java.text.NumberFormat

- `Number parse(String s)`

  returns the numeric value, assuming the specified `String` represents a number.

The `Class` Class

While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects. This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute.

However, you can also access this information by working with a special Java class. The class that holds this information is called, somewhat confusingly, `Class`. The `getClass()` method in the `Object` class returns an instance of `Class` type.

```
Employee e;
. . .
Class cl = e.getClass();
```

Just like an `Employee` object describes the properties of a particular employee, a `Class` object describes the properties of a particular class. Probably the most commonly used method of `Class` is `getName`. This returns the name of the class. For example, the statement

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

prints

```
Employee Harry Hacker
```

if `e` is an employee, or

```
Manager Harry Hacker
```

if `e` is a manager.

You can also obtain a `Class` object corresponding to a string by using the static `forName` method.

```
String className = "Manager";
Class cl = Class.forName(className);
```

You would use this method if the class name is stored in a string that varies at run time. This works if `className` is the name of a class or interface. Otherwise, the `forName` method throws a *checked exception*. See the sidebar on exceptions to see how you need to supply an *exception handler* whenever you use this method.

TIP

At startup, the class containing your `main` method is loaded. It loads all classes that it needs. Each of those loaded classes loads the classes that they need, and so on. That can take a long time for a big application, frustrating the user. You can give users of your program the illusion of a faster start, with the following trick. Make sure that the class containing the `main` method does not explicitly refer to other classes. First display a splash screen. Then manually force the loading of other classes by calling `Class.forName`.

A third method for obtaining an object of type `Class` is a convenient shorthand. If `T` is any Java type, then `T.class` is the matching class object. For example:

```
Class cl1 = Manager.class;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Note that a `Class` object really describes a *type,* which may or may not be a class. For example, `int` is not a class, but `int.class` is nevertheless an object of type `Class`.

NOTE

For historical reasons, the `getName` method returns somewhat strange names for array types:

```
System.out.println(Double[].class.getName());
   // prints [Ljava.lang.Double;
System.out.println(int[].class.getName());
   // prints [I
```

Catching Exceptions

We will cover exception handling fully in Chapter 11, but in the meantime you will occasionally encounter methods that threaten to throw exceptions.

When an error occurs at run time, a program can “throw an exception.” Throwing an exception is more flexible than terminating the program because you can provide a *handler* that “catches” the exception and deals with it.

If you don't provide a handler, the program still terminates and prints a message to the console, giving the type of the exception. You may already have seen exception reports when you accidentally used a `null` reference or overstepped the bounds of an array.

There are two kinds of exceptions: *unchecked* exceptions and *checked* exceptions. With checked exceptions, the compiler checks that you provide a handler. However, many common exceptions, such as accessing a `null` reference, are unchecked.

The compiler does not check whether you provide a handler for these errors—after all, you should spend your mental energy on avoiding these mistakes rather than coding handlers for them.

But not all errors are avoidable. If an exception can occur despite your best efforts, then the compiler insists that you provide a handler. The `Class.forName` method is an example of a method that throws a checked exception. In Chapter 11, you will see several exception handling strategies. For now, we'll just show you the simplest handler implementation.

Place one or more methods that might throw checked exceptions inside a `try` block. Then provide the handler code in the `catch` clause.

```
try
{
   statements that might throw exceptions
}
      catch(Exception e)
{
   handler action
}
```

Here is an example:

```
try
{
   String name = . . .;// get class name
   Class cl = Class.forName(name); // might throw exception
    . . . // do something with cl
}
catch(Exception e)
{
   e.printStackTrace();
}
```

If the class name doesn't exist, the remainder of the code in the `try` block is skipped, and the program enters the `catch` clause. (Here, we print a stack trace by using the `printStackTrace` method of the `Throwable` class. `Throwable` is the superclass of the `Exception` class.) If none of the methods in the `try` block throw an exception, then the handler code in the `catch` clause is skipped.

You only need to supply an exception handler for checked exceptions. It is easy to find out which methods throw checked exceptions—the compiler will complain whenever you call a method that threatens to throw a checked exception and you don't supply a handler.

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the == operator to compare class objects, for example:

```
if (e.getClass() == Employee.class) . . .
```

Another example of a useful method is one that lets you create an instance of a class on the fly. This method is called, naturally enough, `newInstance()`. For example,

```
e.getClass().newInstance();
```

creates a new instance of the same class type as `e`. The `newInstance` method calls the default constructor (the one that takes no parameters) to initialize the newly created object.

Using a combination of `forName` and `newInstance` lets you create an object from a class name stored in a string.

```
String s = "Manager";
Object m = Class.forName(s).newInstance();
```

NOTE

If you need to provide parameters for the constructor of a class you want to create by name in this manner, then you can't use statements like the above. Instead, you must use the `newInstance` method in the `Constructor` class. (This is one of several classes in the `java.lang.reflect` package. We will discuss reflection in the next section.)

C++ NOTE

The `newInstance` method corresponds to the idiom of a *virtual constructor* in C++. However, virtual constructors in C++ are not a language feature but just an idiom that needs to be supported by a specialized library. The `Class` class is similar to the `type_info` class in C++, and the `getClass` method is equivalent to the `typeid` operator. The Java `Class` is quite a bit more versatile than `type_info`, though. The C++ `type_info` can only reveal a string with the name of the type, not create new objects of that type.

`java.lang.Class`

- `static Class forName(String className)`

  returns the `Class` object representing the class with name `className`.

- `Object newInstance()`

  returns a new instance of this class.

`java.lang.reflect.Constructor`

- `Object newInstance(Object[] args)`

  constructs a new instance of the constructor's declaring class.

Parameters:	`args`	the parameters supplied to the constructor. See the section on reflection for more information on how to supply parameters.

`java.lang.Throwable`

- `void printStackTrace()`

  prints the `Throwable` object and the stack trace to the standard error stream.

Reflection

The class `Class` gives you a very rich and elaborate toolset to write programs that manipulate Java code dynamically. This feature is heavily used in *JavaBeans*, the component architecture for Java (see Volume 2 for more on JavaBeans). Using reflection, Java is able to support tools like the ones users of Visual Basic have grown accustomed to. In particular, when new classes are added at design or run time, rapid application development tools that are JavaBeans enabled need to be able to inquire about the capabilities of the classes that were added. (This is equivalent to the process that occurs when you add controls in Visual Basic to the toolbox.)

A program that can analyze the capabilities of classes is called *reflective.* The package that brings this functionality to Java is therefore called, naturally enough, `java.lang.reflect`. The reflection mechanism is extremely powerful. As the next four sections show, you can use it to:

- Analyze the capabilities of classes at run time
- Inspect objects at run time, for example, to write a single `toString` method that works for *all* classes
- Implement generic array manipulation code
- Take advantage of `Method` objects that work just like function pointers in languages such as C++

Reflection is a powerful and complex mechanism; however, it is of interest mainly to tool builders, not application programmers. If you are interested in programming applications rather than tools for other Java programmers, you can safely skip the remainder of this chapter and return to it at a later time.

Using Reflection to Analyze the Capabilities of Classes

Here is a brief overview of the most important parts of the reflection mechanism for letting you examine the structure of a class.

The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class, respectively. All three classes have a method called `getName` that returns the name of the item. The `Field` class has a method `getType` that returns an object, again of type `Class`, that describes the field type. The `Method` and `Constructor` classes have methods to report the return type and the types of the parameters used for these methods. All three of these classes also have a method called `getModifiers` that returns an integer, with various bits turned on and off, that describe the modifiers used, such as `public` and `static`. You can then use the static methods in the `Modifier` class in the `java.lang.reflect` package to analyze the integer that `getModifiers` returns. For example, there are methods like `isPublic`, `isPrivate`, or `isFinal` in the `Modifier` class that you could use to tell whether a method or constructor was `public`, `private`, or `final`. All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifiers` returns. You can also use the `Modifier.toString` method to print the modifiers.

The `getFields`, `getMethods`, and `getConstructors` methods of the `Class` class return arrays of the *public* fields, operations, and constructors that the class supports. This includes public members of superclasses. The `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` methods of the `Class` class return arrays consisting of all fields, operations, and constructors that are declared in the class. This includes private and protected members, but not members of superclasses.

Example 5-5 shows you how to print out all information about a class. The program prompts you for the name of a class and then writes out the signatures of all methods and constructors as well as the names of all data fields of a class. For example, if you enter

```
java.lang.Double
```

then the program prints:

```
class java.lang.Double extends java.lang.Number
{
   public java.lang.Double(java.lang.String);
   public java.lang.Double(double);

   public int hashCode();
   public int compareTo(java.lang.Object);
   public int compareTo(java.lang.Double);
   public boolean equals(java.lang.Object);
   public java.lang.String toString();
   public static java.lang.String toString(double);
   public static java.lang.Double valueOf(java.lang.String);
   public static boolean isNaN(double);
   public boolean isNaN();
   public static boolean isInfinite(double);
   public boolean isInfinite();
   public byte byteValue();
   public short shortValue();
   public int intValue();
   public long longValue();
   public float floatValue();
   public double doubleValue();
   public static double parseDouble(java.lang.String);
   public static native long doubleToLongBits(double);
   public static native long doubleToRawLongBits(double);
   public static native double longBitsToDouble(long);

   public static final double POSITIVE_INFINITY;
   public static final double NEGATIVE_INFINITY;
   public static final double NaN;
   public static final double MAX_VALUE;
   public static final double MIN_VALUE;
   public static final java.lang.Class TYPE;
   private double value;
   private static final long serialVersionUID;
}
```

What is remarkable about this program is that it can analyze any class that the Java interpreter can load, not just the classes that were available when the program was compiled. We will use this program in the next chapter to peek inside the inner classes that the Java compiler generates automatically.

Example 5-5 ReflectionTest.java

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

public class ReflectionTest
{
   public static void main(String[] args)
   {
      // read class name from command line args or user input
      String name;
      if (args.length > 0)
         name = args[0];
      else
         name = JOptionPane.showInputDialog
            ("Class name (e.g. java.util.Date): ");

      try
      {
         // print class name and superclass name (if != Object)
         Class cl = Class.forName(name);
         Class supercl = cl.getSuperclass();
         System.out.print("class " + name);
         if (supercl != null && supercl != Object.class)
            System.out.print(" extends " + supercl.getName());

         System.out.print("\n{\n");
         printConstructors(cl);
         System.out.println();
         printMethods(cl);
         System.out.println();
         printFields(cl);
         System.out.println("}");
      }
      catch(ClassNotFoundException e) { e.printStackTrace(); }
```

```
         System.exit(0);
      }

      /**
         Prints all constructors of a class
         @param cl a class
       */
      public static void printConstructors(Class cl)
      {
         Constructor[] constructors = cl.getDeclaredConstructors();

         for (int i = 0; i < constructors.length; i++)
         {
            Constructor c = constructors[i];
            String name = c.getName();
            System.out.print(Modifier.toString(c.getModifiers()));
            System.out.print("   " + name + "(");

            // print parameter types
            Class[] paramTypes = c.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
               if (j > 0) System.out.print(", ");
               System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
         }
      }

      /**
         Prints all methods of a class
         @param cl a class
       */
      public static void printMethods(Class cl)
      {
         Method[] methods = cl.getDeclaredMethods();

         for (int i = 0; i < methods.length; i++)
         {
            Method m = methods[i];
            Class retType = m.getReturnType();
            String name = m.getName();

            // print modifiers, return type and method name
            System.out.print(Modifier.toString(m.getModifiers()));
            System.out.print("   " + retType.getName() + " " + name
               + "(");

            // print parameter types
            Class[] paramTypes = m.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
               if (j > 0) System.out.print(", ");
               System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
         }
      }

```

```
   /**
      Prints all fields of a class
      @param cl a class
    */
   public static void printFields(Class cl)
   {
      Field[] fields = cl.getDeclaredFields();

      for (int i = 0; i < fields.length; i++)
      {
         Field f = fields[i];
         Class type = f.getType();
         String name = f.getName();
         System.out.print(Modifier.toString(f.getModifiers()));
         System.out.println("   " + type.getName() + " " + name
            + ";");
      }
   }
}
```

java.lang.Class

- `Field[] getFields()`
- `Field[] getDeclaredFields()`

The `getFields` method returns an array containing `Field` objects for the public fields. The `getDeclaredField` method returns an array of `Field` objects for all fields. The methods return an array of length 0 if there are no such fields, or if the `Class` object represents a primitive or array type.

- `Method[] getMethods()`
- `Method[] getDeclaredMethods()`

return an array containing `Method` objects that give you all the public methods (for `getMethods`) or all methods (for `getDeclaredMethods`) of the class or interface. This includes those inherited from classes or interfaces above it in the inheritance chain.

- `Constructor[] getConstructors()`
- `Constructor[] getDeclaredConstructors()`

return an array containing `Constructor` objects that give you all the public constructors (for `getConstructors`) or all constructors (for `getDeclaredConstructors`) of the class represented by this Class object.

`java.lang.reflect.Field`

`java.lang.reflect.Method`

`java.lang.reflect.Constructor`

- `Class getDeclaringClass()`

  returns the `Class` object for the class that defines this constructor, method, or field.

- `Class[] getExceptionTypes()` (in `Constructor` and `Method` classes)

  returns an array of `Class` objects that represent the types of the exceptions thrown by the method.

- `int getModifiers()`

  returns an integer that describes the modifiers of this constructor, method, or field. Use the methods in the `Modifier` class to analyze the return value.

- `String getName()`

  returns a string that is the name of the constructor, method, or field.

- `Class[] getParameterTypes()` (in `Constructor` and `Method` classes)

  returns an array of `Class` objects that represent the types of the parameters.

`java.lang.reflect.Modifier`

- `static String toString(int modifiers)`

  returns a string with the modifiers that correspond to the bits set in `modifiers`.

Using Reflection to Analyze Objects at Run Time

In the preceding section, we saw how we can find out the *names* and *types* of the data fields of any object:

- Get the corresponding `Class` object.
- Call `getDeclaredFields` on the `Class` object.

In this section, we go one step further and actually look at the *contents* of the data fields. Of course, it is easy to look at the contents of a specific field of an object whose name and type are known when you write a program. But reflection lets you look at fields of objects that were not known at compile time.

The key method to achieve this is the `get` method in the `Field` class. If `f` is an object of type `Field` (for example, one obtained from `getDeclaredFields`) and `obj` is an object of the class of which `f` is a field, then `f.get(obj)` returns an object whose value is the current value of the field of `obj`. This is all a bit abstract, so let's run through an example.

```
Employee harry = new Employee("Harry Hacker", 35000,
   10, 1, 1989);
Class cl = harry.getClass();
   // the class object representing Employee
Field f = cl.getField("name");
   // the name field of the Employee class
Object v = f.get(harry);
   // the value of the name field of the harry object
   // i.e. the String object "Harry Hacker"
```

Actually, there is a problem with this code. Since the name field is a private field, the get method will throw an `IllegalAccessException`. You can only use the `get` method to get the values of accessible fields. The security mechanism of Java lets you find out what fields any object has, but it won't let you read the values of those fields unless you have access permission.

The default behavior of the reflection mechanism is to respect Java access control. However, if a Java program is not controlled by a security manager that disallows it, it is possible to override access control. To do this, invoke the `setAccessible` method on a `Field`, `Method`, or `Constructor` object, for example:

```
f.setAccessible(true);
   // now OK to call f.get(harry);
```

The `setAccessible` method is a method of the `AccessibleObject` class, the common superclass of the `Field`, `Method`, and `Constructor`. This feature is provided for debuggers, persistent storage, and similar mechanisms. We will use it for a generic `toString` method later in this section.

There is another issue with the `get` method that we need to deal with. The `name` field is a `String`, and so it is not a problem to return the value as an `Object`. But suppose we want to look at the `salary` field. That is a `double`, and in Java, number types are not objects. To handle this, you can either use the `getDouble` method of the `Field` class, or you can call `get`, where the reflection mechanism automatically wraps the field value into the appropriate wrapper class, in this case, `Double`.

Of course, you can also set the values that you can get. The call `f.set(obj, value)` sets the field represented by `f` of the object `obj` to the new value.

Example 5-6 shows how to write a generic `toString` method that works for *any* class. It uses `getDeclaredFields` to obtain all data fields. It then uses the `setAccessible` convenience method to make all fields accessible. For each field, it obtains the name and the value. Each value is turned into a string by invoking *its* `toString` method. The `toString` method examines all superclass fields until it reaches the `Object` class.

```
class ObjectAnalyzer
{
  public static String toString(Object obj)
   {
      Class cl = obj.getClass();
      String r = cl.getName();

      // inspect the fields of this class and all superclasses
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);

         // get the names and values of all fields
         for (int i = 0; i < fields.length; i++)
         {
            Field f = fields[i];
            r += f.getName() + "=";
            try
            {
               Object val = f.get(obj);
               r += val.toString();
            }
            catch (Exception e) { e.printStackTrace(); }
            if (i < fields.length - 1)
               r += ",";
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != Object.class);

      return r;
   }
   . . .
}
```

You can use this `toString` method to peek inside any object. For example, here is what you get when you look inside the `System.out` object:

```
java.io.PrintStream[autoFlush=true,trouble=false,
textOut=java.io.BufferedWriter@8786b,
charOut=java.io.OutputStreamWriter@19c082,
closing=false][out=java.io.BufferedOutputStream@2dd2dd][]
```

In Example 5-6, the generic `toString` method is used to implement the `toString` of the `Employee` class:

```
public String toString()
{
   return ObjectAnalyzer.toString(this);
}
```

This is a hassle-free method for supplying a `toString` method, and it is highly recommended, especially for debugging. The same recursive approach can also be used to define a generic `equals` method. See the code in the example program Example 5-6 for details.

Example 5-6 ObjectAnalyzerTest.java

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

public class ObjectAnalyzerTest
{
   public static void main(String[] args)
   {
      // test toString method of Employee
      Employee harry  = new Employee("Harry Hacker", 35000,
         1996, 12, 1);
      System.out.println("harry=" + harry);

      // test equals method of Employee
      Employee coder  = new Employee("Harry Hacker", 35000,
         1996, 12, 1);
      System.out.println(
         "Before raise, harry.equals(coder) returns "
         + harry.equals(coder));
      harry.raiseSalary(5);
      System.out.println(
         "After raise, harry.equals(coder) returns "
         + harry.equals(coder));

      Manager carl = new Manager("Carl Cracker", 80000,
         1987, 12, 15);
      Manager boss = new Manager("Carl Cracker", 80000,
         1987, 12, 15);
      boss.setBonus(5000);
      System.out.println("boss=" + boss);
      System.out.println(
         "carl.equals(boss) returns " + carl.equals(boss));
   }
}

class ObjectAnalyzer
{
   /**
      Converts an object to a string representation that lists
      all fields.
      @param obj an object
      @return a string with the object's class name and all
      field names and values
   */
   public static String toString(Object obj)
   {
      Class cl = obj.getClass();
      String r = cl.getName();

      // inspect the fields of this class and all superclasses
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);

         // get the names and values of all fields
         for (int i = 0; i < fields.length; i++)
         {
            Field f = fields[i];
            r += f.getName() + "=";
            try
            {
               Object val = f.get(obj);
               r += val.toString();
            }
            catch (Exception e) { e.printStackTrace(); }
            if (i < fields.length - 1)
               r += ",";
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != Object.class);

      return r;
   }

   /**
      Tests whether two objects are equal by checking if all
      field values are equal
      @param a an object
      @param b another object
      @return true if a and b are equal
   */
   public static boolean equals(Object a, Object b)
   {
      if (a == b) return true;
      if (a == null || b == null) return false;
      Class cl = a.getClass();
      if (cl != b.getClass()) return false;

      // inspect the fields of this class and all superclasses
      do
      {
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         for (int i = 0; i < fields.length; i++)
         {
            Field f = fields[i];
            // if field values don't match, objects aren't equal
            try
            {
               if (!f.get(a).equals(f.get(b)))
                  return false;
            }
            catch (Exception e) { e.printStackTrace(); }
         }
         cl = cl.getSuperclass();
      }
      while (cl != Object.class);

      return true;
   }
}

class Employee
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar
         = new GregorianCalendar(year, month - 1, day);
         // GregorianCalendar uses 0 for January
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return ObjectAnalyzer.toString(this);
   }

   public boolean equals(Object b)
   {
      return ObjectAnalyzer.equals(this, b);
   }

   private String name;
   private double salary;
   private Date hireDay;
}

```

```
class Manager extends Employee
{
   public Manager(String n, double s,
      int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }

   public double getSalary()
   {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }

   public void setBonus(double b)
   {
      bonus = b;
   }

   private double bonus;
}
```

`java.lang.reflect.AccessibleObject`

- `void setAccessible(boolean flag)`

  sets the accessibility flag for this reflection object. A value of `true` indicates that Java language access checking is suppressed, and that the private properties of the object can be queried and set.

- `boolean isAccessible()`

  gets the value of the accessibility flag for this reflection object.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

  is a convenience method to set the accessibility flag for an array of objects.

Using Reflection to Write Generic Array Code

The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. For example, when you use this feature with the `arrayCopy` method from Chapter 3, you can dynamically expand an existing array while preserving the current contents.

The problem we want to solve is pretty typical. Suppose you have an array of some type that is full and you want to grow it. And suppose you are sick of writing the grow-and-copy code by hand. You want to write a generic method to grow an array.

```
Employee[] a = new Employee[100];
. . .
// array is full
a = (Employee[])arrayGrow(a);
```

How can we write such a generic method? It helps that an `Employee[]` array can be converted to an `Object[]` array. That sounds promising. Here is a first attempt to write a generic method. We simply grow the array by 10% + 10 elements (since the 10% growth is not substantial enough for small arrays).

```
static Object[] arrayGrow(Object[] a) // not useful
{
   int newLength = a.length * 11 / 10 + 10;
   Object[] newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0, a.length);
   return newArray;
}
```

However, there is a problem with actually *using* the resulting array. The type of array that this code returns is an array of *objects* (`Object[]`) because we created the array using the line of code:

```
new Object[newLength]
```

An array of objects *cannot* be cast to an array of employees (`Employee[]`). Java would generate a `ClassCast` exception at run time. The point is, as we mentioned earlier, that a Java array remembers the type of its entries, that is, the element type used in the `new` expression that created it. It is legal to cast an `Employee[]` temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into an `Employee[]` array. To write this kind of generic array code, we need to be able to make a new array of the *same* type as the original array. For this, we need the methods of the `Array` class in the `java.lang.reflect` package. The key is the static `newInstance` method of the `Array` class that constructs a new array. You must supply the type for the entries and the desired length as parameters to this method.

```
Object newArray = Array.newInstance(componentType, newLength);
```

To actually carry this out, we need to get the length and component type of the new array.

The length is obtained by calling `Array.getLength(a)`. The static `getLength` method of the `Array` class returns the length of any array. To get the component type of the new array:

1. First, get the class object of `a`.
2. Confirm that it is indeed an array.
3. Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.

Why is `getLength` a method of `Array` but `getComponentType` a method of `Class`? We don't know—the distribution of the reflection methods seems a bit ad hoc at times.

Here's the code:

```
static Object arrayGrow(Object a) // useful
{
   Class cl = a.getClass();
   if (!cl.isArray()) return null;
   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   int newLength = length * 11 / 10 + 10;
   Object newArray = Array.newInstance(componentType,
      newLength);
   System.arraycopy(a, 0, newArray, 0, length);
   return newArray;
}
```

Note that this `arrayGrow` method can be used to grow arrays of any type, not just arrays of objects.

```
int[] ia = { 1, 2, 3, 4 };
ia = (int[])arrayGrow(ia);
```

To make this possible, the parameter of `arrayGrow` is declared to be of type `Object`, *not an array of objects* (`Object[]`). The integer array type `int[]` can be converted to an `Object`, but not to an array of objects!

Example 5-7 shows both array grow methods in action. Note that the cast of the return value of `badArrayGrow` will throw an exception.

Example 5-7 ArrayGrowTest.java

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

public class ArrayGrowTest
{
   public static void main(String[] args)
   {
      int[] a = { 1, 2, 3 };
      a = (int[])goodArrayGrow(a);
      arrayPrint(a);

      String[] b = { "Tom", "Dick", "Harry" };
      b = (String[])goodArrayGrow(b);
      arrayPrint(b);

      System.out.println
         ("The following call will generate an exception.");
      b = (String[])badArrayGrow(b);
   }

   /**
      This method attempts to grow an array by allocating a
      new array and copying all elements.
      @param a the array to grow
      @return a larger array that contains all elements of a.
      However, the returned array has type Object[], not
      the same type as a
   */
```

```
    static Object[] badArrayGrow(Object[] a)
    {
       int newLength = a.length * 11 / 10 + 10;
       Object[] newArray = new Object[newLength];
       System.arraycopy(a, 0, newArray, 0, a.length);
       return newArray;
    }

    /**
       This method grows an array by allocating a
       new array of the same type and copying all elements.
       @param a the array to grow. This can be an object array
       or a fundamental type array
       @return a larger array that contains all elements of a.

    */
    static Object goodArrayGrow(Object a)
    {
       Class cl = a.getClass();
       if (!cl.isArray()) return null;
       Class componentType = cl.getComponentType();
       int length = Array.getLength(a);
       int newLength = length * 11 / 10 + 10;

       Object newArray = Array.newInstance(componentType,
          newLength);
       System.arraycopy(a, 0, newArray, 0, length);
       return newArray;
    }

    /**
       A convenience method to print all elements in an array
       @param a the array to print. can be an object array
       or a fundamental type array
    */
    static void arrayPrint(Object a)
    {
       Class cl = a.getClass();
       if (!cl.isArray()) return;
       Class componentType = cl.getComponentType();
       int length = Array.getLength(a);
       System.out.print(componentType.getName()
          + "[" + length + "] = { ");
       for (int i = 0; i < Array.getLength(a); i++)
          System.out.print(Array.get(a, i)+ " ");
       System.out.println("}");
    }
}
```

Method Pointers!

On the surface, Java does not have method pointers—ways of giving the location of a method to another method so that the second method can invoke it later. In fact, the designers of Java have said that method pointers are dangerous and error-prone and that Java *interfaces* (discussed in the next chapter) are a superior solution. However, it turns out that Java now does have method pointers, as a (perhaps accidental) byproduct of the reflection package.

NOTE

Among the nonstandard language extensions that Microsoft added to its Java derivative J++ (and its successor, C#) is another method pointer type that is different from the `Method` class that we discuss in this section. However, as you will see in chapter 6, inner classes are a more useful and general mechanism.

To see method pointers at work, recall that you can inspect a field of an object with the `get` method of the `Field` class. Similarly, the `Method` class has an `invoke` method that lets you call the method that is wrapped in the current `Method` object. The signature for the `invoke` method is:

```
Object invoke(Object obj, Object[] args)
```

The first parameter is the implicit parameter, and the array of objects provides the explicit parameters. For a static method, the first parameter is ignored—you can set it to `null`. If the method has no explicit parameters, you can pass `null` or an array of length 0 for the `args` parameter. For example, if `m1` represents the `getName` method of the `Employee` class, the following code shows how you can call it:

```
String n = (String)m1.invoke(harry, null);
```

As with the `get` and `set` methods of the `Field` type, there's a problem if the parameter or return type is not a class but a basic type. You must wrap any of the basic types into their corresponding wrappers before inserting them into the `args` array. Conversely, the `invoke` method will return the wrapped type and not the basic type. For example, suppose that `m2` represents the `raiseSalary` method of the `Employee` class. Then, you need to wrap the `double` parameter into a `Double` object.

```
Object[] args = { new Double(5.5) };
m2.invoke(harry, args);
```

How do you obtain a `Method` object? You can, of course, call `getDeclaredMethods` and search through the returned array of `Method` objects until you find the method that you want. Or, you can call the `getMethod` method of the `Class` class. This is similar to the `getField` method that takes a string with the field name and returns a `Field` object. However, there may be several methods with the same name, so you need to be careful that you get the right one. For that reason, you must also supply an array that gives the correct parameter types. For example, here is how you can get method pointers to the `getName` and `raiseSalary` methods of the `Employee` class.

```
Method m1 = Employee.class.getMethod("getName", null);
Methodm2 = Employee.class.getMethod("raiseSalary",
   new Class[] { double.class } );
```

The second parameter of the `getMethod` method is an array of `Class` objects. Since the `raiseSalary` method has one parameter of type `double`, we must supply an array with a single element, `double.class`. It is usually easiest to make that array on the fly, as we did in the example above. The expression

```
new Class[] { double.class }
```

denotes an array of `Class` objects, filled with one element, namely, the class object `double.class`.

Now that you have seen the syntax of `Method` objects, let's put them to work. Example 5-8 is a program that prints a table of values for a mathematical function such as `Math.sqrt` or `Math.sin`. The printout looks like this:

```
public static native double java.lang.Math.sqrt(double)
      1.0000 |      1.0000
      2.0000 |      1.4142
      3.0000 |      1.7321
      4.0000 |      2.0000
      5.0000 |      2.2361
      6.0000 |      2.4495
      7.0000 |      2.6458
      8.0000 |      2.8284
      9.0000 |      3.0000
     10.0000 |      3.1623
```

The code for printing a table is, of course, independent of the actual function that is being tabulated.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = f(x);
      // where f is the function to be tabulated
      // not the actual syntax--see below
   System.out.println(x + " | " + y);
}
```

We want to write a generic `printTable` method that can tabulate any function. We will pass the function as a parameter of type `Method`.

```
static void printTable(double from, double to, int n, Method f)
```

Of course, `f` is an object and not a function, so we cannot simply write `f(x)` to evaluate it. Instead, we must supply `x` in the parameter array (suitably wrapped as a `Double`), use the `invoke` method, and unwrap the return value.

```
Object[] args = { new Double(x) };
Double d = (Double)f.invoke(null, args);
double y = d.doubleValue();
```

The first parameter of `invoke` is `null` because we are calling a static method.

Here is a sample call to `printTable` that tabulates the square root function.

```
printTable(1, 10, 10,
   java.lang.Math.class.getMethod("sqrt",
   new Class[] { double.class }));
```

The hardest part is to get the method object. Here, we get the method of the `java.lang.Math` class that has the name `sqrt` and whose parameter list contains just one type, `double`.

Example 5-8 shows the complete code of the `printTable` method and a couple of test runs.

Example 5-8 MethodPointerTest.java

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

public class MethodPointerTest
{
   public static void main(String[] args) throws Exception
   {
      // get method pointers to the square and sqrt methods
      Method square = MethodPointerTest.class.getMethod("square",
         new Class[] { double.class });
      Method sqrt = java.lang.Math.class.getMethod("sqrt",
         new Class[] { double.class });

      // print tables of x- and y-values

      printTable(1, 10, 10, square);
      printTable(1, 10, 10, sqrt);
   }

   /**
      Returns the square of a number
      @param x a number
      @return x squared
   */
   public static double square(double x)
   {
      return x * x;
   }

   /**
      Prints a table with x- and y-values for a method
      @param from the lower bound for the x-values
      @param to the upper bound for the x-values
      @param n the number of rows in the table
      @param f a method with a double parameter and double
         return value
   */
   public static void printTable(double from, double to,
      int n, Method f)
   {
      // print out the method as table header
      System.out.println(f);

      // construct formatter to print with 4 digits precision

      NumberFormat formatter = NumberFormat.getNumberInstance();
      formatter.setMinimumFractionDigits(4);
      formatter.setMaximumFractionDigits(4);
      double dx = (to - from) / (n - 1);

      for (double x = from; x <= to; x += dx)
      {
         // print x-value
         String output = formatter.format(x);
         // pad with spaces to field width of 10
         for (int i = 10 - output.length(); i > 0; i--)
            System.out.print(' ');
         System.out.print(output + " |");

         try
         {
            // invoke method and print y-value
            Object[] args = { new Double(x) };
            Double d = (Double)f.invoke(null, args);
            double y = d.doubleValue();

            output = formatter.format(y);
            // pad with spaces to field width of 10
            for (int i = 10 - output.length(); i > 0; i--)
               System.out.print(' ');

            System.out.println(output);
         }
         catch (Exception e) {  e.printStackTrace(); }
      }
   }
}
```

As this example shows clearly, you can do anything with `Method` objects that you can do with function pointers in C. Just as in C, this style of programming is usually quite inconvenient and always error-prone. What happens if you invoke a method with the wrong parameters? The `invoke` method throws an exception.

Also, the parameters and return values of `invoke` are necessarily of type `Object`. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code. Therefore, errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than simply calling methods directly.

For that reason, we suggest that you use `Method` objects in your own programs only when absolutely necessary. Using interfaces and inner classes (the subject of the next chapter) is almost always a better idea. In particular, we echo the developers of Java and suggest not using `Method` objects for callback functions. Using interfaces for the callbacks (see the next chapter as well) leads to code that runs faster and is a lot more maintainable.

Design Hints for Inheritance

We want to end this chapter with some hints for using inheritance that we have found useful.

1. *Place common operations and fields in the superclass.*

   This is why we put the name field into the `Person` class, rather than replicating it in `Employee` and `Student`.