Notice that the first argument to **collect( )** is a lambda expression that returns a new **LinkedList**. The second argument uses the standard collection method **add( )** to add an element to the list. The third element uses **addAll( )** to combine two linked lists. As a point of interest, you can use any method defined by **LinkedList** to add an element to the list. For example, you could use **addFirst( )** to add elements to the start of the list, as shown here:

```
(list, element) -> list.addFirst(element)
```

As you may have guessed, it is not always necessary to specify a lambda expression for the arguments to **collect( )**. Often, method and/or constructor references will suffice. For example, again assuming the preceding program, this statement creates a **HashSet** that contains all of the elements in the **nameAndPhone** stream:

```
HashSet<NamePhone> npSet = nameAndPhone.collect(HashSet::new,
                                                HashSet::add,
                                                HashSet::addAll);
```

Notice that the first argument specifies the **HashSet** constructor reference. The second and third specify method references to **HashSet**'s **add( )** and **addAll( )** methods.

One last point: In the language of the stream API, the **collect( )** method performs what is called a *mutable reduction*. This is because the result of the reduction is a mutable (i.e., changeable) storage object, such as a collection.

Iterators and Streams

Although a stream is not a data storage object, you can still use an iterator to cycle through its elements in much the same way as you would use an iterator to cycle through the elements of a collection. The stream API supports two types of iterators. The first is the traditional **Iterator**. The second is **Spliterator**, which was added by JDK 8. It provides significant advantages in certain situations when used with parallel streams.

Use an Iterator with a Stream

As just mentioned, you can use an iterator with a stream in just the same way that you do with a collection. Iterators are discussed in Chapter 19, but a brief

review will be useful here. Iterators are objects that implement the **Iterator** interface declared in **java.util**. Its two key methods are **hasNext( )** and **next( )**. If there is another element to iterate, **hasNext( )** returns **true**, and **false** otherwise. The **next( )** method returns the next element in the iteration.

NOTE There are additional iterator types that handle the primitive streams: **PrimitiveIterator**, **PrimitiveIterator.OfDouble**, **PrimitiveIterator.OfLong**, and **PrimitiveIterator.OfInt**. These iterators all extend the **Iterator** interface and work in the same general way as those based directly on **Iterator**.

To obtain an iterator to a stream, call **iterator( )** on the stream. The version used by **Stream** is shown here.

Iterator<T> iterator( )

Here, **T** specifies the element type. (The primitive streams return iterators of the appropriate primitive type.)

The following program shows how to iterate through the elements of a stream. In this case, the strings in an **ArrayList** are iterated, but the process is the same for any type of stream.

```
// Use an iterator with a stream.

import java.util.*;
import java.util.stream.*;

class StreamDemo8 {

  public static void main(String[] args) {

    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();
```

```
    // Obtain an iterator to the stream.
    Iterator<String> itr = myStream.iterator();

    // Iterate the elements in the stream.
    while(itr.hasNext())
      System.out.println(itr.next());
  }
}
```

The output is shown here:

```
Alpha
Beta
Gamma
Delta
Phi
Omega
```

Use Spliterator

Spliterator offers an alternative to **Iterator**, especially when parallel processing is involved. In general, **Spliterator** is more sophisticated than **Iterator**, and a discussion of **Spliterator** is found in Chapter 19. However, it will be useful to review its key features here. **Spliterator** defines several methods, but we only need to use three. The first is **tryAdvance( )**. It performs an action on the next element and then advances the iterator. It is shown here:

boolean tryAdvance(Consumer<? super T> *action*)

Here, *action* specifies the action that is executed on the next element in the iteration. **tryAdvance( )** returns **true** if there is a next element. It returns **false** if no elements remain. As discussed earlier in this chapter, **Consumer** declares one method called **accept( )** that receives an element of type **T** as an argument and returns **void**.

Because **tryAdvance( )** returns **false** when there are no more elements to process, it makes the iteration loop construct very simple, for example:

```
while(splitItr.tryAdvance( // perform action here );
```

As long as **tryAdvance( )** returns **true**, the action is applied to the next element. When **tryAdvance( )** returns **false**, the iteration is complete. Notice how **tryAdvance( )** consolidates the purposes of **hasNext( )** and **next( )** provided by

Iterator into a single method. This improves the efficiency of the iteration process.

The following version of the preceding program substitutes a **Spliterator** for the **Iterator**:

```
// Use a Spliterator.

import java.util.*;
import java.util.stream.*;

class StreamDemo9 {

  public static void main(String[] args) {

    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Iterate the elements of the stream.
    while(splitItr.tryAdvance((n) -> System.out.println(n)));
  }
}
```

The output is the same as before.

In some cases, you might want to perform some action on each element collectively, rather than one at a time. To handle this type of situation, **Spliterator** provides the **forEachRemaining( )** method, shown here:

default void forEachRemaining(Consumer<? super T> *action*)

This method applies *action* to each unprocessed element and then returns. For

example, assuming the preceding program, the following displays the strings remaining in the stream:

```
splitItr.forEachRemaining((n) -> System.out.println(n));
```

Notice how this method eliminates the need to provide a loop to cycle through the elements one at a time. This is another advantage of **Spliterator**.

One other **Spliterator** method of particular interest is **trySplit( )**. It splits the elements being iterated in two, returning a new **Spliterator** to one of the partitions. The other partition remains accessible by the original **Spliterator**. It is shown here:

Spliterator<T> trySplit( )

If it is not possible to split the invoking **Spliterator**, **null** is returned. Otherwise, a reference to the partition is returned. For example, here is another version of the preceding program that demonstrates **trySplit( )**:

```
// Demonstrate trySplit().

import java.util.*;
import java.util.stream.*;
```

```
class StreamDemo10 {

  public static void main(String[] args) {

    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Now, split the first iterator.
    Spliterator<String> splitItr2 = splitItr.trySplit();

     // If splitItr could be split, use splitItr2 first.
    if(splitItr2 != null) {
      System.out.println("Output from splitItr2: ");
      splitItr2.forEachRemaining((n) -> System.out.println(n));
    }

    // Now, use the splitItr.
    System.out.println("\nOutput from splitItr: ");
    splitItr.forEachRemaining((n) -> System.out.println(n));
  }
}
```

The output is shown here:

```
Output from splitItr2:
Alpha
Beta
Gamma

Output from splitItr:
Delta
Phi
```

```
Omega
```

Although splitting the **Spliterator** in this simple illustration is of no practical value, splitting can be of *great value* when parallel processing over large data sets. However, in many cases, it is better to use one of the other **Stream** methods in conjunction with a parallel stream, rather than manually handling these details with **Spliterator**. **Spliterator** is primarily for the cases in which none of the predefined methods seems appropriate.

More to Explore in the Stream API

This chapter has discussed several key aspects of the stream API and introduced the techniques required to use them, but the stream API has much more to offer. To begin, here are a few of the other methods provided by **Stream** that you will find helpful:

- To determine if one or more elements in a stream satisfy a specified predicate, use **allMatch( )**, **anyMatch( )**, or **noneMatch( )**.
- To obtain the number of elements in the stream, call **count( )**.
- To obtain a stream that contains only unique elements, use **distinct( )**.
- To create a stream that contains a specified set of elements, use **of( )**.

One last point: the stream API is a powerful addition to Java. You will want to explore all of the capabilities that **java.util.stream** has to offer.

CHAPTER

30

Regular Expressions and Other Packages

When Java was originally released, it included a set of eight packages, called the *core API*. Each subsequent release added to the API. Today, the Java API contains a very large number of packages. Many of the packages support areas of specialization that are beyond the scope of this book. However, several packages warrant an examination here. Four are **java.util.regex**, **java.lang.reflect**, **java.rmi**, and **java.text**. They support regular expression processing, reflection, Remote Method Invocation (RMI), and text formatting, respectively. The chapter ends by introducing the date and time API in **java.time** and its subpackages.

The *regular expression* package lets you perform sophisticated pattern matching operations. This chapter provides an introduction to this package along with extensive examples. *Reflection* is the ability of software to analyze itself. It is an essential part of the Java Beans technology that is covered in Chapter 37. *Remote Method Invocation (RMI)* allows you to build Java applications that are distributed among several machines. This chapter provides a simple client/server example that uses RMI. The *text formatting* capabilities of **java.text** have many uses. The one examined here formats date and time strings. The date and time API supplies an up-to-date approach to handling date and time.

Regular Expression Processing

The **java.util.regex** package supports regular expression processing. Beginning with JDK 9, **java.util.regex** is in the **java.base** module. As the term is used here, a *regular expression* is a string of characters that describes a character sequence. This general description, called a *pattern*, can then be used to find matches in other character sequences. Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences.

There are two classes that support regular expression processing: **Pattern** and

Matcher. These classes work together. Use **Pattern** to define a regular expression. Match the pattern against another sequence using **Matcher**.

Pattern

The **Pattern** class defines no constructors. Instead, a pattern is created by calling the **compile( )** factory method. One of its forms is shown here:

static Pattern compile(String *pattern*)

Here, *pattern* is the regular expression that you want to use. The **compile( )** method transforms the string in *pattern* into a pattern that can be used for pattern matching by the **Matcher** class. It returns a **Pattern** object that contains the pattern.

Once you have created a **Pattern** object, you will use it to create a **Matcher**. This is done by calling the **matcher( )** method defined by **Pattern**. It is shown here:

Matcher matcher(CharSequence *str*)

Here *str* is the character sequence that the pattern will be matched against. This is called the *input sequence*. **CharSequence** is an interface that defines a read-only set of characters. It is implemented by the **String** class, among others. Thus, you can pass a string to **matcher( )**.

Matcher

The **Matcher** class has no constructors. Instead, you create a **Matcher** by calling the **matcher( )** factory method defined by **Pattern**, as just explained. Once you have created a **Matcher**, you will use its methods to perform various pattern matching operations. Several are described here.

The simplest pattern matching method is **matches( )**, which determines whether the character sequence matches the pattern. It is shown here:

boolean matches( )

It returns **true** if the sequence and the pattern match, and **false** otherwise. Understand that the entire sequence must match the pattern, not just a subsequence of it.

To determine if a subsequence of the input sequence matches the pattern, use

find( ). One version is shown here:

```
boolean find( )
```

It returns **true** if there is a matching subsequence and **false** otherwise. This method can be called repeatedly, allowing it to find all matching subsequences. Each call to **find( )** begins where the previous one left off.

You can obtain a string containing the last matching sequence by calling **group( )**. One of its forms is shown here:

```
String group( )
```

The matching string is returned. If no match exists, then an **IllegalStateException** is thrown.

You can obtain the index within the input sequence of the current match by calling **start( )**. The index one past the end of the current match is obtained by calling **end( )**. The forms used in this chapter are shown here:

```
int start( )
int end( )
```

Both throw **IllegalStateException** if no match exists.

You can replace all occurrences of a matching sequence with another sequence by calling **replaceAll( )**. One version is shown here:

```
String replaceAll(String newStr)
```

Here, *newStr* specifies the new character sequence that will replace the ones that match the pattern. The updated input sequence is returned as a string.

Regular Expression Syntax

Before demonstrating **Pattern** and **Matcher**, it is necessary to explain how to construct a regular expression. Although no rule is complicated by itself, there are a large number of them, and a complete discussion is beyond the scope of this chapter. However, a few of the more commonly used constructs are described here.

In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers. A normal character is matched as-is. Thus, if a pattern consists of "xy", then the only input

sequence that will match it is "xy". Characters such as newline and tab are specified using the standard escape sequences, which begin with a \ . For example, a newline is specified by **\n**. In the language of regular expressions, a normal character is also called a *literal*.

A character class is a set of characters. A character class is specified by putting the characters in the class between brackets. For example, the class [wxyz] matches w, x, y, or z. To specify an inverted set, precede the characters with a ^. For example, [^wxyz] matches any character except w, x, y, or z. You can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use [1-9].

The wildcard character is the **.** (dot) and it matches any character. Thus, a pattern that consists of "." will match these (and other) input sequences: "A", "a", "x", and so on.

A quantifier determines how many times an expression is matched. The basic quantifiers are shown here:

+	Match one or more.
*	Match zero or more.
?	Match zero or one.

For example, the pattern "x+" will match "x", "xx", and "xxx", among others. As you will see, variations are supported that affect how matching is performed.

One other point: In general, if you specify an invalid expression, a **PatternSyntaxException** will be thrown.

Demonstrating Pattern Matching

The best way to understand how regular expression pattern matching operates is to work through some examples. The first, shown here, looks for a match with a literal pattern:

```
// A simple pattern matching demo.
import java.util.regex.*;
class RegExpr {
  public static void main(String args[]) {
    Pattern pat;
    Matcher mat;
    boolean found;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    found = mat.matches(); // check for a match

    System.out.println("Testing Java against Java.");
    if(found) System.out.println("Matches");
    else System.out.println("No Match");

    System.out.println();

    System.out.println("Testing Java against Java SE.");
    mat = pat.matcher("Java SE"); // create a new matcher

    found = mat.matches(); // check for a match

    if(found) System.out.println("Matches");
    else System.out.println("No Match");
  }
}
```

The output from the program is shown here:

```
Testing Java against Java.
Matches

Testing Java against Java SE.
No Match
```

Let’s look closely at this program. The program begins by creating the pattern that contains the sequence "Java". Next, a **Matcher** is created for that pattern that has the input sequence "Java". Then, the **matches( )** method is called to determine if the input sequence matches the pattern. Because the sequence and the pattern are the same, **matches( )** returns **true**. Next, a new **Matcher** is created with the input sequence "Java SE" and **matches( )** is called again. In this case, the pattern and the input sequence differ, and no match is found.

Remember, the **matches( )** function returns **true** only when the input sequence precisely matches the pattern. It will not return **true** just because a subsequence matches.

You can use **find( )** to determine if the input sequence contains a subsequence that matches the pattern. Consider the following program:

```
// Use find() to find a subsequence.
import java.util.regex.*;

class RegExpr2 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java SE");
    System.out.println("Looking for Java in Java SE.");

    if(mat.find()) System.out.println("subsequence found");
    else System.out.println("No Match");
  }
}
```

The output is shown here:

```
Looking for Java in Java SE.
subsequence found
```

In this case, **find( )** finds the subsequence "Java".

The **find( )** method can be used to search the input sequence for repeated occurrences of the pattern because each call to **find( )** picks up where the previous one left off. For example, the following program finds two occurrences of the pattern "test":

```
// Use find() to find multiple subsequences.
import java.util.regex.*;

class RegExpr3 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("test");
    Matcher mat = pat.matcher("test 1 2 3 test");

    while(mat.find()) {
      System.out.println("test found at index " +
                         mat.start());
    }
  }
}
```

The output is shown here:

```
test found at index 0
test found at index 11
```

As the output shows, two matches were found. The program uses the **start( )** method to obtain the index of each match.

Using Wildcards and Quantifiers

Although the preceding programs show the general technique for using **Pattern** and **Matcher**, they don't show their power. The real benefit of regular expression processing is not seen until wildcards and quantifiers are used. To begin, consider the following example that uses the + quantifier to match any arbitrarily long sequence of Ws:

```
// Use a quantifier.
import java.util.regex.*;

class RegExpr4 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("W+");
    Matcher mat = pat.matcher("W WW WWW");
    while(mat.find())
      System.out.println("Match: " + mat.g
  }
}
```

The output from the program is shown here:

```
Match: W
Match: WW
Match: WWW
```

As the output shows, the regular expression pattern "W+" matches any arbitrarily long sequence of Ws.

The next program uses a wildcard to create a pattern that will match any sequence that begins with *e* and ends with *d*. To do this, it uses the dot wildcard character along with the + quantifier.

```
// Use wildcard and quantifier.
import java.util.regex.*;

class RegExpr5 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("e.+d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

You might be surprised by the output produced by the program, which is shown here:

```
Match: extend cup end
```

Only one match is found, and it is the longest sequence that begins with *e* and ends with *d*. You might have expected two matches: "extend" and "end". The reason that the longer sequence is found is that the pattern "e.+d" matches the longest sequence that fits the pattern. This is called *greedy behavior*. You can specify *reluctant behavior* by adding the **?** to the pattern, as shown in this version of the program. It causes the shortest matching pattern to be obtained.

```
// Use a reluctant quantifier.
import java.util.regex.*;

class RegExpr6 {
  public static void main(String args[]) {
    // Use reluctant matching behavior.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");
    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output from the program is shown here:

```
Match: extend
Match: end
```

As the output shows, the pattern "e.+?d" will match the shortest sequence that begins with *e* and ends with *d*. Thus, two matches are found.

In general, to convert a greedy quantifier into a reluctant quantifier, add a **?**. You can also specify *possessive* behavior by appending a **+**. For example, you might want to try the pattern **"e.?+d"** and observe the result. You can also specify a number of times to match by using ***{min, limit}***, which matches *min* times, up to *limit* times. Also supported are ***{min}*** and ***{min,}*** which match *min* times, and *min* times but possibly more, respectively.

Working with Classes of Characters

Sometimes you will want to match any sequence that contains one or more characters, in any order, that are part of a set of characters. For example, to match whole words, you want to match any sequence of the letters of the alphabet. One of the easiest ways to do this is to use a character class, which defines a set of characters. Recall that a character class is created by putting the characters you want to match between brackets. For example, to match the lowercase characters a through z, use **[a-z]**. The following program demonstrates this technique:

```
// Use a character class.
import java.util.regex.*;

class RegExpr7 {
  public static void main(String args[]) {
    // Match lowercase words.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output is shown here:

```
Match: this
Match: is
Match: a
Match: test
```

Using replaceAll( )

The **replaceAll( )** method supplied by **Matcher** lets you perform powerful search and replace operations that use regular expressions. For example, the following program replaces all occurrences of sequences that begin with "Jon" with "Eric":

```
// Use replaceAll().
import java.util.regex.*;

class RegExpr8 {
  public static void main(String args[]) {
    String str = "Jon Jonathan Frank Ken Todd";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);

    System.out.println("Original sequence: " + str);

    str = mat.replaceAll("Eric ");

    System.out.println("Modified sequence: " + str);

  }
}
```

The output is shown here:

```
Original sequence: Jon Jonathan Frank Ken Todd
Modified sequence: Eric Eric Frank Ken Todd
```

Because the regular expression "Jon.*? " matches any string that begins with Jon followed by zero or more characters, ending in a space, it can be used to match and replace both Jon and Jonathan with the name Eric. Such a substitution is not easily accomplished without pattern matching capabilities.

Using split( )

You can reduce an input sequence into its individual tokens by using the **split( )** method defined by **Pattern**. One form of the **split( )** method is shown here:

String[ ] split(CharSequence *str*)

It processes the input sequence passed in *str*, reducing it into tokens based on the delimiters specified by the pattern.

For example, the following program finds tokens that are separated by spaces, commas, periods, and exclamation points:

```
// Use split().
import java.util.regex.*;

class RegExpr9 {
  public static void main(String args[]) {

    // Match lowercase words.
    Pattern pat = Pattern.compile("[ ,.!]");

    String strs[] = pat.split("one two,alpha9 12!done.");

    for(int i=0; i < strs.length; i++)
      System.out.println("Next token: " + strs[i]);

  }
}
```

The output is shown here:

```
Next token: one
Next token: two
Next token: alpha9
Next token: 12
Next token: done
```

As the output shows, the input sequence is reduced to its individual tokens. Notice that the delimiters are not included.

Two Pattern-Matching Options

Although the pattern-matching techniques described in the foregoing offer the greatest flexibility and power, there are two alternatives which you might find useful in some circumstances. If you only need to perform a one-time pattern match, you can use the **matches( )** method defined by **Pattern**. It is shown here:

static boolean matches(String *pattern*, CharSequence *str*)

It returns **true** if *pattern* matches *str* and **false** otherwise. This method automatically compiles *pattern* and then looks for a match. If you will be using the same pattern repeatedly, then using **matches( )** is less efficient than compiling the pattern and using the pattern-matching methods defined by **Matcher**, as described previously.

You can also perform a pattern match by using the **matches( )** method

implemented by **String**. It is shown here:

boolean matches(String *pattern*)

If the invoking string matches the regular expression in *pattern*, then **matches( )** returns **true**. Otherwise, it returns **false**.

Exploring Regular Expressions

The overview of regular expressions presented in this section only hints at their power. Since text parsing, manipulation, and tokenization are a large part of programming, you will likely find Java's regular expression subsystem a powerful tool that you can use to your advantage. It is, therefore, wise to explore the capabilities of regular expressions. Experiment with several different types of patterns and input sequences. Once you understand how regular expression pattern matching works, you will find it useful in many of your programming endeavors.

Reflection

Reflection is the ability of software to analyze itself. This is provided by the **java.lang.reflect** package and elements in **Class**. Beginning with JDK 9, **java.lang.reflect** is part of the **java.base** module. Reflection is an important capability, especially when using components called Java Beans. It allows you to analyze a software component and describe its capabilities dynamically, at run time rather than at compile time. For example, by using reflection, you can determine what methods, constructors, and fields a class supports. Reflection was introduced in Chapter 12. It is examined further here.

The package **java.lang.reflect** includes several interfaces. Of special interest is **Member**, which defines methods that allow you to get information about a field, constructor, or method of a class. There are also ten classes in this package. These are listed in Table 30-1.

Class	Primary Function
AccessibleObject	Allows you to bypass the default access control checks.
Array	Allows you to dynamically create and manipulate arrays.
Constructor	Provides information about a constructor.
Executable	An abstract superclass extended by **Method** and **Constructor**.
Field	Provides information about a field.
Method	Provides information about a method.
Modifier	Provides information about class and member access modifiers.
Parameter	Provides information about parameters.
Proxy	Supports dynamic proxy classes.
ReflectPermission	Allows reflection of private or protected members of a class.

Table 30-1 Classes Defined in **java.lang.reflect**

The following application illustrates a simple use of the Java reflection capabilities. It prints the constructors, fields, and methods of the class **java.awt.Dimension**. The program begins by using the **forName( )** method of **Class** to get a class object for **java.awt.Dimension**. Once this is obtained, **getConstructors( )**, **getFields( )**, and **getMethods( )** are used to analyze this class object. They return arrays of **Constructor**, **Field**, and **Method** objects that provide the information about the object. The **Constructor**, **Field**, and **Method** classes define several methods that can be used to obtain information about an object. You will want to explore these on your own. However, each supports the **toString( )** method. Therefore, using **Constructor**, **Field**, and **Method** objects as arguments to the **println( )** method is straightforward, as shown in the program.

```
// Demonstrate reflection.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String args[]) {
    try {
      Class<?> c = Class.forName("java.awt.Dimension");
      System.out.println("Constructors:");
      Constructor<?> constructors[] = c.getConstructors();
      for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }

      System.out.println("Fields:");
      Field fields[] = c.getFields();
      for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
      }

      System.out.println("Methods:");
      Method methods[] = c.getMethods();
      for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

Here is the output from this program. (The output you see when you run the program may differ slightly from that shown.)

```
Constructors:
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
 public java.awt.Dimension(java.awt.Dimension)
Fields:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Methods:
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public java.awt.Dimension java.awt.Dimension.getSize()
 public void java.awt.Dimension.setSize(double,double)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(int,int)
 public double java.awt.Dimension.getHeight()
 public double java.awt.Dimension.getWidth()
 public java.lang.Object java.awt.geom.Dimension2D.clone()
 public void java.awt.geom.
             Dimension2D.setSize(java.awt.geom.Dimension2D)
 public final native java.lang.Class java.lang.Object.getClass()
 public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait()
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait(long,int)
   throws java.lang.InterruptedException
 public final native void java.lang.Object.notify()
 public final native void java.lang.Object.notifyAll()
```

The next example uses Java's reflection capabilities to obtain the public methods of a class. The program begins by instantiating class **A**. The **getClass( )** method is applied to this object reference, and it returns the **Class** object for class **A**. The **getDeclaredMethods( )** method returns an array of **Method** objects that describe only the methods declared by this class. Methods inherited from superclasses such as **Object** are not included.

Each element of the **methods** array is then processed. The **getModifiers( )** method returns an **int** containing flags that describe which modifiers apply for this element. The **Modifier** class provides a set of **is*X*** methods, shown in Table 30-2, that can be used to examine this value. For example, the static method **isPublic( )** returns **true** if its argument includes the **public** modifier. Otherwise, it returns **false**. In the following program, if the method supports public access,

its name is obtained by the **getName( )** method and is then printed.

Method	Description
static boolean isAbstract(int *val*)	Returns **true** if *val* has the **abstract** flag set and **false** otherwise.
static boolean isFinal(int *val*)	Returns **true** if *val* has the **final** flag set and **false** otherwise.
static boolean isInterface(int *val*)	Returns **true** if *val* has the **interface** flag set and **false** otherwise.
static boolean isNative(int *val*)	Returns **true** if *val* has the **native** flag set and **false** otherwise.
static boolean isPrivate(int *val*)	Returns **true** if *val* has the **private** flag set and **false** otherwise.
static boolean isProtected(int *val*)	Returns **true** if *val* has the **protected** flag set and **false** otherwise.
static boolean isPublic(int *val*)	Returns **true** if *val* has the **public** flag set and **false** otherwise.
static boolean isStatic(int *val*)	Returns **true** if *val* has the **static** flag set and **false** otherwise.
static boolean isStrict(int *val*)	Returns **true** if *val* has the **strict** flag set and **false** otherwise.
static boolean isSynchronized(int *val*)	Returns **true** if *val* has the **synchronized** flag set and **false** otherwise.
static boolean isTransient(int *val*)	Returns **true** if *val* has the **transient** flag set and **false** otherwise.
static boolean isVolatile(int *val*)	Returns **true** if *val* has the **volatile** flag set and **false** otherwise.

Table 30-2 The “is” Methods Defined by **Modifier** That Determine Modifiers

```
// Show public methods.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {
    try {
      A a = new A();
      Class<?> c = a.getClass();
      System.out.println("Public Methods:");
      Method methods[] = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
          System.out.println(" " + methods[i].getName());
        }
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}

class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
```

Here is the output from this program:

```
Public Methods:
  a1
  a2
```

Modifier also includes a set of static methods that return the type of modifiers that can be applied to a specific type of program element. These methods are

static int classModifiers( )

static int constructorModifiers( )

static int fieldModifiers( )

static int interfaceModifiers( )

static int methodModifiers( )

static int parameterModifiers( )

For example, **methodModifiers( )** returns the modifiers that can be applied to a method. Each method returns flags, packed into an **int**, that indicate which modifiers are legal. The modifier values are defined by constants in **Modifier**, which include **PROTECTED**, **PUBLIC**, **PRIVATE**, **STATIC**, **FINAL**, and so on.

Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications. While a complete discussion of RMI is outside the scope of this book, the following simplified example describes the basic principles involved. RMI is supported by the **java.rmi** package. Beginning with JDK 9, it is part of the **java.rmi** module.

A Simple Client/Server Application Using RMI

This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

Step One: Enter and Compile the Source Code

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one

method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

```
import java.rmi.*;

public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add( )** method is straightforward. Remote objects typically extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

```
import java.rmi.*;
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind( )** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind( )** method is a string that names the server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;

public class AddServer {
  public static void main(String args[]) {

    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add( )** method. The sum is returned from this method and is then printed.

```
import java.rmi.*;

public class AddClient {
  public static void main(String args[]) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf =
                    (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("The first number is: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("The second number is: " + args[2]);

      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("The sum is: " + addServerIntf.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

After you enter all the code, use **javac** to compile the four source files that you created.

Step Two: Manually Generate a Stub if Required

In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. Recall from Chapter 21 that the serialization facilities also recursively process all referenced objects.

If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facilities are also used if objects are returned to a client.

Prior to Java 5, stubs needed to be built manually by using **rmic**. This step **is**

not required for modern versions of Java. However, if you are working in a very old legacy environment, then you can use the **rmic** compiler, as shown here, to build a stub:

```
rmic AddServerImpl
```

This command generates the file **AddServerImpl_Stub.class**. When using **rmic**, be sure that **CLASSPATH** is set to include the current directory.

Step Three: Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class** (if needed), and **AddServerIntf.class** to a directory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class** (if needed), and **AddServer.class** to a directory on the server machine.

NOTE RMI has techniques for dynamic class loading, but they are not used by the example at hand. Instead, all of the files that are used by the client and server applications must be installed manually on those machines.

Step Four: Start the RMI Registry on the Server Machine

The JDK provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. First, check that the **CLASSPATH** environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here:

```
start rmiregistry
```

When this command returns, you should see that a new window has been created. You need to leave this window open until you are done experimenting with the RMI example.

Step Five: Start the Server

The server code is started from the command line, as shown here:

```
java AddServer
```

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

Step Six: Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).

You can try this example without actually having a remote server. To do so, simply install all of the programs on the same machine, start **rmiregistry**, start **AddServer**, and then execute **AddClient** using this command line:

```
java AddClient 127.0.0.1 8 9
```

Here, the address 127.0.0.1 is the “loop back” address for the local machine. Using this address allows you to exercise the entire RMI mechanism without actually having to install the server on a remote computer. (If you are using a firewall, then this approach may not work.)

In either case, sample output from this program is shown here:

```
The first number is: 8
The second number is: 9
The sum is: 17.0
```

NOTE When working with RMI in the real world, it may be necessary for the server to install a security manager.

Formatting Date and Time with java.text

The package **java.text** allows you to format, parse, search, and manipulate text. Beginning with JDK 9, **java.text** is part of the **java.base** module. This section examines two of **java.text**'s most commonly used classes: those that format date and time information. However, it is important to state at the outset that the new date and time API described later in this chapter offers a modern approach to handling date and time that also supports formatting. Of course, legacy code will continue to use the classes shown here for some time.

DateFormat Class

DateFormat is an abstract class that provides the ability to format and parse dates and times. The **getDateInstance( )** method returns an instance of **DateFormat** that can format date information. It is available in these forms:

static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int *style*)
static final DateFormat getDateInstance(int *style*, Locale *locale*)

Here, *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the date to be presented. The parameter *locale* specifies the locale (refer to Chapter 20 for details on **Locale**). If the *style* and/or *locale* is not specified, defaults are used.

One of the most commonly used methods in this class is **format( )**. It has several overloaded forms, one of which is shown here:

final String format(Date *d*)

The argument is a **Date** object that is to be displayed. The method returns a string containing the formatted information.

The following listing illustrates how to format date information. It begins by creating a **Date** object. This captures the current date and time information. Then it outputs the date information by using different styles and locales.

```
// Demonstrate date formats.
import java.text.*;
import java.util.*;

public class DateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Korea: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("United States: " + df.format(date));
  }
}
```

Sample output from this program is shown here:

```
Japan: 2018/06/20
Korea: 2018. 6. 20.
United Kingdom: 20 June 2018
United States: Wednesday, June 20, 2018
```

The **getTimeInstance( )** method returns an instance of **DateFormat** that can format time information. It is available in these versions:

static final DateFormat getTimeInstance( )
static final DateFormat getTimeInstance(int *style*)
static final DateFormat getTimeInstance(int *style*, Locale *locale*)

Here, *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the time to be presented. The parameter *locale* specifies the locale. If the *style* and/or *locale* is not specified, defaults are used.

The following listing illustrates how to format time information. It begins by

creating a **Date** object. This captures the current date and time information. Then it outputs the time information by using different styles and locales.

```
// Demonstrate time formats.
import java.text.*;
import java.util.*;
public class TimeFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    System.out.println("Canada: " + df.format(date));
  }
}
```

Sample output from this program is shown here:

```
Japan: 13:03
United Kingdom: 13:03:31 GMT-06:00
Canada: 1:03:31 PM Central Daylight Time
```

The **DateFormat** class also has a **getDateTimeInstance( )** method that can format both date and time information. You may wish to experiment with it on your own.

SimpleDateFormat Class

SimpleDateFormat is a concrete subclass of **DateFormat**. It allows you to define your own formatting patterns that are used to display date and time information.

One of its constructors is shown here:

SimpleDateFormat(String *formatString*)

The argument *formatString* describes how date and time information is

displayed. An example of its use is given here:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss
zzz");
```

The symbols used in the formatting string determine the information that is displayed. Table 30-3 lists these symbols and gives a description of each.

Symbol	Description
a	AM or PM
d	Day of month (1–31)
h	Hour in AM/PM (1–12)
k	Hour in day (1–24)
m	Minute in hour (0–59)
s	Second in minute (0–59)
u	Day of week, with Monday being 1
w	Week of year (1–52)
y	Year
z	Time zone
D	Day of year (1–366)
E	Day of week (for example, Thursday)
F	Day of week in month
G	Era (for example, AD or BC)
H	Hour in day (0–23)
K	Hour in AM/PM (0–11)
L	Month
M	Month
S	Millisecond in second
W	Week of month (1–5)
X	Time zone in ISO 8601 format
Y	Week year
Z	Time zone in RFC 822 format

Table 30-3 Formatting String Symbols for **SimpleDateFormat**

In most cases, the number of times a symbol is repeated determines how that data is presented. Text information is displayed in an abbreviated form if the pattern letter is repeated less than four times. Otherwise, the unabbreviated form is used. For example, a zzzz pattern can display Pacific Daylight Time, and a zzz pattern can display PDT.

For numbers, the number of times a pattern letter is repeated determines how many digits are presented. For example, hh:mm:ss can present 01:51:15, but h:m:s displays the same time value as 1:51:15.

Finally, M or MM causes the month to be displayed as one or two digits. However, three or more repetitions of M cause the month to be displayed as a text string.

The following program shows how this class is used:

```
// Demonstrate SimpleDateFormat.
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
    System.out.println(sdf.format(date));
  }
}
```

Sample output from this program is shown here:

```
01:30:51
20 Jun 2018 01:30:51 CDT
Wed Jun 20 2018
```

The java.time Time and Date API

In Chapter 20, Java's long-standing approach to handling date and time through the use of classes such as **Calendar** and **GregorianCalendar** was discussed. It is expected that this traditional approach will remain in widespread use for some

time and is, therefore, something that all Java programmers need to be familiar with. However, beginning with JDK 8, Java includes another approach to handling time and date. This new approach is defined in the following packages:

Package	Description
java.time	Provides top-level classes that support time and date.
java.time.chrono	Supports alternative, non-Gregorian calendars.
java.time.format	Supports time and date formatting.
java.time.temporal	Supports extended date and time functionality.
java.time.zone	Supports time zones.

These packages define a large number of classes, interfaces, and enumerations that provide extensive, finely grained support for time and date operations. Because of the number of elements that comprise the new time and date API, it can seem fairly intimidating at first. However, it is well organized and logically structured. Its size reflects the detail of control and flexibility that it provides. Although it is far beyond the scope of this book to examine each element in this extensive API, we will look at several of its main classes. As you will see, these classes are sufficient for many uses. Beginning with JDK 9, these packages are in the **java.base** module.

Time and Date Fundamentals

In **java.time** are defined several top-level classes that give you easy access to the time and date. Three of these are **LocalDate**, **LocalTime**, and **LocalDateTime**. They are value-based, and as their names suggest, they encapsulate the local date, time, and date and time. Using these classes, it is easy to obtain the current date and time, format the date and time, and compare dates and times, among other operations.

LocalDate encapsulates a date that uses the default Gregorian calendar as specified by ISO 8601. **LocalTime** encapsulates a time, as specified by ISO 8601. **LocalDateTime** encapsulates both date and time. These classes contain a large number of methods that give you access to the date and time components, allow you to compare dates and times, add or subtract date or time components, and so on. Because a common naming convention for methods is employed, once you know how to use one of these classes, the others are easy to master.

LocalDate, **LocalTime**, and **LocalDateTime** do not define public

constructors. Rather, to obtain an instance, you will use a factory method. One very convenient method is **now( )**, which is defined for all three classes. It returns the current date and/or time of the system. Each class defines several versions, but we will use its simplest form. Here is the version we will use as defined by **LocalDate**:

```
static LocalDate now( )
```

The version for **LocalTime** is shown here:

```
static LocalTime now( )
```

The version for **LocalDateTime** is shown here:

```
static LocalDateTime now( )
```

As you can see, in each case, an appropriate object is returned. The object returned by **now( )** can be displayed in its default, human-readable form by use of a **println( )** statement, for example. However, it is also possible to take full control over the formatting of date and time.

The following program uses **LocalDate** and **LocalTime** to obtain the current date and time and then displays them. Notice how **now( )** is called to retrieve the current date and time.

```
// A simple example of LocalDate and LocalTime.
import java.time.*;

class DateTimeDemo {
  public static void main(String args[]) {

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate);
    LocalTime curTime = LocalTime.now();
    System.out.println(curTime);
  }
}
```

Sample output is shown here:

```
2018-06-20
17:31:15.274937600
```

The output reflects the default format that is given to the date and time. (The next section shows how to specify a different format.)

Because the preceding program displays both the current date and the current time, it could have been more easily written using the **LocalDateTime** class. In this approach, only a single instance needs to be created and only a single call to **now( )** is required, as shown here:

```
LocalDateTime curDateTime = LocalDateTime.now();
System.out.println(curDateTime);
```

Using this approach, the default output includes both date and time. Here is a sample:

```
2018-06-20T17:34:18.991974600
```

One other point: from a **LocalDateTime** instance, it is possible to obtain a reference to the date or time component by using the **toLocalDate( )** and **toLocalTime( )** methods, shown here:

LocalDate toLocalDate( )

LocalTime toLocalTime( )

Each returns a reference to the indicated element.

Formatting Date and Time

Although the default formats shown in the preceding examples will be adequate for some uses, often you will want to specify a different format. Fortunately, this is easy to do because **LocalDate**, **LocalTime**, and **LocalDateTime** all provide the **format( )** method, shown here:

String format(DateTimeFormatter *fmtr*)

Here, *fmtr* specifies the instance of **DateTimeFormatter** that will provide the format.

DateTimeFormatter is packaged in **java.time.format**. To obtain a **DateTimeFormatter** instance, you will typically use one of its factory methods. Three are shown here:

static DateTimeFormatter ofLocalizedDate(FormatStyle *fmtDate*)

static DateTimeFormatter ofLocalizedTime(FormatStyle *fmtTime*)

static DateTimeFormatter ofLocalizedDateTime(FormatStyle *fmtDate*,
FormatStyle *fmtTime*)

Of course, the type of **DateTimeFormatter** that you create will be based on the type of object it will be operating on. For example, if you want to format the date in a **LocalDate** instance, then use **ofLocalizedDate( )**. The specific format is specified by the **FormatStyle** parameter.

FormatStyle is an enumeration that is packaged in **java.time.format**. It defines the following constants:

FULL

LONG

MEDIUM

SHORT

These specify the level of detail that will be displayed. (Thus, this form of **DateTimeFormatter** works similarly to **java.text.DateFormat**, described earlier in this chapter.)

Here is an example that uses **DateTimeFormatter** to display the current date and time:

```
// Demonstrate DateTimeFormatter.
import java.time.*;
import java.time.format.*;

class DateTimeDemo2 {
  public static void main(String args[]) {

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate.format(
           DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));

    LocalTime curTime = LocalTime.now();
    System.out.println(curTime.format(
           DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
  }
}
```

Sample output is shown here:

```
Wednesday, June 20, 2018
2:16 PM
```

In some situations, you may want a format different from the ones you can specify by use of **FormatStyle**. One way to accomplish this is to use a predefined formatter, such as **ISO_DATE** or **ISO_TIME**, provided by **DateTimeFormatter**. Another way is to create a custom format by specifying a pattern. To do this, you can use the **ofPattern( )** factory method of **DateTimeFormatter**. One version is shown here:

static DateTimeFormatter ofPattern(String *fmtPattern*)

Here, *fmtPattern* specifies a string that contains the date and time pattern that you want. It returns a **DateTimeFormatter** that will format according to that pattern. The default locale is used.

In general, a pattern consists of format specifiers, called *pattern letters*. A pattern letter will be replaced by the date or time component that it specifies. The full list of pattern letters is shown in the API documentation for **ofPattern( )**. Here is a sampling. Note that the pattern letters are case-sensitive.

a	AM/PM indicator
d	Day in month
E	Day in week
h	Hour, 12-hour clock
H	Hour, 24-hour clock
M	Month
m	Minutes
s	Seconds
y	Year

In general, the precise output that you see will be determined by how many times a pattern letter is repeated. (Thus, **DateTimeFormatter** works a bit like **java.text.SimpleDateFormat**, described earlier in this chapter.) For example, assuming that the month is April, the patterns:

```
M MM MMM MMMM
```

produce the following formatted output:

```
4 04 Apr April
```

Frankly, experimentation is the best way to understand what each pattern letter does and how various repetitions affect the output.

When you want to output a pattern letter as text, enclose the text between single quotation marks. In general, it is a good idea to enclose all non-pattern characters within single quotation marks to avoid problems if the set of pattern letters changes in subsequent versions of Java.

The following program demonstrates the use of a date and time pattern:

```
// Create a custom date and time format.
import java.time.*;
import java.time.format.*;

class DateTimeDemo3 {
  public static void main(String args[]) {

    LocalDateTime curDateTime = LocalDateTime.now();
    System.out.println(curDateTime.format(
              DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Sample output is shown here:

```
June 20, 2018 2:22 PM
```

One other point about creating custom date and time output: **LocalDate**, **LocalTime**, and **LocalDateTime** define methods that let you obtain various date and time components. For example, **getHour( )** returns the hour as an **int**; **getMonth( )** returns the month in the form of a **Month** enumeration value; and **getYear( )** returns the year as an **int**. Using these, and other methods, you can manually construct output. You can also use these values for other purposes, such as when creating specialized timers.

Parsing Date and Time Strings

LocalDate, **LocalTime**, and **LocalDateTime** provide the ability to parse date and/or time strings. To do this, call **parse( )** on an instance of one of those classes. It has two forms. The first uses the default formatter that parses the date and/or time formatted in the standard ISO fashion, such as 03:31 for time and 2020-08-02 for date. The form of this version of **parse( )** for **LocalDateTime** is shown here. (Its form for the other classes is similar except for the type of object returned.)

static LocalDateTime parse(CharSequence *dateTimeStr*)

Here, *dateTimeStr* is a string that contains the date and time in the proper format. If the format is invalid, an exception will be thrown.

If you want to parse a date and/or time string that is in a format other than ISO format, you can use a second form of **parse( )** that lets you specify your

own formatter. The version specified by **LocalDateTime** is shown next. (The other classes provide a similar form except for the return type.)

```
static LocalDateTime parse(CharSequence dateTimeStr,
                           DateTimeFormatter dateTimeFmtr)
```

Here, *dateTimeFmtr* specifies the formatter that you want to use.

Here is a simple example that parses a date and time string by use of a custom formatter:

```
// Parse a date and time.
import java.time.*;
import java.time.format.*;

class DateTimeDemo4 {
  public static void main(String args[]) {

    // Obtain a LocalDateTime object by parsing a date and time string.
    LocalDateTime curDateTime =
         LocalDateTime.parse("June 21, 2018 12:01 AM",
                  DateTimeFormatter.ofPattern("MMMM d',' yyyy hh':'mm a"));

     // Now, display the parsed date and time.
    System.out.println(curDateTime.format(
              DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Sample output is shown here:

```
June 21, 2018 12:01 AM
```

Other Things to Explore in java.time

Although you will want to explore all of the date and time packages, a good place to start is with **java.time**. It contains a great deal of functionality that you may find useful. Begin by examining the methods defined by **LocalDate**, **LocalTime**, and **LocalDateTime**. Each has methods that let you add or subtract dates and/or times, adjust dates and/or times by a given component, compare dates and/or times, and create instances based on date and/or time components, among others. Other classes in **java.time** that you may find of particular interest include **Instant**, **Duration**, and **Period**. **Instant** encapsulates an instant in time.

Duration encapsulates a length of time. **Period** encapsulates a length of date.

PART

III Introducing GUI Programming with Swing

CHAPTER

31

Introducing Swing

In Part II, you saw how to build very simple user interfaces with the AWT classes. Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces. Today, programmers typically use Swing for this purpose. Swing is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade.

Coverage of Swing is divided between three chapters. This chapter introduces Swing. It begins by describing Swing's core concepts. It then presents a simple example that shows the general form of a Swing program. This is followed by an example that uses event handling. The chapter concludes by explaining how painting is accomplished in Swing. The next chapter presents several commonly used Swing components. The third chapter introduces Swing-based menus. It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this book. (In fact, full coverage of Swing requires an entire book of its own.) However, these three chapters will give you a basic understanding of this important topic.

NOTE For a comprehensive introduction to Swing, see my book *Swing: A Beginner's Guide* published by McGraw-Hill Professional (2007).

The Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on

different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing. (The AWT is covered in Chapters 25 and 26. Event handling is described in Chapter 24.)

Two Key Swing Features

As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight

With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers,

the look and feel of each component is determined by Swing, not by the underlying operating system. As a result, each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java provides look-and-feels, such as metal and Nimbus, that are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must

implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*. For this reason, Swing's approach is called either the *Model-Delegate* architecture or the *Separable Model* architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the **ButtonModel** interface. UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is **ButtonUI**. Normally, your programs will not interact directly with the UI delegate.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Let's look a bit more closely at components and containers.

Components

In general, Swing components are derived from the **JComponent** class. (The only exceptions to this are the four top-level containers, described in the next section.) **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component**. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package **javax.swing**. The following table shows the class names for Swing components (including those used as containers).

JApplet (Deprecated)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Notice that all component classes begin with the letter **J.** For example, the class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. In the past, the one used for applets was **JApplet**. As explained in Chapter 1, beginning with JDK 9 applets have been deprecated. As a result, **JApplet** is deprecated. Furthermore, beginning with JDK 11, applet support has been removed.

The second type of containers supported by Swing are lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight

containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

The Top-Level Container Panes

Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**. **JRootPane** is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the *glass pane,* the *content pane,* and the *layered pane.*

The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example. In most cases, you won't need to use the glass pane directly, but it is there if you need it.

The layered pane is an instance of **JLayeredPane**. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.) The layered pane holds the content pane and the (optional) menu bar.

Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene. The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane. By default, the content pane is an opaque instance of **JPanel**.

The Swing Packages

Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

Beginning the JDK 9, the Swing packages are part of the **java.desktop** module.

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this book. For example, they use a different set of components and a different container hierarchy than does the AWT. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. Before we begin, it is necessary to point out that in the past there were two types of Java programs in which Swing was typically used. The first is a desktop application. This type of Swing application is widely used, and is the type of Swing program described here. The second is the applet. Because applets are now deprecated and not suitable for use in new code, they are not discussed in this book.

Although quite short, the following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing. It uses two Swing components: **JFrame** and **JLabel**. **JFrame** is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
// A simple Swing application.

import javax.swing.*;

class SwingDemo {

  SwingDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("A Simple Swing Application");

    // Give the frame an initial size.
    jfrm.setSize(275, 100);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a text-based label.
    JLabel jlab = new JLabel(" Swing means powerful GUIs.");

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac SwingDemo.java
```

To run the program, use this command line:

```
java SwingDemo
```

When the program is run, it will produce a window similar to that shown in Figure 31-1.

Figure 31-1 The window produced by the **SwingDemo** program

Because the **SwingDemo** program illustrates several core Swing concepts, we will examine it carefully, line by line. The program begins by importing **javax.swing**. As mentioned, this package contains the components and models defined by Swing. For example, **javax.swing** defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.

Next, the program declares the **SwingDemo** class and a constructor for that class. The constructor is where most of the action of the program occurs. It begins by creating a **JFrame**, using this line of code:

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

Next, the window is sized using this statement:

```
jfrm.setSize(275, 100);
```

The **setSize( )** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

void setSize(int *width*, int *height*)

In this example, the width of the window is set to 275 and the height is set to 100.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated. While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation( )**, as the program does:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation( )** is shown here:

void setDefaultCloseOperation(int *what*)

The value passed in *what* determines what happens when the window is closed. There are several other options in addition to **JFrame.EXIT_ON_CLOSE**. They are shown here:

DISPOSE_ON_CLOSE

HIDE_ON_CLOSE

DO_NOTHING_ON_CLOSE

Their names reflect their actions. These constants are declared in **WindowConstants**, which is an interface declared in **javax.swing** that is implemented by **JFrame**.

The next line of code creates a Swing **JLabel** component:

```
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two. The label created by the program contains only text, which is passed to its constructor.

The next line of code adds the label to the content pane of the frame:

```
jfrm.add(jlab);
```

As explained earlier, all top-level containers have a content pane in which

components are stored. Thus, to add a component to a frame, you must add it to the frame's content pane. This is accomplished by calling **add( )** on the **JFrame** reference (**jfrm** in this case). The general form of **add( )** is shown here:

Component add(Component *comp*)

The **add( )** method is inherited by **JFrame** from the AWT class **Container.**

By default, the content pane associated with a **JFrame** uses border layout. The version of **add( )** just shown adds the label to the center location. Other versions of **add( )** enable you to specify one of the border regions. When a component is added to the center, its size is adjusted automatically to fit the size of the center.

Before continuing, an important historical point needs to be made. Prior to JDK 5, when adding a component to the content pane, you could not invoke the **add( )** method directly on a **JFrame** instance. Instead, you needed to call **add( )** on the content pane of the **JFrame** object. The content pane can be obtained by calling **getContentPane( )** on a **JFrame** instance. The **getContentPane( )** method is shown here:

Container getContentPane( )

It returns a **Container** reference to the content pane. The **add( )** method was then called on that reference to add a component to a content pane. Thus, in the past, you had to use the following statement to add **jlab** to **jfrm**:

```
jfrm.getContentPane().add(jlab); // old-style
```

Here, **getContentPane( )** first obtains a reference to content pane, and then **add( )** adds the component to the container linked to this pane. This same procedure was also required to invoke **remove( )** to remove a component and **setLayout( )** to set the layout manager for the content pane. This is why you will see explicit calls to **getContentPane( )** frequently throughout pre-5.0 legacy code. Today, the use of **getContentPane( )** is no longer necessary. You can simply call **add( )**, **remove( )**, and **setLayout( )** directly on **JFrame** because these methods have been changed so that they operate on the content pane automatically.

The last statement in the **SwingDemo** constructor causes the window to become visible:

```
jfrm.setVisible(true);
```

The **setVisible( )** method is inherited from the AWT **Component** class. If its argument is **true**, the window will be displayed. Otherwise, it will be hidden. By default, a **JFrame** is invisible, so **setVisible(true)** must be called to show it.

Inside **main( )**, a **SwingDemo** object is created, which causes the window and the label to be displayed. Notice that the **SwingDemo** constructor is invoked using these lines of code:

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    new SwingDemo();
  }
});
```

This sequence causes a **SwingDemo** object to be created on the *event dispatching thread* rather than on the main thread of the application. Here's why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application. Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.

To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application. However, **main( )** is executed on the main thread. Thus, **main( )** cannot directly instantiate a **SwingDemo** object. Instead, it must create a **Runnable** object that executes on the event dispatching thread and have this object create the GUI.

To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the **SwingUtilities** class. These methods are **invokeLater( )** and **invokeAndWait( )**. They are shown here:

```
static void invokeLater(Runnable obj)

static void invokeAndWait(Runnable obj)
  throws InterruptedException, InvocationTargetException
```

Here, *obj* is a **Runnable** object that will have its **run( )** method called by the event dispatching thread. The difference between the two methods is that **invokeLater( )** returns immediately, but **invokeAndWait( )** waits until **obj.run( )** returns. You can use one of these methods to call a method that constructs the

GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread. You will normally want to use **invokeLater( )**, as the preceding program does. However, when the initial GUI for an applet is constructed, **invokeAndWait( )** is required. Thus, you will see its use in legacy applet code.

Event Handling

The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because **JLabel** does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components *do* respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the *delegation event model*, and it is described in Chapter 24. In many cases, Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event**. Events specific to Swing are stored in **javax.swing.event**.

Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in Figure 31-2.

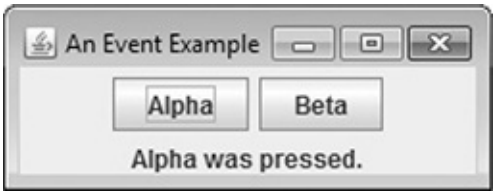


Figure 31-2 Output from the **EventDemo** program

```
// Handle an event in a Swing program.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {

  JLabel jlab;

  EventDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("An Event Example");

    // Specify FlowLayout for the layout manager.
    jfrm.setLayout(new FlowLayout());

    // Give the frame an initial size.
    jfrm.setSize(220, 90);
```

```
    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Make two buttons.
    JButton jbtnAlpha = new JButton("Alpha");
    JButton jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
      }
    });

    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
      }
    });

    // Add the buttons to the content pane.
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);

    // Create a text-based label.
    jlab = new JLabel("Press a button.");

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new EventDemo();
      }
    });
  }
}
```

First, notice that the program now imports both the **java.awt** and **java.awt.event** packages. The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame. (See Chapter 26 for coverage of layout managers.) The **java.awt.event** package is needed because it defines the **ActionListener** interface and the **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**. By default, the content pane uses **BorderLayout** as its layout manager. However, for this example, **FlowLayout** is more convenient.

After setting the size and default close operation, **EventDemo( )** creates two push buttons, as shown here:

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

The first button will contain the text "Alpha" and the second will contain the text "Beta". Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

JButton(String *msg*)

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent.** Thus, **JButton** provides the **addActionListener( )** method, which is used to add an action listener. (**JButton** also provides **removeActionListener( )** to remove a listener, but this method is not used by the program.) As explained in Chapter 24, the **ActionListener** interface defines only one method: **actionPerformed( )**. It is shown again here for your convenience:

void actionPerformed(ActionEvent *ae*)

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button’s action events are added by the code shown here:

```
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Alpha was pressed.");
  }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta was pressed.");
  }
});
```

Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in **jlab** is changed to reflect which button was pressed.

Beginning with JDK 8, lambda expressions can also be used to implement some types of event handlers. For example, the event handler for the Alpha button could be written like this:

```
jbtnAlpha.addActionListener( (ae) -> jlab.setText("Alpha was
pressed."));
```

As you can see, this code is shorter. Of course, the approach you choose will be determined by the situation and your own preferences.

Next, the buttons are added to the content pane of **jfrm**:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Finally, **jlab** is added to the content pane and the window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.

One last point: Remember that all event handlers, such as **actionPerformed( )**, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

Painting in Swing

Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing's other components, such as **JLabel.** Although many (perhaps most) uses of Swing will *not* involve drawing directly to the surface of a component, it is available for those applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as **drawLine( )** or **drawRect( )**. Thus, most of the techniques and methods described in Chapter 25 also apply to Swing. However, there are also some very important differences, and the process is discussed in detail in this section.

Painting Fundamentals

Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finally grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class **Component** defines a method called **paint( )** that is used to draw output directly to the surface of a component. For the most part, **paint( )** is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, **paint( )** is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The **paint( )** method is also called when a program begins running. When writing AWT-based code, an application will override **paint( )** when it needs to write output directly to the surface of the component.

Because **JComponent** inherits **Component**, all Swing's lightweight components inherit the **paint( )** method. However, you *will not* override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: **paintComponent( )**, **paintBorder( )**, and **paintChildren( )**. These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method **paint( )** simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of the component and then override its **paintComponent( )** method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding **paintComponent( )**, the first thing you must do is call **super.paintComponent( )**, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The **paintComponent( )** method is shown here:

protected void paintComponent(Graphics *g*)

The parameter *g* is the graphics context to which output is written.

To cause a component to be painted under program control, call **repaint( )**. It works in Swing just as it does for the AWT. The **repaint( )** method is defined by **Component**. Calling it causes the system to call **paint( )** as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to **paint( )** results in a call to **paintComponent( )**. Therefore, to output to the surface of a component, your program will store the output until **paintComponent( )** is called. Inside the overridden **paintComponent( )**, you will draw the stored output.

Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the *paintable area* of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly.

To obtain the border width, call **getInsets( )**, shown here:

Insets getInsets( )

This method is defined by **Container** and overridden by **JComponent**. It returns

an **Insets** object that contains the dimensions of the border. The inset values can be obtained by using these fields:

```
int top;

int bottom;

int left;

int right;
```

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling **getWidth( )** and **getHeight( )** on the component. They are shown here:

```
int getWidth( )

int getHeight( )
```

By subtracting the value of the insets, you can compute the usable width and height of the component.

A Paint Example

Here is a program that puts into action the preceding discussion. It creates a class called **PaintPanel** that extends **JPanel**. The program then uses an object of that class to display lines whose endpoints have been generated randomly. Sample output is shown in Figure 31-3.

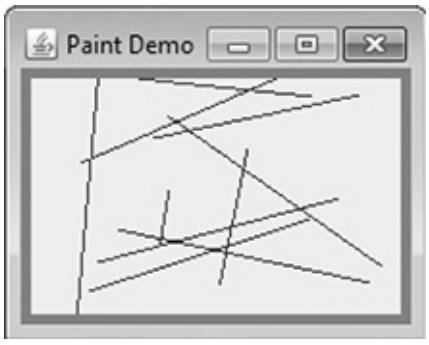


Figure 31-3 Sample output from the **PaintPanel** program

```
// Paint lines to a panel.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// This class extends JPanel. It overrides
// the paintComponent() method so that random
// lines are plotted in the panel.
class PaintPanel extends JPanel {
  Insets ins; // holds the panel's insets

  Random rand; // used to generate random numbers

  // Construct a panel.
  PaintPanel() {

    // Put a border around the panel.
    setBorder(
      BorderFactory.createLineBorder(Color.RED, 5));

    rand = new Random();
  }
```

```
  // Override the paintComponent() method.
  protected void paintComponent(Graphics g) {
    // Always call the superclass method first.
    super.paintComponent(g);

    int x, y, x2, y2;

    // Get the height and width of the component.
    int height = getHeight();
    int width = getWidth();

    // Get the insets.
    ins = getInsets();

    // Draw ten lines whose endpoints are randomly generated.
    for(int i=0; i < 10; i++) {
      // Obtain random coordinates that define
      // the endpoints of each line.
      x = rand.nextInt(width-ins.left);
      y = rand.nextInt(height-ins.bottom);
      x2 = rand.nextInt(width-ins.left);
      y2 = rand.nextInt(height-ins.bottom);

      // Draw the line.
      g.drawLine(x, y, x2, y2);
    }
  }
}

// Demonstrate painting directly onto a panel.
class PaintDemo {

   JLabel jlab;
   PaintPanel pp;

   PaintDemo() {

     // Create a new JFrame container.
     JFrame jfrm = new JFrame("Paint Demo");

     // Give the frame an initial size.
     jfrm.setSize(200, 150);

     // Terminate the program when the user closes the application.
     jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     // Create the panel that will be painted.
     pp = new PaintPanel();

     // Add the panel to the content pane. Because the default
     // border layout is used, the panel will automatically be
     // sized to fit the center region.
     jfrm.add(pp);
```

```
    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new PaintDemo();
      }
    });
  }
}
```

Let's examine this program closely. The **PaintPanel** class extends **JPanel**. **JPanel** is one of Swing's lightweight containers, which means that it is a component that can be added to the content pane of a **JFrame**. To handle painting, **PaintPanel** overrides the **paintComponent( )** method. This enables **PaintPanel** to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly.

Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the **setBorder( )** method, shown here:

void setBorder(Border *border*)

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the **BorderFactory** class. The one used in the program is **createLineBorder( )**, which creates a simple line border. It is shown here:

static Border createLineBorder(Color *clr*, int *width*)

Here, *clr* specifies the color of the border and *width* specifies its width in pixels.

Inside the override of **paintComponent( )**, notice that it first calls **super.paintComponent( )**. As explained, this is necessary to ensure that the component is properly drawn. Next, the width and height of the panel are

obtained along with the insets. These values are used to ensure the lines lie within the drawing area of the panel. The drawing area is the overall width and height of a component less the border width. The computations are designed to work with differently sized **PaintPanel**s and borders. To prove this, try changing the size of the window. The lines will still all lie within the borders of the panel.

The **PaintDemo** class creates a **PaintPanel** and then adds the panel to the content pane. When the application is first displayed, the overridden **paintComponent( )** method is called, and the lines are drawn. Each time you resize or hide and restore the window, a new set of lines are drawn. In all cases, the lines fall within the paintable area.

CHAPTER

Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of a Swing application. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, which means that they are all derived from **JComponent**.

Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image. Both are defined by Swing and packaged in **javax.swing**.

JLabel and ImageIcon

JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at **JLabel** a bit more closely. **JLabel** can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

JLabel(Icon *icon*)
JLabel(String *str*)
JLabel(String *str*, Icon *icon*, int *align*)

Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT, CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

ImageIcon(String *filename*)

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

Icon getIcon( )
String getText( )

The icon and text associated with a label can be set by these methods:

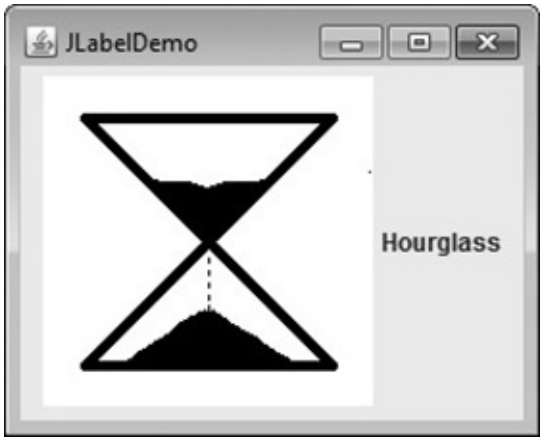
void setIcon(Icon *icon*)
void setText(String *str*)

Here, *icon* and *str* are the icon and text, respectively. Therefore, using **setText( )** it is possible to change the text inside a label during program execution.

The following program illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **hourglass.png**, which depicts an hourglass. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;

public class JLabelDemo {

   public JLabelDemo() {

     // Set up the JFrame.
     JFrame jfrm = new JFrame("JLabelDemo");
     jfrm.setLayout(new FlowLayout());
     jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     jfrm.setSize(260, 210);

     // Create an icon.
     ImageIcon ii = new ImageIcon("hourglass.png");

     // Create a label.
     JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

     // Add the label to the content pane.
     jfrm.add(jl);

     // Display the frame.
     jfrm.setVisible(true);
   }

  public static void main(String[] args) {
     // Create the frame on the event dispatching thread.

     SwingUtilities.invokeLater(
       new Runnable() {
         public void run() {
           new JLabelDemo();
         }
       }
     );

   }
}
```

Output from the label example is shown here:



JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model. Three of **JTextField**'s constructors are shown here:

JTextField(int *cols*)
JTextField(String *str*, int *cols*)
JTextField(String *str*)

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

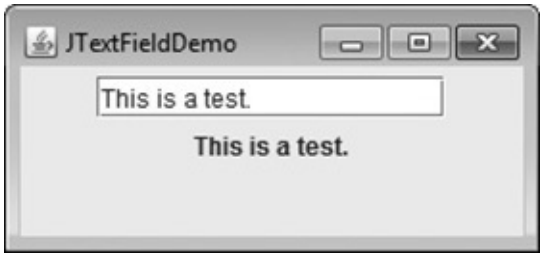
JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses enter. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call **getText( )**.

The following example illustrates **JTextField**. It creates a **JTextField** and adds it to the content pane. When the user presses enter, an action event is

generated. This is handled by displaying the text in a label.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JTextFieldDemo {

  public JTextFieldDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JTextFieldDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(260, 120);

    // Add a text field to content pane.
    JTextField jtf = new JTextField(15);
    jfrm.add(jtf);

    // Add a label.
    JLabel jlab = new JLabel();
    jfrm.add(jlab);

    // Handle action events.
    jtf.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        // Show text when user presses ENTER.
        jlab.setText(jtf.getText());
      }
    });

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTextFieldDemo();
        }
      }
    );

  }
}
```

Output from the text field example is shown here:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

void setDisabledIcon(Icon *di*)
void setPressedIcon(Icon *pi*)
void setSelectedIcon(Icon *si*)
void setRolloverIcon(Icon *ri*)

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

String getText( )
void setText(String *str*)

Here, *str* is the text to be associated with the button.

The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

JButton

The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

JButton(Icon *icon*)
JButton(String *str*)
JButton(String *str*, Icon *icon*)

Here, *str* and *icon* are the string and icon used for the button.

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand( )** on the button. You can obtain the action command by calling **getActionCommand( )** on the event object. It is declared like this:

String getActionCommand( )

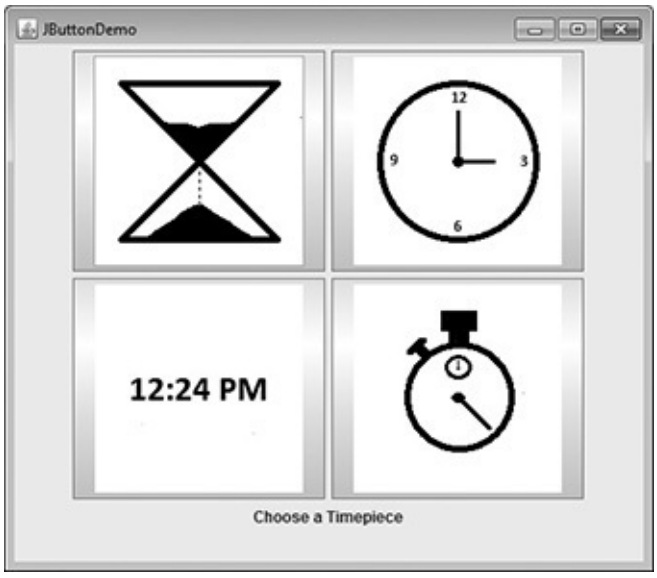
The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JButtonDemo implements ActionListener {
  JLabel jlab;

  public JButtonDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(500, 450);

    // Add buttons to content pane.
    ImageIcon hourglass = new ImageIcon("hourglass.png");
    JButton jb = new JButton(hourglass);
    jb.setActionCommand("Hourglass");
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon analog = new ImageIcon("analog.png");
    jb = new JButton(analog);
    jb.setActionCommand("Analog Clock");
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon digital = new ImageIcon("digital.png");
    jb = new JButton(digital);
    jb.setActionCommand("Digital Clock");
    jb.addActionListener(this);
    jfrm.add(jb);

    ImageIcon stopwatch = new ImageIcon("stopwatch.png");
    jb = new JButton(stopwatch);
    jb.setActionCommand("Stopwatch");
    jb.addActionListener(this);
    jfrm.add(jb);
```

```
    // Create and add the label to content pane.
    jlab = new JLabel("Choose a Timepiece");
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle button events.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JButtonDemo();
        }
      }
    );

  }
}
```

Output from the button example is shown here:



JToggleButton

A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**, which are described later in this chapter. Thus, **JToggleButton** defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String *str*)

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called **JToggleButton.Toggle-ButtonModel**. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the **ItemListener** interface. Recall from Chapter 24, that each time an item event is generated, it is passed to the **itemStateChanged( )** method defined by **ItemListener**. Inside **itemStateChanged( )**, the **getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

```
Object getItem( )
```

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected( )** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

```
boolean isSelected( )
```

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected( )** to determine the button's state.

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JToggleButtonDemo {

  public JToggleButtonDemo() {
```

```
    // Set up the JFrame.
    JFrame jfrm = new JFrame("JToggleButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 100);

    // Create a label.
    JLabel jlab = new JLabel("Button is off.");

    // Make a toggle button.
    JToggleButton jtbn =  new JToggleButton("On/Off");

    // Add an item listener for the toggle button.
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("Button is on.");
        else
          jlab.setText("Button is off.");
      }
    });

    // Add the toggle button and label to the content pane.
    jfrm.add(jtbn);
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JToggleButtonDemo();
        }
      }
    );

  }
}
```

The output from the toggle button example is shown here:

Check Boxes

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

JCheckBox(String *str*)

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected( )** determines if the box was selected or cleared. The **getText( )** method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JCheckBoxDemo implements ItemListener {
  JLabel jlab;

  public JCheckBoxDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JCheckBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);

    // Add check boxes to the content pane.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    jfrm.add(cb);
```

```
    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    jfrm.add(cb);

    // Create the label and add it to the content pane.
    jlab = new JLabel("Select languages");
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle item events for the check boxes.
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText(cb.getText() + " is selected");
    else
      jlab.setText(cb.getText() + " is cleared");
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JCheckBoxDemo();
        }
      }
    );

  }
}
```

Output from this example is shown here:

Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends **JToggleButton**. **JRadioButton** provides several constructors. The one used in the example is shown here:

JRadioButton(String *str*)

Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface. Recall that the only method defined by **ActionListener** is **actionPerformed( )**. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand( )**. By default, the action command is the same as the button label, but you can set the action command to something else by calling **setActionCommand( )** on the radio button. Second, you can call **getSource( )** on the **ActionEvent** object and check that reference against the buttons. Third, you can check each radio button to find out which one is

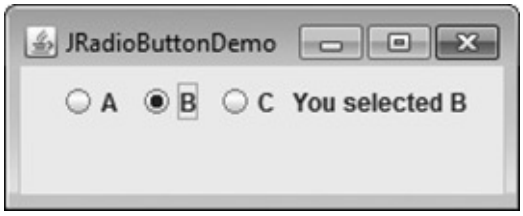
currently selected by calling **isSelected( )** on each button. Finally, each button could use its own action event handler implemented as either an anonymous inner class or a lambda expression. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by **actionPerformed( )**. Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo implements ActionListener {
  JLabel jlab;

  public JRadioButtonDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JRadioButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);
```

```
    // Create radio buttons and add them to content pane.
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    jfrm.add(b1);

    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    jfrm.add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    jfrm.add(b3);

    // Define a button group.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Create a label and add it to the content pane.
    jlab = new JLabel("Select One");
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle button selection.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JRadioButtonDemo();
        }
      }
    );

  }
}
```

Output from the radio button example is shown here:



JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. **JTabbedPane** uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab( )**. Here is one of its forms:

void addTab(String *name*, Component *comp*)

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

1. Create an instance of **JTabbedPane**.
2. Add eachn tab by calling **addTab( )**.
3. Add the tabbed pane to the content pane.

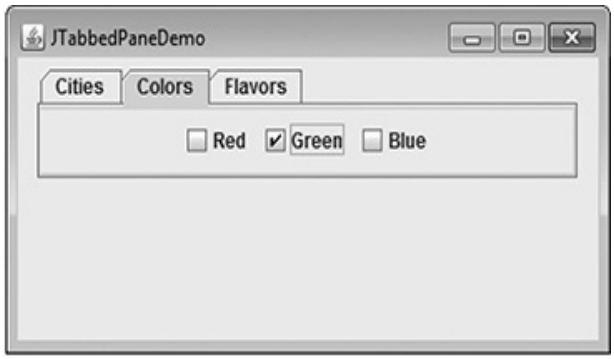
The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one combo

box. This enables the user to select one of three flavors.

```
// Demonstrate JTabbedPane.
import javax.swing.*;
import java.awt.*;

public class JTabbedPaneDemo {

  public JTabbedPaneDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JTabbedPaneDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 200);

    // Create the tabbed pane.
    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    jfrm.add(jtp);
```

```
        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Create the frame on the event dispatching thread.

        SwingUtilities.invokeLater(
          new Runnable() {
            public void run() {
              new JTabbedPaneDemo();
            }
          }
        );

    }
}

// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {

    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}

class ColorsPanel extends JPanel {

    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}

class FlavorsPanel extends JPanel {

    public FlavorsPanel() {
        JComboBox<String> jcb = new JComboBox<String>();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");

        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```

Output from the tabbed pane example is shown in the following three illustrations:



JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

JScrollPane(Component *comp*)

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

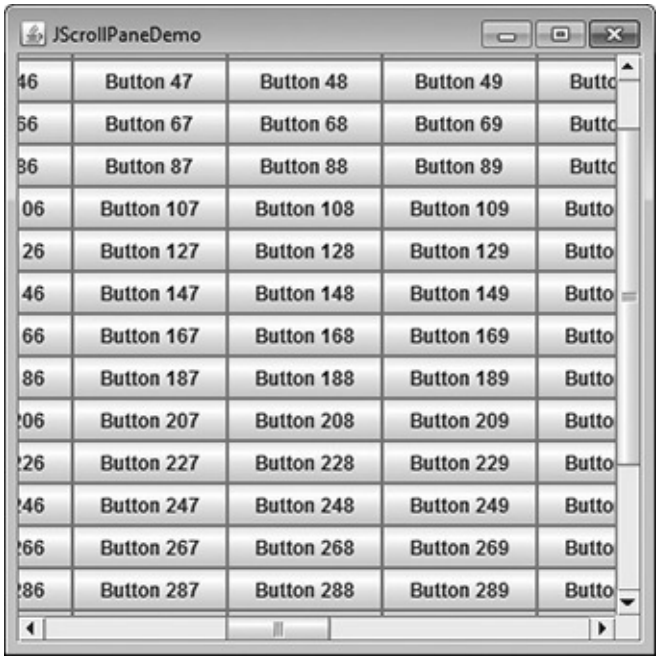
1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;

public class JScrollPaneDemo {

  public JScrollPaneDemo() {

    // Set up the JFrame.  Use the default BorderLayot.
    JFrame jfrm = new JFrame("JScrollPaneDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 400);

    // Create a panel and add 400 buttons to it.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));

    int b = 0;
    for(int i = 0; i < 20; i++) {
      for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
      }
    }

    // Create the scroll pane.
    JScrollPane jsp = new JScrollPane(jp);

    // Add the scroll pane to the content pane.
    // Because the default border layout is used,
    // the scroll pane will be added to the center.
    jfrm.add(jsp, BorderLayout.CENTER);

    // Display the frame.
    jfrm.setVisible(true);
  }
```

```
  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JScrollPaneDemo();
        }
      }
    );

  }
}
```

Output from the scroll pane example is shown here:



JList

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.

In the past, the items in a **JList** were represented as **Object** references.

However, beginning with JDK 7, **JList** was made generic and is now declared like this:

class JList<E>

Here, **E** represents the type of the items in the list.

JList provides several constructors. The one used here is

JList(E[ ] *items*)

This creates a **JList** that contains the items in the array specified by *items*.

JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged( )**, which is shown here:

void valueChanged(ListSelectionEvent *le*)

Here, *le* is a reference to the event. Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:

```
int getSelectedIndex( )
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue( )**:

```
E getSelectedValue( )
```

It returns a reference to the first selected value. If no value has been selected, it returns **null**.

The following program demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged( )** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class JListDemo {

  // Create an array of cities.
  String Cities[] = { "New York", "Chicago", "Houston",
                      "Denver", "Los Angeles", "Seattle",
                      "London", "Paris", "New Delhi",
                      "Hong Kong", "Tokyo", "Sydney" };

  public JListDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JListDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 200);

    // Create a JList.
    JList<String> jlst = new JList<String>(Cities);

    // Set the list selection mode to single-selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add the list to a scroll pane.
    JScrollPane jscrlp = new JScrollPane(jlst);

    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Make a label that displays the selection.
    JLabel jlab = new JLabel("Choose a City");

    // Add selection listener for the list.
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Get the index of the changed item.
        int idx = jlst.getSelectedIndex();

        // Display selection, if item was selected.
        if(idx != -1)
          jlab.setText("Current selection: " + Cities[idx]);
```

```
        else // Otherwise, reprompt.
          jlab.setText("Choose a City");
      }
    });

    // Add the list and label to the content pane.
    jfrm.add(jscrlp);
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JListDemo();
        }
      }
    );

  }
}
```

Output from the list example is shown here:

JComboBox

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.

In the past, the items in a **JComboBox** were represented as **Object** references. However, beginning with JDK 7, **JComboBox** was made generic and is now declared like this:

class JComboBox<E>

Here, **E** represents the type of the items in the combo box.

The **JComboBox** constructor used by the example is shown here:

JComboBox(E[ ] *items*)

Here, *items* is an array that initializes the combo box. Other constructors are available.

JComboBox uses the **ComboBoxModel**. Mutable combo boxes (those whose entries can be changed) use the **MutableComboBoxModel**.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem( )** method, shown here:

void addItem(E *obj*)

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call **getSelectedItem( )** on the combo box. It is shown here:

Object getSelectedItem( )

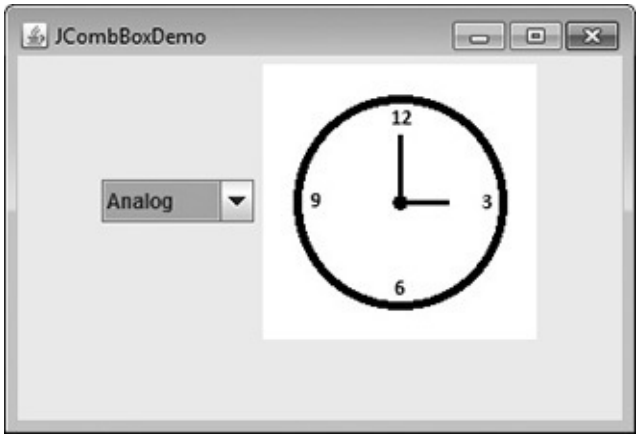
You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerful component.

```
// Demonstrate JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JComboBoxDemo {

  String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };

  public JComboBoxDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JCombBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 250);

    // Instantiate a combo box and add it to the content pane.
    JComboBox<String> jcb = new JComboBox<String>(timepieces);
    jfrm.add(jcb);
```

```
    // Create a label and add it to the content pane.
    JLabel jlab = new JLabel(new ImageIcon("hourglass.png"));
    jfrm.add(jlab);

    // Handle selections.
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        String s = (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".png"));
      }
    });

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JComboBoxDemo();
        }
      }
    );

  }
}
```

Output from the combo box example is shown here:



Trees

A *tree* is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

JTree(Object *obj* [ ])
JTree(Vector<?> *v*)
JTree(TreeNode *tn*)

In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.

Although **JTree** is packaged in **javax.swing**, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support **JTree** is quite large.

JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A **JTree** generates a variety of events, but three relate specifically to trees: **TreeExpansionEvent**, **TreeSelectionEvent**, and **TreeModelEvent**. **TreeExpansionEvent** events occur when a node is expanded or collapsed. A **TreeSelectionEvent** is generated when the user selects or deselects a node within the tree. A **TreeModelEvent** is fired when the data or structure of the tree changes. The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively. The tree event classes and listener interfaces are packaged in **javax.swing.event**.

The event handled by the sample program shown in this section is

TreeSelectionEvent. To listen for this event, implement **TreeSelectionListener**. It defines only one method, called **valueChanged( )**, which receives the **TreeSelectionEvent** object. You can obtain the path to the selected object by calling **getPath( )**, shown here, on the event object:

```
TreePath getPath( )
```

It returns a **TreePath** object that describes the path to the changed node. The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString( )** method is used. It returns a string that describes the path.

The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add( )** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, *child* is a mutable tree node that is to be added as a child to the current node.

JTree does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

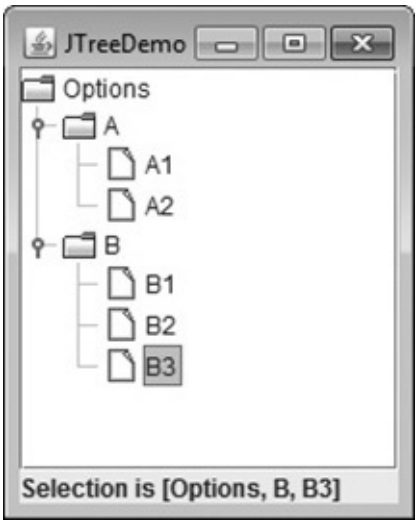
1. Create an instance of **JTree**.
2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled "Options". This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add( )** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree. Inside the **valueChanged( )** method, the path to the current selection is obtained and displayed.

```
// Demonstrate JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class JTreeDemo {

  public JTreeDemo() {

    // Set up the JFrame. Use default BorderLayout.
    JFrame jfrm = new JFrame("JTreeDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 250);

    // Create top node of tree.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

    // Create subtree of "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);

    // Create subtree of "B".
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Create the tree.
    JTree tree = new JTree(top);
```

```
    // Add the tree to a scroll pane.
    JScrollPane jsp = new JScrollPane(tree);

    // Add the scroll pane to the content pane.
    jfrm.add(jsp);

    // Add the label to the content pane.
    JLabel jlab = new JLabel();
    jfrm.add(jlab, BorderLayout.SOUTH);

    // Handle tree selection events.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
      public void valueChanged(TreeSelectionEvent tse) {
        jlab.setText("Selection is " + tse.getPath());
      }
    });

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTreeDemo();
        }
      }
    );

  }
}
```

Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. **JTable** is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, **JTable** still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like **JTree**, **JTable** has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors. The one used here is

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A **JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple **JTable** that can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

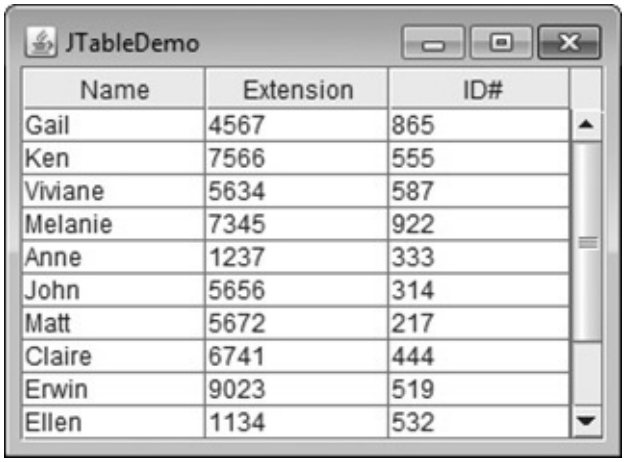
The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data**

in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;

public class JTableDemo {

  // Initialize column headings.
  String[] colHeads = { "Name", "Extension", "ID#" };

  // Initialize data.
  Object[][] data = {
    { "Gail", "4567", "865" },
    { "Ken", "7566", "555" },
    { "Viviane", "5634", "587" },
    { "Melanie", "7345", "922" },
    { "Anne", "1237", "333" },
    { "John", "5656", "314" },
    { "Matt", "5672", "217" },
    { "Claire", "6741", "444" },
    { "Erwin", "9023", "519" },
    { "Ellen", "1134", "532" },
    { "Jennifer", "5689", "112" },
    { "Ed", "9030", "133" },
    { "Helen", "6751", "145" }
  };

  public JTableDemo() {

    // Set up the JFrame. Use default BorderLayout.
    JFrame jfrm = new JFrame("JTableDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(300, 300);

    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);
```

```
    // Add the scroll pane to the content pane.
    jfrm.add(jsp);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JTableDemo();
        }
      }
    );

  }
}
```

Output from this example is shown here:



CHAPTER

Introducing Swing Menus

This chapter introduces another fundamental aspect of the Swing GUI environment: the menu. Menus form an integral part of many applications because they present the program's functionality to the user. Because of their importance, Swing provides extensive support for menus. They are an area in which Swing's power is readily apparent.

The Swing menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The popup menu, which is usually activated by right-clicking the mouse.
- The toolbar, which provides rapid access to program functionality, often paralleling menu items.
- The action, which enables two or more different components to be managed by a single object. Actions are commonly used with menus and toolbars.

Swing menus also support accelerator keys, which enable menu items to be selected without having to activate the menu, and mnemonics, which allow a menu item to be selected by the keyboard once the menu options are displayed.

Menu Basics

The Swing menu system is supported by a group of related classes. The ones used in this chapter are shown in Table 33-1, and they represent the core of the menu system. Although they may seem a bit confusing at first, Swing menus are quite easy to use. Swing allows a high degree of customization, if desired; however, you will normally use the menu classes as-is because they support all of the most needed options. For example, you can easily add images and keyboard shortcuts to a menu.

Class	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more **JMenuItem**s.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item
JSeparator	The visual separator between menu items.
JPopupMenu	A menu that is typically activated by right-clicking the mouse.

Table 33-1 The Core Swing Menu Classes

Here is a brief overview of how the classes fit together. To create the top-level menu for an application, you first create a **JMenuBar** object. This class is, loosely speaking, a container for menus. To the **JMenuBar** instance, you will add instances of **JMenu**. Each **JMenu** object defines a menu. That is, each **JMenu** object contains one or more selectable items. The items displayed by a **JMenu** are objects of **JMenuItem**. Thus, a **JMenuItem** defines a selection that can be chosen by the user.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, popup menus. To create a popup menu, first create an object of type **JPopupMenu**. Then, add **JMenuItem**s to it. A popup menu is normally activated by clicking the right mouse button when the mouse is over a component for which a popup menu has been defined.

In addition to "standard" menu items, you can also include check boxes and radio buttons in a menu. A check box menu item is created by **JCheckBoxMenuItem**. A radio button menu item is created by **JRadioButtonMenuItem**. Both of these classes extend **JMenuItem**. They can be used in standard menus and popup menus.

JToolBar creates a stand-alone component that is related to the menu. It is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

JSeparator is a convenience class that creates a separator line in a menu.

One key point to understand about Swing menus is that each menu item extends **AbstractButton**. Recall that **AbstractButton** is also the superclass of all of Swing's button components, such as **JButton**. Thus, all menu items are, essentially, buttons. Obviously, they won't actually look like buttons when used

in a menu, but they will, in many ways, act like buttons. For example, selecting a menu item generates an action event in the same way that pressing a button does.

Another key point is that **JMenuItem** is a superclass of **JMenu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **JMenu** object and then add it to another **JMenu** object. You will see this process in action in the following section.

As mentioned in passing previously, when a menu item is selected, an action event is generated. The action command string associated with that action event will, by default, be the name of the selection. Thus, you can determine which item was selected by examining the action command. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item's action events. In this case, the menu selection is already known, and there is no need to examine the action command string to determine which item was selected.

Menus can also generate other types of events. For example, each time that a menu is activated, selected, or canceled, a **MenuEvent** is generated that can be listened for via a **MenuListener**. Other menu-related events include **MenuKeyEvent**, **MenuDragMouseEvent**, and **PopupMenuEvent**. In many cases, however, you need only watch for action events, and in this chapter, we will use only action events.

An Overview of JMenuBar, JMenu, and JMenuItem

Before you can create a menu, you need to know something about the three core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **JMenu** and **JMenuItem** are also used by popup menus. Thus, these classes form the foundation of the menu system.

JMenuBar

As mentioned, **JMenuBar** is essentially a container for menus. Like all components, it inherits **JComponent** (which inherits **Container** and **Component**). It has only one constructor, which is the default constructor. Therefore, initially the menu bar will be empty, and you will need to populate it

with menus prior to use. Each application has one and only one menu bar.

JMenuBar defines several methods, but often you will only need to use one: **add( )**. The **add( )** method adds a **JMenu** to the menu bar. It is shown here:

JMenu add(JMenu *menu*)

Here, *menu* is a **JMenu** instance that is added to the menu bar. A reference to the menu is returned. Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add( )**, which is inherited from **Container**:

Component add(Component *menu*, int *idx*)

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove( )**, which is inherited from **Container**. It has these two forms:

void remove(Component *menu*)

void remove(int *idx*)

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

Another method that is sometimes useful is **getMenuCount( )**, shown here:

int getMenuCount( )

It returns the number of elements contained within the menu bar.

JMenuBar defines some other methods that you might find helpful in specialized applications. For example, you can obtain an array of references to the menus in the bar by calling **getSubElements( )**. You can determine if a menu is selected by calling **isSelected( )**.

Once a menu bar has been created and populated, it is added to a **JFrame** by calling **setJMenuBar( )** on the **JFrame** instance. (Menu bars *are not* added to the content pane.) The **setJMenuBar( )** method is shown here:

void setJMenuBar(JMenuBar *mb*)

Here, *mb* is a reference to the menu bar. The menu bar will be displayed in a position determined by the look and feel. Usually, this is at the top of the window.

JMenu

JMenu encapsulates a menu, which is populated with **JMenuItem**s. As mentioned, it is derived from **JMenuItem**. This means that one **JMenu** can be a selection in another **JMenu**. This enables one menu to be a submenu of another. **JMenu** defines a number of constructors. For example, here is the one used in the examples in this chapter:

JMenu(String *name*)

This constructor creates a menu that has the title specified by *name*. Of course, you don't have to give a menu a name. To create an unnamed menu, you can use the default constructor:

JMenu( )

Other constructors are also supported. In each case, the menu is empty until menu items are added to it.

JMenu defines many methods. Here is a brief description of some commonly used ones. To add an item to the menu, use the **add( )** method, which has a number of forms, including the two shown here:

JMenuItem add(JMenuItem *item*)

Component add(Component *item*, int *idx*)

Here, *item* is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by *idx*. As expected, indexing starts at zero. Both forms return a reference to the item added. As a point of interest, you can also use **insert( )** to add menu items to a menu.

You can add a separator (an object of type **JSeparator**) to a menu by calling **addSeparator( )**, shown here:

void addSeparator( )

The separator is added onto the end of the menu. You can insert a separator into

a menu by calling **insertSeparator( )**, shown next:

void insertSeparator(int *idx*)

Here, *idx* specifies the zero-based index at which the separator will be added.

You can remove an item from a menu by calling **remove( )**. Two of its forms are shown here:

void remove(JMenuItem *menu*)

void remove(int *idx*)

In this case, *menu* is a reference to the item to remove and *idx* is the index of the item to remove.

You can obtain the number of items in the menu by calling **getMenuComponentCount( )**, shown here:

int getMenuComponentCount( )

You can get an array of the items in the menu by calling **getMenuComponents( )**, shown next:

Component[ ] getMenuComponents( )

An array containing the components is returned.

JMenuItem

JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. As mentioned, **JMenuItem** is derived from **AbstractButton**, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated. (This is similar to the way a **JButton** fires an action event when it is pressed.)
JMenuItem defines many constructors. The ones used in this chapter are shown here:

JMenuItem(String *name*)

JMenuItem(Icon *image*)

JMenuItem(String *name*, Icon *image*)

JMenuItem(String *name*, int *mnem*)

JMenuItem(Action *action*)

The first constructor creates a menu item with the name specified by *name*. The second creates a menu item that displays the image specified by *image*. The third creates a menu item with the name specified by *name* and the image specified by *image*. The fourth creates a menu item with the name specified by *name* and uses the keyboard mnemonic specified by *mnem*. This mnemonic enables you to select an item from the menu by pressing the specified key. The last constructor creates a menu item using the information specified in *action*. A default constructor is also supported.

Because menu items inherit **AbstractButton**, you have access to the functionality provided by **AbstractButton**. One such method that is often useful with menus is **setEnabled( )**, which you can use to enable or disable a menu item. It is shown here:

void setEnabled(boolean *enable*)

If *enable* is **true**, the menu item is enabled. If *enable* is **false**, the item is disabled and cannot be selected.

Create a Main Menu

Traditionally, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. Fortunately, Swing makes creating and managing the main menu easy. This section shows you how to construct a basic main menu. Subsequent sections will show you how to add options to it.

Constructing the main menu requires several steps. First, create the **JMenuBar** object that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **JMenu** object and then adding **JMenuItem**s to it. After the menus have been created, add them to the menu bar. The menu bar, itself, must then be added to the frame by calling **setJMenuBar( )**. Finally, for each menu item, you must add an action listener that handles the action event fired when the menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label in the content pane. Sample output is shown in Figure 33-1.

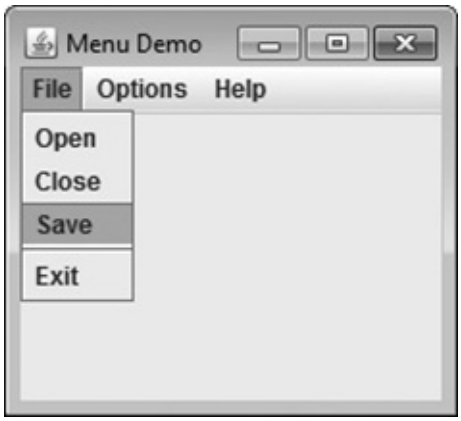


Figure 33-1 Sample output from the **MenuDemo** program

```
// Demonstrate a simple main menu.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

  JLabel jlab;

  MenuDemo() {
    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Menu Demo");
```

```
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());

// Give the frame an initial size.
jfrm.setSize(220, 200);

// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Create a label that will display the menu selection.
jlab = new JLabel();

// Create the menu bar.
JMenuBar jmb = new JMenuBar();

// Create the File menu.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Create the Options menu.
JMenu jmOptions = new JMenu("Options");

// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
JMenuItem jmiRed = new JMenuItem("Red");
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);
```

```
    // Finally, add the entire options menu to
    // the menu bar
    jmb.add(jmOptions);

    // Create the Help menu.
    JMenu jmHelp = new JMenu("Help");
    JMenuItem jmiAbout = new JMenuItem("About");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);

    // Add action listeners for the menu items.
    jmiOpen.addActionListener(this);
    jmiClose.addActionListener(this);
    jmiSave.addActionListener(this);
    jmiExit.addActionListener(this);
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
    jmiAbout.addActionListener(this);

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Add the menu bar to the frame.
    jfrm.setJMenuBar(jmb);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle menu item action events.
  public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```

Let’s examine, in detail, how the menus in this program are created, beginning with the **MenuDemo** constructor. It starts by creating a **JFrame** and setting its layout manager, size, and default close operation. (These operations are described in Chapter 31.) A **JLabel** is then constructed. It will be used to display a menu selection. Next, the menu bar is constructed and a reference to it is assigned to **jmb** by this statement:

```
// Create the menu bar.
JMenuBar jmb = new JMenuBar();
```

Then, the File menu **jmFile** and its menu entries are created by this sequence:

```
// Create the File menu.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. Next, the menu entries are added to the file menu by this sequence:

```
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
```

Finally, the File menu is added to the menu bar with this line:

```
jmb.add(jmFile);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit. However, notice that a separator has been added before Exit. This visually separates Exit from the preceding three selections.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the same process.

Notice that **MenuDemo** implements the **ActionListener** interface and action events generated by a menu selection are handled by the **actionPerformed( )** method defined by **MenuDemo**. Therefore, the program adds **this** as the action listener for the menu items. Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the frame by the following line:

```
jfrm.setJMenuBar(jmb);
```

As mentioned, menu bars are not added to the content pane. They are added directly to the **JFrame**.

The **actionPerformed( )** method handles the action events generated by the menu. It obtains the action command string associated with the selection by calling **getActionCommand( )** on the event. It stores a reference to this string in **comStr**. Then, it tests the action command against "Exit", as shown here:

```
if(comStr.equals("Exit")) System.exit(0);
```

If the action command is "Exit", then the program terminates by calling **System.exit( )**. This method causes the immediate termination of a program and passes its argument as a status code to the calling process, which is usually the operating system. By convention, a status code of zero means normal termination. Anything else indicates that the program terminated abnormally. For all other menu selections, the choice is displayed.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the course of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: mnemonics and accelerators. As

it applies to menus, a *mnemonic* defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed. An *accelerator* is a key that lets you select a menu item without having to first activate the menu.

A mnemonic can be specified for both **JMenuItem** and **JMenu** objects. There are two ways to set the mnemonic for **JMenuItem**. First, it can be specified when an object is constructed using this constructor:

JMenuItem(String *name*, int *mnem*)

In this case, the name is passed in *name* and the mnemonic is passed in *mnen*. Second, you can set the mnemonic by calling **setMnemonic( )**. To specify a mnemonic for **JMenu**, you must call **setMnemonic( )**. This method is inherited by both classes from **AbstractButton** and is shown next:

void setMnemonic(int *mnem*)

Here, *mnem* specifies the mnemonic. It should be one of the constants defined in **java.awt.event.KeyEvent**, such as **KeyEvent.VK_F** or **KeyEvent.VK_Z**. (There is another version of **setMnemonic( )** that takes a **char** argument, but it is considered obsolete.) Mnemonics are not case sensitive, so in the case of **VK_A**, typing either *a* or *A* will work.

By default, the first matching letter in the menu item will be underscored. In cases in which you want to underscore a letter other than the first match, specify the index of the letter as an argument to **setDisplayedMnemonicIndex( )**, which is inherited by both **JMenu** and **JMenuItem** from **AbstractButton**. It is shown here:

void setDisplayedMnemonicIndex(int *idx*)

The index of the letter to underscore is specified by *idx*.

An accelerator can be associated with a **JMenuItem** object. It is specified by calling **setAccelerator( )**, shown next:

void setAccelerator(KeyStroke *ks*)

Here, *ks* is the key combination that is pressed to select the menu item. **KeyStroke** is a class that contains several factory methods that construct various

types of keystroke accelerators. The following are three examples:

static KeyStroke getKeyStroke(char *ch*)

static KeyStroke getKeyStroke(Character *ch*, int *modifier*)

static KeyStroke getKeyStroke(int *ch*, int *modifier*)

Here, *ch* specifies the accelerator character. In the first version, the character is specified as a **char** value. In the second, it is specified as an object of type **Character**. In the third, it is a value of type **KeyEvent**, previously described. The value of *modifier* must be one or more of the following constants, defined in the **java.awt.event.InputEvent** class:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Therefore, if you pass **VK_A** for the key character and **InputEvent.CTRL_DOWN_MASK** for the modifier, the accelerator key combination is CTRL-A.

The following sequence adds both mnemonics and accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can select the File menu by typing ALT-F. Then, you can use the mnemonics O, C, S, or E to select an option. Alternatively, you can directly select a File menu option by pressing CTRL-O, CTRL-C, CTRL-S, or CTRL-E. Figure 33-2 shows how this menu looks when activated.

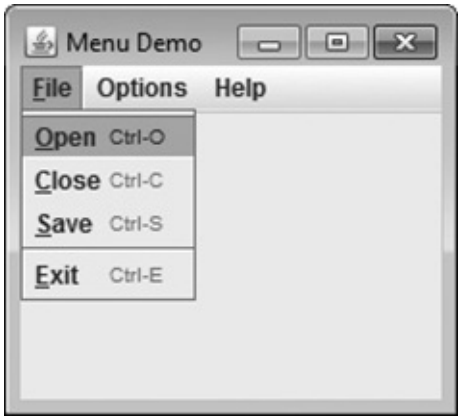


Figure 33-2 The File menu after adding mnemonics and accelerators

```
// Create the File menu with mnemonics and accelerators.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);

JMenuItem jmiOpen = new JMenuItem("Open",
                                  KeyEvent.VK_O);
jmiOpen.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiClose = new JMenuItem("Close",
                                   KeyEvent.VK_C);
jmiClose.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_C,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save",
                                  KeyEvent.VK_S);
jmiSave.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                  KeyEvent.VK_E);
jmiExit.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_E,
                                 InputEvent.CTRL_DOWN_MASK));
```

Add Images and Tooltips to Menu Items

You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using one of these constructors:

JMenuItem(Icon *image*)

JMenuItem(String *name*, Icon *image*)

The first creates a menu item that displays the image specified by *image*. The second creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created:

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

After this addition, the icon specified by **icon** will be displayed next to the text "About" when the Help menu is displayed. This is shown in Figure 33-3. You can also add an icon to a menu item after the item has been created by calling **setIcon( )**, which is inherited from **AbstractButton**. You can specify the horizontal alignment of the image relative to the text by calling **setHorizontalTextPosition( )**.

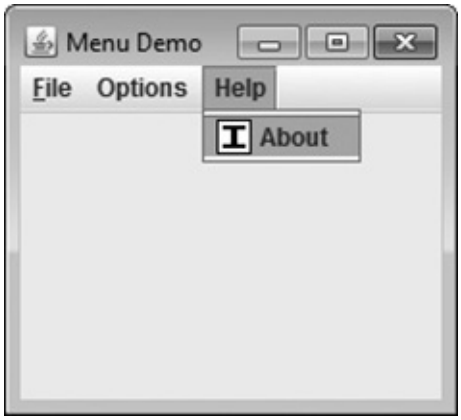


Figure 33-3 The About item with the addition of an icon

You can specify a disabled icon, which is shown when the menu item is disabled, by calling **setDisabledIcon( )**. Normally, when a menu item is disabled, the default icon is shown in gray. If a disabled icon is specified, then that icon is displayed when the menu item is disabled.

A *tooltip* is a small message that describes an item. It is automatically displayed if the mouse remains over the item for a moment. You can add a tooltip to a menu item by calling **setToolTipText( )** on the item, specifying the text you want displayed. It is shown here:

void setToolTipText(String *msg*)

In this case, *msg* is the string that will be displayed when the tooltip is activated. For example, this creates a tooltip for the About item:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

As a point of interest, **setToolTipText( )** is inherited by **JMenuItem** from **JComponent**. This means you can add a tooltip to other types of components, such as a push button. You might want to try this on your own.

Use JRadioButtonMenuItem and JCheckBoxMenuItem

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, Swing defines two others: check boxes and radio buttons. These items can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes, including check boxes or radio buttons in a menu simply seems the most natural place for a specific set of features. Whatever your reason, Swing makes it easy to use check boxes and radio buttons in menus, and both are examined here.

To add a check box to a menu, create a **JCheckBoxMenuItem**. It defines several constructors. This is the one used in this chapter:

JCheckBoxMenuItem(String *name*)

Here, *name* specifies the name of the item. The initial state of the check box is unchecked. If you want to specify the initial state, you can use this constructor:

JCheckBoxMenuItem(String *name*, boolean *state*)

In this case, if *state* is **true**, the box is initially checked. Otherwise, it is cleared. **JCheckBoxMenuItem** also provides constructors that let you specify an icon. Here is one example:

JCheckBoxMenuItem(String *name*, Icon *icon*)

In this case, *name* specifies the name of the item and the image associated with the item is passed in *icon*. The item is initially unchecked. Other constructors are also supported.

Check boxes in menus work like stand-alone check boxes. For example, they generate action events and item events when their state changes. Check boxes

are especially useful in menus when you have options that can be selected and you want to display their selected/deselected status.

A radio button can be added to a menu by creating an object of type **JRadioButtonMenuItem**. **JRadioButtonMenuItem** inherits **JMenuItem**. It provides a rich assortment of constructors. The ones used in this chapter are shown here:

JRadioButtonMenuItem(String *name*)

JRadioButtonMenuItem(String *name*, boolean *state*)

The first constructor creates an unselected radio button menu item that is associated with the name passed in *name*. The second lets you specify the initial state of the button. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Other constructors let you specify an icon. Here is one example:

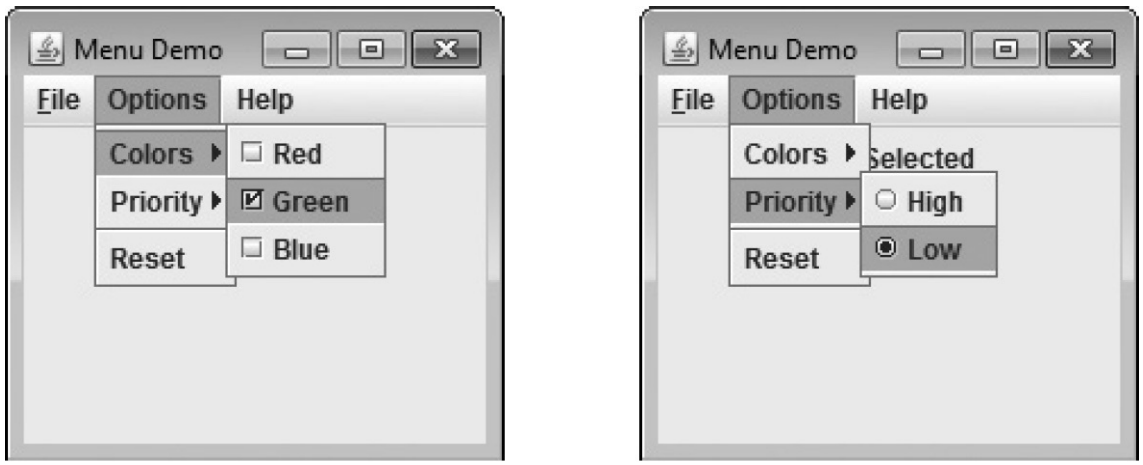
JRadioButtonMenuItem(String *name*, Icon *icon*, boolean *state*)

This creates a radio button menu item that is associated with the name passed in *name* and the image passed in *icon*. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Several other constructors are supported.

A **JRadioButtonMenuItem** works like a stand-alone radio button, generating item and action events. Like stand-alone radio buttons, menu-based radio buttons must be put into a button group in order for them to exhibit mutually exclusive selection behavior.

Because both J**CheckBoxMenuItem** and **JRadioButtonMenuItem** inherit **JMenuItem**, each has all of the functionality provided by **JMenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

To try check box and radio button menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check boxes for the Colors submenu and radio buttons for the Priority submenu. After making the substitution, the Options menu will look like those shown in Figure 33-4.



(a) (b)

Figure 33-4 The effects of check box (a) and radio button (b) menu items

```
// Create the Options menu.
JMenu jmOptions = new JMenu("Options");

// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
```

```
// Use check boxes for colors. This allows
// the user to select more than one color.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");

// Use radio buttons for the priority setting.
// This lets the menu show which priority is used
// but also ensures that one and only one priority
// can be selected at any one time. Notice that
// the High radio button is initially selected.
JRadioButtonMenuItem jmiHigh =
  new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
  new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create button group for the radio button menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);
```

Create a Popup Menu

A popular alternative or addition to the menu bar is the popup menu. Typically, a popup menu is activated by clicking the right mouse button when over a component. Popup menus are supported in Swing by the **JPopupMenu** class. **JPopupMenu** has two constructors. In this chapter, only the default constructor is used:

JPopupMenu( )

It creates a default popup menu. The other constructor lets you specify a title for the menu. Whether this title is displayed is subject to the look and feel.

In general, popup menus are constructed like regular menus. First, create a **JPopupMenu** object, and then add menu items to it. Menu item selections are also handled in the same way: by listening for action events. The main difference between a popup menu and regular menu is the activation process.

Activating a popup menu requires three steps:

1. You must register a listener for mouse events.
2. Inside the mouse event handler, you must watch for the popup trigger.
3. When a popup trigger is received, you must show the popup menu by calling **show( )**.

Let's examine each of these steps closely.

A popup menu is normally activated by clicking the right mouse button when the mouse pointer is over a component for which a popup menu is defined. Thus, the *popup trigger* is usually caused by right-clicking the mouse on a popup menu–enabled component. To listen for the popup trigger, implement the **MouseListener** interface and then register the listener by calling the **addMouseListener( )** method. As described in Chapter 24, **MouseListener** defines the methods shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

Of these, two are very important relative to the popup menu: **mousePressed( )** and **mouseReleased( )**. Depending on the installed look and feel, either of these two events can trigger a popup menu. For this reason, it is often easier to use a **MouseAdapter** to implement the **MouseListener** interface and simply override **mousePressed( )** and **mouseReleased( )**.

The **MouseEvent** class defines several methods, but only four are commonly needed when activating a popup menu. They are shown here:

int getX( )

int getY( )

boolean isPopupTrigger( )

Component getComponent( )

The current X,Y location of the mouse relative to the source of the event is found by calling **getX( )** and **getY( )**. These are used to specify the upper-left corner of the popup menu when it is displayed. The **isPopupTrigger( )** method returns **true** if the mouse event represents a popup trigger and **false** otherwise. You will use this method to determine when to pop up the menu. To obtain a reference to the component that generated the mouse event, call **getComponent( )**.

To actually display the popup menu, call the **show( )** method defined by **JPopupMenu**, shown next:

void show(Component *invoker*, int *upperX*, int *upperY*)

Here, *invoker* is the component relative to which the menu will be displayed. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to *invoker*. A common way to obtain the invoker is to call **getComponent( )** on the event object passed to the mouse event handler.

The preceding theory can be put into practice by adding a popup Edit menu to the **MenuDemo** program shown at the start of this chapter. This menu will have three items called Cut, Copy, and Paste. Begin by adding the following instance variable to **MenuDemo**:

```
JPopupMenu jpu;
```

The **jpu** variable will hold a reference to the popup menu.

Next, add the following code sequence to the **MenuDemo** constructor:

```
// Create an Edit popup menu.
jpu = new JPopupMenu();

// Create the popup menu items.
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// Add the menu items to the popup menu.
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// Add a listener for the popup trigger.
jfrm.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
  public void mouseReleased(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
});
```

This sequence begins by constructing an instance of **JPopupMenu** and storing it in **jpu**. Then, it creates the three menu items, Cut, Copy, and Paste, in the usual way, and adds them to **jpu**. This finishes the construction of the popup Edit menu. Popup menus are not added to the menu bar or any other object.

Next, a **MouseListener** is added by creating an anonymous inner class. This class is based on the **MouseAdapter** class, which means that the listener need only override those methods that are relevant to the popup menu: **mousePressed( )** and **mouseReleased( )**. The adapter provides default implementations of the other **MouseListener** methods. Notice that the mouse

listener is added to **jfrm**. This means that a right-button click inside any part of the content pane will trigger the popup menu.

The **mousePressed( )** and **mouseReleased( )** methods call **isPopupTrigger( )** to determine if the mouse event is a popup trigger event. If it is, the popup menu is displayed by calling **show( )**. The invoker is obtained by calling **getComponent( )** on the mouse event. In this case, the invoker will be the content pane. The X,Y coordinates of the upper-left corner are obtained by calling **getX( )** and **getY( )**. This makes the menu pop up with its upper-left corner directly under the mouse pointer.

Finally, you also need to add these action listeners to the program. They handle the action events fired when the user selects an item from the popup menu.

```
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
```

After you have made these additions, the popup menu can be activated by clicking the right mouse button anywhere inside the content pane of the application. Figure 33-5 shows the result.

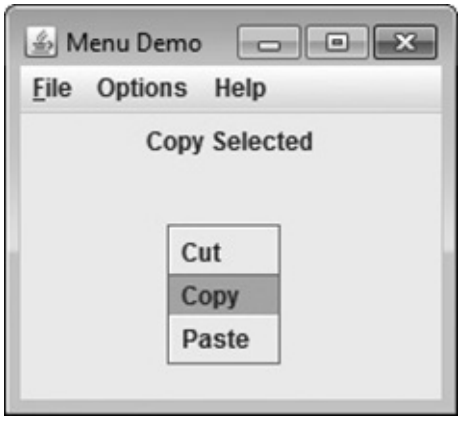


Figure 33-5 A popup Edit menu

One other point about the preceding example. Because the invoker of the popup menu is always **jfrm**, in this case, you could pass it explicitly rather than calling **getComponent( )**. To do so, you must make **jfrm** into an instance variable of the **MenuDemo** class (rather than a local variable) so that it is accessible to the inner class. Then you can use this call to **show( )** to display the popup menu:

```
jpu.show(jfrm, me.getX(), me.getY());
```

Although this works in this example, the advantage of using **getComponent( )** is that the popup menu will automatically pop up relative to the invoking component. Thus, the same code could be used to display any popup menu relative to its invoking object.

Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. A toolbar contains a list of buttons (or other components) that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without needing to drop through a menu. Typically, toolbar buttons show icons rather than text, although either or both are allowed. Furthermore, tooltips are often associated with icon-based toolbar buttons. Toolbars can be positioned on any side of a window by dragging the toolbar, or they can be dragged out of the window entirely, in which case they become free floating.

In Swing, toolbars are instances of the **JToolBar** class. Its constructors enable you to create a toolbar with or without a title. You can also specify the layout of the toolbar, which will be either horizontal or vertical. The **JToolBar** constructors are shown here:

JToolBar( )

JToolBar(String *title*)

JToolBar(int *how*)

JToolBar(String *title*, int *how*)

The first constructor creates a horizontal toolbar with no title. The second creates a horizontal toolbar with the title specified by *title*. The title will show only when the toolbar is dragged out of its window. The third creates a toolbar that is oriented as specified by *how*. The value of *how* must be either **JToolBar.VERTICAL** or **JToolBar.HORIZONTAL**. The fourth constructor creates a toolbar that has the title specified by *title* and is oriented as specified by *how*.

A toolbar is typically used with a window that uses a border layout. There are two reasons for this. First, it allows the toolbar to be initially positioned along one of the four border positions. Frequently, the top position is used. Second, it allows the toolbar to be dragged to any side of the window.

In addition to dragging the toolbar to different locations within a window, you can also drag it out of the window. Doing so creates an *undocked* toolbar. If you specify a title when you create the toolbar, then that title will be shown when the toolbar is undocked.

You add buttons (or other components) to a toolbar in much the same way that you add them to a menu bar. Simply call **add( )**. The components are shown in the toolbar in the order in which they are added.

Once you have created a toolbar, you *do not* add it to the menu bar (if one exists). Instead, you add it to the window container. As mentioned, typically you will add a toolbar to the top (that is, north) position of a border layout, using a horizontal orientation. The component that will be affected is added to the center of the border layout. Using this approach causes the program to begin running with the toolbar in the expected location. However, you can drag the toolbar to any of the other positions. Of course, you can also drag the toolbar out of the window.

To illustrate the toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: set a breakpoint, clear a breakpoint, and resume program execution. Three steps are needed to add the toolbar.

First, remove this line from the program:

```
jfrm.setLayout(new FlowLayout());
```

By removing this line, the **JFrame** automatically uses a border layout.

Second, because **BorderLayout** is being used, change the line that adds the label **jlab** to the frame, as shown next:

```
jfrm.add(jlab, BorderLayout.CENTER);
```

This line explicitly adds **jlab** to the center of the border layout. (Explicitly specifying the center position is technically not necessary because, by default, components are added to the center when a border layout is used. However, explicitly specifying the center makes it clear to anyone reading the code that a border layout is being used and that **jlab** goes in the center.)

Next, add the following code, which creates the Debug toolbar:

```
// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Debug");

// Load the images.
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");

// Create the toolbar buttons.
JButton jbtnSet = new JButton(set);
jbtnSet.setActionCommand("Set Breakpoint");
jbtnSet.setToolTipText("Set Breakpoint");

JButton jbtnClear = new JButton(clear);
jbtnClear.setActionCommand("Clear Breakpoint");
jbtnClear.setToolTipText("Clear Breakpoint");

JButton jbtnResume = new JButton(resume);
jbtnResume.setActionCommand("Resume");
jbtnResume.setToolTipText("Resume");

// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

Let’s look at this code closely. First, a **JToolBar** is created and given the title "Debug". Then, a set of **ImageIcon** objects are created that hold the images for the toolbar buttons. Next, three toolbar buttons are created. Notice that each has an image, but no text. Also, each is explicitly given an action command and a tooltip. The action commands are set because the buttons are not given names when they are constructed. Tooltips are especially useful when applied to icon-based toolbar components because sometimes it’s hard to design images that are intuitive to all users. The buttons are then added to the toolbar, and the toolbar is added to the north side of the border layout of the frame.

Finally, add the action listeners for the toolbar, as shown here:

```
// Add the toolbar action listeners.
jbtnSet.addActionListener(this);
jbtnClear.addActionListener(this);
jbtnResume.addActionListener(this);
```

Each time the user presses a toolbar button, an action event is fired, and it is handled in the same way as the other menu-related events. Figure 33-6 shows the toolbar in action.

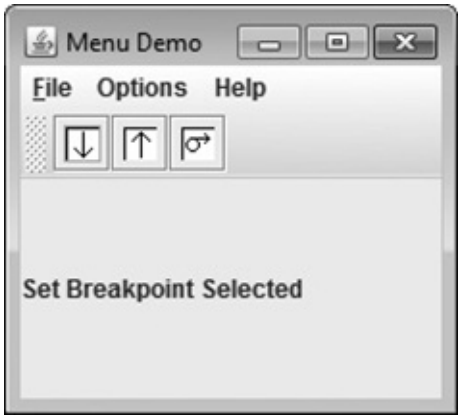


Figure 33-6 The Debug toolbar in action

Use Actions

Often, a toolbar and a menu item contain items in common. For example, the same functions provided by the Debug toolbar in the preceding example might also be offered through a menu selection. In such a case, selecting an option (such as setting a breakpoint) causes the same action to occur, independently of whether the menu or the toolbar was used. Also, both the toolbar button and the menu item would (most likely) use the same icon. Furthermore, when a toolbar button is disabled, the corresponding menu item would also need to be disabled. Such a situation would normally lead to a fair amount of duplicated, interdependent code, which is less than optimal. Fortunately, Swing provides a solution: the *action*.

An action is an instance of the **Action** interface. **Action** extends the **ActionListener** interface and provides a means of combining state information with the **actionPerformed( )** event handler. This combination allows one action to manage two or more components. For example, an action lets you centralize the control and handling of a toolbar button and a menu item. Instead of having

to duplicate code, your program need only create an action that automatically handles both components.

Because **Action** extends **ActionListener**, an action must provide an implementation of the **actionPerformed( )** method. This handler will process the action events generated by the objects linked to the action.

In addition to the inherited **actionPerformed( )** method, **Action** defines several methods of its own. One of particular interest is **putValue( )**. It sets the value of the various properties associated with an action and is shown here:

void putValue(String *key*, Object *val*)

It assigns *val* to the property specified by *key* that represents the desired property. Although not used by the example that follows, it is helpful to note that **Action** also supplies the **getValue( )** method that obtains a specified property. It is shown here:

Object getValue(String *key*)

It returns a reference to the property specified by *key*.

The key values used by **putValue( )** and **getValue( )** include those shown here:

Key Value	Description
static final String ACCELERATOR_KEY	Represents the accelerator property. Accelerators are specified as **KeyStroke** objects.
static final String ACTION_COMMAND_KEY	Represents the action command property. An action command is specified as a string.
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Represents the index of the character displayed as the mnemonic. This is an **Integer** value.
static final String LARGE_ICON_KEY	Represents the large icon associated with the action. The icon is specified as an object of type **Icon**.
static final String LONG_DESCRIPTION	Represents a long description of the action. This description is specified as a string.
static final String MNEMONIC_KEY	Represents the mnemonic property. A mnemonic is specified as a **KeyEvent** constant.
static final String NAME	Represents the name of the action (which also becomes the name of the button or menu item to which the action is linked). The name is specified as a string.
static final String SELECTED_KEY	Represents the selection status. If set, the item is selected. The state is represented by a **Boolean** value.
static final String SHORT_DESCRIPTION	Represents the tooltip text associated with the action. The tooltip text is specified as a string.
static final String SMALL_ICON	Represents the icon associated with the action. The icon is specified as an object of type **Icon**.

For example, to set the mnemonic to the letter *X*, use this call to **putValue( )**:

```
actionOb.putValue(MNEMONIC_KEY, KeyEvent.VK_X);
```

One **Action** property that is not accessible through **putValue( )** and **getValue( )** is the enabled/disabled status. For this, you use the **setEnabled( )** and **isEnabled( )** methods. They are shown here:

void setEnabled(boolean *enabled*)

boolean isEnabled( )

For **setEnabled( )**, if *enabled* is **true**, the action is enabled. Otherwise, it is disabled. If the action is enabled, **isEnabled( )** returns **true**. Otherwise, it returns **false**.

Although you can implement all of the **Action** interface yourself, you won't usually need to. Instead, Swing provides a partial implementation called **AbstractAction** that you can extend. By extending **AbstractAction**, you need implement only one method: **actionPerformed( )**. The other **Action** methods are provided for you. **AbstractAction** provides three constructors. The one used in this chapter is shown here:

AbstractAction(String *name*, Icon *image*)

It constructs an **AbstractAction** that has the name specified by *name* and the icon specified by *image*.

Once you have created an action, it can be added to a **JToolBar** and used to construct a **JMenuItem**. To add an action to a **JToolBar**, use this version of **add( )**:

JButton add(Action *actObj*)

Here, *actObj* is the action that is being added to the toolbar. The properties defined by *actObj* are used to create a toolbar button. To create a menu item from an action, use this **JMenuItem** constructor:

JMenuItem(Action *actObj*)

Here, *actObj* is the action used to construct a menu item according to its properties.

NOTE In addition to **JToolBar** and **JMenuItem**, actions are also supported by several other Swing components, such as **JPopupMenu**, **JButton**, **JRadioButton**, and **JCheckBox**. **JRadioButtonMenuItem** and **JCheckBoxMenuItem** also support actions.

To illustrate the benefit of actions, we will use them to manage the Debug toolbar created in the previous section. We will also add a Debug submenu under the Options main menu. The Debug submenu will contain the same selections as the Debug toolbar: Set Breakpoint, Clear Breakpoint, and Resume. The same actions that support these items in the toolbar will also support these items in the menu. Therefore, instead of having to create duplicate code to handle both the toolbar and menu, both are handled by the actions.

Begin by creating an inner class called **DebugAction** that extends **AbstractAction**, as shown here:

```
// A class to create an action for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
             KeyStroke.getKeyStroke(accel,
                                    InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, mnem);
    putValue(SHORT_DESCRIPTION, tTip);
  }

  // Handle events for both the toolbar and the
  // Debug menu.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Toggle the enabled status of the
    // Set and Clear Breakpoint options.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}
```

DebugAction extends **AbstractAction**. It creates an action class that will be used to define the properties associated with the Debug menu and toolbar. Its constructor has five parameters that let you specify the following items:

- Name

- Icon
- Mnemonic
- Accelerator
- Tooltip

The first two are passed to **AbstractAction**'s constructor via **super**. The other three properties are set through calls to **putValue( )**.

The **actionPerformed( )** method of **DebugAction** handles events for the action. This means that when an instance of **DebugAction** is used to create a toolbar button and a menu item, events generated by either of those components are handled by the **actionPerformed( )** method in **DebugAction**. Notice that this handler displays the selection in **jlab**. In addition, if the Set Breakpoint option is selected, then the Clear Breakpoint option is enabled and the Set Breakpoint option is disabled. If the Clear Breakpoint option is selected, then the Set Breakpoint option is enabled and the Clear Breakpoint option is disabled. This illustrates how an action can be used to enable or disable a component. When an action is disabled, it is disabled for all uses of that action. In this case, if Set Breakpoint is disabled, then it is disabled both in the toolbar and in the menu.

Next, add these **DebugAction** instance variables to **MenuDemo**:

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Next, create three **ImageIcon**s that represent the Debug options, as shown here:

```
// Load the images for the actions.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Now, create the actions that manage the Debug options, as shown here:

```
// Create actions.
setAct =
  new DebugAction("Set Breakpoint",
                  setIcon,
                  KeyEvent.VK_S,
                  KeyEvent.VK_B,
                  "Set a break point.");

clearAct =
  new DebugAction("Clear Breakpoint",
                  clearIcon,
                  KeyEvent.VK_C,
                  KeyEvent.VK_L,
                  "Clear a break point.");

resumeAct =
  new DebugAction("Resume",
                  resumeIcon,
                  KeyEvent.VK_R,
                  KeyEvent.VK_R,
                  "Resume execution after breakpoint.");

// Initially disable the Clear Breakpoint option.
clearAct.setEnabled(false);
```

Notice that the accelerator for Set Breakpoint is B and the accelerator for Clear Breakpoint is L. The reason these keys are used rather than S and C is that these keys are already allocated by the File menu for Save and Close. However, they can still be used as mnemonics because each mnemonic is localized to its own menu. Also notice that the action that represents Clear Breakpoint is initially disabled. It will be enabled only after a breakpoint has been set.

Next, use the actions to create buttons for the toolbar and then add those buttons to the toolbar, as shown here:

```
// Create the toolbar buttons by using the actions.
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Breakpoints");
```

```
// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

Finally, create the Debug menu, as shown next:

```
// Now, create a Debug menu that goes under the Options
// menu bar item. Use the actions to create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);
```

After making these changes and additions, the actions that you created will be used to manage both the Debug menu and the toolbar. Thus, changing a property in the action (such as disabling it) will affect all uses of that action. The program will now look as shown in Figure 33-7.

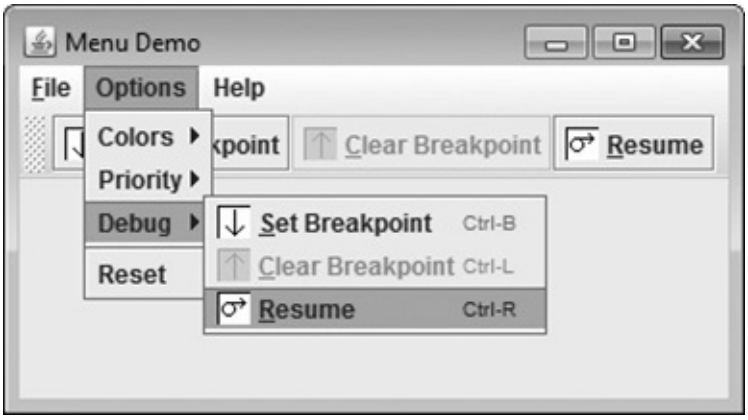


Figure 33-7 Using actions to manage the Debug toolbar and menu

Put the Entire MenuDemo Program Together

Throughout the course of this discussion, many changes and additions have been

made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **jmb**, **jmFile**, and **jtb**, have been made into instance variables.

```
// The complete MenuDemo program.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

  JLabel jlab;

  JMenuBar jmb;

  JToolBar jtb;

  JPopupMenu jpu;

  DebugAction setAct;
  DebugAction clearAct;
  DebugAction resumeAct;

  MenuDemo() {
    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Complete Menu Demo");

    // Use default border layout.

    // Give the frame an initial size.
    jfrm.setSize(360, 200);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a label that will display the menu selection.
    jlab = new JLabel();

    // Create the menu bar.
    jmb = new JMenuBar();

    // Make the File menu.
    makeFileMenu();
```

```
    // Construct the Debug actions.
    makeActions();

    // Make the toolbar.
    makeToolBar();

    // Make the Options menu.
    makeOptionsMenu();

    // Make the Help menu.
    makeHelpMenu();

    // Make the Edit popup menu.
    makeEditPUMenu();

    // Add a listener for the popup trigger.
    jfrm.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
      }
      public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
      }
    });

    // Add the label to the center of the content pane.
    jfrm.add(jlab, SwingConstants.CENTER);

    // Add the toolbar to the north position of
    // the content pane.
    jfrm.add(jtb, BorderLayout.NORTH);

    // Add the menu bar to the frame.
    jfrm.setJMenuBar(jmb);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle menu item action events.
  // This does NOT handle events generated
  // by the Debug options.
  public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
  }
```

```
// An action class for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
             KeyStroke.getKeyStroke(accel,
                                    InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, mnem);
    putValue(SHORT_DESCRIPTION, tTip);
  }

  // Handle events for both the toolbar and the
  // Debug menu.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Toggle the enabled status of the
    // Set and Clear Breakpoint options.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}

// Create the File menu with mnemonics and accelerators.
void makeFileMenu() {
  JMenu jmFile = new JMenu("File");
  jmFile.setMnemonic(KeyEvent.VK_F);

  JMenuItem jmiOpen = new JMenuItem("Open",
                                    KeyEvent.VK_O);
  jmiOpen.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                   InputEvent.CTRL_DOWN_MASK));

  JMenuItem jmiClose = new JMenuItem("Close",
                                     KeyEvent.VK_C);
  jmiClose.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_C,
                                   InputEvent.CTRL_DOWN_MASK));

  JMenuItem jmiSave = new JMenuItem("Save",
                                    KeyEvent.VK_S);
  jmiSave.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                   InputEvent.CTRL_DOWN_MASK));
```

```
    JMenuItem jmiExit = new JMenuItem("Exit",
                                      KeyEvent.VK_E);
    jmiExit.setAccelerator(
              KeyStroke.getKeyStroke(KeyEvent.VK_E,
                                     InputEvent.CTRL_DOWN_MASK));

    jmFile.add(jmiOpen);
    jmFile.add(jmiClose);
    jmFile.add(jmiSave);
    jmFile.addSeparator();
    jmFile.add(jmiExit);
    jmb.add(jmFile);

    // Add the action listeners for the File menu.
    jmiOpen.addActionListener(this);
    jmiClose.addActionListener(this);
    jmiSave.addActionListener(this);
    jmiExit.addActionListener(this);
  }

  // Create the Options menu.
  void makeOptionsMenu() {
    JMenu jmOptions = new JMenu("Options");

    // Create the Colors submenu.
    JMenu jmColors = new JMenu("Colors");

    // Use check boxes for colors. This allows
    // the user to select more than one color.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    // Add the items to the Colors menu.
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);

    // Create the Priority submenu.
    JMenu jmPriority = new JMenu("Priority");

    // Use radio buttons for the priority setting.
    // This lets the menu show which priority is used
    // but also ensures that one and only one priority
    // can be selected at any one time. Notice that
    // the High radio button is initially selected.
    JRadioButtonMenuItem jmiHigh =
      new JRadioButtonMenuItem("High", true);
    JRadioButtonMenuItem jmiLow =
      new JRadioButtonMenuItem("Low");

    // Add the items to the Priority menu.
    jmPriority.add(jmiHigh);
```

```
    jmPriority.add(jmiLow);
    jmOptions.add(jmPriority);

    // Create a button group for the radio button
    //  menu items.
    ButtonGroup bg = new ButtonGroup();
    bg.add(jmiHigh);
    bg.add(jmiLow);

    // Now, create a Debug submenu that goes under
    // the Options menu bar item. Use actions to
    // create the items.
    JMenu jmDebug = new JMenu("Debug");
    JMenuItem jmiSetBP = new JMenuItem(setAct);
    JMenuItem jmiClearBP = new JMenuItem(clearAct);
    JMenuItem jmiResume = new JMenuItem(resumeAct);

    // Add the items to the Debug menu.
    jmDebug.add(jmiSetBP);
    jmDebug.add(jmiClearBP);
    jmDebug.add(jmiResume);
    jmOptions.add(jmDebug);

    // Create the Reset menu item.
    JMenuItem jmiReset = new JMenuItem("Reset");
    jmOptions.addSeparator();
    jmOptions.add(jmiReset);

    // Finally, add the entire options menu to
    // the menu bar
    jmb.add(jmOptions);

    // Add the action listeners for the Options menu,
    // except for those supported by the Debug menu.
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
  }

  // Create the Help menu.
  void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Add an icon to the About menu item.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);
```

```
    // Add action listener for About.
    jmiAbout.addActionListener(this);
  }

  // Construct the actions needed by the Debug menu
  // and toolbar.
  void makeActions() {
    // Load the images for the actions.
    ImageIcon setIcon = new ImageIcon("setBP.gif");
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");
    ImageIcon resumeIcon = new ImageIcon("resume.gif");

    // Create actions.
    setAct =
      new DebugAction("Set Breakpoint",
                      setIcon,
                      KeyEvent.VK_S,
                      KeyEvent.VK_B,
                      "Set a break point.");

    clearAct =
      new DebugAction("Clear Breakpoint",
                      clearIcon,
                      KeyEvent.VK_C,
                      KeyEvent.VK_L,
                      "Clear a break point.");

    resumeAct =
      new DebugAction("Resume",
                      resumeIcon,
                      KeyEvent.VK_R,
                      KeyEvent.VK_R,
                      "Resume execution after breakpoint.");

    // Initially disable the Clear Breakpoint option.
    clearAct.setEnabled(false);
  }

  // Create the Debug toolbar.
  void makeToolBar() {
    // Create the toolbar buttons by using the actions.
    JButton jbtnSet = new JButton(setAct);
    JButton jbtnClear = new JButton(clearAct);
    JButton jbtnResume = new JButton(resumeAct);

    // Create the Debug toolbar.
    jtb = new JToolBar("Breakpoints");

    // Add the buttons to the toolbar.
    jtb.add(jbtnSet);
    jtb.add(jbtnClear);
    jtb.add(jbtnResume);
  }
```

```
  // Create the Edit popup menu.
  void makeEditPUMenu() {
    jpu = new JPopupMenu();

    // Create the popup menu items
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // Add the menu items to the popup menu.
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);

    // Add the Edit popup menu action listeners.
    jmiCut.addActionListener(this);
    jmiCopy.addActionListener(this);
    jmiPaste.addActionListener(this);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```

Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, it supplies dialog classes, such as **JOptionPane** and **JDialog**, that you can use to streamline the construction of dialog windows. It also provides additional controls beyond those introduced in Chapter 31. Two you will want to explore are **JSpinner** (which creates a spin

control) and **JFormattedTextField** (which supports formatted text). You will also want to experiment with defining your own models for the various components. Frankly, the best way to become familiar with Swing's capabilities is to experiment with it.

PART

IV

Applying Java

CHAPTER

34

Java Beans

This chapter provides an overview of creating Java Beans. Beans are important because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans use an architecture called *JavaBeans* that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

The software industry also sought the benefits of reusability and interoperability of a component-based approach. To realize these benefits, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. JavaBeans provides just such an architecture.

What Is a Java Bean?

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously

on a user's workstation or to work in cooperation with a set of other distributed components. Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Beans

The following list enumerates some of the benefits that JavaBeans technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

At the core of Bean programming is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the JavaBeans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the JavaBeans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a *setter* method. A property is obtained by a *getter* method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```
public T getN( )
public void setN(T arg)
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:

```
private double depth, height, width;

public double getDepth( ) {
  return depth;
}
public void setDepth(double d) {
  depth = d;
}

public double getHeight( ) {
  return height;
}
public void setHeight(double h) {
  height = h;
}

public double getWidth( ) {
  return width;
}
public void setWidth(double w) {
  width = w;
}
```

NOTE For a **boolean** property, a method of the form **is***PropertyName*( ) can also be used as an accessor.

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

public T getN(int *index*);
public void setN(int *index*, T *value*);
public T[ ] getN( );
public void setN(T *values*[ ]);

Here is an indexed property called **data** along with its getter and setter methods:

```
private double data[ ];

public double getData(int index) {
  return data[index];
}
public void setData(int index, double value) {
  data[index] = value;
}
public double[ ] getData( ) {
  return data;
}
public void setData(double[ ] values) {
  data = new double[values.length];
  System.arraycopy(values, 0, data, 0, values.length);
}
```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where **T** is the type of the event:

public void addTListener(TListener *eventListener*)
public void addTListener(TListener *eventListener*)
throws java.util.TooManyListenersException
public void removeTListener(TListener *eventListener*)

These methods are used to add or remove a listener for the specified event. The version of **addTListener( )** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener( )** is used to remove the listener. For example, assuming an event interface type called **TemperatureListener**, a Bean that monitors temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener tl) {
...
}
public void removeTemperatureListener(TemperatureListener tl) {
```

```
...
}
```

Methods and Design Patterns

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available. The **BeanInfo** interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, you must call that class *bname*BeanInfo, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.

To simplify the use of **BeanInfo**, JavaBeans supplies the **SimpleBeanInfo** class. It provides default implementations of the **BeanInfo** interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don’t override a method, then design-pattern introspection will be used. For example, if you don’t override **getPropertyDescriptors( )**, then design patterns are used to discover a Bean’s properties. You will see **SimpleBeanInfo** in action later in this chapter.

Bound and Constrained Properties

A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

Customizers

A Bean developer can provide a *customizer* that helps another developer

configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

The JavaBeans API

The JavaBeans functionality is provided by a set of classes and interfaces in the **java.beans** package. Beginning with JDK 9, this package is in the **java.desktop** module. This section provides a brief overview of its contents. Table 34-1 lists the interfaces in **java.beans** and provides a brief description of their functionality. Table 34-2 lists the classes in **java.beans**.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets. (Deprecated by JDK 9.)
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Table 34-1 The Interfaces in **java.beans**

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of **PersistenceDelegate**.
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes, among others.

IndexedPropertyChangeEvent	A subclass of **PropertyChangeEvent** that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a **BeanInfo** object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the **PropertyChangeListener** or **VetoableChangeListener** interfaces.
PropertyChangeListenerProxy	Extends **EventListenerProxy** and implements **PropertyChangeListener**.
PropertyChangeSupport	Beans that support bound properties can use this class to notify **PropertyChangeListener** objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a **PropertyEditor** object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing **BeanInfo** classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends **EventListenerProxy** and implements **VetoableChangeListener**.
VetoableChangeSupport	Beans that support constrained properties can use this class to notify **VetoableChangeListener** objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Table 34-2 The Classes in **java.beans**

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector**, **PropertyDescriptor**,

EventSetDescriptor, and **MethodDescriptor**. Each is briefly examined here.

Introspector

The **Introspector** class provides several static methods that support introspection. Of most interest is **getBeanInfo( )**. This method returns a **BeanInfo** object that can be used to obtain information about the Bean. The **getBeanInfo( )** method has several forms, including the one shown here:

static BeanInfo getBeanInfo(Class<?> *bean*) throws IntrospectionException

The returned object contains information about the Bean specified by *bean.*

PropertyDescriptor

The **PropertyDescriptor** class describes the characteristics of a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling **isBound( )**. To determine if a property is constrained, call **isConstrained( )**. You can obtain the name of a property by calling **getName( )**.

EventSetDescriptor

The **EventSetDescriptor** class represents a set of Bean events. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listeners, call **getAddListenerMethod( )**. To obtain the method used to remove listeners, call **getRemoveListenerMethod( )**. To obtain the type of a listener, call **getListenerType( )**. You can obtain the name of an event set by calling **getName( )**.

MethodDescriptor

The **MethodDescriptor** class represents a Bean method. To obtain the name of the method, call **getName( )**. You can obtain information about the method by calling **getMethod( )**, shown here:

Method getMethod( )

An object of type **Method** that describes the method is returned.

A Bean Example

This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a **BeanInfo** class. It also makes use of the **Introspector**, **PropertyDescriptor**, and **EventSetDescriptor** classes. The example uses three classes. The first is a Bean called **Colors**, shown here:

```
// A simple Bean.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
  transient private Color color; // not persistent
  private boolean rectangular; // is persistent
```

```
  public Colors() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize(200, 100);
    change();
  }

  public boolean getRectangular() {
    return rectangular;
  }

  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
  }

  public void change() {
    color = randomColor();
    repaint();
  }

  private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
  }

  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }
    else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

The **Colors** Bean displays a colored object within a frame. The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**. The constructor defines an anonymous inner class that extends **MouseAdapter** and overrides its **mousePressed( )** method. The **change( )** method is invoked in response to mouse presses. It selects a random color and then repaints the component. The **getRectangular( )** and **setRectangular( )** methods provide access to the one property of this Bean. The **change( )** method calls **randomColor( )** to choose a color and then calls **repaint( )** to make the change visible. Notice that the **paint( )** method uses the **rectangular** and **color** variables to determine how to present the Bean.

The next class is **ColorsBeanInfo**. It is a subclass of **SimpleBeanInfo** that provides explicit information about **Colors**. It overrides **getPropertyDescriptors( )** in order to designate which properties are presented to a Bean user. In this case, the only property exposed is **rectangular**. The method creates and returns a **PropertyDescriptor** object for the **rectangular** property. The **PropertyDescriptor** constructor that is used is shown here:

PropertyDescriptor(String *property*, Class<?> *beanCls*)
throws IntrospectionException

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor rectangular = new
         PropertyDescriptor("rectangular", Colors.class);
      PropertyDescriptor pd[] = {rectangular};
      return pd;
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
    return null;
  }
}
```

The final class is called **IntrospectorDemo**. It uses introspection to display the properties and events that are available within the **Colors** Bean.

```
// Show properties and events.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
  public static void main(String args[]) {
    try {
      Class<?> c = Class.forName("Colors");
      BeanInfo beanInfo = Introspector.getBeanInfo(c);

      System.out.println("Properties:");
      PropertyDescriptor propertyDescriptor[] =
        beanInfo.getPropertyDescriptors();
      for(int i = 0; i < propertyDescriptor.length; i++) {
        System.out.println("\t" + propertyDescriptor[i].getName());
      }
```

```
      System.out.println("Events:");
      EventSetDescriptor eventSetDescriptor[] =
        beanInfo.getEventSetDescriptors();
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
  }
}
```

The output from this program is the following:

```
Properties:
        rectangular
Events:
        mouseWheel
        mouse
        mouseMotion
        component
        hierarchyBounds
        focus
        hierarchy
        propertyChange
        inputMethod
        key
```

Notice two things in the output. First, because **ColorsBeanInfo** overrides **getPropertyDescriptors( )** such that the only property returned is **rectangular**, only the **rectangular** property is displayed. However, because **getEventSetDescriptors( )** is not overridden by **ColorsBeanInfo**, design-pattern introspection is used, and all events are found, including those in **Colors**' superclass, **Canvas**. Remember, if you don't override one of the "get" methods defined by **SimpleBeanInfo**, then the default, design-pattern introspection is used. To observe the difference that **ColorsBeanInfo** makes, erase its class file and then run **IntrospectorDemo** again. This time it will report more properties.

CHAPTER

Introducing Servlets

This chapter presents an introduction to *servlets.* Servlets are small programs that execute on the server side of a web connection. The topic of servlets is quite large, and it is beyond the scope of this chapter to cover it all. Instead, we will focus on the core concepts, interfaces, and classes, and develop several examples.

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in

terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with other software via the sockets and RMI mechanisms that you have seen already.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init( )**, **service( )**, and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init( )** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service( )** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server’s address space and is available to process any other HTTP requests received from clients. The **service( )** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The

algorithms by which this determination is made are specific to each server. The server calls the **destroy( )** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To experiment with servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Apache Tomcat. Glassfish is an open source project sponsored by Oracle and is provided with the Java EE SDK. It is supported by NetBeans. Apache Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

This chapter uses Tomcat in the examples. It provides a simple, yet effective way to experiment with servlets using only the command line tools. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don’t need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses a different servlet container, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and/or a different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 8.5.31. This version of Tomcat is used here because it will work for a very wide range of readers. You can download Tomcat from **tomcat.apache.org**. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 8.5.31 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\
```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

NOTE All of the directories shown in this section assume Tomcat 8.5.31. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-8.5.31** directory. To stop Tomcat, execute **shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib
```

To make **servlet-api.jar** accessible, update your **CLASSPATH** environment variable so that it includes

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-
8.5.31\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-8.5.31-
windows-
x64\apache-tomcat-8.5.31\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, the examples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\webapps\
examples\WEB-INF\classes
```

Next, add the servlet's name and mapping to the **web.xml** file in the following directory:

```
C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\webapps\
examples\WEB-INF
```

For instance, assuming the first example, called **HelloServlet**, you will add the following lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlets/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.
2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the following program:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

  public void service(ServletRequest request,
    ServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>Hello!");
    pw.close();
  }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init( )** and **destroy( )**, which may be used as is. You need supply only the **service( )** method.

Inside **HelloServlet**, the **service( )** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument

is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType( )** establishes the MIME type of the HTTP response. In this program, the MIME type is text/html. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter(**) method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println( )** is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet

Start a web browser and enter the URL shown here:

```
http://localhost:8080/examples/servlets/servlet/HelloServlet
```

Alternatively, you may enter the URL shown here:

```
http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet
```

This can be done because 127.0.0.1 is defined as the IP address of the local machine.

You will observe the output of the servlet in the browser display area. It will contain the string **Hello!** in bold type.

The Servlet API

Two packages contain the classes and interfaces that are required to build the servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also

provided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The servlet specification supported by Tomcat 8.5.31 is version 3.1. (As a point of interest, Tomcat 9 supports servlet specification 4.) This chapter discusses the core of the Servlet API, which will be available to most readers and works with all modern versions of the servlet specification.

The javax.servlet Package

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes several key interfaces that are provided in this package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the **javax.servlet** package:

Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the **Servlet** interface. It declares the **init( )**, **service( )**, and **destroy( )** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in Table 35-1.

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. A **ServletException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

Table 35-1 The Methods Defined by **Servlet**

The **init( )**, **service( )**, and **destroy( )** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig( )** method is called by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo( )** method to provide a string with useful information (for example, the version number). This method is also invoked by the server.

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration<String> getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface enables servlets to obtain information about their environment. Several of its methods are summarized in Table 35-2.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real (i.e., absolute) path that corresponds to the relative path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Writes *s* and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

Table 35-2 Various Methods Defined by **ServletContext**

The ServletRequest Interface

The **ServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are summarized in Table 35-3.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has been previously invoked on this object.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration<String> getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has been previously invoked on this object.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

Table 35-3 Various Methods Defined by **ServletRequest**

The ServletResponse Interface

The **ServletResponse** interface enables a servlet to formulate a response for a client. Several of its methods are summarized in Table 35-4.

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has been previously invoked on this object.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has been previously invoked on this object.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

Table 35-4 Various Methods Defined by **ServletResponse**

The GenericServlet Class

The **GenericServlet** class provides implementations of the basic life cycle methods for a servlet. **GenericServlet** implements the **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

void log(String *s*)
void log(String *s*, Throwable *e*)

Here, *s* is the string to be appended to the log, and *e* is an exception that occurred.

The ServletInputStream Class

The **ServletInputStream** class extends **InputStream**. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. In addition to the input methods inherited from **InputStream**, a method is provided to read bytes from the stream. It is shown here:

int readLine(byte[ ] *buffer*, int *offset*, int *size*) throws IOException

Here, *buffer* is the array into which *size* bytes are placed starting at *offset.* The method returns the actual number of bytes read or –1 if an end-of-stream condition is encountered.

The ServletOutputStream Class

The **ServletOutputStream** class extends **OutputStream**. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. In addition to the output methods provided by **OutputStream**, it also defines the **print( )** and **println( )** methods, which output data to the stream.

The Servlet Exception Classes

javax.servlet defines two exceptions. The first is **ServletException**, which indicates that a servlet problem has occurred. The second is **UnavailableException**, which extends **ServletException**. It indicates that a servlet is unavailable.

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/
          servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service( )** method is overridden to process client requests. The **getParameterNames( )** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter( )** method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet
extends GenericServlet {

  public void service(ServletRequest request,
    ServletResponse response)
  throws ServletException, IOException {

    // Get print writer.
    PrintWriter pw = response.getWriter();

    // Get enumeration of parameter names.
    Enumeration<String> e = request.getParameterNames();

    // Display parameter names and values.
    while(e.hasMoreElements()) {
      String pname = e.nextElement();
      pw.print(pname + " = ");
      String pvalue = request.getParameter(pname);
      pw.println(pvalue);
    }
    pw.close();
  }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

The javax.servlet.http Package

The preceding examples have used the classes and interfaces defined in **javax.servlet**, such as **ServletRequest**, **ServletResponse**, and **GenericServlet**, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in **javax.servlet.http**. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the interfaces used in this chapter:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.

The following table summarizes the classes used in this chapter. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.

The HttpServletRequest Interface

The **HttpServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are shown in Table 35-5.

Method	Description
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration<String> getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

Table 35-5 Various Methods Defined by **HttpServletRequest**

The HttpServletResponse Interface

The **HttpServletResponse** interface enables a servlet to formulate an HTTP

response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, **SC_OK** indicates that the HTTP request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in Table 35-6.

Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

Table 35-6 Various Methods Defined by **HttpServletResponse**

The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 35-7. All of these methods throw an **IllegalStateException**

if the session has already been invalidated.

Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration<String> getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

Table 35-7 Various Methods Defined by **HttpSession**

The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store.

A servlet can write a cookie to a user's machine via the **addCookie( )** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that can be saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for **Cookie**. It has the signature shown here:

Cookie(String *name*, String *value*)

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in Table 35-8.

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
boolean isHttpOnly( )	Returns **true** if the cookie has the **HttpOnly** attribute.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setHttpOnly(boolean *httpOnly*)	If *httpOnly* is **true**, then the **HttpOnly** attribute is added to the cookie. If *httpOnly* is **false**, the **HttpOnly** attribute is removed.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

Table 35-8 The Methods Defined by **Cookie**

The HttpServlet Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in Table 35-9.

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP OPTIONS request.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table 35-9 The Methods Defined by **HttpServlet**

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut( )**, and **doTrace( )**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.html**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.html** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
  action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing. The **doGet( )** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");
    pw.println(color);
    pw.close();
  }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

```
http://localhost:8080/examples/servlets/servlet/ColorGetServlet?
color=Red
```

The characters to the right of the question mark are known as the *query string.*

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost( )** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");
    pw.println(color);
    pw.close();
  }
}
```

Compile the servlet and perform the same steps as described in the previous section to test it.

NOTE Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. The parameter names and values are sent in the body of the HTTP request.

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

File	Description
AddCookie.html	Allows a user to specify a value for the cookie named **MyCookie**.
AddCookieServlet.java	Processes the submission of **AddCookie.html**.
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following

listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie( )** method. A feedback message is then written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {
```

```
        // Get parameter from HTTP request.
        String data = request.getParameter("data");

        // Create cookie.
        Cookie cookie = new Cookie("MyCookie", data);

        // Add cookie to HTTP response.
        response.addCookie(cookie);

        // Write output to browser.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
        pw.close();
    }
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies( )** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    // Get cookies from header of HTTP request.
    Cookie[] cookies = request.getCookies();

    // Display these cookies.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>");
    for(int i = 0; i < cookies.length; i++) {
      String name = cookies[i].getName();
      String value = cookies[i].getValue();
      pw.println("name = " + name +
        "; value = " + value);
    }
    pw.close();
  }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.html** in a browser.
3. Enter a value for **MyCookie**.
4. Submit the web page.

After completing these steps, you will observe that a feedback message is displayed by the browser.

Next, request the following URL via the browser:

```
http://localhost:8080/examples/servlets/servlet/GetCookiesServlet
```

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the **setMaxAge( )** method of **Cookie**. Therefore, the cookie expires when the browser session ends. You can experiment by using **setMaxAge( )** and observe that the cookie is then saved on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession( )** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )**, and **removeAttribute( )** methods of **HttpSession** manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The **getSession( )** method gets the current session. A new session is created if one does not already exist. The **getAttribute( )** method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute( )** method is called to bind the name "date" to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {
```

```
  public void doGet(HttpServletRequest request,
     HttpServletResponse response)
  throws ServletException, IOException {

     // Get the HttpSession object.
     HttpSession hs = request.getSession(true);

     // Get writer.
     response.setContentType("text/html");
     PrintWriter pw = response.getWriter();
     pw.print("<B>");

     // Display date/time of last access.
     Date date = (Date)hs.getAttribute("date");
     if(date != null) {
       pw.print("Last access: " + date + "<br>");
     }

     // Display current date/time.
     date = new Date();
     hs.setAttribute("date", date);
     pw.println("Current date: " + date);
  }
}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

PART

Appendixes

APPENDIX

Using Java's Documentation Comments

As explained in Part I, Java supports three types of comments. The first two are the // and the /* */. The third type is called a *documentation comment.* It begins with the character sequence /**. It ends with */. Documentation comments allow you to embed information about your program into the program itself. You can then use the **javadoc** utility program (supplied with the JDK) to extract the information and put it into an HTML file. Documentation comments make it convenient to document your programs. You have almost certainly seen documentation that uses such comments because that is the way the Java API library was documented. Beginning with JDK 9, **javadoc** includes support for modules.

The javadoc Tags

The **javadoc** utility recognizes several tags, including those shown here:

Tag	Meaning
@author	Identifies the author.
{@code}	Displays information as-is, without processing HTML styles, in code font.
@deprecated	Specifies that a program element is deprecated.
{@docRoot}	Specifies the path to the root directory of the current documentation.
@exception	Identifies an exception thrown by a method or constructor.
@hidden	Prevents an element from appearing in the documentation.
{@index}	Specifies a term for indexing.
{@inheritDoc}	Inherits a comment from the immediate superclass.
{@link}	Inserts an in-line link to another topic.
{@linkplain}	Inserts an in-line link to another topic, but the link is displayed in a plain-text font.
{@literal}	Displays information as-is, without processing HTML styles.
@param	Documents a parameter.

Tag	Meaning
@provides	Documents a service provided by a module.
@return	Documents a method's return value.
@see	Specifies a link to another topic.
@serial	Documents a default serializable field.
@serialData	Documents the data written by the **writeObject( )** or **writeExternal( )** methods.
@serialField	Documents an **ObjectStreamField** component.
@since	States the release when a specific change was introduced.
{@summary}	Documents a summary of an item. (Added by JDK 10.)
@throws	Same as **@exception**.
@uses	Documents a service needed by a module.
{@value}	Displays the value of a constant, which must be a **static** field.
@version	Specifies the version of a program element.

Document tags that begin with an "at" sign (@) are called *stand-alone* tags (also called *block* tags), and they must be used on their own line. Tags that begin with a brace, such as **{@code}**, are called *in-line* tags, and they can be used within a larger description. You may also use other, standard HTML tags in a documentation comment. However, some tags, such as headings, should not be used because they disrupt the look of the HTML file produced by **javadoc**.

As it relates to documenting source code, you can use documentation comments to document classes, interfaces, fields, constructors, methods, and modules. In all cases, the documentation comment must immediately precede the item being documented. Some tags, such as **@see**, **@since**, and **@deprecated**, can be used to document any element. Other tags apply only to the relevant elements. Each tag is examined next.

NOTE Documentation comments can also be used for documenting a package and preparing an overview, but the procedures differ from those used to document source code. See the **javadoc** documentation for details on these uses. Beginning with JDK 9, *javadoc* can also document a **module-info.java** file.

@author

The **@author** tag documents the author of a program element. It has the following syntax:

@author *description*

Here, *description* will usually be the name of the author. You will need to specify the **-author** option when executing **javadoc** in order for the **@author** field to be included in the HTML documentation.

{@code}

The **{@code}** tag enables you to embed text, such as a snippet of code, into a comment. That text is then displayed as-is in code font, without any further processing, such as HTML rendering. It has the following syntax:

{@code *code-snippet*}

@deprecated

The **@deprecated** tag specifies that a program element is deprecated. It is recommended that you include **@see** or **{@link}** tags to inform the programmer about available alternatives. The syntax is the following:

@deprecated *description*

Here, *description* is the message that describes the deprecation. The **@deprecated** tag can be used in documentation for fields, methods, constructors, classes, modules, and interfaces.

{@docRoot}

{@docRoot} specifies the path to the root directory of the current documentation.

@exception

The **@exception** tag describes an exception to a method. It has the following syntax:

@exception *exception-name explanation*

Here, the fully qualified name of the exception is specified by *exception-name*, and *explanation* is a string that describes how the exception can occur. The **@exception** tag can only be used in documentation for a method or constructor.

@hidden

The **@hidden** tag prevents an element from appearing in the documentation.

{@index}

The **{@index}** tag specifies an item that will be indexed, and thus found when using the search feature. It has the following syntax:

{ @index *term usage-str* }

Here, *term* is the item (which can be a quoted string) to be indexed. *usage-str* is optional. Thus, in the following **@exception** tag, **{@index}** causes the term "error" to be added to the index:

```
@exception IOException On input {@index error}.
```

Note that the word "error" is still displayed as part of the description. It's just that now it is also indexed. If you include the optional *usage-str*, then that description will be shown in the index and in the search box to indicate how the term is used. For example, **{@index error Serious execution failure}** will show "Serious execution failure" under "error" in the index and in the search box.

{@inheritDoc}

This tag inherits a comment from the immediate superclass.

{@link}

The **{@link}** tag provides an in-line link to additional information. It has the following syntax:

{@link *pkg.class#member text*}

Here, *pkg.class#member* specifies the name of a class or method to which a link is added, and *text* is the string that is displayed.

{@linkplain}

Inserts an in-line link to another topic. The link is displayed in plain-text font.

Otherwise, it is similar to **{@link}**.

{@literal}

The **{@literal}** tag enables you to embed text into a comment. That text is then displayed as-is, without any further processing, such as HTML rendering. It has the following syntax:

{@literal *description*}

Here, *description* is the text that is embedded.

@param

The **@param** tag documents a parameter. It has the following syntax:

@param *parameter-name explanation*

Here, *parameter-name* specifies the name of a parameter. The meaning of that parameter is described by *explanation*. The **@param** tag can be used only in documentation for a method or constructor, or a generic class or interface.

@provides

The **@provides** tag documents a service provided by a module. It has the following syntax:

@provides *type explanation*

Here, *type* specifies a service provider type and *explanation* describes the service provider.

@return

The **@return** tag describes the return value of a method. It has the following syntax:

@return *explanation*

Here, *explanation* describes the type and meaning of the value returned by a

method. The **@return** tag can be used only in documentation for a method.

@see

The **@see** tag provides a reference to additional information. Two commonly used forms are shown here:

@see *anchor*
@see *pkg.class#member text*

In the first form, *anchor* is a link to an absolute or relative URL. In the second form, *pkg.class#member* specifies the name of the item, and *text* is the text displayed for that item. The text parameter is optional, and if not used, then the item specified by *pkg.class#member* is displayed. The member name, too, is optional. Thus, you can specify a reference to a package, class, or interface in addition to a reference to a specific method or field. The name can be fully qualified or partially qualified. However, the dot that precedes the member name (if it exists) must be replaced by a hash character.

@serial

The **@serial** tag defines the comment for a default serializable field. It has the following syntax:

@serial *description*

Here, *description* is the comment for that field.

@serialData

The **@serialData** tag documents the data written by the **writeObject( )** and **writeExternal( )** methods. It has the following syntax:

@serialData *description*

Here, *description* is the comment for that data.

@serialField

For a class that implements **Serializable**, the **@serialField** tag provides comments for an **ObjectStreamField** component. It has the following syntax:

@serialField *name type description*

Here, *name* is the name of the field, *type* is its type, and *description* is the comment for that field.

@since

The **@since** tag states that an element was introduced in a specific release. It has the following syntax:

@since *release*

Here, *release* is a string that designates the release or version in which this feature became available.

{@summary}

The **{@summary}** tag explicitly specifies the summary text that will be used for an item. It must be the first tag in the documentation for the item. It has the following syntax:

@summary *explanation*

Here, *explanation* provides a summary of the tagged item, which can span multiple lines. This tag was added by JDK 10. Without the use of **{@summary}**, the first line in an item's documentation comment is used as the summary.

@throws

The **@throws** tag has the same meaning as the **@exception** tag.

@uses

The **@uses** tag documents a service provider needed by a module. It has the following syntax:

@uses *type explanation*

Here, *type* specifies a service provider type and *explanation* describes the service.

{@value}

{@value} has two forms. The first displays the value of the constant that it precedes, which must be a **static** field. It has this form:

{@value}

The second form displays the value of a specified **static** field. It has this form:

{@value *pkg.class#field*}

Here, *pkg.class#field* specifies the name of the **static** field.

@version

The **@version** tag specifies the version of a program element. It has the following syntax:

@version *info*

Here, *info* is a string that contains version information, typically a version number, such as 2.2. You will need to specify the **-version** option when executing **javadoc** in order for the **@version** field to be included in the HTML documentation.

The General Form of a Documentation Comment

After the beginning **/****, the first line or lines become the main description of your class, interface, field, constructor, method, or module. After that, you can include one or more of the various **@** tags. Each **@** tag must start at the beginning of a new line or follow one or more asterisks (*****) that are at the start of a line. Multiple tags of the same type should be grouped together. For example,

if you have three **@see** tags, put them one after the other. In-line tags (those that begin with a brace) can be used within any description.

Here is an example of a documentation comment for a class:

```
/**
 * This class draws a bar chart.
 * @author Herbert Schildt
 * @version 3.2
*/
```

What javadoc Outputs

The **javadoc** program takes as input your Java program's source file and outputs several HTML files that contain the program's documentation. Information about each class will be in its own HTML file. **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated. Beginning with JDK 9, a search box feature is also included.

An Example that Uses Documentation Comments

Following is a sample program that uses documentation comments. Notice the way each comment immediately precedes the item that it describes. After being processed by **javadoc**, the documentation about the **SquareNum** class will be found in **SquareNum.html**.

```
import java.io.*;
/**
 * This class demonstrates documentation comments.
 * @author Herbert Schildt
 * @version 1.2
*/
public class SquareNum {
  /**
   * This method returns the square of num.
   * This is a multiline description. You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
  */
  public double square(double num) {
    return num * num;
  }

  /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
  */
  public double getNumber() throws IOException {
    // create a BufferedReader using System.in
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;
```

```
    str = inData.readLine();
    return (new Double(str)).doubleValue();
  }
  /**
   * This method demonstrates square().
   * @param args Unused.
   * @exception IOException On input error.
   * @see IOException
  */

  public static void main(String args[])
    throws IOException
  {
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Enter value to be squared: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Squared value is " + val);
  }
}
```

APPENDIX

Introducing JShell

Beginning with JDK 9, Java has included a tool called JShell. It provides an interactive environment that enables you to quickly and easily experiment with Java code. JShell implements what is referred to as *read-evaluate-print loop* (REPL) execution. Using this mechanism, you are prompted to enter a fragment of code. This fragment is then read and evaluated. Next, JShell displays output related to the code, such as the output produced by a **println( )** statement, the result of an expression, or the current value of a variable. JShell then prompts for the next piece of code, and the process continues (i.e., loops). In the language of JShell, each code sequence you enter is called a *snippet*.

A key point to understand about JShell is that you do not need to enter a complete Java program to use it. Each snippet you enter is simply evaluated as you enter it. This is possible because JShell handles many of the details associated with a Java program for you automatically. This lets you concentrate on a specific feature without having to write a complete program, which makes JShell especially helpful when you are first learning Java.

As you might expect, JShell can also be useful to experienced programmers. Because JShell stores state information, it is possible to enter multiline code sequences and run them inside JShell. This makes JShell quite useful when you need to prototype a concept because it lets you interactively experiment with your code without having to develop and compile a complete program.

This appendix introduces JShell and explores several of its key features, with the primary focus being on those features most useful to beginning Java programmers.

JShell Basics

JShell is a command-line tool. Thus, it runs in a command-prompt window. To start a JShell session, execute **jshell** from the command line. After doing so, you will see the JShell prompt:

```
jshell>
```

When this prompt is displayed, you can enter a code snippet or a JShell command.

In its simplest form, JShell lets you enter an individual statement and immediately see the result. To begin, think back to the first example Java program in this book. It is shown again here:

```
class Example {
  // Your program begins with a call to main().
  public static void main(String args[]) {
    System.out.println("This is a simple Java program.");
  }
}
```

In this program, only the **println( )** statement actually performs an action, which is displaying its message on the screen. The rest of the code simply provides the required class and method declarations. In JShell, it is not necessary to explicitly specify the class or method in order to execute the **println( )** statement. JShell can execute it directly on its own. To see how, enter the following line at the JShell prompt:

```
System.out.println("This is a simple Java program.");
```

Then, press enter. This output is displayed:

```
This is a simple Java program.

jshell>
```

As you can see, the call to **println( )** is evaluated and its string argument is output. Then, the prompt is redisplayed.

Before moving on, it is useful to explain why JShell can execute a single statement, such as the call to **println( )**, when the Java compiler, **javac**, requires a complete program. JShell is able to evaluate a single statement because JShell automatically provides the necessary program framework for you, behind the scenes. This consists of a *synthetic class* and a *synthetic method*. Thus, in this case, the **println( )** statement is embedded in a synthetic method that is part of a synthetic class. As a result, the preceding code is still part of a valid Java program even though you don’t see all of the details. This approach provides a very fast and convenient way to experiment with Java code.

Next, let’s look at how variables are supported. In JShell, you can declare a

variable, assign the variable a value, and use it in any valid expressions. For example, enter the following line at the prompt:

```
int count;
```

After doing so you will see the following response:

```
count ==> 0
```

This indicates that **count** has been added to the synthetic class and initialized to zero. Furthermore, it has been added as a **static** variable of the synthetic class.

Next, give **count** the value 10 by entering this statement:

```
count = 10;
```

You will see this response:

```
count ==> 10
```

As you can see, **count**'s value is now 10. Because **count** is **static**, it can be used without reference to an object.

Now that **count** has been declared, it can be used in an expression. For example, enter this **println( )** statement:

```
System.out.println("Reciprocal of count: " + 1.0 / count);
```

JShell responds with:

```
Reciprocal of count: 0.1
```

Here, the result of the expression **1.0 / count** is 0.1 because **count** was previously assigned the value 10.

In addition to demonstrating the use of a variable, the preceding example illustrates another important aspect of JShell: it maintains state information. In this case, **count** is assigned the value 10 in one statement and then this value is used in the expression **1.0 / count** in the subsequent call to **println( )** in a second statement. Between these two statements, JShell stores **count**'s value. In general, JShell maintains the current state and effect of the code snippets that you enter. This lets you experiment with larger code fragments that span multiple lines.

Before moving on, let's try one more example. In this case, we will create a **for** loop that uses the **count** variable. Begin by entering this line at the prompt:

```
for(count = 0; count < 5; count++)
```

At this point, JShell responds with the following prompt:

```
...>
```

This indicates that additional code is required to finish the statement. In this case, the target of the **for** loop must be provided. Enter the following:

```
System.out.println(count);
```

After entering this line, the **for** statement is complete and both lines are executed. You will see the following output:

```
0
1
2
3
4
```

In addition to statements and variable declarations, JShell lets you declare classes and methods, and use import statements. Examples are shown in the following sections. One other point: Any code that is valid for JShell will also be valid for compilation by **javac**, assuming the necessary framework is provided to create a complete program. Thus, if a code fragment can be executed by JShell, then that fragment represents valid Java code. In other words, JShell code *is* Java code.

List, Edit, and Rerun Code

JShell supports a large number of commands that let you control the operation of JShell. At this point, three are of particular interest because they let you list the code that you have entered, edit a line of code, and rerun a code snippet. As the subsequent examples become longer, you will find these commands to be very helpful.

In JShell, all commands start with a **/** followed by the command. Perhaps the most commonly used command is **/list**, which lists the code that you have entered. Assuming that you have followed along with the examples shown in the preceding section, you can list your code by entering **/list** at this time. Your JShell session will respond with a numbered list of the snippets you entered. Pay special attention to the entry that shows the **for** loop. Although it consists of two

lines, it constitutes one statement. Thus, only one snippet number is used. In the language of JShell, the snippet numbers are referred to as *snippet IDs*. In addition to the basic form of **/list** just shown, other forms are supported, including those that let you list specific snippets by name or number. For example, you can list the **count** declaration by using **/list count**.

You can edit a snippet by using the **/edit** command. This command causes an edit window to open in which you can modify your code. Here are three forms of the **/edit** command that you will find helpful at this time. First, if you specify **/edit** by itself, the edit window contains all of the lines you have entered and lets you edit any part of it. Second, you can specify a specific snippet to edit by using **/edit** *n*, where *n* specifies the snippet's number. For example, to edit snippet 3, use **/edit 3**. Finally, you can specify a named element, such as a variable. For example, to change the value of **count**, use **/edit count**.

As you have seen, JShell executes code as you enter it. However, you can also rerun what you have entered. To rerun the last fragment that you entered, use **/!**. To rerun a specific snippet, specify its number using this form: /*n*, where *n* specifies the snippet to run. For example, to rerun the fourth snippet, enter **/4**. You can rerun a snippet by specifying its position relative to the current fragment by use of a negative offset. For example, to rerun a fragment that is three snippets before the current one, use **/-3**.

Before moving on, it is helpful to point out that several commands, including those just shown, allow you to specify a list of names or numbers. For example, to edit lines 2 and 4, you could use **/edit 2 4**. For recent versions of JShell, several commands allow you specify a range of snippets. These include the **/list**, **/edit**, and /*n* commands just described. For example, to list snippets 4 through 6, you would use **/list 4-6**.

There is one other important command that you need to know about now: **/exit**. This terminates JShell.

Add a Method

As explained in Chapter 6, methods occur within classes. However, when using JShell it is possible to experiment with a method without having to *explicitly* declare it within a class. As mentioned earlier, this is because JShell automatically wraps code fragments within a synthetic class. As a result, you can easily and quickly write a method without having to provide a class framework. You can also call the method without having to create an object. This feature of JShell is especially beneficial when learning the basics of methods in Java or

when prototyping new code. To understand the process, we will work through an example.

To begin, start a new JShell session and enter the following method at the prompt:

```
double reciprocal(double val) {
  return 1.0/val;
}
```

This creates a method that returns the reciprocal of its argument. After you enter this, JShell responds with the following:

```
| created method reciprocal(double)
```

This indicates the method has been added to JShell's synthetic class and is ready for use.

To call **reciprocal( )**, simply specify its name, without any object or class reference. For example, try this:

```
System.out.println(reciprocal(4.0));
```

JShell responds by displaying 0.25.

You might be wondering why you can call **reciprocal( )** without using the dot operator and an object reference. Here is the answer. When you create a stand-alone method in JShell, such as **reciprocal( )**, JShell automatically makes that method a **static** member of the synthetic class. As you know from Chapter 7, **static** methods are called relative to their class, not on a specific object. So, no object is required. This is similar to the way that stand-alone variables become **static** variables of the synthetic class, as described earlier.

Another important aspect of JShell is its support for a *forward reference* inside a method. This feature lets one method call another method, even if the second method has not yet been defined. This enables you to enter a method that depends on another method without having to worry about which one you enter first. Here is a simple example. Enter this line in JShell:

```
void myMeth() { myMeth2(); }
```

JShell responds with the following:

```
|  created method myMeth(), however, it cannot be invoked until myMeth2()
   is declared
```

As you can see, JShell knows that **myMeth2( )** has not yet been declared, but it still lets you define **myMeth( )**. As you would expect, if you try to call **myMeth( )** at this time, you will see an error message since **myMeth2( )** is not yet defined, but you are still able to enter the code for **myMeth( )**.

Next, define **myMeth2( )** like this:

```
void myMeth2() { System.out.println("JShell is powerful."); }
```

Now that **myMeth2( )** has been defined, you can call **myMeth( )**.

In addition to its use in a method, you can use a forward reference in a field initializer in a class.

Create a Class

Although JShell automatically supplies a synthetic class that wraps code snippets, you can also create your own class in JShell. Furthermore, you can instantiate objects of your class. This allows you to experiment with classes inside JShell's interactive environment. The following example illustrates the process.

Start a new JShell session and enter the following class, line by line:

```
class MyClass {
  double v;

  MyClass(double d) { v = d; }

  // Return the reciprocal of v.
  double reciprocal() { return 1.0 / v; }
}
```

When you finish entering the code, JShell will respond with

```
| created class MyClass
```

Now that you have added **MyClass**, you can use it. For example, you can create a **MyClass** object with the following line:

```
MyClass ob = new MyClass(10.0);
```

JShell will respond by telling you that it added **ob** as a variable of type **MyClass**.

Next, try the following line:

```
System.out.println(ob.reciprocal());
```

JShell responds by displaying the value 0.1.

As a point of interest, when you add a class to JShell, it becomes a **static** nested member of a synthetic class.

Use an Interface

Interfaces are supported by JShell in the same way as classes. Therefore, you can declare an interface and implement it by a class within JShell. Let's work through a simple example. Before beginning, start a new JShell session.

The interface that we will use declares a method called **isLegalVal( )** that is used to determine if a value is valid for some purpose. It returns **true** if the value is legal and **false** otherwise. Of course, what constitutes a legal value will be determined by each class that implements the interface. Begin by entering the following interface into JShell:

```
interface MyIF {
  boolean isLegalVal(double v);
}
```

JShell responds with

```
| created interface MyIf
```

Next, enter the following class, which implements MyIF:

```
class MyClass implements MyIF {

  double start;
  double end;

  MyClass(double a, double b) { start = a; end = b; }

  // Determine if v is within the range start to end, inclusive.
  public boolean isLegalVal(double v) {
    if((v >= start) && (v <= end)) return true;
    return false;
  }

}
```

JShell responds with

```
| created class MyClass
```

Notice that **MyClass** implements **isLegalVal( )** by determining if the value **v** is within the range (inclusive) of the values in the **MyClass** instance variables **start** and **end**.

Now that both **MyIF** and **MyClass** have been added, you can create a **MyClass** object and call **isLegalVal( )** on it, as shown here:

```
MyClass ob = new MyClass(0.0, 10.0);

System.out.println(ob.isLegalVal(5.0));
```

In this case, the value **true** is displayed because 5 is within the range 0 through 10.

Because **MyIF** has been added to JShell, you can also create a reference to an object of type **MyIF**. For example, the following is also valid code:

```
MyIF ob2 = new MyClass(1.0, 3.0);
boolean result = ob2.isLegalVal(1.1);
```

In this case, the value of **result** will be **true** and will be reported as such by JShell.

One other point: enumerations and annotations are supported in JShell in the same way as classes and interfaces.

Evaluate Expressions and Use Built-in Variables

JShell includes the ability to directly evaluate an expression without it needing to be part of a full Java statement. This is especially useful when you are experimenting with code and don't need to execute the expression in a larger context. Here is a simple example. Using a new JShell session, enter the following at the prompt:

```
3.0 / 16.0
```

JShell responds with:

```
$1 ==> 0.1875
```

As you can see, the result of the expression is computed and displayed. However, note that this value is also assigned to a temporary variable called **$1**. In general, each time an expression is evaluated directly, its result is stored in a temporary variable of the proper type. Temporary variable names all begin with a **$** followed by a number, which is increased each time a new temporary variable is needed. You can use these temporary variables like any other variable. For example, the following displays the value of **$1**, which is 0.1875 in this case.

```
System.out.println($1);
```

Here is another example:

```
double v = $1 * 2;
```

Here, the value **$1** times 2 is assigned to **v**. Thus, **v** will contain 0.375.

You can change the value of a temporary variable. For example, this reverses the sign of **$1**:

```
$1 = -$1
```

JShell responds with

```
$1 ==> -0.1875
```

Expressions are not limited to numeric values. For example, here is one that

concatenates a **String** with the value returned by **Math.abs($1)**.

```
"The absolute value of $1 is " + Math.abs($1)
```

This results in a temporary variable that contains the string

```
The absolute value of $1 is 0.1875
```

Importing Packages

As described in Chapter 9, an **import** statement is used to bring members of a package into view. Furthermore, any time you use a package other than **java.lang**, you must import it. The situation is much the same in JShell except that by default, JShell imports several commonly used packages automatically. These include **java.io** and **java.util**, among several others. Since these packages are already imported, no explicit **import** statement is required to use them.

For example, because **java.io** is automatically imported, the following statement can be entered:

```
FileInputStream fin = new FileInputStream("myfile.txt");
```

Recall that **FileInputStream** is packaged in **java.io**. Since **java.io** is automatically imported, it can be used without having to include an explicit **import** statement. Assuming that you actually have a file called **myfile.txt** in the current directory, JShell will respond by adding the variable **fin** and opening the file. You can then read and display the file by entering these statements:

```
int i;
do {
  i = fin.read();
  if(i != -1) System.out.print((char) i);
} while(i != -1);
```

This is the same basic code that was discussed in Chapter 13, but no explicit **import java.io** statement is required.

Keep in mind that JShell automatically imports only a handful of packages. If you want to use a package not automatically imported by JShell, then you must explicitly import it as you do with a normal Java program. One other point: you can see a list of the current imports by using the **/imports** command.

Exceptions

In the I/O example shown in the preceding section on imports, the code snippets also illustrate another very important aspect of JShell. Notice that there are no **try**/**catch** blocks that handle I/O exceptions. If you look back at the similar code in Chapter 13, the code that opens the file catches a **FileNotFoundException**, and the code that reads the file watches for an **IOException**. The reason that you don't need to catch these exceptions in the snippets shown earlier is because JShell automatically handles them for you. More generally, JShell will automatically handle checked exceptions in many cases.

Some More JShell Commands

In addition to the commands discussed earlier, JShell supports several others. One command that you will want to try immediately is **/help**. It displays a list of the commands. You can also use **/?** to obtain help. Some of the more commonly used commands are examined here.

You can reset JShell by using the **/reset** command. This is especially useful when you want to change to a new project. By use of **/reset** you avoid the need to exit and then restart JShell. Be aware, however, that **/reset** resets the entire JShell environment, so all state information is lost.

You can save a session by using **/save**. Its simplest form is shown here:

/save *filename*

Here, *filename* specifies the name of the file to save into. By default, **/save** saves your current source code, but it supports several options, of which two are of particular interest. By specifying **-all** you save all lines that you enter, including those that you entered incorrectly. You can use the **-history** option to save your session history (i.e., the list of the commands that you have entered).

You can load a saved session by using **/open**. Its form is shown next:

/open *filename*

Here, *filename* is the name of the file to load.

JShell provides several commands that let you list various elements of your work. They are shown here:

Command	Effect
/types	Shows classes, interfaces, and enums.
/imports	Shows import statements.
/methods	Shows methods.
/vars	Shows variables.

For example, if you entered the following lines:

```
int start = 0;
int end = 10;
int count = 5;
```

and then entered the **/vars** command, you would see

```
| int start = 0;
| int end = 10;
| int count = 5;
```

Another often useful command is **/history**. It lets you view the history of the current session. The history contains a list of what you have typed at the command prompt.

Exploring JShell Further

The best way to get proficient with JShell is to work with it. Try entering several different Java constructs and watching the way that JShell responds. As you experiment with JShell, you will find the usage patterns that work best for you. This will enable you to find effective ways to integrate JShell into your learning or development process. Also, keep in mind that JShell is not just for beginners. It also excels when prototyping code. Thus, even if you are an experienced pro, you will still find JShell helpful whenever you need to explore new areas.

Simply put: JShell is an important tool that further enhances the overall Java development experience.

APPENDIX

Compile and Run Simple Single-File Programs in One Step

In Chapter 2, you were shown how to compile a Java program into bytecode using the **javac** compiler and then run the resulting **.class** file(s) using the Java launcher **java**. This is how Java programs have been compiled and run since Java's beginning, and it is the method that you will use when developing applications. However, beginning with JDK 11, it is possible to compile and run some types of simple Java programs directly from the source file without having to first invoke **javac**. To do this, pass the name of the source file, using the **.java** file extension, to **java**. This causes **java** to automatically invoke the compiler and execute the program.

For example, the following automatically compiles and runs the first example in this book:

```
java Example.java
```

In this case, the **Example** class is compiled and then run in a single step. There is no need to use **javac**. Be aware, however, that no **.class** file is created. Instead, the compilation is done behind the scenes. As a result, to rerun the program, you must execute the source file again. You can't execute its **.class** file because one won't be created. One other point: if you have already used **javac** to compile **Example.java**, then you must erase the resulting **.class** file **Example.class** before trying the source-file launch feature. When running a source file, there cannot already be a **.class** file with the same name as the class that contains **main( )** in the source file.

One use of the source-file launch capability is to facilitate the use of Java programs in script files. It can also be useful for short one-time-use programs. In some cases, it makes it a little easier to run simple example programs when you are experimenting with Java. It is not, however, a general-purpose substitute for Java's normal compilation/execution process.

Although this new ability to launch a Java program directly from its source file is appealing, it comes with several restrictions. First, the entire program must be contained in a single source file. However, most real-world programs use

multiple source files. Second, it will always execute the first class it finds in the file, and that class must contain a **main( )** method. If the first class in the file does not contain a **main( )** method, the launch will fail. This means that you must follow a strict organization for your code, even if you would prefer to organize it otherwise. Third, because no **.class** files are created, using **java** to run a single-file program does not result in a class file that can be reused, possibly by other programs. Finally, as a general rule, if there is already a **.class** file that has the same name as a class in the source file, then the single-file launch will fail. As a result of these restrictions, using **java** to run a single-file source program can be useful, but it constitutes what is, essentially, a special-case technique.

As it relates to this book, it is possible to use the single source-file launch feature to try many of the examples; just be sure that the class with the **main( )** method is first in your file. That said, it is not, however, applicable or appropriate in all cases. Furthermore, the discussions (and many of the examples) in the book assume that you are using the normal compilation process of invoking **javac** to compile a source file into bytecode and then using **java** to run that bytecode. This is the mechanism used for real-world development, and understanding this process is an important part of learning Java. It is imperative that you are thoroughly familiar with it. For these reasons, when trying the examples in this book, it is strongly recommended that in all cases you use the normal approach to compiling and running a Java program. Doing so ensures that you have a solid foundation in the way Java works. Of course, you might find it fun to experiment with the single source-file launch option!

Index

Please note that index links point to page beginnings from the print edition. Locations are approximate in e-readers, and you may need to page down one or more times after clicking a link to get to the indexed material.

A

B

C

D

E

F

G

H

I