Chapter 9 Interfaces and Polymorphism

CHAPTER GOALS

- To learn about interfaces
- To be able to convert between class and interface references
- To understand the concept of polymorphism
- To appreciate how interfaces can be used to decouple classes
- To learn how to implement helper classes as inner classes
- To understand how inner classes access variables from the surrounding scope

G To implement event listeners in graphical applications

In order to increase programming productivity, we want to be able to *reuse* software components in multiple projects. However, some adaptations are often required to make reuse possible. In this chapter, you will learn an important strategy for separating the reusable part of a computation from the parts that vary in each reuse scenario. The reusable part invokes methods of an *interface*. It is combined with a class that implements the interface methods. To produce a different application, you simply plug in another class that implements the same methods. The program's behavior varies according to the class that was plugged in—this phenomenon is called *polymorphism*.

9.1 Using Interfaces for Code Reuse

It is often possible to make code more general and more reusable by focusing on the essential operations that are carried out. *Interface types* are used to express these common operations.

Use interface types to make code more reusable.

Consider the `DataSet` class of Chapter 6. We used that class to compute the average and maximum of a set of input values. However, the class was suitable only for computing the average of a set of *numbers*. If we wanted to process bank accounts to find the bank account with the highest balance, we would have to modify the class, like this:

```
public class DataSet// Modified for BankAccount objects
{
   . . .
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() <
x.getBalance())
         maximum = x;
      count ++;
   }
   public BankAccount getMaximum()
   {
      return maximum;
   }
   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Or suppose we wanted to find the coin with the highest value among a set of coins. We would need to modify the `DataSet` class again.

```
public class DataSet // Modified for Coin objects
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() <
x.getValue())
         maximum = x;
      count++;
   }
   public Coin getMaximum()
   {
      return maximum;
```

```
   }
   private double sum;
   private Coin maximum;
   private int count;
}
```

Clearly, the fundamental mechanics of analyzing the data is the same in all cases, but the details of measurement differ.

Suppose that the various classes agree on a single method `getMeasure` that obtains the measure to be used in the data analysis. For bank accounts, `getMeasure` returns the balance. For coins, `getMeasure` returns the coin value, and so on. Then we can implement a single reusable `DataSet` class whose add method looks like this:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() <
x.getMeasure())
   maximum = x;
count++;
```

What is the type of the variable `x`? Ideally, `x` should refer to any class that has a `getMeasure` method.

A Java interface type declares a set of methods and their signatures.

In Java, an *interface type* is used to specify required operations. We will define an interface type that we call `Measurable`:

```
public interface Measurable
{
   double getMeasure();
}
```

389

390

The interface declaration lists all methods that the interface type requires. The `Measurable` interface type requires a single method, but in general, an interface type can require multiple methods.

Note that the `Measurable` type is not a type in the standard library—it is a type that was created specifically for this book, in order to make the `DataSet` class more reusable.

Unlike a class, an interface type provides no implementation.

An interface type is similar to a class, but there are several important differences:

- All methods in an interface type are *abstract;* that is, they have a name, parameters, and a return type, but they don't have an implementation.
- All methods in an interface type are automatically public.
- An interface type does not have instance fields.

Now we can use the interface type `Measurable` to declare the variables `x` and `maximum`.

```
public class DataSet
{
   . . .
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() <
x.getMeasure())
         maximum = x;
      count++;
   }
   public Measurable getMaximum()
   {
      return maximum;
   }
   private double sum;
   private Measurable maximum;
   private int count;
}
```

Use the `implements` keyword to indicate that a class implements an interface type.

This `DataSet` class is usable for analyzing objects of any class that *implements* the `Measurable` interface. A class implements an interface type if it declares the

interface in an `implements` clause. It should then implement the method or methods that the interface requires.

```
class ClassName implements Measurable
{
   public double getMeasure()
   {
      Implementation
   }
   Additional methods and fields
}
```

390

391

A class can implement more than one interface type. Of course, the class must then define all the methods that are required by all the interfaces it implements.

Let us modify the `BankAccount` class to implement the `Measurable` interface.

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

Note that the class must declare the method as `public`, whereas the interface need not—all methods in an interface are public.

Similarly, it is an easy matter to modify the `Coin` class to implement the `Measurable` interface.

```
public class Coin implements Measurable
{
   public double getMeasure()
   {
      return value;
   }
   . . .
}
```

In summary, the `Measurable` interface expresses what all measurable objects have in common. This commonality makes the `DataSet` class reusable. Objects of the `DataSet` class can be used to analyze collections of objects of *any* class that

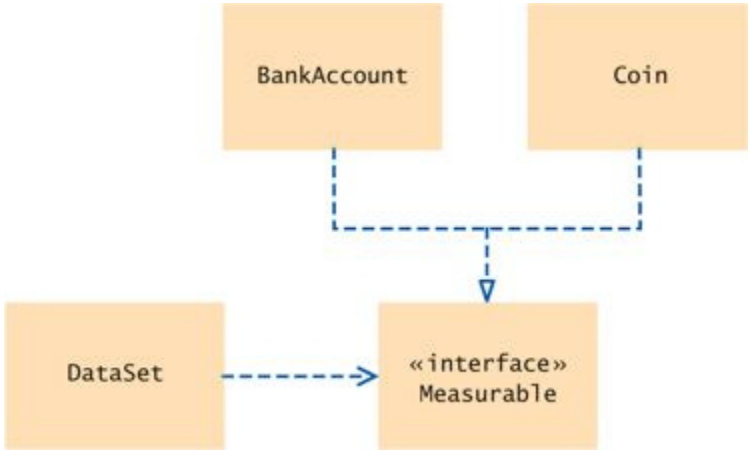
implements the `Measurable` interface. Following is a test program that illustrates that fact.

Interfaces can reduce the coupling between classes.

Figure 1 shows the relationships between the `Measurable` interface, the classes that implement the interface, and the `DataSet` class that uses the interface. In the UML notation, interfaces are tagged with a "stereotype" indicator «`interface`». A dotted arrow with a triangular tip denotes the "*is-a*" relationship between a class and an interface. You have to look carefully at the arrow tips—a dotted line with an open arrow tip denotes the "*uses*" relationship or dependency.

391
392

Figure 1



UML Diagram of the `DataSet` Class and the Classes that Implement the `Measurable` Interface

This diagram shows that the `DataSet` class depends only on the `Measurable` interface. It is decoupled from the `BankAccount` and `Coin` classes.

SYNTAX 9.1: Defining an Interface

```
public interface InterfaceName
{
   method signatures
}
```

Example

```
public interface Measurable
{
   double getMeasure();
}
```

Purpose

To define an interface and its method signatures. The methods are automatically public.

SYNTAX 9.2: Implementing an Interface

```
public class ClassName
      implements InterfaceName, InterfaceName, . . .
{
   methods
   fields
}
```

Example

```
public class BankAccount implements Measurable
{
    // Other BankAccount methods
    public double getMeasure()
    {
       // Method implementation
    }
}
```

Purpose

To define a new class that implements the methods of an interface

ch09/measure1/DataSetTester.java

```
1  /**
2     This program tests the DataSet class.
3  */
4  public class DataSetTester
```

```
 5  {
 6     public static void main(String[] args)
 7     {
 8        DataSet bankData = new DataSet();
 9
10        bankData.add(new BankAccount(0));
11        bankData.add(new BankAccount(10000));
12        bankData.add(new BankAccount(2000));
13
14        System.out.println("Average balance: "
15              + bankData.getAverage());
16        System.out.println("Expected: 4000");
17        Measurable max = bankData.getMaximum();
18        System.out.println("Highest balance: "
19              + max.getMeasure());
20        System.out.println("Expected: 10000");
21
22        DataSet coinData = new DataSet();
23
24        coinData.add(new Coin(0.25, "quarter"));
25        coinData.add(new Coin(0.1, "dime"));
26        coinData.add(new Coin(0.05, "nickel"));
27
28        System.out.println("Average coin value: "
29              + coinData.getAverage());
30        System.out.println("Expected: 0.133");
31        max = coinData.getMaximum();
32        System.out.println("Highest coin value: "
33              + max.getMeasure());
34        System.out.println("Expected: 0.25");
35     }
36  }
```

Output

```
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

Self Check

1. Suppose you want to use the `DataSet` class to find the `Country` object with the largest population. What condition must the `Country` class fulfill?
2. Why can't the `add` method of the `DataSet` class have a parameter of type `Object`?

Common Error 9.1: Forgetting to Define Implementing Methods as Public

The methods in an interface are not declared as `public`, because they are public by default. However, the methods in a class are not public by default—their default access level is "package" access, which we discuss in Chapter 10. It is a Common Error to forget the `public` keyword when defining a method from an interface:

```
public class BankAccount implements Measurable
{
   double getMeasure() // Oops should be public
   {
      return balance;
   }
   . . .
}
```

Then the compiler complains that the method has a weaker access level, namely package access instead of public access. The remedy is to declare the method as public.

Advanced Topic 9.1: Constants in Interfaces

Interfaces cannot have instance fields, but it is legal to specify *constants*. For example, the `SwingConstants` interface defines various constants, such as `SwingConstants.NORTH`, `SwingConstants.EAST`, and so on.

When defining a constant in an interface, you can (and should) omit the keywords `public static final`, because all fields in an interface are automatically `public static final`. For example,

```
public interface SwingConstants
{
   int NORTH = 1;
   int NORTHEAST = 2;
   int EAST = 3;
   . . .
}
```

394

395

9.2 Converting Between Class and Interface Types

Interfaces are used to express the commonality between classes. In this section, we discuss when it is legal to convert between class and interface types.

Have a close look at the call

```
bankData.add(new BankAccount(10000));
```

from the test program of the preceding section. Here we pass an object of type `BankAccount` to the `add` method of the `DataSet` class. However, that method has a parameter of type `Measurable`:

```
public void add(Measurable x)
```

Is it legal to convert from the `BankAccount` type to the `Measurable` type?

You can convert from a class type to an interface type, provided the class implements the interface.

In Java, such a type conversion is legal. You can convert from a class type to the type of any interface that the class implements. For example,

```
BankAccount account = new BankAccount(10000);
Measurable x = account; // OK
```

Alternatively, `x` can refer to a `Coin` object, provided the `Coin` class has been modified to implement the `Measurable` interface.

```
Coin dime = new Coin(0.1, "dime");
Measurable x = dime; // Also OK
```

Thus, when you have an object variable of type `Measurable`, you don't actually know the exact type of the object to which `x` refers. All you know is that the object has a `getMeasure` method.

However, you cannot convert between unrelated types:

```
Measurable x = new Rectangle(5, 10, 20, 30);// Error
```

That assignment is an error, because the `Rectangle` class doesn't implement the `Measurable` interface.

Occasionally, it happens that you convert an object to an interface reference and you need to convert it back. This happens in the `getMaximum` method of the `DataSet` class. The `DataSet` stores the object with the largest measure, *as a* `Measurable` *reference*.

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum();
```

Now what can you do with the `max` reference? *You* know it refers to a `Coin` object, but the compiler doesn't. For example, you cannot call the `getName` method:

```
String coinName = max.getName(); // Error
```

That call is an error, because the `Measurable` type has no `getName` method.

395

396

However, as long as you are absolutely sure that `max` refers to a `Coin` object, you can use the *cast* notation to convert it back:

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

You need a cast to convert from an interface type to a class type.

If you are wrong, and the object doesn't actually refer to a coin, your program will throw an exception and terminate.

This cast notation is the same notation that you saw in Chapter 4 to convert between number types. For example, if `x` is a floating-point number, then `(int) x` is the integer part of the number. The intent is similar—to convert from one type to another. However, there is one big difference between casting of number types and casting of class types. When casting number types, you *lose information*, and you use the cast to tell the compiler that you agree to the information loss. When casting object types, on the other hand, you *take a risk* of causing an exception, and you tell the compiler that you agree to that risk.

Self Check

3. Can you use a cast `(BankAccount) x` to convert a `Measurable` variable `x` to a `BankAccount` reference?

4. If both `BankAccount` and `Coin` implement the `Measurable` interface, can a `Coin` reference be converted to a `BankAccount` reference?

Common Error 9.2: Trying to Instantiate an Interface

You can define variables whose type is an interface, for example:

```
Measurable x;
```

However, you can *never* construct an interface:

```
Measurable x = new Measurable(); // Error
```

Interfaces aren't classes. There are no objects whose types are interfaces. If an interface variable refers to an object, then the object must belong to some class—a class that implements the interface:

```
Measurable x = new BankAccount(); // OK
```

9.3 Polymorphism

When multiple classes implement the same interface, each class implements the methods of the interface in different ways. How is the correct method executed when the interface method is invoked? We will answer that question in this section.

396

397

It is worth emphasizing once again that it is perfectly legal—and in fact very common—to have variables whose type is an interface, such as

```
Measurable x;
```

Just remember that the object to which `x` refers doesn't have type `Measurable`. In fact, *no object* has type `Measurable`. Instead, the type of the object is some class that implements the `Measurable` interface, such as `BankAccount` or `Coin`.

Note that `x` can refer to objects of *different* types during its lifetime. Here the variable `x` first contains a reference to a bank account, then a reference to a coin.

```
x = new BankAccount(10000); // OK
x = new Coin(0.1, "dime");// OK
```

What can you do with an interface variable, given that you don't know the class of the object that it references? You can invoke the methods of the interface:

```
double m = x.getMeasure();
```

The `DataSet` class took advantage of this capability by computing the measure of the added object, without worrying exactly what kind of object was added.

Now let's think through the call to the `getMeasure` method more carefully. *Which* `getMeasure` method? The `BankAccount` and `Coin` classes provide two *different* implementations of that method. How did the correct method get called if the caller didn't even know the exact class to which `x` belongs?

The Java virtual machine makes a special effort to locate the correct method that belongs to the class of the actual object. That is, if `x` refers to a `BankAccount` object, then the `BankAccount.getMeasure` method is called. If `x` refers to a `Coin` object, then the `Coin.getMeasure` method is called. This means that one method call

```
double m = x.getMeasure();
```

can call different methods depending on the momentary contents of `x`.

> Polymorphism denotes the principle that behavior can vary depending on the actual type of an object.

The principle that the actual type of the object determines the method to be called is called *polymorphism*. The term "polymorphism" comes from the Greek words for "many shapes". The same computation works for objects of many shapes, and adapts itself to the nature of the objects. In Java, all instance methods are polymorphic.

When you see a polymorphic method call, such as `x.getMeasure()`, there are several possible `getMeasure` methods that can be called. You have already seen another case in which the same method name can refer to different methods, namely when a method name is *overloaded*: that is, when a single class has several methods with the same name but different parameter types. For example, you can have two constructors `BankAccount()` and `BankAccount(double)`. The compiler selects the appropriate method when compiling the program, simply by looking at the types of the parameters:

```
account = new BankAccount();
   // Compiler selects BankAccount()
account = new BankAccount(10000);
   // Compiler selects BankAccount(double)
```

397

398

There is an important difference between polymorphism and overloading. The compiler picks an overloaded method when translating the program, before the program ever runs. This method selection is called *early binding*. However, when selecting the appropriate `getMeasure` method in a call `x.getMeasure()`, the compiler does not make any decision when translating the method. The program has to run before anyone can know what is stored in `x`. Therefore, the virtual machine, and not the compiler, selects the appropriate method. This method selection is called *late binding*.

Early binding of methods occurs if the compiler selects a method from several possible candidates. Late binding occurs if the method selection takes place when the program runs.

SELF CHECK

5. Why is it impossible to construct a `Measurable` object?
6. Why can you nevertheless declare a variable whose type is `Measurable`?
7. What do overloading and polymorphism have in common? Where do they differ?

9.4 Using Interfaces for Callbacks

In this section, we discuss how the `DataSet` class can be made even more reusable by supplying a different interface type. This type of interface provides a "callback" mechanism, allowing the `DataSet` class to call back a specific method when it needs more information.

To understand why a further improvement to the `DataSet` class is desirable, consider these limitations of the `Measurable` interface:

- You can add the `Measurable` interface only to classes under your control. If you want to process a set of `Rectangle` objects, you cannot make the `Rectangle` class implement another interface—it is a system class, which you cannot change.
- You can measure an object in only one way. If you want to analyze a set of savings accounts both by bank balance and by interest rate, you are stuck.

Therefore, let us rethink the `DataSet` class. The data set needs to measure the objects that are added. When the objects are required to be of type `Measurable`, the responsibility of measuring lies with the added objects themselves, which is the cause of the limitations that we noted. It would be better if another object could carry out the measurement. Let's move the measurement method into a different interface:

```
public interface Measurer
{
   double measure(Object anObject);
}
```

The `measure` method measures an object and returns its measurement. Here we use the fact that all objects can be converted to the type `Object`, the "lowest common denominator" of all classes in Java. We will discuss the `Object` type in greater detail in Chapter 10.

398 399

The improved `DataSet` class is constructed with a `Measurer` object (that is, an object of some class that implements the `Measurer` interface). That object is saved in a `measurer` instance field and used to carry out the measurements, like this:

```
public void add(Object x)
{
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) <
measurer.measure(x))
      maximum = x;
   count++;
}
```

The `DataSet` class simply makes a callback to the `measure` method whenever it needs to measure any object.

Now you can define measurers to take on any kind of measurement. For example, here is how you can measure rectangles by area. Define a class

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() *
aRectangle.getHeight();
      return area;
   }
}
```

Note that the `measure` method must accept a parameter of type `Object`, even though this particular measurer just wants to measure rectangles. The method

signature must match the signature of the `measure` method in the `Measurer` interface. Therefore, the `Object` parameter is cast to the `Rectangle` type:

```
Rectangle aRectangle = (Rectangle) anObject;
```

What can you do with a `RectangleMeasurer`? You need it for a `DataSet` that compares rectangles by area. Construct an object of the `RectangleMeasurer` class and pass it to the `DataSet` constructor.

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
```

Next, add rectangles to the data set.

```
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
. . .
```

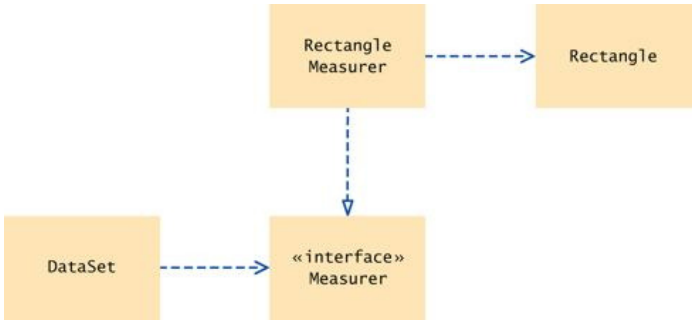
The data set will ask the `RectangleMeasurer` object to measure the rectangles. In other words, the data set uses the `RectangleMeasurer` object to carry out callbacks.

Figure 2 shows the UML diagram of the classes and interfaces of this solution. As in Figure 1, the `DataSet` class is decoupled from the `Rectangle` class whose objects it processes. However, unlike in Figure 1, the `Rectangle` class is no longer coupled with another class. Instead, to process rectangles, you have to come up with a small "helper" class `RectangleMeasurer`. This helper class has only one purpose: to tell the `DataSet` how to measure its objects.

399

400

Figure 2



UML Diagram of the `DataSet` Class and the `Measurer` Interface

ch09/measure2/DataSet.java

```
1  /**
2     Computes the average of a set of data values.
3  */
4  public class DataSet
5  {
6     /**
7        Constructs an empty data set with a given measurer.
8        @param aMeasurer the measurer that is used to measure
   data values
9     */
10    public DataSet(Measurer aMeasurer)
11    {
12       sum = 0;
13       count = 0;
14       maximum = null;
15       measurer = aMeasurer;
16    }
17
18    /**
19       Adds a data value to the data set.
20       @param xa data value
21    */
22    public void add(Object x)
23    {
24       sum = sum + measurer.measure(x);
25       if (count == 0
26             | | measurer.measure(maximum) <
   measurer.measure(x))
27          maximum = x;
28       count++;
29    }
30
31    /**
32       Gets the average of the added data.
33       @return the average or 0 if no data has been added
34    */
35    public double getAverage()
36    {
37       if (count == 0) return 0;
38       else return sum / count;
```

400

401

```
39     }
40
41     /**
42        Gets the largest of the added data.
43        @return the maximum or 0 if no data has been added
44     */
45     public Object getMaximum()
46     {
47        return maximum;
48     }
49
50     private double sum;
51     private Object maximum;
52     private int count;
53     private Measurer measurer;
54  }
```

ch09/measure2/DataSetTester2.java

```
 1  import java.awt.Rectangle;
 2
 3  /**
 4     This program demonstrates the use of a Measurer.
 5  */
 6  public class DataSetTester2
 7  {
 8     public static void main(String[] args)
 9     {
47       Measurer m = new RectangleMeasurer();
11
12       DataSet data = new DataSet(m);
13
14       data.add(new Rectangle(5, 10, 20, 30));
15       data.add(new Rectangle(10, 20, 30, 40));
16       data.add(new Rectangle(20, 30, 5, 15));
17
18       System.out.println("Average area: " +
data.getAverage());
19       System.out.println("Expected: 625");
20
21       Rectangle max = (Rectangle)
data.getMaximum();
22       System.out.println("Maximum area
rectangle: " + max);
```

```
23         System.out.println("Expected:
java.awt.Rectangle[
24               x=10,y=20,width=30,height=40]");
25      }
26   }
```

401

402

ch09/measure2/Measurer.java

```
1   /**
2      Describes any class whose objects can measure other objects.
3   */
4   public interface Measurer
5   {
6      /**
7         Computes the measure of an object.
8         @param anObject the object to be measured
9         @return the measure
10     */
11     double measure(Object anObject);
12  }
```

ch09/measure2/RectangleMeasurer.java

```
1   import java.awt.Rectangle;
2
3   /**
4      Objects of this class measure rectangles by area.
5   */
6   public class RectangleMeasurer implements
Measurer
7   {
8      public double measure(Object anObject)
9      {
10        Rectangle aRectangle = (Rectangle)
anObject;
11        double area = aRectangle.getWidth() *
aRectangle.getHeight();
12        return area;
13     }
14  }
```

Output

```
Average area: 625
Expected: 625
Maximum area rectangle:
java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected:
java.awt.Rectangle[x=10,y=20,width=30,height=40]
```

SELF CHECK

8. Suppose you want to use the `DataSet` class of Section 9.1 to find the longest `String` from a set of inputs. Why can't this work?
9. How can you use the `DataSet` class of this section to find the longest `String` from a set of inputs?
10. Why does the `measure` method of the `Measurer` interface have one more parameter than the `getMeasure` method of the `Measurable` interface?

402

403

9.5 Inner Classes

The `RectangleMeasurer` class is a very trivial class. We need this class only because the `DataSet` class needs an object of some class that implements the `Measurer` interface. When you have a class that serves a very limited purpose, such as this one, you can declare the class inside the method that needs it:

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      . . .
   }
}
```

Such a class is called an *inner class*. An inner class is any class that is defined inside another class. This arrangement signals to the reader of your program that the `RectangleMeasurer` class is not interesting beyond the scope of this method. Since an inner class inside a method is not a publicly accessible feature, you don't need to document it as thoroughly.

An inner class is declared inside another class. Inner classes are commonly used for tactical classes that should not be visible elsewhere in a program.

You can also define an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class.

SYNTAX 9.3: Inner Classes

Declared inside a method:

```
class OuterClassName
{
   method signature
   {
      . . .
      class InnerClassName
      {
         methods
         fields
      }
      . . .
   }
   . . .
}
```

Declared inside the class:

```
class OuterClassName
{
   methods
   fields
   accessSpecifier class
InnerClassName
   {
      methods
      fields
   }
   . . .
}
```

403

404

Example

```
public class Tester
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      . . .
   }
}
```

Purpose

To define an inner class whose scope is restricted to a single method or the methods of a single class

When you compile the source files for a program that uses inner classes, have a look at the class files in your program directory—you will find that the inner classes are stored in files with curious names, such as `DataSetTester3$1$RectangleMeasurer.class`. The exact names aren't important. The point is that the compiler turns an inner class into a regular class file.

ch09/measure3/DataSetTester3.java

```
1  import java.awt.Rectangle;
2
3  /**
4     This program demonstrates the use of an inner class.
5  */
6  public class DataSetTester3
7  {
8     public static void main(String[] args)
9     {
10       class RectangleMeasurer implements
Measurer
11       {
12          public double measure(Object anObject)
13          {
14             Rectangle aRectangle = (Rectangle)
anObject;
15             double area
16                   = aRectangle.getWidth() *
aRectangle.getHeight();
17             return area;
18          }
19       }
20
21       Measurer m = new RectangleMeasurer();
22
23       DataSet data = new DataSet(m);
24
25       data.add(new Rectangle(5, 10, 20, 30));
26       data.add(new Rectangle(10, 20, 30, 40));
```

```
27         data.add(new Rectangle(20, 30, 5, 15));
28
29         System.out.println("Average area: " +
data.getAverage());
30         System.out.println("Expected: 625");
31
32         Rectangle max = (Rectangle)
data.getMaximum();
33         System.out.println("Maximum area
rectangle: " + max);
34         System.out.println("Expected:
java.awt.Rectangle[
35               x=10,y=20,width=30,height=40]");
36      }
37 }
```

SELF CHECK

11. Why would you use an inner class instead of a regular class?

12. How many class files are produced when you compile the `DataSetTester3` program?

ADVANCED TOPIC 9.2: Anonymous Classes

An entity is *anonymous* if it does not have a name. In a program, something that is only used once doesn't usually need a name. For example, you can replace

```
Coin aCoin = new Coin(0.1, "dime");
data.add(aCoin);
```

with

```
data.add(new Coin(0.1, "dime"));
```

if the coin is not used elsewhere in the same method. The object `new Coin(0.1, "dime")` is an *anonymous object*. Programmers like anonymous objects, because they don't have to go through the trouble of coming up with a name. If you have struggled with the decision whether to call a coin `c`, `dime`, or `aCoin`, you'll understand this sentiment.

Inner classes often give rise to a similar situation. After a single object of the `Rectangle-Measurer` has been constructed, the class is never used again. In Java, it is possible to define *anonymous classes* if all you ever need is a single object of the class.

```
public static void main(String[] args)
{
   // Construct an object of an anonymous class
   Measurer m = new Measurer()
      // Class definition starts here
      {
         public double measure(Object anObject)
         {
            Rectangle aRectangle = (Rectangle)
anObject;
            double area = aRectangle.getWidth() *
aRectangle.getHeight();
            return area;
         }
      };
   DataSet data = new DataSet(m);
   . . .
}
```

405

406

This means: Construct an object of a class that implements the `Measurer` interface by defining the `measure` method as specified. Many programmers like this style, but we will not use it in this book.

RANDOM FACT 9.1: Operating Systems

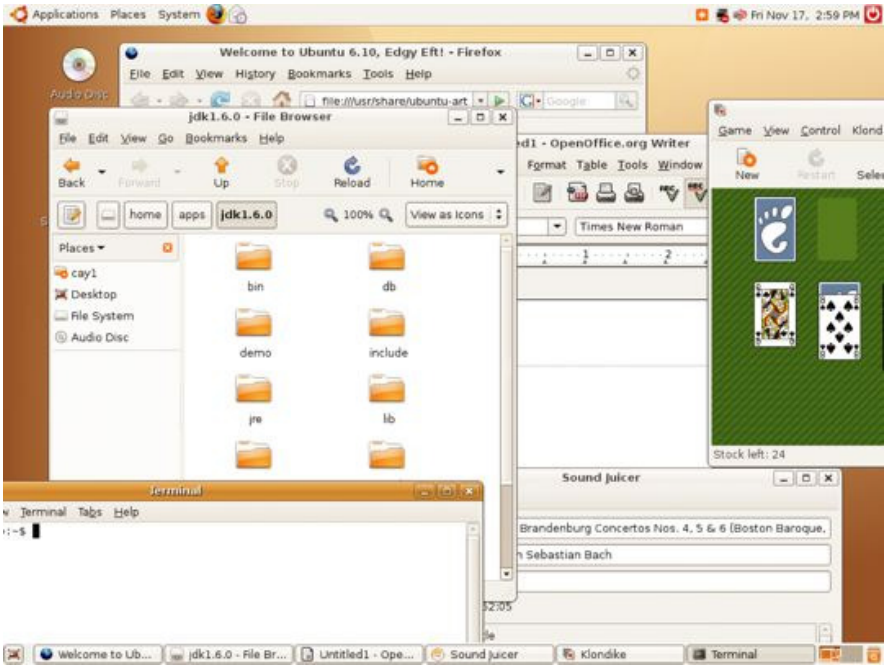
Without an operating system, a computer would not be useful. Minimally, you need an operating system to locate files and to start programs. The programs that you run need services from the operating system to access devices and to interact with other programs. Operating systems on large computers need to provide more services than those on personal computers do.

Here are some typical services:

- *Program loading*. Every operating system provides some way of launching application programs. The user indicates what program should be run,

usually by typing the name of the program or by clicking on an icon. The operating system locates the program code, loads it into memory, and starts it.

- *Managing files*. A storage device, such as a hard disk is, electronically, simply a device capable of storing a huge sequence of zeroes and ones. It is up to the operating system to bring some structure to the storage layout and organize it into files, folders, and so on. The operating system also needs to impose some amount of security and redundancy into the file system so that a power outage does not jeopardize the contents of an entire hard disk. Some operating systems do a better job in this regard than others.
- *Virtual memory*. RAM is expensive, and few computers have enough RAM to hold all programs and their data that a user would like to run simultaneously. Most operating systems extend the available memory by storing some data on the hard disk. The application programs do not realize whether a particular data item is in memory or in the virtual memory disk storage. When a program accesses a data item that is currently not in RAM, the processor senses this and notifies the operating system. The operating system swaps the needed data from the hard disk into RAM, simultaneously swapping out a memory block of equal size that had not been accessed for some time.
- *Handling multiple users*. The operating systems of large and powerful computers allow simultaneous access by multiple users. Each user is connected to the computer through a separate terminal. The operating system authenticates users by checking that each one has a valid account and password. It gives each user a small slice of processor time, then serves the next user.
- *Multitasking*. Even if you are the sole user of a computer, you may want to run multiple applications—for example, to read your e-mail in one window and run the Java compiler in another. The operating system is responsible for dividing processor time between the applications you are running, so that each can make progress.



A Graphical Software Environment for the Linux Operating System

- *Printing*. The operating system queues up the print requests that are sent by multiple applications. This is necessary to make sure that the printed pages do not contain a mixture of words sent simultaneously from separate programs.
- *Windows*. Many operating systems present their users with a desktop made up of multiple windows. The operating system manages the location and appearance of the window frames; the applications are responsible for the interiors.
- *Fonts*. To render text on the screen and the printer, the shapes of characters must be defined. This is especially important for programs that can display multiple type styles and sizes. Modern operating systems contain a central font repository.
- *Communicating between programs*. The operating system can facilitate the transfer of information between programs. That transfer can happen through

cut and paste or *interprocess communication*. Cut and paste is a user-initiated data transfer in which the user copies data from one application into a transfer buffer (often called a "clipboard") managed by the operating system and inserts the buffer's contents into another application. Interprocess communication is initiated by applications that transfer data without direct user involvement.

- *Networking*. The operating system provides protocols and services for enabling applications to reach information on other computers attached to the network.

Today, the most popular operating systems for personal computers are Linux (see figure), the Macintosh OS, and Microsoft Windows.

407

408

9.6 Events, Event Sources, and Event Listeners

In the applications that you have written so far, user input was under control of the *program*. The program asked the user for input in a specific order. For example, a program might ask the user to supply first a name, then a dollar amount. But the programs that you use every day on your computer don't work like that. In a program with a modern graphical user interface, the *user* is in control. The user can use both the mouse and the keyboard and can manipulate many parts of the user interface in any desired order. For example, the user can enter information into text fields, pull down menus, click buttons, and drag scroll bars in any order. The program must react to the user commands, in whatever order they arrive. Having to deal with many possible inputs in random order is quite a bit harder than simply forcing the user to supply input in a fixed order.

In the following sections, you will learn how to write Java programs that can react to user interface events, such as button pushes and mouse clicks. The Java windowing toolkit has a very sophisticated mechanism that allows a program to specify the events in which it is interested and which objects to notify when one of these events occurs.

User interface events include key presses, mouse moves, button clicks, menu selections, and so on.

Whenever the user of a graphical program types characters or uses the mouse anywhere inside one of the windows of the program, the Java window manager sends a notification to the program that an *event* has occurred. The window manager generates huge numbers of events. For example, whenever the mouse moves a tiny interval over a window, a “mouse move” event is generated. Events are also generated when the user presses a key, clicks a button, or selects a menu item.

Most programs don't want to be flooded by boring events. For example, when a button is clicked with the mouse, the mouse moves over the button, then the mouse button is pressed, and finally the button is released. Rather than receiving lots of irrelevant mouse events, a program can indicate that it only cares about button clicks, not about the underlying mouse events. However, if the mouse input is used for drawing shapes on a virtual canvas, it is necessary to closely track mouse events.

An event listener belongs to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs.

Every program must indicate which events it needs to receive. It does that by installing *event listener* objects. An event listener object belongs to a class that you define. The methods of your event listener classes contain the instructions that you want to have executed when the events occur.

To install a listener, you need to know the *event source*. The event source is the user interface component that generates a particular event. You add an event listener object to the appropriate event sources. Whenever the event occurs, the event source calls the appropriate methods of all attached event listeners.

Event sources report on events. When an event occurs, the event source notifies all event listeners.

408

409

Use `JButton` components for buttons. Attach an `ActionListener` to each button.

This sounds somewhat abstract, so let's run through an extremely simple program that prints a message whenever a button is clicked. Button listeners must belong to a class that implements the `ActionListener` interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

This particular interface has a single method, `actionPerformed`. It is your job to supply a class whose `actionPerformed` method contains the instructions that you want executed whenever the button is clicked. Here is a very simple example of such a listener class:

ch09/button1/ClickListener.java

```
 1  import java.awt.event.ActionEvent;
 2  import java.awt.event.ActionListener;
 3
 4  /**
 5     An action listener that prints a message.
 6  */
 7  public class ClickListener implements
ActionListener
 8  {
 9     public void actionPerformed(ActionEvent
event)
10     {
11        System.out.println("I was clicked.");
12     }
13 }
```

We ignore the `event` parameter of the `actionPerformed` method—it contains additional details about the event, such as the time at which it occurred.

Once the listener class has been defined, we need to construct an object of the class and add it to the button:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Whenever the button is clicked, it calls

```
listener.actionPerformed(event);
```

As a result, the message is printed.

You can think of the `actionPerformed` method as another example of a callback, similar to the `measure` method of the `Measurer` class. The windowing toolkit calls the `actionPerformed` method whenever the button is pressed, whereas the `DataSet` calls the `measure` method whenever it needs to measure an object.

You can test this program out by opening a console window, starting the `ButtonViewer` program from that console window, clicking the button, and watching the messages in the console window (see Figure 3).

409

410

Figure 3

Implementing an Action Listener

ch09/button1/ButtonViewer.java

```
1  import java.awt.event.ActionListener;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4
5  /**
6     This program demonstrates how to install an action listener.
7  */
8  public class ButtonViewer
9  {
10   public static void main(String[] args)
```

```
11     {
12        JFrame frame = new JFrame();
13        JButton button = new JButton("Click me!");
14        frame.add(button);
15
16        ActionListener listener = new
ClickListener();
17        button.addActionListener(listener);
18
19        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
20        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
21        frame.setVisible(true);
22     }
23
24     private static final int FRAME_WIDTH = 100;
25     private static final int FRAME_HEIGHT = 60;
26  }
```

Self Check

13. Which objects are the event source and the event listener in the `ButtonViewer` program?

14. Why is it legal to assign a `ClickListener` object to a variable of type `ActionListener`?

410

411

Common Error 9.3: Modifying the Signature in the Implementing Method

When you implement an interface, you must define each method *exactly* as it is specified in the interface. Accidentally making small changes to the parameter or return types is a Common Error. Here is the classic example,

```
class MyListener implements ActionListener
{
   public void actionPerformed()
   // Oops . . . forgot ActionEvent parameter
   {
      . . .
   }
}
```

As far as the compiler is concerned, this class has two methods:

```
public void actionPerformed(ActionEvent event)
public void actionPerformed()
```

The first method is undefined. The compiler will complain that the method is missing. You have to read the error message carefully and pay attention to the parameter and return types to find your error.

9.7 Using Inner Classes for Listeners

In the preceding section, you saw how the code that is executed when a button is clicked is placed into a listener class. It is common to implement listener classes as inner classes like this:

```
JButton button = new JButton(". . .");

// This inner class is declared in the same method as the button variable
class MyListener implements ActionListener
{
   . . .
};

ActionListener listener = new MyListener();
button.addActionListener(listener);
```

There are two reasons for this arrangement. First, it places the trivial listener class exactly where it is needed, without cluttering up the remainder of the project. Moreover, inner classes have a very attractive feature: Their methods can access variables that are defined in surrounding blocks. In this regard, method definitions of inner classes behave similarly to nested blocks.

411

412

Recall that a *block* is a statement group enclosed by braces. If a block is nested inside another, the inner block has access to all variables from the surrounding block:

```
{   // Surrounding block
   BankAccount account = new BankAccount();
   if (. . .)
   {   // Inner block
      . . .
      // OK to access variable from surrounding block
```

```
            account.deposit(interest);
            . . .
         }  // End of inner block
         . . .
      }  // End of surrounding block
```

The same nesting works for inner classes. Except for some technical restrictions, which we will examine later in this section, the methods of an inner class can access the variables from the enclosing scope. This feature is very useful when implementing event handlers. It allows the inner class to access variables without having to pass them as constructor or method parameters.

> Methods of an inner class can access local variables from surrounding blocks and fields from surrounding classes.

Let's look at an example. Suppose we want to add interest to a bank account whenever a button is clicked.

```
JButton button = new JButton("Add Interest");
final BankAccount account = new
BankAccount(INITIAL_BALANCE);

// This inner class is declared in the same method as the account and button variables.
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // The listener method accesses the account variable
      // from the surrounding block
      double interest = account.getBalance()
            * INTEREST_RATE / 100;
      account.deposit(interest);
   }
};

ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

There is a technical wrinkle. An inner class can access surrounding *local* variables only if they are declared as `final`. That sounds like a restriction, but it is usually not

an issue in practice. Keep in mind that an object variable is `final` when the variable always refers to the same object. The state of the object can change, but the variable can't refer to a different object. For example, in our program, we never intended to have the `account` variable refer to multiple bank accounts, so there was no harm in declaring it as `final`.

Local variables that are accessed by an inner-class method must be declared as `final`.

412

413

An inner class can also access *fields* of the surrounding class, again with a restriction. The field must belong to the object that constructed the inner class object. If the inner class object was created inside a static method, it can only access static surrounding fields.

Here is the source code for the program.

ch09/button2/InvestmentViewer1.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5
6  /**
7     This program demonstrates how an action listener can access
8     a variable from a surrounding block.
9  */
10 public class InvestmentViewer1
11 {
12    public static void main(String[] args)
13    {
14       JFrame frame = new JFrame();
15
16       // The button to trigger the calculation
17       JButton button = new JButton("Add
Interest");
18       frame.add(button);
19
20       // The application adds interest to this bank account
```

```
21         final BankAccount account = new
BankAccount(INITIAL_BALANCE);
22
23         class AddInterestListener implements
ActionListener
24         {
25            public void
actionPerformed(ActionEvent event)
26            {
27               // The listener method accesses the account variable
28               // from the surrounding block
29               double interest =
account.getBalance()
30                     * INTEREST_RATE / 100;
31               account.deposit(interest);
32               System.out.println("balance: " +
account.getBalance());
33            }
34         }
35
36         ActionListener listener = new
AddInterestListener();
37         button.addActionListener(listener);
38
39         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
40         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
41         frame.setVisible(true);
42      }
43
44      private static final double INTEREST_RATE =
10;
45      private static final double INITIAL_BALANCE
= 1000;                                                    413
46                                                         414
47      private static final int FRAME_WIDTH = 120;
48      private static final int FRAME_HEIGHT = 60;
49 }
```

Output

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```

SELF CHECK

15. Why would an inner class method want to access a variable from a surrounding scope?

16. If an inner class accesses a local variable from a surrounding scope, what special rule applies?

9.8 Building Applications with Buttons

In this section, you will learn how to structure a graphical application that contains buttons. We will put a button to work in our simple investment viewer program. Whenever the button is clicked, interest is added to a bank account, and the new balance is displayed (see Figure 4).

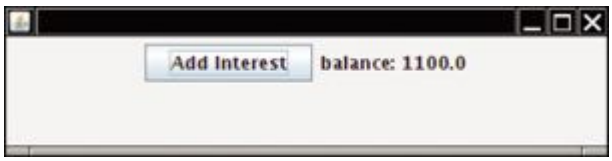
First, we construct an object of the `JButton` class. Pass the button label to the constructor:

```
JButton button = new JButton("Add Interest");
```

We also need a user interface component that displays a message, namely the current bank balance. Such a component is called a *label*. You pass the initial message string to the `JLabel` constructor, like this:

```
JLabel label = new JLabel("balance: " +
account.getBalance());
```

Figure 4



An Application with a Button

The frame of our application contains both the button and the label. However, we cannot simply add both components directly to the frame—they would be placed on

top of each other. The solution is to put them into a *panel*, a container for other user-interface components, and then add the panel to the frame:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Use a `JPanel` container to group multiple user-interface components together.

Now we are ready for the hard part—the event listener that handles button clicks. As in the preceding section, it is necessary to define a class that implements the `ActionListener` interface, and to place the button action into the `actionPerformed` method. Our listener class adds interest and displays the new balance:

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double interest = account.getBalance() *
INTEREST_RATE / 100;
      account.deposit(interest);
      label.setText("balance: " +
account.getBalance());
   }
}
```

There is just a minor technicality. The `actionPerformed` method manipulates the `account` and `label` variables. These are local variables of the `main` method of the investment viewer program, not instance fields of the `AddInterestListener` class. We therefore need to declare the `account` and `label` variables as `final` so that the `actionPerformed` method can access them.

You often install event listeners as inner classes so that they can have access to the surrounding fields, methods, and final variables.

Let's put the pieces together.

```
public static void main(String[] args)
{
```

```
      . . .
      JButton button = new JButton("Add Interest");
      final BankAccount account = new
   BankAccount(INITIAL_BALANCE);
      final JLabel label = new JLabel("balance: " +
   account.getBalance());
      class AddInterestListener implements
   ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            double interest = account.getBalance()
                  * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance: " +
   account.getBalance());
         }
      }
      ActionListener listener = new
   AddInterestListener();
      button.addActionListener(listener);                    415
      . . .                                                  416
   }
```

With a bit of practice, you will learn to glance at this code and translate it into plain English: "When the button is clicked, add interest and set the label text."

Here is the complete program. It demonstrates how to add multiple components to a frame, by using a panel, and how to implement listeners as inner classes.

ch09/button3/InvestmentViewer2.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7  import javax.swing.JTextField;
8
9  /**
10    This program displays the growth of an investment.
11 */
12 public class InvestmentViewer2
```

```
13  {
14     public static void main(String[] args)
15     {
16        JFrame frame = new JFrame();
17
18        // The button to trigger the calculation
19        JButton button = new JButton("Add
Interest");
20
21        // The application adds interest to this bank account
22        final BankAccount account = new
BankAccount(INITIAL_BALANCE);
23
24        // The label for displaying the results
25        final JLabel label = new JLabel(
26              "balance: " +
account.getBalance());
27
28        // The panel that holds the user interface components
29        JPanel panel = new JPanel();
30        panel.add(button);
31        panel.add(label);
32        frame.add(panel);
33
34        class AddInterestListener implements
ActionListener
35        {
36           public void
actionPerformed(ActionEvent event)
37           {
38              double interest =
account.getBalance()
39                    * INTEREST_RATE / 100;
40              account.deposit(interest);
41              label.setText(
42                    "balance: " +
account.getBalance());
43           }
44        }
45
46       ActionListener listener = new
AddInterestListener();
47       button.addActionListener(listener);
48
```

416

417

```
49         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
50         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
51         frame.setVisible(true);
52      }
53
54      private static final double INTEREST_RATE =
        10;
55      private static final double INITIAL_BALANCE
        = 1000;
56
57      private static final int FRAME_WIDTH = 400;
58      private static final int FRAME_HEIGHT = 100;
59   }
```

SELF CHECK

17. How do you place the "`balance: . . .`" message to the left of the `"Add Interest"` button?

18. Why was it not necessary to declare the `button` variable as `final`?

COMMON ERROR 9.4: Forgetting to Attach a Listener

If you run your program and find that your buttons seem to be dead, double-check that you attached the button listener. The same holds for other user interface components. It is a surprisingly COMMON ERROR to program the listener class and the event handler action without actually attaching the listener to the event source.

PRODUCTIVITY HINT 9.1: Don't Use a Container as a Listener

In this book, we use inner classes for event listeners. That approach works for many different event types. Once you master the technique, you don't have to think about it anymore. Many development environments automatically generate code with inner classes, so it is a good idea to be familiar with them.

However, some programmers bypass the event listener classes and instead turn a container (such as a panel or frame) into a listener. Here is a typical example. The

`actionPerformed` method is added to the viewer class. That is, the viewer implements the `ActionListener` interface.

```
public class InvestmentViewer
      implements ActionListener// This approach is not recommended
{                                                                    417
public InvestmentViewer()                                            418
{
   JButton button = new JButton("Add Interest");
   button.addActionListener(this);
   . . .
}
public void actionPerformed(ActionEvent event)
{
}
. . .
}
```

Now the `actionPerformed` method is a part of the `InvestmentViewer` class rather than part of a separate listener class. The listener is installed as `this`.

This technique has two major flaws. First, it separates the button definition from the button action. Also, it doesn't *scale* well. If the viewer class contains two buttons that each generate action events, then the `actionPerformed` method must investigate the event source, which leads to code that is tedious and error-prone.

9.9 Processing Timer Events

In this section we will study timer events and show how they allow you to implement simple animations.

The `Timer` class in the `javax.swing` package generates a sequence of action events, spaced apart at even time intervals. (You can think of a timer as an invisible button that is automatically clicked.) This is useful whenever you want to have an object updated in regular intervals. For example, in an animation, you may want to update a scene ten times per second and redisplay the image, to give the illusion of movement.

A timer generates timer events at fixed intervals.

When you use a timer, you specify the frequency of the events and an object of a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Finally, start the timer.

```
class MyListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // This action will be executed at each timer event
      Place listener action here
   }
}
MyListener listener = new MyListener();
Timer t = new Timer (interval, listener);
t.start();
```

418

419

Then the timer calls the `actionPerformed` method of the `listener` object every `interval` milliseconds.

Our sample program will display a moving rectangle. We first supply a `RectangleComponent` class with a `moveBy` method that moves the rectangle by a given amount.

ch09/timer/RectangleComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import java.awt.Rectangle;
4  import javax.swing.JComponent;
5
6  /**
7     This component displays a rectangle that can be moved.
8  */
9  public class RectangleComponent extends
JComponent
10 {
11    public RectangleComponent()
12    {
13       // The rectangle that the paint method draws
```

```
14         box = new Rectangle(BOX_X, BOX_Y,
15               BOX_WIDTH, BOX_HEIGHT);
16      }
17
18      public void paintComponent(Graphics g)
19      {
20         super.paintComponent(g);
21         Graphics2D g2 = (Graphics2D) g;
22
23         g2.draw(box);
24      }
25
26      /**
27         Moves the rectangle by a given amount.
28         @param x the amount to move in the x-direction
29         @param y the amount to move in the y-direction
30      */
31      public void moveBy(int dx, int dy)
32      {
33         box.translate(dx, dy);
34         repaint();
35      }
36
37      private Rectangle box;
38
39      private static final int BOX_X = 100;
40      private static final int BOX_Y = 100;
41      private static final int BOX_WIDTH = 20;
42      private static final int BOX_HEIGHT = 30;
43   }
```

419

420

Note the call to `repaint` in the `moveBy` method. This call is necessary to ensure that the component is repainted after the state of the rectangle object has been changed. Keep in mind that the component object does not contain the pixels that show the drawing. The component merely contains a `Rectangle` object, which itself contains four coordinate values. Calling `translate` updates the rectangle coordinate values. The call to `repaint` forces a call to the `paintComponent` method. The `paintComponent` method redraws the component, causing the rectangle to appear at the updated location.

The `repaint` method causes a component to repaint itself. Call this method whenever you modify the shapes that the `paintComponent` method draws.

The `actionPerformed` method of the timer listener simply calls `component.moveBy(1, 1)`. This moves the rectangle one pixel down and to the right. Since the `actionPerformed` method is called many times per second, the rectangle appears to move smoothly across the frame.

ch09/timer/RectangleMover.java

```
 1  import java.awt.event.ActionEvent;
 2  import java.awt.event.ActionListener;
 3  import javax.swing.JFrame;
 4  import javax.swing.Timer;
 5
 6  /**
 7     This program moves the rectangle.
 8  */
 9  public class RectangleMover
10 {
11    public static void main(String[] args)
12    {
13       JFrame frame = new JFrame();
14
15       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
16       frame.setTitle("An animated rectangle");
17       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
18
19       final RectangleComponent component = new
RectangleComponent();
20       frame.add(component);
21
22       frame.setVisible(true);
23
24       class TimerListener implements
ActionListener
25       {
26          public void
actionPerformed(ActionEvent event)
27          {
28             component.moveBy(1, 1);
```

```
29            }
30         }
31
32         ActionListener listener = new
TimerListener();
33
34         final int DELAY = 100;// Milliseconds between timer
ticks
35         Timer t = new Timer(DELAY, listener);
36         t.start();
37      }
38
39      private static final int FRAME_WIDTH = 300;
40      private static final int FRAME_HEIGHT = 400;
41   }
```

420

421

Self Check

19. Why does a timer require a listener object?

20. What would happen if you omitted the call to `repaint` in the `moveBy` method?

Common Error 9.5: Forgetting to Repaint

You have to be careful when your event handlers change the data in a painted component. When you make a change to the data, the component is not automatically painted with the new data. You must tell the Swing framework that the component needs to be repainted, by calling the `repaint` method either in the event handler or in the component's mutator methods. Your component's `paintComponent` method will then be invoked at an opportune moment, with an appropriate `Graphics` object. Note that you should not call the `paintComponent` method directly.

This is a concern only for your own painted components. When you make a change to a standard Swing component such as a `JLabel`, the component is automatically repainted.

9.10 Mouse Events

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to process mouse events. Mouse events are more complex than button clicks or timer ticks.

You use a mouse listener to capture mouse events.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
        // Called when a mouse button has been pressed on a component
   void mouseReleased(MouseEvent event);
        // Called when a mouse button has been released on a component
   void mouseClicked(MouseEvent event);
        // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
        // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
        // Called when the mouse exits a component
}
```

421

422

The `mousePressed` and `mouseReleased` methods are called whenever a mouse button is pressed or released. If a button is pressed and released in quick succession, and the mouse has not moved, then the `mouseClicked` method is called as well. The `mouseEntered` and `mouseExited` methods can be used to paint a user-interface component in a special way whenever the mouse is pointing inside it.

The most commonly used method is `mousePressed`. Users generally expect that their actions are processed as soon as the mouse button is pressed.

You add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
```

```
      // Implements five methods
}

MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

In our sample program, a user clicks on a component containing a rectangle. Whenever the mouse button is pressed, the rectangle is moved to the mouse location. We first enhance the `RectangleComponent` class and add a `moveTo` method to move the rectangle to a new position.

ch09/mouse/RectangleComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import java.awt.Rectangle;
4  import javax.swing.JComponent;
5
6  /**
7     This component displays a rectangle that can be moved.
8  */
9  public class RectangleComponent extends
JComponent
10 {
11    public RectangleComponent()
12    {
13       // The rectangle that the paint method draws
14       box = new Rectangle(BOX_X, BOX_Y,
15             BOX_WIDTH, BOX_HEIGHT);
16    }
17
18    public void paintComponent(Graphics g)
19    {
20       super.paintComponent(g);
21       Graphics2D g2 = (Graphics2D) g;
22
23       g2.draw(box);
24    }
25
26    /**
27       Moves the rectangle to the given location.
28       @param xthe x-position of the new location
29       @param ythe y-position of the new location
```

422

423

```
30     */
31     public void moveTo(int x, int y)
32     {
33        box.setLocation(x, y);
34        repaint();
35     }
36
37     private Rectangle box;
38
39     private static final int BOX_X = 100;
40     private static final int BOX_Y = 100;
41     private static final int BOX_WIDTH = 20;
42     private static final int BOX_HEIGHT = 30;
43  }
```

Note the call to `repaint` in the `moveTo` method. As explained in the preceding section, this call causes the component to repaint itself and show the rectangle in the new position.

Now, add a mouse listener to the component. Whenever the mouse is pressed, the listener moves the rectangle to the mouse location.

```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
   }
   // Do-nothing methods
   public void mouseReleased(MouseEvent event) {}
   public void mouseClicked(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

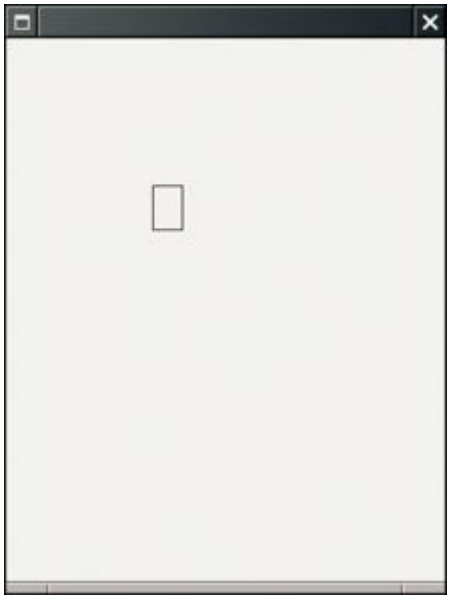
It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all five methods of the interface must be implemented. The unused methods are simply implemented as do-nothing methods.

Go ahead and run the `RectangleComponentViewer` program. Whenever you click the mouse inside the frame, the top left corner of the rectangle moves to the mouse pointer (see Figure 5).

423

424

Figure 5



Clicking the Mouse Moves the Rectangle

ch09/mouse/RectangleComponentViewer.java

```
1  import java.awt.event.MouseListener;
2  import java.awt.event.MouseEvent;
3  import javax.swing.JFrame;
4
5  /**
6     This program displays a RectangleComponent.
7  */
8  public class RectangleComponentViewer
9  {
10    public static void main(String[] args)
11    {
12       final RectangleComponent component = new
   RectangleComponent();
13
```

```
14          // Add mouse press listener
15
16          class MousePressListener implements
MouseListener
17          {
18             public void mousePressed(MouseEvent
event)
19             {
20                int x = event.getX();
21                int y = event.getY();
22                component.moveTo(x, y);
23             }
24
25             // Do-nothing methods
26             public void mouseReleased(MouseEvent
event) {}
27             public void mouseClicked(MouseEvent
event) {}
28             public void mouseEntered(MouseEvent
event) {}
29             public void mouseExited(MouseEvent
event) {}
30          }
31
32          MouseListener listener = new
MousePressListener();
33          component.addMouseListener(listener);
34
35          JFrame frame = new JFrame();
36          frame.add(component);
37
38          frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
39          frame.setDefaultCloseOperation(JFrame.EXIT_ON_
40          frame.setVisible(true);
41       }
42
43       private static final int FRAME_WIDTH = 300;
44       private static final int FRAME_HEIGHT = 400;
45    }
```

424

425

Self Check

21. Why was the `moveBy` method in the `RectangleComponent` replaced with a `moveTo` method?

22. Why must the `MousePressListener` class supply five methods?

ADVANCED TOPIC 9.3: Event Adapters

In the preceding section you saw how to install a mouse listener into a mouse event source and how the listener methods are called when an event occurs. Usually, a program is not interested in all listener notifications. For example, a program may only be interested in mouse clicks and may not care that these mouse clicks are composed of "mouse pressed" and "mouse released" events. Of course, the program could supply a listener that defines all those methods in which it has no interest as "do-nothing" methods, for example:

```
class MouseClickListener implements MouseListener
{
   public void mouseClicked(MouseEvent event)
   {
      // Mouse click action here
   }
   // Four do-nothing methods
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
}
```

This is boring. For that reason, some friendly soul has created a `MouseAdapter` class that implements the `MouseListener` interface such that all methods do nothing. You can *extend* that class, inheriting the do-nothing methods and overriding the methods that you care about, like this:

425
426

```
class MouseClickListener extends MouseAdapter
{
   public void mouseClicked(MouseEvent event)
   {
      // Mouse click action here
   }
}
```

See Chapter 10 for more information on the process of extending classes.

RANDOM FACT 9.2: Programming Languages

Many hundreds of programming languages exist today, which is actually quite surprising. The idea behind a high-level programming language is to provide a medium for programming that is independent from the instruction set of a particular processor, so that one can move programs from one computer to another without rewriting them. Moving a program from one programming language to another is a difficult process, however, and it is rarely done. Thus, it seems that there would be little use for so many programming languages.

Unlike human languages, programming languages are created with specific purposes. Some programming languages make it particularly easy to express tasks from a particular problem domain. Some languages specialize in database processing; others in "artificial intelligence" programs that try to infer new facts from a given base of knowledge; others in multimedia programming. The Pascal language was purposefully kept simple because it was designed as a teaching language. The C language was developed to be translated efficiently into fast machine code, with a minimum of housekeeping overhead. The C++ language builds on C by adding features for object-oriented programming. The Java language was designed for securely deploying programs across the Internet.

In the early 1970s the U.S. Department of Defense (DoD) was seriously concerned about the high cost of the software components of its weapons equipment. It was estimated that more than half of the total DoD budget was spent on the development of this *embedded-systems* software—that is, software that is embedded in some machinery, such as an airplane or missile, to control it. One of the perceived problems was the great diversity of programming languages that were used to produce that software. Many of these languages, such as TACPOL, CMS-2, SPL/1, and JOVIAL, were virtually unknown outside the defense sector.

In 1976 a committee of computer scientists and defense industry representatives was asked to evaluate existing programming languages. The committee was to determine whether any of them could be made the DoD standard for all future military programming. To nobody's surprise, the committee decided that a new language would need to be created. Contractors were then invited to submit designs for such a new language. Of 17 initial proposals, four were chosen to

develop their languages. To ensure an unbiased evaluation, the languages received code names: Red (by Intermetrics), Green (by CII Honeywell Bull), Blue (by Softech), and Yellow (by SRI International). All four languages were based on Pascal. The Green language emerged as the winner in 1979. It was named Ada in honor of the world's first programmer, Ada Lovelace (see Random Fact 14.1). 426

427

The Ada language was roundly derided by academics as a typical bloated Defense Department product. Military contractors routinely sought, and obtained, exemptions from the requirement that they had to use Ada on their projects. Outside the defense industry, few companies used Ada. Perhaps that is unfair. Ada had been *designed* to be complex enough to be useful for many applications, whereas other, more popular languages, notably C++, have *grown* to be just as complex and ended up being unmanageable.

The initial version of the C language was designed around 1972. Unlike Ada, C is a simple language that lets you program “close to the machine”. It is also quite unsafe. Because different compiler writers added different features, the language actually sprouted various dialects. Some programming instructions were understood by one compiler but rejected by another. Such divergence is an immense pain to a programmer who wants to move code from one computer to another, and an effort got underway to iron out the differences and come up with a standard version of C. The design process ended in 1989 with the completion of the ANSI (American National Standards Institute) Standard. In the meantime, Bjarne Stroustrup of AT&T added features of the language Simula (an object-oriented language designed for carrying out simulations) to C. The resulting language was called C++. From 1985 until today, C++ has grown by the addition of many features, and a standardization process was completed in 1998. C++ has been enormously popular because programmers can take their existing C code and move it to C++ with only minimal changes. In order to keep compatibility with existing code, every innovation in C++ had to work around the existing language constructs, yielding a language that is powerful but somewhat cumbersome to use.

In 1995, Java was designed to be conceptually simpler and more internally consistent than C++, while retaining the syntax that is familiar to millions of C and C++ programmers. The Java *language* was a great design success. It is indeed clean and simple. As for the Java *library*, you know from your own experience that it is neither.

Keep in mind that a programming language is only part of the technology for writing programs. To be successful, a programming language needs feature-rich libraries, powerful tools, and a community of knowledgeable and enthusiastic users. Several very well-designed programming languages have withered on the vine, whereas other programming languages whose design was merely "good enough" have thrived in the marketplace.

427

428

CHAPTER SUMMARY

1. Use interface types to make code more reusable.
2. A Java interface type declares a set of methods and their signatures.
3. Unlike a class, an interface type provides no implementation.
4. Use the `implements` keyword to indicate that a class implements an interface type.
5. Interfaces can reduce the coupling between classes.
6. You can convert from a class type to an interface type, provided the class implements the interface.
7. You need a cast to convert from an interface type to a class type.
8. Polymorphism denotes the principle that behavior can vary depending on the actual type of an object.
9. Early binding of methods occurs if the compiler selects a method from several possible candidates. Late binding occurs if the method selection takes place when the program runs.
10. An inner class is declared inside another class. Inner classes are commonly used for tactical classes that should not be visible elsewhere in a program.
11. User interface events include key presses, mouse moves, button clicks, menu selections, and so on.
12. An event listener belongs to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs.

13. Event sources report on events. When an event occurs, the event source notifies all event listeners.
14. Use `JButton` components for buttons. Attach an `ActionListener` to each button.
15. Methods of an inner class can access local variables from surrounding blocks and fields from surrounding classes.
16. Local variables that are accessed by an inner-class method must be declared as `final`.
17. Use a `JPanel` container to group multiple user-interface components together.
18. You often install event listeners as inner classes so that they can have access to the surrounding fields, methods, and final variables.
19. A timer generates timer events at fixed intervals.

428
429

20. The `repaint` method causes a component to repaint itself. Call this method whenever you modify the shapes that the `paintComponent` method draws.
21. You use a mouse listener to capture mouse events.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.awt.Component
   addMouseListener
   repaint
java.awt.Container
   add
java.awt.Rectangle
   setLocation
java.awt.event.MouseEvent
   getX
   getY
java.awt.event.ActionListener
   actionPerformed
java.awt.event.MouseListener
   mouseClicked
```

```
      mouseEntered
      mouseExited
      mousePressed
      mouseReleased
javax.swing.AbstractButton
      addActionListener
javax.swing.JButton
javax.swing.JLabel
javax.swing.JPanel
javax.swing.Timer
      start
      stop
```

REVIEW EXERCISES

★ **Exercise R9.1.** Suppose `C` is a class that implements the interfaces `I` and `J`. Which of the following assignments require a cast?

```
C c = . . .;
I i = . . .;
J j = . . .;
```

a. `c = i;`

b. `j = c;`

c. `i = j;`

★ **Exercise R9.2.** Suppose `C` is a class that implements the interfaces `I` and `J`, and suppose `i` is declared as

```
I i = new C();
```

Which of the following statements will throw an exception?

a. `C c = (C) i;`

b. `J j = (J) i;`

c. `i = (I) null;`

★ **Exercise R9.3.** Suppose the class `Sandwich` implements the `Edible` interface, and you are given the variable definitions

```
Sandwich sub = new Sandwich();
Rectangle cerealBox = new Rectangle(5, 10,
20, 30);
Edible e = null;
```

Which of the following assignment statements are legal?

a. `e = sub;`

b. `sub = e;`

c. `sub = (Sandwich) e;`

d. `sub = (Sandwich) cerealBox;`

e. `e = cerealBox;`

f. `e = (Edible) cerealBox;`

g. `e = (Rectangle) cerealBox;`

h. `e = (Rectangle) null;`

★★ **Exercise R9.4.** How does a cast such as `(BankAccount) x` differ from a cast of number values such as `(int) x`?

★★ **Exercise R9.5.** The classes `Rectangle2D.Double`, `Ellipse2D.Double`, and `Line2D.Double` implement the `Shape` interface. The `Graphics2D` class depends on the `Shape` interface but not on the rectangle, ellipse, and line classes. Draw a UML diagram denoting these facts.

★★ **Exercise R9.6.** Suppose `r` contains a reference to a `new Rectangle(5, 10, 20, 30)`. Which of the following assignments is legal? (Look inside the API documentation to check which interfaces the `Rectangle` class implements.)

a. `Rectangle a = r;`

b. `Shape b = r;`

c. `String c = r;`

d. `ActionListener d = r;`

e. `Measurable e = r;`

f. `Serializable f = r;`

g. `Object g = r;`

430

431

★★ **Exercise R9.7.** Classes such as `Rectangle2D.Double`, `Ellipse2D.Double` and `Line2D.Double` implement the `Shape` interface. The `Shape` interface has a method

```
Rectangle getBounds()
```

that returns a rectangle completely enclosing the shape. Consider the method call:

```
Shape s = . . .;
Rectangle r = s.getBounds();
```

Explain why this is an example of polymorphism.

★★★ **Exercise R9.8.** In Java, a method call such as `x.f()` uses late binding—the exact method to be called depends on the type of the object to which `x` refers. Give two kinds of method calls that use early binding in Java.

★★ **Exercise R9.9.** Suppose you need to process an array of employees to find the average and the highest salaries. Discuss what you need to do to use the implementation of the `DataSet` class in Section 9.1 (which processes `Measurable` objects). What do you need to do to use the second implementation (in Section 9.4)? Which is easier?

★★★ **Exercise R9.10.** What happens if you add a `String` object to the implementation of the `DataSet` class in Section 9.1? What happens if you add a `String` object to a `DataSet` object of the implementation in Section 9.4 that uses a `RectangleMeasurer` class?

★ **Exercise R9.11.** How would you reorganize the `DataSetTester3` program if you needed to make `RectangleMeasurer` into a top-level class (that is, not an inner class)?

★★ **Exercise R9.12.** What is a callback? Can you think of another use for a callback for the `DataSet` class? (*Hint*: Exercise P9.8.)

★★ **Exercise R9.13.** Consider this top-level and inner class. Which variables can the `f` method access?

```
public class T
{
   public void m(final int x, int y)
   {
      int a;
      final int b;
      class C implements I
      {
         public void f()
         {
            . . .
         }
      }
      final int c;
      . . .
   }
   private int t;
}
```

431

432

★★ **Exercise R9.14.** What happens when an inner class tries to access a non-`final` local variable? Try it out and explain your findings.

★★★G **Exercise R9.15.** How would you reorganize the `InvestmentViewer1` program if you needed to make `AddInterestListener` into a top-level class (that is, not an inner class)?

★G **Exercise R9.16.** What is an event object? An event source? An event listener?

★G **Exercise R9.17.** From a programmer's perspective, what is the most important difference between the user interfaces of a console application and a graphical application?

★G **Exercise R9.18.** What is the difference between an `ActionEvent` and a `MouseEvent`?

★★G **Exercise R9.19.** Why does the `ActionListener` interface have only one method, whereas the `MouseListener` has five methods?

★★G **Exercise R9.20.** Can a class be an event source for multiple event types? If so, give an example.

★★G **Exercise R9.21.** What information does an action event object carry? What additional information does a mouse event object carry?

★★★G **Exercise R9.22.** Why are we using inner classes for event listeners? If Java did not have inner classes, could we still implement event listeners? How?

★★G **Exercise R9.23.** What is the difference between the `paintComponent` and `repaint` methods?

★G **Exercise R9.24.** What is the difference between a frame and a panel?

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★ **Exercise P9.1.** Have the `Die` class of Chapter 6 implement the `Measurable` interface. Generate dice, cast them, and add them to the implementation of the `DataSet` class in Section 9.1. Display the average.

★ **Exercise P9.2.** Define a class `Quiz` that implements the `Measurable` interface. A quiz has a score and a letter grade (such as `B+`). Use the implementation of the `DataSet` class in Section 9.1 to process a collection of quizzes. Display the average score and the quiz with the highest score (both letter grade and score).

★ **Exercise P9.3.** A person has a name and a height in centimeters. Use the implementation of the `DataSet` class in Section 9.4 to process a collection of `Person` objects. Display the average height and the name of the tallest person.

★ **Exercise P9.4.** Modify the implementation of the `DataSet` class in Section 9.1 (the one processing `Measurable` objects) to also compute the minimum data element.

432

433

★ **Exercise P9.5.** Modify the implementation of the `DataSet` class in Section 9.4 (the one using a `Measurer` object) to also compute the minimum data element.

★ **Exercise P9.6.** Using a different `Measurer` object, process a set of `Rectangle` objects to find the rectangle with the largest perimeter.

★★★ **Exercise P9.7.** Enhance the `DataSet` class so that it can either be used with a `Measurer` object or for processing `Measurable` objects. *Hint:* Supply a default constructor that implements a `Measurer` that processes `Measurable` objects.

★★ **Exercise P9.8.** Define an interface `Filter` as follows:

```
public interface Filter
{
   boolean accept(Object x);
}
```

Modify the implementation of the `DataSet` class in Section 9.4 to use both a `Measurer` and a `Filter` object. Only objects that the filter accepts should be processed. Demonstrate your modification by having a data set process a collection of bank accounts, filtering out all accounts with balances less than $1,000.

★★ **Exercise P9.9.** Look up the definition of the standard `Comparable` interface in the API documentation. Modify the DataSet class of Section 9.1 to accept `Comparable` objects. With this interface, it is no longer meaningful to compute the average. The `DataSet` class should record the minimum and maximum data values. Test your modified `DataSet` class

by adding a number of `String` objects. (The `String` class implements the `Comparable` interface.)

★ **Exercise P9.10.** Modify the `Coin` class to have it implement the `Comparable` interface.

★★★ **Exercise P9.11.** The `System.out.printf` method has predefined formats for printing integers, floating-point numbers, and other data types. But it is also extensible. If you use the `S` format, you can print any class that implements the `Formattable` interface. That interface has a single method:

```
void formatTo(Formatter formatter, int
flags, int width, int precision)
```

In this exercise, you should make the `BankAccount` class implement the `Formattable` interface. Ignore the flags and precision and simply format the bank balance, using the given width. In order to achieve this task, you need to get an `Appendable` reference like this:

```
Appendable a = formatter.out();
```

`Appendable` is another interface with a method

```
void append(CharSequence sequence)
```

`CharSequence` is yet another interface that is implemented by (among others) the `String` class. Construct a string by first converting the bank balance into a string and then padding it with spaces so that it has the desired width. Pass that string to the `append` method.

433

434

★★★ **Exercise P9.12.** Enhance the `formatTo` method of Exercise P9.11 by taking into account the precision.

★★G **Exercise P9.13.** Write a method `randomShape` that randomly generates objects implementing the Shape interface: some mixture of rectangles, ellipses, and lines, with random positions. Call it 10 times and draw all of them.

★G **Exercise P9.14.** Enhance the `ButtonViewer` program so that it prints a message "I was clicked *n* times!" whenever the button is clicked. The value *n* should be incremented with each click.

★★G **Exercise P9.15.** Enhance the `ButtonViewer` program so that it has two buttons, each of which prints a message "I was clicked *n* times!" whenever the button is clicked. Each button should have a separate click count.

★★G **Exercise P9.16.** Enhance the `ButtonViewer` program so that it has two buttons labeled A and B, each of which prints a message "Button *x* was clicked!", where *x* is A or B.

★★★G **Exercise P9.17.** Implement a `ButtonViewer` program as in Exercise P9.16, using only a single listener class.

★G **Exercise P9.18.** Enhance the `ButtonViewer` program so that it prints the time at which the button was clicked.

★★★G **Exercise P9.19.** Implement the `AddInterestListener` in the `InvestmentViewer1` program as a regular class (that is, not an inner class). *Hint:* Store a reference to the bank account. Add a constructor to the listener class that sets the reference.

★★★G **Exercise P9.20.** Implement the `AddInterestListener` in the `InvestmentViewer2` program as a regular class (that is, not an inner class). *Hint:* Store references to the bank account and the label in the listener. Add a constructor to the listener class that sets the references.

★★G **Exercise P9.21.** Write a program that uses a timer to print the current time once a second. *Hint:* The following code prints the current time:

```
Date now = new Date();
System.out.println(now);
```

The `Date` class is in the `java.util` package.

★★★G **Exercise P9.22.** Change the `RectangleComponent` for the animation program in Section 9.9 so that the rectangle bounces off the edges of the component rather than simply moving outside.

★★G **Exercise P9.23.** Write a program that animates a car so that it moves across a frame.

★★★G **Exercise P9.24.** Write a program that animates two cars moving across a frame in opposite directions (but at different heights so that they don't collide.)

★★G **Exercise P9.25.** Change the `RectangleComponent` for the mouse listener program in Section 9.10 so that a new rectangle is added to the component whenever the mouse is clicked. *Hint:* Keep an `ArrayList<Rectangle>` and draw all rectangles in the `paint-Component` method.

434

435

★★G **Exercise P9.26.** Write a program that demonstrates the growth of a roach population. Start with two roaches and double the number of roaches with each button click.

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 9.1.** Design an interface `MoveableShape` that can be used as a generic mechanism for animating a shape. A moveable shape must have two methods: `move` and `draw`. Write a generic `AnimationPanel` that paints and moves any `MoveableShape` (or array list of `MoveableShape` objects if you covered Chapter 7). Supply moveable rectangle and car shapes.

★★★ **Project 9.2.** Your task is to design a general program for managing board games with two players. Your program should be flexible enough to handle games such as tic-tac-toe, chess, or the Game of Nim of Project 6.2.

Design an interface `Game` that describes a board game. Think about what your program needs to do. It asks the first player to input a move—a string in a game-specific format, such as `Be3` in chess. Your program knows nothing about specific games, so the `Game` interface must have a method such as

```
boolean isValidMove(String move)
```

Once the move is found to be valid, it needs to be executed—the interface needs another method `executeMove`. Next, your program needs to check whether the game is over. If not, the other player's move is processed. You should also provide some mechanism for displaying the current state of the board.

Design the `Game` interface and provide two implementations of your choice—such as `Nim` and `Chess` (or `TicTacToe` if you are less ambitious). Your `GamePlayer` class should manage a `Game` reference without knowing which game is played, and process the moves from both players. Supply two programs that differ only in the initialization of the `Game` reference.

435

436

ANSWERS TO SELF-CHECK QUESTIONS

1. It must implement the `Measurable` interface, and its `getMeasure` method must return the population.
2. The `Object` class doesn't have a `getMeasure` method, and the add method invokes the `getMeasure` method.
3. Only if `x` actually refers to a `BankAccount` object.
4. No—a `Coin` reference can be converted to a `Measurable` reference, but if you attempt to cast that reference to a `BankAccount`, an exception occurs.
5. `Measurable` is an interface. Interfaces have no fields and no method implementations.

6. That variable never refers to a `Measurable` object. It refers to an object of some class—a class that implements the `Measurable` interface.

7. Both describe a situation where one method name can denote multiple methods. However, overloading is resolved early by the compiler, by looking at the types of the parameter variables. Polymorphism is resolved late, by looking at the type of the implicit parameter object just before making the call.

8. The `String` class doesn't implement the `Measurable` interface.

9. Implement a class `StringMeasurer` that implements the `Measurer` interface.

10. A measurer measures an object, whereas `getMeasure` measures "itself", that is, the implicit parameter.

11. Inner classes are convenient for insignificant classes. Also, their methods can access variables and fields from the surrounding scope.

12. Four: one for the outer class, one for the inner class, and two for the `DataSet` and `Measurer` classes.

13. The `button` object is the event source. The `listener` object is the event listener.

14. The `ClickListener` class implements the `ActionListener` interface.

15. Direct access is simpler than the alternative—passing the variable as a parameter to a constructor or method.

16. The local variable must be declared as `final`.

17. First add `label` to the `panel`, then add `button`.

18. The `actionPerformed` method does not access that variable.

19. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.

20. The moved rectangles won't be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.

21. Because you know the current mouse position, not the amount by which the mouse has moved.

22. It implements the `MouseListener` interface, which has five methods.