example, the `Class` object that represents the `Button` class is retrieved:

```
Class c = java.awt.Button.class;
```

- If the class name is unknown at compile time, but available at runtime, you can use the `forName` method. In the following example, if the `String` named `strg` is set to "java.awt.Button" then `forName` returns the `Class` object associated with the `Button` class:

```
Class c = Class.forName(strg);
```

Trail: The Reflection API
Lesson: Examining Classes

Examining Interfaces

`Class` objects represent interfaces as well as classes. If you aren't sure whether a `Class` object represents an interface or a class, call the `isInterface` method.

You invoke `Class` methods to get information about an interface. To find the public constants of an interface, invoke the `getFields` method upon the `Class` object that represents the interface. The section Identifying Class Fields has an example containing `getFields`. You can use `getMethods` to get information about an interface's methods. See the section Obtaining Method Information. To find out about an interface's modifiers, invoke the `getModifiers` method. See the section Discovering Class Modifiers for an example.

By calling `isInterface`, the following program reveals that `Observer` is an interface and that `Observable` is a class:

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

class SampleCheckInterface {

   public static void main(String[] args) {
      Class observer = Observer.class;
      Class observable = Observable.class;
      verifyInterface(observer);
      verifyInterface(observable);
   }

   static void verifyInterface(Class c) {
      String name = c.getName();
      if (c.isInterface()) {
         System.out.println(name + " is an interface.");
      } else {
         System.out.println(name + " is a class.");
      }
   }
}
```

The output of the preceding program is:

```
java.util.Observer is an interface.
java.util.Observable is a class.
```


Trail: The Reflection API
Lesson: Examining Classes

Getting the Class Name

Every class in the Java programming language has a name. When you declare a class, the name immediately follows the `class` keyword. In the following class declaration, the class name is `Point`:

```
public class Point {int x, y;}
```

At runtime, you can determine the name of a `Class` object by invoking the `getName` method. The `String` returned by `getName` is the fully-qualified name of the class.

The following program gets the class name of an object. First, it retrieves the corresponding `Class` object, and then it invokes the `getName` method on that `Class` object.

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

class SampleName {

   public static void main(String[] args) {
      Button b = new Button();
      printName(b);
   }

   static void printName(Object o) {
       Class c = o.getClass();
       String s = c.getName();
       System.out.println(s);
   }
}
```

The sample program prints the following line:

```
java.awt.Button
```

Trail: The Reflection API
Lesson: Examining Classes

Discovering Class Modifiers

A class declaration may include the following modifiers: `public`, `abstract`, or `final`. The class modifiers precede the `class` keyword in the class definition. In the following example, the class modifiers are `public` and `final`:

```
public final Coordinate {int x, int y, int z}
```

To identify the modifiers of a class at runtime you perform these steps:

1. Invoke `getModifiers` on a `Class` object to retrieve a set of modifiers.
2. Check the modifiers by calling `isPublic`, `isAbstract`, and `isFinal`.

The following program identifies the modifiers of the `String` class.

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

class SampleModifier {

   public static void main(String[] args) {
      String s = new String();
      printModifiers(s);
   }

   public static void printModifiers(Object o) {
      Class c = o.getClass();
      int m = c.getModifiers();
      if (Modifier.isPublic(m))
         System.out.println("public");
      if (Modifier.isAbstract(m))
         System.out.println("abstract");
      if (Modifier.isFinal(m))
         System.out.println("final");
   }
}
```

The output of the sample program reveals that the modifiers of the `String` class are `public` and `final`:

```
public
final
```


The Java™ Tutorial

Trail: The Reflection API
Lesson: Examining Classes

Finding Superclasses

Because the Java programming language supports inheritance, an application such as a class browser must be able to identify superclasses. To determine the superclass of a class, you invoke the `getSuperclass` method. This method returns a `Class` object representing the superclass, or returns null if the class has no superclass. To identify all ancestors of a class, call `getSuperclass` iteratively until it returns null.

The program that follows finds the names of the `Button` class's ancestors by calling `getSuperclass` iteratively.

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

class SampleSuper {

   public static void main(String[] args) {
      Button b = new Button();
      printSuperclasses(b);
   }

   static void printSuperclasses(Object o) {
      Class subclass = o.getClass();
      Class superclass = subclass.getSuperclass();
      while (superclass != null) {
         String className = superclass.getName();
         System.out.println(className);
         subclass = superclass;
         superclass = subclass.getSuperclass();
      }
   }
}
```

The output of the sample program verifies that the parent of `Button` is `Component`, and that the parent of `Component` is `Object`:

```
java.awt.Component
java.lang.Object
```

Identifying the Interfaces Implemented by a Class

The type of an object is determined by not only its class and superclass, but also by its interfaces. In a class declaration, the interfaces are listed after the `implements` keyword. For example, the `RandomAccessFile` class implements the `DataOutput` and `DataInput` interfaces:

```
public class RandomAccessFile implements DataOutput, DataInput
```

You invoke the `getInterfaces` method to determine which interfaces a class implements. The `getInterfaces` method returns an array of `Class` objects. The reflection API represents interfaces with `Class` objects. Each `Class` object in the array returned by `getInterfaces` represents one of the interfaces implemented by the class. You can invoke the `getName` method on the `Class` objects in the array returned by `getInterfaces` to retrieve the interface names. To find out how to get additional information about interfaces, see the section Examining Interfaces.

The program that follows prints the interfaces implemented by the `RandomAccessFile` class.

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

class SampleInterface {

   public static void main(String[] args) {
      try {
          RandomAccessFile r = new RandomAccessFile("myfile", "r");
          printInterfaceNames(r);
      } catch (IOException e) {
          System.out.println(e);
      }
   }

   static void printInterfaceNames(Object o) {
      Class c = o.getClass();
      Class[] theInterfaces = c.getInterfaces();
      for (int i = 0; i < theInterfaces.length; i++) {
         String interfaceName = theInterfaces[i].getName();
         System.out.println(interfaceName);
      }
   }
}
```

Note that the interface names printed by the sample program are fully qualified:

```
java.io.DataOutput
java.io.DataInput
```


Trail: The Reflection API
Lesson: Examining Classes

Identifying Class Fields

If you are writing an application such as a class browser, you might want to find out what fields belong to a particular class. You can identify a class's fields by invoking the `getFields` method on a `Class` object. The `getFields` method returns an array of `Field` objects containing one object per accessible public field.

A public field is accessible if it is a member of either:

- this class
- a superclass of this class
- an interface implemented by this class
- an interface extended from an interface implemented by this class

The methods provided by the `Field` class allow you to retrieve the field's name, type, and set of modifiers. You can even get and set the value of a field, as described in the sections Getting Field Values and Setting Field Values.

The following program prints the names and types of fields belonging to the `GridBagConstraints` class. Note that the program first retrieves the `Field` objects for the class by calling `getFields`, and then invokes the `getName` and `getType` methods on each of these `Field` objects.

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

class SampleField {

   public static void main(String[] args) {
      GridBagConstraints g = new GridBagConstraints();
      printFieldNames(g);
   }

   static void printFieldNames(Object o) {
      Class c = o.getClass();
      Field[] publicFields = c.getFields();
      for (int i = 0; i < publicFields.length; i++) {
         String fieldName = publicFields[i].getName();
```

```
                Class typeClass = publicFields[i].getType();
                String fieldType = typeClass.getName();
                System.out.println("Name: " + fieldName +
                  ", Type: " + fieldType);
                }
            }
        }
```

A truncated listing of the output generated by the preceding program follows:

```
Name: RELATIVE, Type: int
Name: REMAINDER, Type: int
Name: NONE, Type: int
Name: BOTH, Type: int
Name: HORIZONTAL, Type: int
Name: VERTICAL, Type: int
.
.
.
```


Trail: The Reflection API
Lesson: Examining Classes

Discovering Class Constructors

To create an instance of a class, you invoke a special method called a constructor. Like methods, constructors can be overloaded and are distinguished from one another by their signatures.

You can get information about a class's constructors by invoking the `getConstructors` method, which returns an array of `Constructor` objects. You can use the methods provided by the `Constructor` class to determine the constructor's name, set of modifiers, parameter types, and set of throwable exceptions. You can also create a new instance of the `Constructor` object's class with the `Constructor.newInstance` method. You'll learn how to invoke `Constructor.newInstance` in the section Manipulating Objects.

The sample program that follows prints out the parameter types for each constructor in the `Rectangle` class. The program performs the following steps:

1. It retrieves an array of `Constructor` objects from the `Class` object by calling `getConstructors`.
2. For every element in the `Constructor` array, it creates an array of `Class` objects by invoking `getParameterTypes`. The `Class` objects in the array represent the parameters of the constructor.
3. The program calls `getName` to fetch the class name for every parameter in the `Class` array created in the preceding step.

It's not as complicated as it sounds. Here's the source code for the sample program:

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

class SampleConstructor {

   public static void main(String[] args) {
      Rectangle r = new Rectangle();
      showConstructors(r);
   }

   static void showConstructors(Object o) {
      Class c = o.getClass();
      Constructor[] theConstructors = c.getConstructors();
      for (int i = 0; i < theConstructors.length; i++) {
         System.out.print("( ");
         Class[] parameterTypes =
            theConstructors[i].getParameterTypes();
```

```
            for (int k = 0; k < parameterTypes.length; k ++) {
               String parameterString = parameterTypes[k].getName();
               System.out.print(parameterString + " ");
               }
            System.out.println(")");
            }
      }
   }
```

In the first line of output generated by the sample program, no parameter types appear because that particular `Constructor` object represents a no-argument constructor. In subsequent lines, the parameters listed are either `int` types or fully qualified object names. The output of the sample program is:

```
( )
( int int )
( int int int int )
( java.awt.Dimension )
( java.awt.Point )
( java.awt.Point java.awt.Dimension )
( java.awt.Rectangle )
```


Trail: The Reflection API
Lesson: Examining Classes

Obtaining Method Information

To find out what public methods belong to a class, invoke the method named `getMethods`. The array returned by `getMethods` contains `Method` objects. You can use a `Method` object to uncover a method's name, return type, parameter types, set of modifiers, and set of throwable exceptions. All of this information would be useful if you were writing a class browser or a debugger. With `Method.invoke`, you can even call the method itself. To see how to do this, see the section Invoking Methods.

The following sample program prints the name, return type, and parameter types of every public method in the `Polygon` class. The program performs the following tasks:

1. It retrieves an array of `Method` objects from the `Class` object by calling `getMethods`.
2. For every element in the `Method` array, the program:
   a. retrieves the method name by calling `getName`
   b. gets the return type by invoking `getReturnType`
   c. creates an array of `Class` objects by invoking `getParameterTypes`
3. The array of `Class` objects created in the preceding step represents the parameters of the method. To retrieve the class name for every one of these parameters, the program invokes `getName` against each `Class` object in the array.

Not many lines of source code are required to accomplish these tasks:

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

class SampleMethod {

   public static void main(String[] args) {
      Polygon p = new Polygon();
      showMethods(p);
   }

   static void showMethods(Object o) {
      Class c = o.getClass();
      Method[] theMethods = c.getMethods();
      for (int i = 0; i < theMethods.length; i++) {
         String methodString = theMethods[i].getName();
         System.out.println("Name: " + methodString);
         String returnString =
           theMethods[i].getReturnType().getName();
         System.out.println("   Return Type: " + returnString);
         Class[] parameterTypes = theMethods[i].getParameterTypes();
         System.out.print("   Parameter Types:");
```

```
                for (int k = 0; k < parameterTypes.length; k ++) {
                   String parameterString = parameterTypes[k].getName();
                   System.out.print(" " + parameterString);
                }
                System.out.println();
             }
          }
       }
```

An abbreviated version of the output generated by the sample program is as follows:

```
Name: equals
   Return Type: boolean
   Parameter Types: java.lang.Object
Name: getClass
   Return Type: java.lang.Class
   Parameter Types:
Name: hashCode
   Return Type: int
   Parameter Types:
.
.
.
Name: intersects
   Return Type: boolean
   Parameter Types: double double double double
Name: intersects
   Return Type: boolean
   Parameter Types: java.awt.geom.Rectangle2D
Name: translate
   Return Type: void
   Parameter Types: int int
```


Trail: The Reflection API
Lesson: Manipulating Objects

Getting Field Values

If you are writing a development tool such as a debugger, you must be able to obtain field values. This is a three-step process:

1. Create a `Class` object. The section Retrieving Class Objects shows you how to do this.
2. Create a `Field` object by invoking `getField` on the `Class` object. For more information, see the section Identifying Class Fields.
3. Invoke one of the `get` methods on the `Field` object.

The `Field` class has specialized methods for getting the values of primitive types. For example, the `getInt` method returns the contents as an `int` value, `getFloat` returns a `float`, and so forth. If the field stores an object instead of a primitive, then use the `get` method to retrieve the object.

The following sample program demonstrates the three steps listed previously. This program gets the value of the `height` field from a `Rectangle` object. Because the `height` is a primitive type (`int`), the object returned by the `get` method is a wrapper object (`Integer`).

In the sample program, the name of the `height` field is known at compile time. However, in a development tool such as a GUI builder, the field name might not be known until runtime. To find out what fields belong to a class, you can use the techniques described in the section Identifying Class Fields.

Here is the source code for the sample program:

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

class SampleGet {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 325);
      printHeight(r);

   }

   static void printHeight(Rectangle r) {
      Field heightField;
      Integer heightValue;
      Class c = r.getClass();
```

```
         try {
           heightField = c.getField("height");
           heightValue = (Integer) heightField.get(r);
           System.out.println("Height: " + heightValue.toString());
         } catch (NoSuchFieldException e) {
             System.out.println(e);
         } catch (SecurityException e) {
             System.out.println(e);
         } catch (IllegalAccessException e) {
             System.out.println(e);
         }
      }
   }
```

The output of the sample program verifies the value of the `height` field:

```
Height: 325
```


The Java™ Tutorial

Trail: The Reflection API
Lesson: Manipulating Objects

Setting Field Values

Some debuggers allow users to change field values during a debugging session. If you are writing a tool that has this capability, you must call one of the `Field` class's `set` methods. To modify the value of a field, perform the following steps:

1. Create a `Class` object. For more information, see the section Retrieving Class Objects.
2. Create a `Field` object by invoking `getField` on the `Class` object. The section Identifying Class Fields shows you how.
3. Invoke the appropriate `set` method on the `Field` object.

The `Field` class provides several `set` methods. Specialized methods, such as `setBoolean` and `setInt`, are for modifying primitive types. If the field you want to change is an object invoke the `set` method. You can call `set` to modify a primitive type, but you must use the appropriate wrapper object for the value parameter.

The sample program that follows modifies the `width` field of a `Rectangle` object by invoking the `set` method. Since the `width` is a primitive type, an `int`, the value passed by `set` is an `Integer`, which is an object wrapper.

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

class SampleSet {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
      System.out.println("original: " + r.toString());
      modifyWidth(r, new Integer(300));
      System.out.println("modified: " + r.toString());
   }

   static void modifyWidth(Rectangle r, Integer widthParam ) {
      Field widthField;
      Integer widthValue;
      Class c = r.getClass();
      try {
        widthField = c.getField("width");
        widthField.set(r, widthParam);
```

```
            } catch (NoSuchFieldException e) {
                System.out.println(e);
            } catch (IllegalAccessException e) {
                System.out.println(e);
            }
        }
    }
```

The output of the sample program verifies that the `width` changed from 100 to 300:

```
original: java.awt.Rectangle[x=0,y=0,width=100,height=20]
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]
```


The Java^{TM} Tutorial

Trail: The Reflection API

Lesson: Manipulating Objects

Software development tools, such as GUI builders and debuggers, need to manipulate objects at runtime. For example, a GUI builder may allow the end-user to select a `Button` from a menu of components, create the `Button` object, and then click the `Button` while running the application within the GUI builder. If you're in the business of creating software development tools, you'll want to take advantage of the reflection API features described in this lesson.

This lesson has the following sections:

Creating Objects

How can you create an object if you don't know its class name until runtime? You'll find the answer in this section.

Getting Field Values

In this section you'll learn how to get the values of an object's fields, even if you don't know the name of the object, or even its class, until runtime.

Setting Field Values

Not only can you get field values at runtime, you can also set them. This section shows you how.

Invoking Methods

This section shows you how to dynamically invoke methods. Given an object, you can find out what methods its class defines, and then invoke the methods.

Trail: The Reflection API
Lesson: Manipulating Objects

Invoking Methods

Suppose that you are writing a debugger that allows the user to select and then invoke methods during a debugging session. Since you don't know at compile time which methods the user will invoke, you cannot hardcode the method name in your source code. Instead, you must follow these steps:

1. Create a `Class` object that corresponds to the object whose method you want to invoke. See the section Retrieving Class Objects for more information.
2. Create a `Method` object by invoking `getMethod` on the `Class` object. The `getMethod` method has two arguments: a `String` containing the method name, and an array of `Class` objects. Each element in the array corresponds to a parameter of the method you want to invoke. For more information on retrieving `Method` objects, see the section Obtaining Method Information
3. Invoke the method by calling `invoke`. The `invoke` method has two arguments: an array of argument values to be passed to the invoked method, and an object whose class declares or inherits the method.

The sample program that follows shows you how to invoke a method dynamically. The program retrieves the `Method` object for the `String.concat` method and then uses `invoke` to concatenate two `String` objects.

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

class SampleInvoke {

   public static void main(String[] args) {
      String firstWord = "Hello ";
      String secondWord = "everybody.";
      String bothWords = append(firstWord, secondWord);
      System.out.println(bothWords);
   }

   public static String append(String firstWord, String secondWord) {
      String result = null;
      Class c = String.class;
      Class[] parameterTypes = new Class[] {String.class};
      Method concatMethod;
      Object[] arguments = new Object[] {secondWord};
      try {
        concatMethod = c.getMethod("concat", parameterTypes);
        result = (String) concatMethod.invoke(firstWord, arguments);
      } catch (NoSuchMethodException e) {
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println(e);
```



```
      } catch (InvocationTargetException e) {
          System.out.println(e);
      }
      return result;
   }
}
```

The output of the preceding program is:

```
Hello everybody.
```

Trail: The Reflection API
Lesson: Manipulating Objects

Creating Objects

The simplest way to create an object in the Java programming language is to use the `new` operator:

```
Rectangle r = new Rectangle();
```

This technique is adequate for nearly all applications, because usually you know the class of the object at compile time. However, if you are writing development tools, you may not know the class of an object until runtime. For example, a GUI builder might allow the user to drag and drop a variety of GUI components onto the page being designed. In this situation, you may be tempted to create the GUI components as follows:

```
String className;

// . . . load className from the user interface

Object o = new (className); // WRONG!
```

The preceding statement is invalid because the `new` operator does not accept arguments. Fortunately, with the reflection API you can create an object whose class is unknown until runtime. The method you invoke to create an object dynamically depends on whether or not the constructor you want to use has arguments. This section discusses these topics:

- Using No-Argument Constructors
- Using Constructors that Have Arguments

Trail: The Reflection API
Lesson: Manipulating Objects

Using No-Argument Constructors

If you need to create an object with the no-argument constructor, you can invoke the `newInstance` method on a `Class` object. The `newInstance` method throws a `NoSuchMethodException` if the class does not have a no-argument constructor. For more information on working with `Constructor` objects, see the section Discovering Class Constructors.

The following sample program creates an instance of the `Rectangle` class using the no-argument constructor by calling the `newInstance` method:

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

class SampleNoArg {

   public static void main(String[] args) {
      Rectangle r = (Rectangle) createObject("java.awt.Rectangle");
      System.out.println(r.toString());
   }

   static Object createObject(String className) {
      Object object = null;
      try {
          Class classDefinition = Class.forName(className);
          object = classDefinition.newInstance();
      } catch (InstantiationException e) {
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println(e);
      } catch (ClassNotFoundException e) {
          System.out.println(e);
      }
      return object;
   }
}
```

The output of the preceding program is:

```
java.awt.Rectangle[x=0,y=0,width=0,height=0]
```

Trail: The Reflection API
Lesson: Manipulating Objects

Using Constructors that Have Arguments

To create an object with a constructor that has arguments, you invoke the `newInstance` method on a `Constructor` object, not a `Class` object. This technique involves several steps:

1. Create a `Class` object for the object you want to create.
2. Create a `Constructor` object by invoking `getConstructor` on the `Class` object. The `getConstructor` method has one parameter: an array of `Class` objects that correspond to the constructor's parameters.
3. Create the object by invoking `newInstance` on the `Constructor` object. The `newInstance` method has one parameter: an `Object` array whose elements are the argument values being passed to the constructor.

The sample program that follows creates a `Rectangle` with the constructor that accepts two integers as parameters. Invoking `newInstance` on this constructor is analogous to this statement:

```
Rectangle rectangle = new Rectangle(12, 34);
```

This constructor's arguments are primitive types, but the argument values passed to `newInstance` must be objects. Therefore, each of the primitive `int` types is wrapped in an `Integer` object.

The sample program hardcodes the argument passed to the `getConstructor` method. In a real-life application such as a debugger, you would probably let the user select the constructor. To verify the user's selection, you could use the methods described in the section Discovering Class Constructors.

The source code for the sample program follows:

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

class SampleInstance {

   public static void main(String[] args) {

      Rectangle rectangle;
      Class rectangleDefinition;
      Class[] intArgsClass = new Class[] {int.class, int.class};
      Integer height = new Integer(12);
      Integer width = new Integer(34);
      Object[] intArgs = new Object[] {height, width};
      Constructor intArgsConstructor;

      try {
```

```
            rectangleDefinition = Class.forName("java.awt.Rectangle");
            intArgsConstructor =
                rectangleDefinition.getConstructor(intArgsClass);
            rectangle =
                (Rectangle) createObject(intArgsConstructor, intArgs);
          } catch (ClassNotFoundException e) {
              System.out.println(e);
          } catch (NoSuchMethodException e) {
              System.out.println(e);
          }
       }

       public static Object createObject(Constructor constructor,
                                         Object[] arguments) {

          System.out.println ("Constructor: " + constructor.toString());
          Object object = null;

          try {
            object = constructor.newInstance(arguments);
            System.out.println ("Object: " + object.toString());
            return object;
          } catch (InstantiationException e) {
              System.out.println(e);
          } catch (IllegalAccessException e) {
              System.out.println(e);
          } catch (IllegalArgumentException e) {
              System.out.println(e);
          } catch (InvocationTargetException e) {
              System.out.println(e);
          }
          return object;
       }
    }
```

The sample program prints a description of the constructor and the object that it creates:

```
Constructor: public java.awt.Rectangle(int,int)
Object: java.awt.Rectangle[x=0,y=0,width=12,height=34]
```

Start of Tutorial > Start of Trail > Start of Lesson

Search
Feedback Form

The Java^{TM} Tutorial

Start of Tutorial > Start of Trail

Search
Feedback Form

Trail: The Reflection API

Lesson: Working with Arrays

The `Array` class provides methods that allow you to dynamically create and access arrays. In this lesson yu'll learn how to use these methods.

Identifying Arrays

This section shows you how to determine if an object really is an array.

Retrieving Component Types

If you want to find out the component type of an array, you'll want to check out the programming example in this section.

Creating Arrays

This section shows you how simple it is to create arrays at run time.

Getting and Setting Element Values

Even if you don't know the name of an array until run time, you can examine or modify the values of its elements. This section shows you how.

Start of Tutorial > Start of Trail

Search
Feedback Form

The Java[TM] Tutorial

Trail: The Reflection API
Lesson: Working with Arrays

Identifying Arrays

If you aren't certain that a particular object is an array, you can check it with the `Class.isArray` method. Let's take a look at an example.

The sample program that follows prints the names of the arrays that are encapsulated in an object. The program performs these steps:

1. It retrieves the `Class` object that represents the target object.
2. It gets the `Field` objects for the `Class` object retrieved in step 1.
3. For each `Field` object, the program gets a corresponding `Class` object by invoking the `getType` method.
4. To verify that the `Class` object retrieved in the preceding step represents an array, the program invokes the `isArray` method.

Here's the source code for the sample program:

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

class SampleArray {

   public static void main(String[] args) {
      KeyPad target = new KeyPad();
      printArrayNames(target);
   }

   static void printArrayNames(Object target) {
      Class targetClass = target.getClass();
      Field[] publicFields = targetClass.getFields();
      for (int i = 0; i < publicFields.length; i++) {
         String fieldName = publicFields[i].getName();
         Class typeClass = publicFields[i].getType();
         String fieldType = typeClass.getName();
         if (typeClass.isArray()) {
            System.out.println("Name: " + fieldName +
               ", Type: " + fieldType);
         }
```



```
         }
      }
   }

class KeyPad {

   public boolean alive;
   public Button power;
   public Button[] letters;
   public int[] codes;
   public TextField[] rows;
   public boolean[] states;
}
```

The output of the sample program follows. Note that the left bracket indicates that the object is an array. For a detailed description of the type descriptors that `getName` returns, see section 4.3.1 of *The Java Virtual Machine Specification*.

```
Name: letters, Type: [Ljava.awt.Button;
Name: codes, Type: [I
Name: rows, Type: [Ljava.awt.TextField;
Name: states, Type: [Z
```

Trail: The Reflection API
Lesson: Working with Arrays

Retrieving Component Types

The component type is the type of an array's elements. For example, the component type of the `arrowKeys` array in the following line of code is `Button`:

```
Button[] arrowKeys = new Button[4];
```

The component type of a multidimensional array is an array. In the next line of code, the component type of the array named `matrix` is `int[]`:

```
int[][] matrix = new int[100][100];
```

By invoking the `getComponentType` method against the `Class` object that represents an array, you can retrieve the component type of the array's elements.

The sample program that follows invokes the `getComponentType` method and prints out the class name of each array's component type.

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

class SampleComponent {

   public static void main(String[] args) {
      int[] ints = new int[2];
      Button[]  buttons = new Button[6];
      String[][] twoDim = new String[4][5];

      printComponentType(ints);
      printComponentType(buttons);
      printComponentType(twoDim);
   }

   static void printComponentType(Object array) {
      Class arrayClass = array.getClass();
      String arrayName = arrayClass.getName();
      Class componentClass = arrayClass.getComponentType();
      String componentName = componentClass.getName();
```



```
      System.out.println("Array: " + arrayName +
         ", Component: " + componentName);
   }
}
```

The output of the sample program is:

```
Array: [I, Component: int
Array: [Ljava.awt.Button;, Component: java.awt.Button
Array: [[Ljava.lang.String;, Component: [Ljava.lang.String;
```

Trail: The Reflection API
Lesson: Working with Arrays

Creating Arrays

If you are writing a development tool such as an application builder, you may want to allow the end user to create arrays at runtime. Your program can provide this capability by invoking the `Array.newInstance` method.

The following sample program uses the `newInstance` method to create a copy of an array that is twice the size of the original array. The `newInstance` method accepts as arguments the length and component type of the new array. The source code follows:

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

class SampleCreateArray {

   public static void main(String[] args) {
      int[] originalArray =  {55, 66};
      int[] biggerArray = (int[]) doubleArray(originalArray);
      System.out.println("originalArray:");
      for (int k = 0; k < Array.getLength(originalArray); k++)
         System.out.println(originalArray[k]);
      System.out.println("biggerArray:");
      for (int k = 0; k < Array.getLength(biggerArray); k++)
         System.out.println(biggerArray[k]);
   }

   static Object doubleArray(Object source) {
      int sourceLength = Array.getLength(source);
      Class arrayClass = source.getClass();
      Class componentClass = arrayClass.getComponentType();
      Object result = Array.newInstance(componentClass,
                                        sourceLength * 2);
      System.arraycopy(source, 0, result, 0, sourceLength);
      return result;
   }
}
```

The output of the preceding program is:

```
originalArray:
55
66
biggerArray:
55
66
0
0
```


You can also use the `newInstance` method to create multidimensional arrays. In this case, the parameters of the method are the component type and an array of `int` types representing the dimensions of the new array.

The next sample program shows how to use `newInstance` to create multidimensional arrays:

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

class SampleMultiArray {

   public static void main(String[] args) {

      // The oneDimA and oneDimB objects are one
      // dimensional int arrays with 5 elements.

      int[] dim1 = {5};
      int[] oneDimA = (int[]) Array.newInstance(int.class, dim1);
      int[] oneDimB = (int[]) Array.newInstance(int.class, 5);

      // The twoDimStr object is a 5 X 10 array of String objects.

      int[] dimStr = {5, 10};
      String[][] twoDimStr =
                 (String[][]) Array.newInstance(String.class,dimStr);

      // The twoDimA object is an array of 12 int arrays. The tail
      // dimension is not defined. It is equivalent to the array
      // created as follows:
      //    int[][] ints = new int[12][];

      int[] dimA = {12};
      int[][] twoDimA = (int[][]) Array.newInstance(int[].class, dimA);
   }
}
```

The Java[TM] Tutorial

Trail: The Reflection API
Lesson: Working with Arrays

Getting and Setting Element Values

In most programs, to access array elements you merely use an assignment expression as follows:

```
int[10] codes;
codes[3] = 22;
aValue = codes[3];
```

This technique will not work if you don't know the name of the array until runtime.

Fortunately, you can use the `Array` class `set` and `get` methods to access array elements when the name of the array is unknown at compile time. In addition to `get` and `set`, the `Array` class has specialized methods that work with specific primitive types. For example, the value parameter of `setInt` is an `int`, and the object returned by `getBoolean` is a wrapper for a `boolean` type.

The sample program that follows uses the `set` and `get` methods to copy the contents of one array to another.

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

class SampleGetArray {

   public static void main(String[] args) {
      int[] sourceInts = {12, 78};
      int[] destInts = new int[2];
      copyArray(sourceInts, destInts);
      String[] sourceStrgs = {"Hello ", "there ", "everybody"};
      String[] destStrgs = new String[3];
      copyArray(sourceStrgs, destStrgs);
   }

   public static void copyArray(Object source, Object dest) {
      for (int i = 0; i < Array.getLength(source); i++) {
         Array.set(dest, i, Array.get(source, i));
         System.out.println(Array.get(dest, i));
      }
   }
}
```


The output of the sample program is:

```
12
78
Hello
there
everybody
```

The Java[TM] Tutorial

Trail: The Reflection API

Lesson: Summary of Classes

The following table summarizes the classes that compose the reflection API. The `Class` and `Object` classes are in the java.lang package. The other classes are contained in the java.lang.reflect package.

Class	Description
Array	Provides static methods to dynamically create and access arrays.
Class	Represents, or reflects, classes and interfaces.
Constructor	Provides information about, and access to, a constructor for a class. Allows you to instantiate a class dynamically.
Field	Provides information about, and dynamic access to, a field of a class or an interface.
Method	Provides information about, and access to, a single method on a class or interface. Allows you to invoke the method dynamically.
Modifier	Provides static methods and constants that allow you to get information about the access modifiers of a class and its members.
Object	Provides the `getClass` method.

The Java[TM] Tutorial

Trail: The Reflection API: End of Trail

You've reached the end of the "The Reflection API" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

Take a break -- have a cup of steaming hot java.

What next?

Once you've caught your breath, you have several choices of where to go next. You can go to the tutorial's front page or to the Trail Map to see all of your choices, or you can go directly to one of the following related trails:

Java Native interface: Another advanced topic used only by specialized programs.