The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

Instead of relying on an enumeration to cycle through the objects (as the preceding program does), you can use an iterator. For example, the following iterator-based code can be substituted into the program:

```
// Use an iterator to display contents.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
  System.out.print(vItr.next() + " ");
System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:

```
// Use an enhanced for loop to display contents
System.out.println("\nElements in vector:");
for(int i : v)
  System.out.print(i + " ");

System.out.println();
```

Because the **Enumeration** interface is not recommended for new code, you will usually use an iterator or a for-each **for** loop to enumerate the contents of a vector. Of course, legacy code will employ **Enumeration**. Fortunately, enumerations and iterators work in nearly the same manner.

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

class Stack<E>

Here, **E** specifies the type of element stored in the stack.

Stack includes all the methods defined by **Vector** and adds several of its own, shown in Table 18-17.

Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
E peek( )	Returns the element on the top of the stack, but does not remove it.
E pop( )	Returns the element on the top of the stack, removing it in the process.
E push(E *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

Table 18-17 The Methods Defined by **Stack**

To put an object on the top of the stack, call **push( )**. To remove and return the top element, call **pop( )**. You can use **peek( )** to return, but not remove, the top object. An **EmptyStackException** is thrown if you call **pop( )** or **peek( )** when the invoking stack is empty. The **empty( )** method returns **true** if nothing is on the stack. The **search( )** method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
  static void showpush(Stack<Integer> st, int a) {
    st.push(a);
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack<Integer> st) {
    System.out.print("pop -> ");
    Integer a = st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[]) {
    Stack<Integer> st = new Stack<Integer>();

    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }
  }
}
```

The following is the output produced by the program; notice how the exception handler for **EmptyStackException** is caught so that you can gracefully handle a stack underflow:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

One other point: although **Stack** is not deprecated, **ArrayDeque** is a better choice.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is discussed here.

With the advent of JDK 5, **Dictionary** was made generic. It is declared as shown here:

class Dictionary<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 18-18.

To add a key and a value, use the **put( )** method. Use **get( )** to retrieve the value of a given key. The keys and values can each be returned as an **Enumeration** by the **keys( )** and **elements( )** methods, respectively. The **size( )** method returns the number of key/value pairs stored in a dictionary, and **isEmpty( )** returns **true** when the dictionary is empty. You can use the **remove( )** method to delete a key/value pair.

REMEMBER The **Dictionary** class is obsolete. You should implement the **Map** interface to obtain key/value storage functionality.

Method	Purpose
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

Table 18-18 The Abstract Methods Defined by **Dictionary**

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a **Dictionary**. However, with the advent of collections, **Hashtable** was reengineered to also implement the **Map** interface. Thus, **Hashtable** is integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

```
class Hashtable<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

A hash table can only store objects that override the **hashCode( )** and **equals( )** methods that are defined by **Object**. The **hashCode( )** method must compute and return the hash code for the object. Of course, **equals( )** compares two objects. Fortunately, many of Java's built-in classes already implement the **hashCode( )** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode( )** and **equals( )**.

The **Hashtable** constructors are shown here:

```
Hashtable( )
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

The first version is the default constructor. The second version creates a hash table that has an initial size specified by *size*. (The default size is 11.) The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in *m*. The default load factor of 0.75 is used.

In addition to the methods defined by the **Map** interface, which **Hashtable** now implements, **Hashtable** defines the legacy methods listed in Table 18-19. Several methods throw **NullPointerException** if an attempt is made to use a **null** key or value.

Method	Description
void clear( )	Resets and empties the hash table.
Object clone( )	Returns a duplicate of the invoking object.
boolean contains(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
boolean containsKey(Object *key*)	Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found.
boolean containsValue(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
Enumeration<V> elements( )	Returns an enumeration of the values contained in the hash table.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the hash table is empty; returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the hash table.
V put(K *key*, V *value*)	Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table.
void rehash( )	Increases the size of the hash table and rehashes all of its keys.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
int size( )	Returns the number of entries in the hash table.
String toString( )	Returns the string equivalent of a hash table.

Table 18-19 The Legacy Methods Defined by **Hashtable**

The following example reworks the bank account program, shown earlier, so that it uses a **Hashtable** to store the names of bank depositors and their current balances:

```
// Demonstrate a Hashtable.
import java.util.*;

class HTDemo {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    Enumeration<String> names;
    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable.
    names = balance.keys();
    while(names.hasMoreElements()) {
      str = names.nextElement();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Deposit 1,000 into John Doe's account.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

The output from this program is shown here:

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe's new balance: 4434.34
```

One important point: Like the map classes, **Hashtable** does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of **balance**. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by **Map**, such as **entrySet( )** or **keySet( )**. For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced **for** loop. Here is a reworked version of the program that shows this technique:

```
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable.
    // First, get a set view of the keys.
    Set<String> set = balance.keySet();

    // Get an iterator.
    Iterator<String> itr = set.iterator();
    while(itr.hasNext()) {
      str = itr.next();
      System.out.println(str + ": " +
                         balance.get(str));

    }

    System.out.println();

    // Deposit 1,000 into John Doe's account.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

Properties

Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**. The **Properties** class is used by some other Java classes. For example, it is the type of object returned by **System.getProperties( )** when obtaining environmental values. Although the **Properties** class, itself, is not generic, several of its methods are.

Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties** defines these constructors:

Properties( )
Properties(Properties *propDefault*)

The first version creates a **Properties** object that has no default values. The second creates an object that uses *propDefault* for its default values. In both cases, the property list is empty.

In addition to the methods that **Properties** inherits from **Hashtable**, **Properties** defines the methods listed in Table 18-20. **Properties** also contains one deprecated method: **save( )**. This was replaced by **store( )** because **save( )** did not handle errors correctly.

Method	Description
String getProperty(String *key*)	Returns the value associated with *key*. A **null** object is returned if *key* is neither in the list nor in the default property list.
String getProperty(String *key*, String *defaultProperty*)	Returns the value associated with *key*. *defaultProperty* is returned if *key* is neither in the list nor in the default property list.
void list(PrintStream *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void list(PrintWriter *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void load(InputStream *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*.
void load(Reader *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*.
void loadFromXML(InputStream *streamIn*) throws IOException, InvalidPropertiesFormatException	Inputs a property list from an XML document linked to *streamIn*.
Enumeration<?> propertyNames( )	Returns an enumeration of the keys. This includes those keys found in the default property list, too.
Object setProperty(String *key*, String *value*)	Associates *value* with *key*. Returns the previous value associated with *key*, or returns **null** if no such association exists.
void store(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
void store(Writer *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
void storeToXML(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the XML document linked to *streamOut*.

Table 18-20 The Methods Defined by **Properties**

Method	Description
void storeToXML(OutputStream *streamOut*, String *description*, String *enc*)	The property list and the string specified by *description* is written to the XML document linked to *streamOut* using the specified character encoding.
Set<String> stringPropertyNames( )	Returns a set of keys.

Table 18-20 The Methods Defined by **Properties** *(continued)*

One useful capability of the **Properties** class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the **getProperty( )** method—such as **getProperty("name" ,"default value")**. If the "name" value is not found, then "default value" is returned. When you construct a **Properties** object, you can pass another instance of **Properties** to be used as the default properties for the new instance. In this case, if you call **getProperty("foo")** on a given **Properties** object, and "foo" does not exist, Java looks for "foo" in the default **Properties** object. This allows for arbitrary nesting of levels of default properties.

The following example demonstrates **Properties**. It creates a property list in which the keys are the names of states and the values are the names of their capitals. Notice that the attempt to find the capital for Florida includes a default value.

```
// Demonstrate a Property list.
import java.util.*;

class PropDemo {
  public static void main(String args[]) {
    Properties capitals = new Properties();

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Get a set-view of the keys.
    Set<?> states = capitals.keySet();

    // Show all of the states and capitals.
    for(Object name : states)
      System.out.println("The capital of " +
                         name + " is " +
                         capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Look for state not in list -- specify default.
    String str = capitals.getProperty("Florida", "Not Found");
    System.out.println("The capital of Florida is " + str + ".");
  }
}
```

The output from this program is shown here:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

Since Florida is not in the list, the default value is used.

Although it is perfectly valid to use a default value when you call **getProperty( )**, as the preceding example shows, there is a better way of handling default values for most applications of property lists. For greater flexibility, specify a default property list when constructing a **Properties** object. The default list will be searched if the desired key is not found in the main list. For example, the following is a slightly reworked version of the preceding program, with a default list of states specified. Now, when Florida is sought, it will be found in the default list:

```
// Use a default property list.
import java.util.*;

class PropDemoDef {
  public static void main(String args[]) {
    Properties defList = new Properties();
    defList.put("Florida", "Tallahassee");
    defList.put("Wisconsin", "Madison");

    Properties capitals = new Properties(defList);

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Get a set-view of the keys.
    Set<?> states = capitals.keySet();

    // Show all of the states and capitals.
    for(Object name : states)
      System.out.println("The capital of " +
                         name + " is " +
                         capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Florida will now be found in the default list.
    String str = capitals.getProperty("Florida");
    System.out.println("The capital of Florida is "
                       + str + ".");
  }
}
```

Using store( ) and load( )

One of the most useful aspects of **Properties** is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store( )** and **load( )** methods. At any time, you can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases. For example, the following program uses a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person's number, you enter his or her name. The program uses the **store( )** and **load( )** methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called **phonebook.dat**. If this file exists, the list is loaded. You can then add to the list. If you do, the new list is saved when you terminate the program. Notice how little code is required to implement a small, but functional, computerized phone book.

```
/* A simple telephone number database that uses
   a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
  public static void main(String args[])
    throws IOException
  {
    Properties ht = new Properties();
    BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));
    String name, number;
    FileInputStream fin = null;
    boolean changed = false;

    // Try to open phonebook.dat file.
    try {
      fin = new FileInputStream("phonebook.dat");
    } catch(FileNotFoundException e) {
      // ignore missing file
    }

    /* If phonebook file already exists,
       load existing telephone numbers. */
    try {
      if(fin != null) {
        ht.load(fin);
        fin.close();
      }
    } catch(IOException e) {
      System.out.println("Error reading file.");
    }

    // Let user enter new names and numbers.
    do {
      System.out.println("Enter new name" +
                         " ('quit' to stop): ");
        name = br.readLine();
        if(name.equals("quit")) continue;

        System.out.println("Enter number: ");
        number = br.readLine();

        ht.put(name, number);
        changed = true;
      } while(!name.equals("quit"));

      // If phone book data has changed, save it.
      if(changed) {
        FileOutputStream fout = new FileOutputStream("phonebook.dat");

        ht.store(fout, "Telephone Book");
        fout.close();
      }

      // Look up numbers given a name.
      do {
        System.out.println("Enter name to find" +
                           " ('quit' to quit): ");
        name = br.readLine();
        if(name.equals("quit")) continue;

        number = (String) ht.get(name);
        System.out.println(number);
      } while(!name.equals("quit"));
    }
  }
```

Parting Thoughts on Collections

The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming's most common tasks. Consider using a collection the next time that you need to store and retrieve information. Remember, collections need not be reserved for only the "large jobs," such as corporate databases, mailing lists, or inventory systems. They are also effective when applied to smaller jobs. For example, a **TreeMap** might make an excellent collection to hold the directory structure of a set of files. A **TreeSet** could be quite useful for storing project-management information. Frankly, the types of problems that will benefit from a collections-based solution are limited only by your imagination. One last point: In Chapter 29, the new stream API is discussed. Because streams are now integrated with collections, consider using a stream when operating on a collection.

CHAPTER

19 java.util Part 2: More Utility Classes

This chapter continues our discussion of **java.util** by examining those classes and interfaces that are not part of the Collections Framework. These include classes that tokenize strings, work with dates, compute random numbers, bundle resources, and observe events. Also covered are the **Formatter** and **Scanner** classes which make it easy to write and read formatted data, and the new **Optional** class, which makes it easier to handle situations in which a value may be absent. Finally, the subpackages of **java.util** are summarized at the end of this chapter. Of particular interest is **java.util.function**, which defines several standard functional interfaces.

StringTokenizer

The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens*, which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner*. **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using **StringTokenizer**.

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, **",;:"** sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, form feed, newline, and carriage return.

The **StringTokenizer** constructors are shown here:

StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimiters*)
StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken( )** method is used to extract consecutive tokens. The **hasMoreTokens( )** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements( )** and **nextElement( )** methods are also implemented, and they act the same as **hasMoreTokens( )** and **nextToken( )**, respectively. The **StringTokenizer** methods are shown in Table 19-1.

Here is an example that creates a **StringTokenizer** to parse "key=value" pairs. Consecutive sets of "key=value" pairs are separated by a semicolon.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
    "author=Schildt;" +
    "publisher=McGraw-Hill;" +
    "copyright=2014";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) {
      String key = st.nextToken();
      String val = st.nextToken();
      System.out.println(key + "\t" + val);
    }
  }
}
```

The output from this program is shown here:

```
title  Java: The Complete Reference
author  Schildt
publisher  McGraw-Hill
copyright  2014
```

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

Table 19-1 The Methods Defined by **StringTokenizer**

BitSet

A **BitSet** class creates a special type of array that holds bit values in the form of **boolean** values. This array can increase in size as needed. This makes it similar to a vector of bits. The **BitSet** constructors are shown here:

BitSet( )
BitSet(int *size*)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to **false**.

BitSet defines the methods listed in Table 19-2.

Method	Description
void and(BitSet *bitSet*)	ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object.
void andNot(BitSet *bitSet*)	For each set bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void clear( )	Zeros all bits.
void clear(int *index*)	Zeros the bit specified by *index*.
void clear(int *startIndex*, int *endIndex*)	Zeros the bits from *startIndex* to *endIndex*–1.
Object clone( )	Duplicates the invoking **BitSet** object.
boolean equals(Object *bitSet*)	Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**.
void flip(int *index*)	Reverses the bit specified by *index*.
void flip(int *startIndex*, int *endIndex*)	Reverses the bits from *startIndex* to *endIndex*–1.
boolean get(int *index*)	Returns the current state of the bit at the specified index.
BitSet get(int *startIndex*, int *endIndex*)	Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed.
int hashCode( )	Returns the hash code for the invoking object.
boolean intersects(BitSet *bitSet*)	Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are set.
boolean isEmpty( )	Returns **true** if all bits in the invoking object are cleared.
int length( )	Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last set bit.
int nextClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit), starting from the index specified by *startIndex*.

Table 19-2 The Methods Defined by **BitSet**

Method	Description
int nextSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned.
void or(BitSet *bitSet*)	ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.
int previousClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit) at or prior to the index specified by *startIndex*. If no cleared bit is found, –1 is returned.
int previousSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit) at or prior to the index specified by *startIndex*. If no set bit is found, –1 is returned.
void set(int *index*)	Sets the bit specified by *index*.
void set(int *index*, boolean *v*)	Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit; **false** clears the bit.
void set(int *startIndex*, int *endIndex*)	Sets the bits from *startIndex* to *endIndex*–1.
void set(int *startIndex*, int *endIndex*, boolean *v*)	Sets the bits from *startIndex* to *endIndex*–1 to the value passed in *v*. **true** sets the bits; **false** clears the bits.
int size( )	Returns the number of bits in the invoking **BitSet** object.
IntStream stream( )	Returns a stream that contains the bit positions, from low to high, that have set bits. (Added by JDK 8.)
byte[ ] toByteArray( )	Returns a **byte** array that contains the invoking **BitSet** object.
long[ ] toLongArray( )	Returns a **long** array that contains the invoking **BitSet** object.
String toString( )	Returns the string equivalent of the invoking **BitSet** object.
static BitSet valueOf(byte[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(ByteBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(long[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(LongBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
void xor(BitSet *bitSet*)	XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.

Table 19-2 The Methods Defined by **BitSet** *(continued)*

Here is an example that demonstrates **BitSet**:

```
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);

    // AND bits
    bits2.and(bits1);
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

The output from this program is shown here. When **toString( )** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position. Cleared bits are not shown.

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Optional, OptionalDouble, OptionalInt, and OptionalLong

JDK 8 adds classes called **Optional**, **OptionalDouble**, **OptionalInt**, and **OptionalLong** that offer a way to handle situations in which a value may or may not be present. In the past, you would normally use the value **null** to indicate that no value is present. However, this can lead to null pointer exceptions if an attempt is made to dereference a null reference. As a result, frequent checks for a **null** value were necessary to avoid generating an exception. These classes provide a better way to handle such situations.

The first and most general of these classes is **Optional**. For this reason, it is the primary focus of this discussion. It is shown here:

```
class Optional<T>
```

Here, **T** specifies the type of value stored. It is important to understand that an **Optional** instance can either contain a value of type **T** or be empty. In other words, an **Optional** object does not necessarily contain a value. **Optional** does not define any constructors, but it does define several methods that let you work with **Optional** objects. For example, you can determine if a value is present, obtain the value if it is present, obtain a default value when no value is present, and construct an **Optional** value. The **Optional** methods are shown in Table 19-3.

Method	Description
static <T> Optional<T> empty( )	Returns an object for which **isPresent( )** returns **false**.
boolean equals(Object *optional*)	Returns **true** if the invoking object equals *optional*. Otherwise, returns **false**.
Optional<T> filter(Predicate<? super T> *condition*)	Returns an **Optional** instance that contains the same value as the invoking object if that value satisfies *condition*. Otherwise, an empty object is returned.
U Optional<U> flatMap(Function<? super T, Optional<U>> *mapFunc*)	Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
T get( )	Returns the value in the invoking object. However, if no value is present, **NoSuchElementException** is thrown.
int hashCode( )	Returns a hashcode for the invoking object.
void ifPresent(Consumer<? super T> *func*)	Calls *func* if a value is present in the invoking object, passing the object to *func*. If no value is present, no action occurs.
boolean isPresent( )	Returns **true** if the invoking object contains a value. Returns **false** if no value is present.
U Optional<U> map(Function<? super T, ? extends U>> *mapFunc*)	Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
static <T> Optional<T> of(T *val*)	Creates an **Optional** instance that contains *val* and returns the result. The value of *val* must not be **null**.

Table 19-3 The Methods Defined by **Optional**

Method	Description
static <T> Optional<T> ofNullable(T *val*)	Creates an **Optional** instance that contains *val* and returns the result. However, if *val* is **null**, then an empty **Optional** instance is returned.
T orElse(T *defVal*)	If the invoking object contains a value, the value is returned. Otherwise, the value specified by *defVal* is returned.
T orElseGet(Supplier<? extends T> *getFunc*)	If the invoking object contains a value, the value is returned. Otherwise, the value obtained from *getFunc* is returned.
<X extends Throwable> T orElseThrow(Supplier<? extends X> *excFunc*) throws X extends Throwable	Returns the value in the invoking object. However, if no value is present, the exception generated by *excFunc* is thrown.
String toString( )	Returns a string corresponding to the invoking object.

Table 19-3 The Methods Defined by **Optional** *(continued)*

The best way to understand **Optional** is to work through an example that uses its core methods. At the foundation of **Optional** are **isPresent( )** and **get( )**. You can determine if a value is present by calling **isPresent( )**. If a value is available, it will return **true**. Otherwise, **false** is returned. If a value is present in an **Optional** instance, you can obtain it by calling **get( )**. However, if you call **get( )** on an object that does not contain a value, **NoSuchElementException** is thrown. For this reason, you should always first confirm that a value is present before calling **get( )** on an **Optional** object.

Of course, having to call two methods to retrieve a value adds overhead to each access. Fortunately, **Optional** defines methods that combine the check for a value with the retrieval of the value. One such method is **orElse( )**. If the object on which it is called contains a value, the value is returned. Otherwise, a default value is returned.

Optional does not define any constructors. Instead, you will use one of its methods to create an instance. For example, you can create an **Optional** instance with a specified value by using **of( )**. You can create an instance of **Optional** that does not contain a value by using **empty( )**.

The following program demonstrates these methods:

```
// Demonstrate several Optional<T> methods

import java.util.*;

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

    Optional<String> noVal = Optional.empty();

    Optional<String> hasVal = Optional.of("ABCDEFG");
    if(noVal.isPresent()) System.out.println("This won't be displayed");
    else System.out.println("noVal has no value");

    if(hasVal.isPresent()) System.out.println("The string in hasVal is: " +
                                               hasVal.get());

    String defStr = noVal.orElse("Default String");
    System.out.println(defStr);
  }
}
```

The output is shown here:

```
noVal has no value
The string in hasVal is: ABCDEFG
Default String
```

As the output shows, a value can be obtained from an **Optional** object only if one is present. This basic mechanism enables **Optional** to prevent null pointer exceptions.

The **OptionalDouble**, **OptionalInt**, and **OptionalLong** classes work much like **Optional**, except that they are designed expressly for use on **double**, **int**, and **long** values, respectively. As such, they specify the methods **getAsDouble( )**, **getAsInt( )**, and **getAsLong( )**, respectively, rather than **get( )**. Also, they do not support the **filter( )**, **ofNullable( )**, **map( )** and **flatMap( )** methods.

Date

The **Date** class encapsulates the current date and time. Before beginning our examination of **Date**, it is important to point out that it has changed substantially from its original version defined by Java 1.0. When Java 1.1 was released, many of the functions carried out by the original **Date** class were moved into the **Calendar** and **DateFormat** classes, and as a result, many of the original 1.0 **Date** methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

Date supports the following non-deprecated constructors:

Date( )
Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The nondeprecated methods defined by **Date** are shown in Table 19-4. **Date** also implements the **Comparable** interface.

Method	Description
boolean after(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**.
boolean before(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**.
Object clone( )	Duplicates the invoking **Date** object.
int compareTo(Date *date*)	Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*.
boolean equals(Object *date*)	Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**.
static Date from(Instant t)	Returns a **Date** object corresponding to the **Instant** object passed in *t*. (Added by JDK 8.)
long getTime( )	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode( )	Returns a hash code for the invoking object.
void setTime(long *time*)	Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Date** object. (Added by JDK 8.)
String toString( )	Converts the invoking **Date** object into a string and returns the result.

Table 19-4 The Nondeprecated Methods Defined by **Date**

As you can see by examining Table 19-4, the non-deprecated **Date** features do not allow you to obtain the individual components of the date or time. As the following program demonstrates, you can only obtain the date and time in terms of milliseconds, in its default string representation as returned by **toString( )**, or (beginning with JDK 8) as an **Instant** object. To obtain more-detailed information about the date and time, you will use the **Calendar** class.

```
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println(date);

    // Display number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
  }
}
```

Sample output is shown here:

```
Wed Jan 01 11:11:44 CST 2014
Milliseconds since Jan. 1, 1970 GMT = 1388596304803
```

Calendar

The abstract **Calendar** class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second. It is intended that subclasses of **Calendar** will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is **GregorianCalendar**.

NOTE JDK 8 defines a new date and time API in **java.time**, which new applications may want to employ. See Chapter 30.

Calendar provides no public constructors. **Calendar** defines several protected instance variables. **areFieldsSet** is a **boolean** that indicates if the time components have been set. **fields** is an array of **int**s that holds the components of the time. **isSet** is a **boolean** array that indicates if a specific time component has been set. **time** is a **long** that holds the current time for this object. **isTimeSet** is a **boolean** that indicates if the current time has been set.

A sampling of methods defined by **Calendar** are shown in Table 19-5.

Method	Description
abstract void add(int *which*, int *val*)	Adds *val* to the time or date component specified by *which*. To subtract, add a negative value. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
boolean after(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is later than the one specified by *calendarObj*. Otherwise, it returns **false**.
boolean before(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is earlier than the one specified by *calendarObj*. Otherwise, it returns **false**.
final void clear( )	Zeros all time components in the invoking object.
final void clear(int *which*)	Zeros the time component specified by *which* in the invoking object.
Object clone( )	Returns a duplicate of the invoking object.
boolean equals(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is equal to the one specified by *calendarObj*. Otherwise, it returns **false**.

Table 19-5 A Sampling of the Methods Defined by **Calendar**

Method	Description
int get(int *calendarField*)	Returns the value of one component of the invoking object. The component is indicated by *calendarField*. Examples of the components that can be requested are **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.MINUTE**, and so forth.
static Locale[ ] getAvailableLocales( )	Returns an array of **Locale** objects that contains the locales for which calendars are available.
static Calendar getInstance( )	Returns a **Calendar** object for the default locale and time zone.
static Calendar getInstance(TimeZone *tz*)	Returns a **Calendar** object for the time zone specified by *tz*. The default locale is used.
static Calendar getInstance(Locale *locale*)	Returns a **Calendar** object for the locale specified by *locale*. The default time zone is used.
static Calendar getInstance(TimeZone *tz*, Locale *locale*)	Returns a **Calendar** object for the time zone specified by *tz* and the locale specified by *locale*.
final Date getTime( )	Returns a **Date** object equivalent to the time of the invoking object.
TimeZone getTimeZone( )	Returns the time zone for the invoking object.
final boolean isSet(int *which*)	Returns **true** if the specified time component is set. Otherwise, it returns **false**.
void set(int *which*, int *val*)	Sets the date or time component specified by *which* to the value specified by *val* in the invoking object. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
final void set(int *year*, int *month*, int *dayOfMonth*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)	Sets various date and time components of the invoking object.
final void setTime(Date *d*)	Sets various date and time components of the invoking object. This information is obtained from the **Date** object *d*.
void setTimeZone(TimeZone *tz*)	Sets the time zone for the invoking object to that specified by *tz*.
final Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Calendar** instance. (Added by JDK 8.)

Table 19-5 A Sampling of the Methods Defined by **Calendar** *(continued)*

Calendar defines the following **int** constants, which are used when you get or set components of the calendar. (The ones with the suffix **FORMAT** or **STANDALONE** were added by JDK 8.)

ALL_STYLES	HOUR_OF_DAY	PM
AM	JANUARY	SATURDAY
AM_PM	JULY	SECOND
APRIL	JUNE	SEPTEMBER
AUGUST	LONG	SHORT
DATE	LONG_FORMAT	SHORT_FORMAT
DAY_OF_MONTH	LONG_STANDALONE	SHORT_STANDALONE
DAY_OF_WEEK	MARCH	SUNDAY
DAY_OF_WEEK_IN_MONTH	MAY	THURSDAY
DAY_OF_YEAR	MILLISECOND	TUESDAY
DECEMBER	MINUTE	UNDECIMBER
DST_OFFSET	MONDAY	WEDNESDAY
ERA	MONTH	WEEK_OF_MONTH
FEBRUARY	NARROW_FORMAT	WEEK_OF_YEAR
FIELD_COUNT	NARROW_STANDALONE	YEAR
FRIDAY	NOVEMBER	ZONE_OFFSET
HOUR	OCTOBER	

The following program demonstrates several **Calendar** methods:

```
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months[] = {
             "Jan", "Feb", "Mar", "Apr",
             "May", "Jun", "Jul", "Aug",
             "Sep", "Oct", "Nov", "Dec"};

    // Create a calendar initialized with the
    // current date and time in the default
    // locale and timezone.
    Calendar calendar = Calendar.getInstance();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Set the time and date information and display it.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
    System.out.print("Updated time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 11:29:39
Updated time: 10:29:22
```

GregorianCalendar

GregorianCalendar is a concrete implementation of a **Calendar** that implements the normal Gregorian calendar with which you are familiar. The **getInstance( )** method of **Calendar** will typically return a **GregorianCalendar** initialized with the current date and time in the default locale and time zone.

GregorianCalendar defines two fields: **AD** and **BC**. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for **GregorianCalendar** objects. The default, **GregorianCalendar( )**, initializes the object with the current date and time in the default locale and time zone. Three more constructors offer increasing levels of specificity:

GregorianCalendar(int *year*, int *month*, int *dayOfMonth*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*,
int *minutes*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*,
int *minutes*, int *seconds*)

All three versions set the day, month, and year. Here, *year* specifies the year. The month is specified by *month*, with zero indicating January. The day of the month is specified by *dayOfMonth*. The first version sets the time to midnight. The second version also sets the hours and the minutes. The third version adds seconds.

You can also construct a **GregorianCalendar** object by specifying the locale and/or time zone. The following constructors create objects initialized with the current date and time using the specified time zone and/or locale:

GregorianCalendar(Locale *locale*)
GregorianCalendar(TimeZone *timeZone*)
GregorianCalendar(TimeZone *timeZone*, Locale *locale*)

GregorianCalendar provides an implementation of all the abstract methods in **Calendar**. It also provides some additional methods. Perhaps the most interesting is **isLeapYear( )**, which tests if the year is a leap year. Its form is

boolean isLeapYear(int *year*)

This method returns **true** if *year* is a leap year and **false** otherwise. JDK 8 also adds the following methods: **from( )** and **toZonedDateTime( )**, which support the new date and time API, and **getCalendarType( )**, which returns the calendar type as a string, which is "gregory".

The following program demonstrates **GregorianCalendar**:

```
// Demonstrate GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
  public static void main(String args[]) {
    String months[] = {
             "Jan", "Feb", "Mar", "Apr",
             "May", "Jun", "Jul", "Aug",
             "Sep", "Oct", "Nov", "Dec"};
    int year;

    // Create a Gregorian calendar initialized
    // with the current date and time in the
    // default locale and timezone.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));

    // Test if the current year is a leap year
    if(gcalendar.isLeapYear(year)) {
      System.out.println("The current year is a leap year");
    }
    else {
      System.out.println("The current year is not a leap year");
    }
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 1:45:5
The current year is not a leap year
```

TimeZone

Another time-related class is **TimeZone**. The abstract **TimeZone** class allows you to work with time zone offsets from Greenwich mean time (GMT), also referred to as Coordinated Universal Time (UTC). It also computes daylight saving time. **TimeZone** only supplies the default constructor.

A sampling of methods defined by **TimeZone** is given in Table 19-6.

Method	Description
Object clone( )	Returns a **TimeZone**-specific version of **clone( )**.
static String[ ] getAvailableIDs( )	Returns an array of **String** objects representing the names of all time zones.
static String[ ] getAvailableIDs(int *timeDelta*)	Returns an array of **String** objects representing the names of all time zones that are *timeDelta* offset from GMT.
static TimeZone getDefault( )	Returns a **TimeZone** object that represents the default time zone used on the host computer.
String getID( )	Returns the name of the invoking **TimeZone** object.
abstract int getOffset(int *era*, int *year*, int *month*, int *dayOfMonth*, int *dayOfWeek*, int *millisec*)	Returns the offset that should be added to GMT to compute local time. This value is adjusted for daylight saving time. The parameters to the method represent date and time components.
abstract int getRawOffset( )	Returns the raw offset (in milliseconds) that should be added to GMT to compute local time. This value is not adjusted for daylight saving time.
static TimeZone getTimeZone(String *tzName*)	Returns the **TimeZone** object for the time zone named *tzName*.
abstract boolean inDaylightTime(Date *d*)	Returns **true** if the date represented by *d* is in daylight saving time in the invoking object. Otherwise, it returns **false**.
static void setDefault(TimeZone *tz*)	Sets the default time zone to be used on this host. *tz* is a reference to the **TimeZone** object to be used.
void setID(String *tzName*)	Sets the name of the time zone (that is, its ID) to that specified by *tzName*.
abstract void setRawOffset(int *millis*)	Sets the offset in milliseconds from GMT.
ZoneId toZoneId( )	Converts the invoking object into a **ZoneId** and returns the result. **ZoneId** is packaged in **java.time.** (Added by JDK 8.)
abstract boolean useDaylightTime( )	Returns **true** if the invoking object uses daylight saving time. Otherwise, it returns **false**.

Table 19-6 A Sampling of the Methods Defined by **TimeZone**

SimpleTimeZone

The **SimpleTimeZone** class is a convenient subclass of **TimeZone**. It implements **TimeZone**'s abstract methods and allows you to work with time zones for a Gregorian calendar. It also computes daylight saving time.

SimpleTimeZone defines four constructors. One is

```
SimpleTimeZone(int timeDelta, String tzName)
```

This constructor creates a **SimpleTimeZone** object. The offset relative to Greenwich mean time (GMT) is *timeDelta*. The time zone is named *tzName*.

The second **SimpleTimeZone** constructor is

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                    int dstDayInMonth0, int dstDay0, int time0,
                    int dstMonth1, int dstDayInMonth1, int dstDay1,
                    int time1)
```

Here, the offset relative to GMT is specified in *timeDelta*. The time zone name is passed in *tzId*. The start of daylight saving time is indicated by the parameters *dstMonth0*, *dstDayInMonth0*, *dstDay0*, and *time0*. The end of daylight saving time is indicated by the parameters *dstMonth1*, *dstDayInMonth1*, *dstDay1*, and *time1*.

The third **SimpleTimeZone** constructor is

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                    int dstDayInMonth0, int dstDay0, int time0,
                    int dstMonth1, int dstDayInMonth1,
                    int dstDay1, int time1, int dstDelta)
```

Here, *dstDelta* is the number of milliseconds saved during daylight saving time.

The fourth **SimpleTimeZone** constructor is:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                    int dstDayInMonth0, int dstDay0, int time0,
                    int time0mode, int dstMonth1, int dstDayInMonth1,
                    int dstDay1, int time1, int time1mode, int dstDelta)
```

Here, *time0mode* specifies the mode of the starting time, and *time1mode* specifies the mode of the ending time. Valid mode values include:

STANDARD_TIME	WALL_TIME	UTC_TIME

The time mode indicates how the time values are interpreted. The default mode used by the other constructors is **WALL_TIME**.

Locale

The **Locale** class is instantiated to produce objects that describe a geographical or cultural region. It is one of several classes that provide you with the ability to write programs that can execute in different international environments. For example, the formats used to display dates, times, and numbers are different in various regions.

Internationalization is a large topic that is beyond the scope of this book. However, many programs will only need to deal with its basics, which include setting the current locale.

The **Locale** class defines the following constants that are useful for dealing with several common locales:

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

For example, the expression **Locale.CANADA** represents the **Locale** object for Canada.

The constructors for **Locale** are

Locale(String *language*)
Locale(String *language*, String *country*)
Locale(String *language*, String *country*, String *variant*)

These constructors build a **Locale** object to represent a specific *language* and in the case of the last two, *country*. These values must contain standard language and country codes. Auxiliary variant information can be provided in *variant*.

Locale defines several methods. One of the most important is **setDefault( )**, shown here:

static void setDefault(Locale *localeObj*)

This sets the default locale used by the JVM to that specified by *localeObj*.

Some other interesting methods are the following:

final String getDisplayCountry( )
final String getDisplayLanguage( )
final String getDisplayName( )

These return human-readable strings that can be used to display the name of the country, the name of the language, and the complete description of the locale.

The default locale can be obtained using **getDefault( )**, shown here:

static Locale getDefault( )

JDK 7 added significant upgrades to the **Locale** class that support Internet Engineering Task Force (IETF) BCP 47, which defines tags for identifying languages, and Unicode Technical Standard (UTS) 35, which defines the Locale Data Markup Language (LDML). Support for BCP 47 and UTS 35 caused several features to be added to **Locale**, including several new methods and the **Locale.Builder** class. Among others, new methods include **getScript( )**, which obtains the locale's script, and **toLanguageTag( )**, which obtains a string that contains the locale's language tag. The **Locale.Builder** class constructs **Locale** instances. It ensures that a locale specification is well-formed as defined by BCP 47. (The **Locale** constructors do not provide such a check.) Several new methods have also been added to **Locale** by JDK 8. Among these are methods that support filtering, extensions, and lookups.

Calendar and **GregorianCalendar** are examples of classes that operate in a locale-sensitive manner. **DateFormat** and **SimpleDateFormat** also depend on the locale.

Random

The **Random** class is a generator of pseudorandom numbers. These are called *pseudorandom* numbers because they are simply uniformly distributed sequences. **Random** defines the following constructors:

Random( )
Random(long *seed*)

The first version creates a number generator that uses a reasonably unique seed. The second form allows you to specify a seed value manually.

If you initialize a **Random** object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another **Random** object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. One way to do this is to use the current time to seed a **Random** object. This approach reduces the possibility of getting repeated sequences.

The core public methods defined by **Random** are shown in Table 19-7. These are the methods that have been available in **Random** for several years (many since Java 1.0) and are widely used.

As you can see, there are seven types of random numbers that you can extract from a **Random** object. Random Boolean values are available from **nextBoolean( )**. Random bytes can be obtained by calling **nextBytes( )**. Integers can be extracted via the **nextInt( )** method. Long integers, uniformly distributed over their range, can be obtained with **nextLong( )**. The **nextFloat( )** and **nextDouble( )** methods return a uniformly distributed **float** and **double**, respectively, between 0.0 and 1.0. Finally, **nextGaussian( )** returns a **double** value centered at 0.0 with a standard deviation of 1.0. This is what is known as a *bell curve*.

Here is an example that demonstrates the sequence produced by **nextGaussian( )**. It obtains 100 random Gaussian values and averages these values. The program also counts the

Method	Description
boolean nextBoolean( )	Returns the next **boolean** random number.
void nextBytes(byte *vals*[ ])	Fills *vals* with randomly generated values.
double nextDouble( )	Returns the next **double** random number.
float nextFloat( )	Returns the next **float** random number.
double nextGaussian( )	Returns the next Gaussian random number.
int nextInt( )	Returns the next **int** random number.
int nextInt(int *n*)	Returns the next **int** random number within the range zero to *n*.
long nextLong( )	Returns the next **long** random number.
void setSeed(long *newSeed*)	Sets the seed value (that is, the starting point for the random number generator) to that specified by *newSeed*.

Table 19-7 The Core Methods Defined by **Random**

number of values that fall within two standard deviations, plus or minus, using increments of 0.5 for each category. The result is graphically displayed sideways on the screen.

```
// Demonstrate random Gaussian values.
import java.util.Random;
class RandDemo {
  public static void main(String args[]) {
    Random r = new Random();
    double val;
    double sum = 0;
    int bell[] = new int[10];

    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;

      for(int x=0; x<10; x++, t += 0.5)
        if(val < t) {
          bell[x]++;
          break;
        }
    }
    System.out.println("Average of values: " +
                       (sum/100));

    // display bell curve, sideways
    for(int i=0; i<10; i++) {
      for(int x=bell[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

Here is a sample program run. As you can see, a bell-like distribution of numbers is obtained.

```
Average of values: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

JDK 8 adds three new methods to **Random** that support the new stream API (see Chapter 29). They are called **doubles( )**, **ints( )**, and **longs( )**, and each returns a reference to a stream that contains a sequence of pseudorandom values of the specified type. Each method defines several overloads. Here are their simplest forms:

DoubleStream doubles( )

IntStream ints( )

LongStream longs( )

The **doubles( )** method returns a stream that contains pseudorandom **double** values. (The range of these values will be less than 1.0 but greater than 0.0.) The **ints( )** method returns a stream that contains pseudorandom **int** values. The **longs( )** method returns a stream that contains pseudorandom **long** values. For these three methods, the stream returned is effectively infinite. Several overloads of each method are provided that let you specify the size of the stream, an origin, and an upper bound.

Observable

The **Observable** class is used to create subclasses that other parts of your program can observe. When an object of such a subclass undergoes a change, observing classes are notified. Observing classes must implement the **Observer** interface, which defines the **update( )** method. The **update( )** method is called when an observer is notified of a change in an observed object.

Observable defines the methods shown in Table 19-8. An object that is being observed must follow two simple rules. First, if it has changed, it must call **setChanged( )**. Second, when it is ready to notify observers of this change, it must call **notifyObservers( )**. This causes the **update( )** method in the observing object(s) to be called. Be careful—if the

Method	Description
void addObserver(Observer *obj*)	Adds *obj* to the list of objects observing the invoking object.
protected void clearChanged( )	Calling this method returns the status of the invoking object to "unchanged."
int countObservers( )	Returns the number of objects observing the invoking object.
void deleteObserver(Observer *obj*)	Removes *obj* from the list of objects observing the invoking object.
void deleteObservers( )	Removes all observers for the invoking object.
boolean hasChanged( )	Returns **true** if the invoking object has been modified and **false** if it has not.
void notifyObservers( )	Notifies all observers of the invoking object that it has changed by calling **update( )**. A **null** is passed as the second argument to **update( )**.
void notifyObservers(Object *obj*)	Notifies all observers of the invoking object that it has changed by calling **update( )**. *obj* is passed as an argument to **update( )**.
protected void setChanged( )	Called when the invoking object has changed.

Table 19-8 The Methods Defined by **Observable**

object calls **notifyObservers( )** without having previously called **setChanged( )**, no action will take place. The observed object must call both **setChanged( )** and **notifyObservers( )** before **update( )** will be called.

Notice that **notifyObservers( )** has two forms: one that takes an argument and one that does not. If you call **notifyObservers( )** with an argument, this object is passed to the observer's **update( )** method as its second parameter. Otherwise, **null** is passed to **update( )**. You can use the second parameter for passing any type of object that is appropriate for your application.

The Observer Interface

To observe an observable object, you must implement the **Observer** interface. This interface defines only the one method shown here:

void update(Observable *observOb*, Object *arg*)

Here, *observOb* is the object being observed, and *arg* is the value passed by **notifyObservers( )**. The **update( )** method is called when a change in the observed object takes place.

An Observer Example

Here is an example that demonstrates an observable object. It creates an observer class, called **Watcher**, that implements the **Observer** interface. The class being monitored is called **BeingWatched**. It extends **Observable**. Inside **BeingWatched** is the method **counter( )**, which simply counts down from a specified value. It uses **sleep( )** to wait a tenth of a second between counts. Each time the count changes, **notifyObservers( )** is called with the current count passed as its argument. This causes the **update( )** method inside **Watcher** to be called, which displays the current count. Inside **main( )**, a **Watcher** and a **BeingWatched** object, called **observing** and **observed**, respectively, are created. Then, **observing** is added to the list of observers for **observed**. This means that **observing.update( )** will be called each time **counter( )** calls **notifyObservers( )**.

```
/* Demonstrate the Observable class and the
   Observer interface.
*/
import java.util.*;

// This is the observing class.
class Watcher implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                       ((Integer)arg).intValue());
  }
}

// This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }

}

class ObserverDemo {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher observing = new Watcher();

    /* Add the observing to the list of observers for
       observed object.  */
    observed.addObserver(observing);

    observed.counter(10);
  }
}
```

The output from this program is shown here:

```
update() called, count is 10
update() called, count is 9
update() called, count is 8
update() called, count is 7
update() called, count is 6
update() called, count is 5
update() called, count is 4
update() called, count is 3
update() called, count is 2
update() called, count is 1
update() called, count is 0
```

More than one object can be an observer. For example, the following program implements two observing classes and adds an object of each class to the **BeingWatched** observer list. The second observer waits until the count reaches zero and then rings the bell.

```
/* An object may be observed by two or more
   observers.
*/

import java.util.*;


// This is the first observing class.
class Watcher1 implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                       ((Integer)arg).intValue());
  }
}

// This is the second observing class.
class Watcher2 implements Observer {
  public void update(Observable obj, Object arg) {
    // Ring bell when done
    if(((Integer)arg).intValue() == 0)
      System.out.println("Done" + '\7');
  }
}

// This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }
}

class TwoObservers {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher1 observing1 = new Watcher1();
    Watcher2 observing2 = new Watcher2();

    // add both observers
    observed.addObserver(observing1);
    observed.addObserver(observing2);

    observed.counter(10);
  }
}
```

The **Observable** class and the **Observer** interface allow you to implement sophisticated program architectures based on the document/view methodology.

Timer and TimerTask

An interesting and useful feature offered by **java.util** is the ability to schedule a task for execution at some future time. The classes that support this are **Timer** and **TimerTask**. Using these classes, you can create a thread that runs in the background, waiting for a specific time. When the time arrives, the task linked to that thread is executed. Various options allow you to schedule a task for repeated execution, and to schedule a task to run on a specific date. Although it was always possible to manually create a task that would be executed at a specific time using the **Thread** class, **Timer** and **TimerTask** greatly simplify this process.

Timer and **TimerTask** work together. **Timer** is the class that you will use to schedule a task for execution. The task being scheduled must be an instance of **TimerTask**. Thus, to schedule a task, you will first create a **TimerTask** object and then schedule it for execution using an instance of **Timer**.

TimerTask implements the **Runnable** interface; thus, it can be used to create a thread of execution. Its constructor is shown here:

protected TimerTask( )

TimerTask defines the methods shown in Table 19-9. Notice that **run( )** is abstract, which means that it must be overridden. The **run( )** method, defined by the **Runnable** interface, contains the code that will be executed. Thus, the easiest way to create a timer task is to extend **TimerTask** and override **run( )**.

Once a task has been created, it is scheduled for execution by an object of type **Timer**. The constructors for **Timer** are shown here:

Timer( )
Timer(boolean *DThread*)
Timer(String *tName*)
Timer(String *tName*, boolean *DThread*)

The first version creates a **Timer** object that runs as a normal thread. The second uses a daemon thread if *DThread* is **true**. A daemon thread will execute only as long as the rest of the program continues to execute. The third and fourth constructors allow you to specify a name for the **Timer** thread. The methods defined by **Timer** are shown in Table 19-9.

Once a **Timer** has been created, you will schedule a task by calling **schedule( )** on the **Timer** that you created. As Table 19-10 shows, there are several forms of **schedule( )** which allow you to schedule tasks in a variety of ways.

Method	Description
boolean cancel( )	Terminates the task. Returns **true** if an execution of the task is prevented. Otherwise, returns **false**.
abstract void run( )	Contains the code for the timer task.
long scheduledExecutionTime( )	Returns the time at which the last execution of the task was scheduled to have occurred.

Table 19-9 The Methods Defined by **TimerTask**

Method	**Description**
void cancel( )	Cancels the timer thread.
int purge( )	Deletes canceled tasks from the timer's queue.
void schedule(TimerTask *TTask*, long *wait*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The *wait* parameter is specified in milliseconds.
void schedule(TimerTask *TTask*, long *wait*, long *repeat*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds.
void schedule(TimerTask *TTask*, Date *targetTime*)	*TTask* is scheduled for execution at the time specified by *targetTime*.
void schedule(TimerTask *TTask*, Date *targetTime*, long *repeat*)	*TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds.
void scheduleAtFixedRate(TimerTask *TTask*, long *wait*, long *repeat*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed.
void scheduleAtFixedRate(TimerTask *TTask*, Date *targetTime*, long *repeat*)	*TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed.

Table 19-10 The Methods Defined by **Timer**

If you create a non-daemon task, then you will want to call **cancel( )** to end the task when your program ends. If you don't do this, then your program may "hang" for a period of time.

The following program demonstrates **Timer** and **TimerTask**. It defines a timer task whose **run( )** method displays the message "Timer task executed." This task is scheduled to run once every half second after an initial delay of one second.

```
// Demonstrate Timer and TimerTask.

import java.util.*;

class MyTimerTask extends TimerTask {
  public void run() {
    System.out.println("Timer task executed.");
  }
}

class TTest {
    public static void main(String args[]) {
      MyTimerTask myTask = new MyTimerTask();
      Timer myTimer = new Timer();

      /* Set an initial delay of 1 second,
         then repeat every half second.
      */
      myTimer.schedule(myTask, 1000, 500);

      try {
        Thread.sleep(5000);
      } catch (InterruptedException exc) {}

      myTimer.cancel();
    }
  }
```

Currency

The **Currency** class encapsulates information about a currency. It defines no constructors. The methods supported by **Currency** are shown in Table 19-11. The following program demonstrates **Currency**:

```
// Demonstrate Currency.
import java.util.*;
```

Method	Description
static Set<Currency> getAvailableCurrencies( )	Returns a set of the supported currencies.
String getCurrencyCode( )	Returns the code (as defined by ISO 4217) that describes the invoking currency.
int getDefaultFractionDigits( )	Returns the number of digits after the decimal point that are normally used by the invoking currency. For example, there are two fractional digits normally used for dollars.
String getDisplayName( )	Returns the name of the invoking currency for the default locale.
String getDisplayName(Locale *loc*)	Returns the name of the invoking currency for the specified locale.
static Currency getInstance(Locale *localeObj*)	Returns a **Currency** object for the locale specified by *localeObj*.
static Currency getInstance(String *code*)	Returns a **Currency** object associated with the currency code passed in *code*.
int getNumericCode( )	Returns the numeric code (as defined by ISO 4217) for the invoking currency.
String getSymbol( )	Returns the currency symbol (such as $) for the invoking object.
String getSymbol(Locale *localeObj*)	Returns the currency symbol (such as $) for the locale passed in *localeObj*.
String toString( )	Returns the currency code for the invoking object.

Table 19-11 The Methods Defined by **Currency**

```
class CurDemo {
  public static void main(String args[]) {
    Currency c;

    c = Currency.getInstance(Locale.US);

    System.out.println("Symbol: " + c.getSymbol());
    System.out.println("Default fractional digits: " +
                       c.getDefaultFractionDigits());
  }
}
```

The output is shown here:

```
Symbol: $
Default fractional digits: 2
```

Formatter

At the core of Java's support for creating formatted output is the **Formatter** class. It provides *format conversions* that let you display numbers, strings, and time and date in virtually any format you like. It operates in a manner similar to the C/C++ **printf( )** function, which means that if you are familiar with C/C++, then learning to use **Formatter** will be very easy. It also further streamlines the conversion of C/C++ code to Java. If you are not familiar with C/C++, it is still quite easy to format data.

NOTE Although Java's **Formatter** class operates in a manner very similar to the C/C++ **printf( )** function, there are some differences, and some new features. Therefore, if you have a C/C++ background, a careful reading is advised.

The Formatter Constructors

Before you can use **Formatter** to format output, you must create a **Formatter** object. In general, **Formatter** works by converting the binary form of data used by a program into formatted text. It stores the formatted text in a buffer, the contents of which can be obtained by your program whenever they are needed. It is possible to let **Formatter** supply this buffer automatically, or you can specify the buffer explicitly when a **Formatter** object is created. It is also possible to have **Formatter** output its buffer to a file.

The **Formatter** class defines many constructors, which enable you to construct a **Formatter** in a variety of ways. Here is a sampling:

Formatter( )

Formatter(Appendable *buf*)

Formatter(Appendable *buf*, Locale *loc*)

Formatter(String *filename*)
  throws FileNotFoundException

Formatter(String *filename*, String *charset*)
  throws FileNotFoundException, UnsupportedEncodingException

```
Formatter(File outF)
  throws FileNotFoundException
Formatter(OutputStream outStrm)
```

Here, *buf* specifies a buffer for the formatted output. If *buf* is null, then **Formatter** automatically allocates a **StringBuilder** to hold the formatted output. The *loc* parameter specifies a locale. If no locale is specified, the default locale is used. The *filename* parameter specifies the name of a file that will receive the formatted output. The *charset* parameter specifies the character set. If no character set is specified, then the default character set is used. The *outF* parameter specifies a reference to an open file that will receive output. The *outStrm* parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a **StringBuilder** to hold the formatted output.

The Formatter Methods

Formatter defines the methods shown in Table 19-12.

Method	Description
void close( )	Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**.
void flush( )	Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file.
Formatter format(String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object.
Formatter format(Locale *loc*, String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object.
IOException ioException( )	If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, null is returned.
Locale locale( )	Returns the invoking object's locale.
Appendable out( )	Returns a reference to the underlying object that is the destination for output.
String toString( )	Returns a **String** containing the formatted output.

Table 19-12 The Methods Defined by **Formatter**

Formatting Basics

After you have created a **Formatter**, you can use it to create a formatted string. To do so, use the **format( )** method. The most commonly used version is shown here:

Formatter format(String *fmtString*, Object ... *args*)

The *fmtSring* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains *format specifiers* that define the way the subsequent arguments are displayed.

In its simplest form, a format specifier begins with a percent sign followed by the format *conversion specifier*. All format conversion specifiers consist of a single character. For example, the format specifier for floating-point data is **%f**. In general, there must be the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, consider this fragment:

```
Formatter fmt = new Formatter();
fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

This sequence creates a **Formatter** that contains the following string:

```
Formatting with Java is easy 10 98.600000
```

In this example, the format specifiers, **%s**, **%d**, and **%f**, are replaced with the arguments that follow the format string. Thus, **%s** is replaced by "with Java", **%d** is replaced by 10, and **%f** is replaced by 98.6. All other characters are simply used as-is. As you might guess, the format specifier **%s** specifies a string, and **%d** specifies an integer value. As mentioned earlier, the **%f** specifies a floating-point value.

The **format( )** method accepts a wide variety of format specifiers, which are shown in Table 19-13. Notice that many specifiers have both upper- and lowercase forms. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point

Table 19-13 The Format Specifiers

Format Specifier	Conversion Applied
%g %G	Uses **%e** or **%f**, based on the value being formatted and the precision
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a **%** sign

Table 19-13 The Format Specifiers *(continued)*

lowercase specifiers perform the same conversion. It is important to understand that Java type-checks each format specifier against its corresponding argument. If the argument doesn't match, an **IllegalFormatException** is thrown.

Once you have formatted a string, you can obtain it by calling **toString( )**. For example, continuing with the preceding example, the following statement obtains the formatted string contained in **fmt**:

```
String str = fmt.toString();
```

Of course, if you simply want to display the formatted string, there is no reason to first assign it to a **String** object. When a **Formatter** object is passed to **println( )**, for example, its **toString( )** method is automatically called.

Here is a short program that puts together all of the pieces, showing how to create and display a formatted string:

```
// A very simple example that uses Formatter.
import java.util.*;

class FormatDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);

    System.out.println(fmt);
    fmt.close();
  }
}
```

One other point: You can obtain a reference to the underlying output buffer by calling **out( )**. It returns a reference to an **Appendable** object.

Now that you know the general mechanism used to create a formatted string, the remainder of this section discusses in detail each conversion. It also describes various options, such as justification, minimum field width, and precision.

Formatting Strings and Characters

To format an individual character, use **%c**. This causes the matching character argument to be output, unmodified. To format a string, use **%s**.

Formatting Numbers

To format an integer in decimal format, use **%d**. To format a floating-point value in decimal format, use **%f**. To format a floating-point value in scientific notation, use **%e**. Numbers represented in scientific notation take this general form:

*x.dddddd*e+/–yy

The **%g** format specifier causes **Formatter** to use either **%f** or **%e**, based on the value being formatted and the precision, which is 6 by default. The following program demonstrates the effect of the **%f** and **%e** format specifiers:

```
// Demonstrate the %f and %e format specifiers.
import java.util.*;

class FormatDemo2 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    for(double i=1.23; i < 1.0e+6; i *= 100) {
      fmt.format("%f %e", i, i);
      System.out.println(fmt);
    }
    fmt.close();

  }
}
```

It produces the following output:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000 1.230000e+04
```

You can display integers in octal or hexadecimal format by using **%o** and **%x**, respectively. For example, this fragment:

```
fmt.format("Hex: %x, Octal: %o", 196, 196);
```

produces this output:

```
   Hex: c4, Octal: 304
```

You can display floating-point values in hexadecimal format by using **%a**. The format produced by **%a** appears a bit strange at first glance. This is because its representation uses a form similar to scientific notation that consists of a hexadecimal significand and a decimal exponent of powers of 2. Here is the general format:

0x1.*sig*p*exp*

Here, *sig* contains the fractional portion of the significand and *exp* contains the exponent. The **p** indicates the start of the exponent. For example, this call:

```
fmt.format("%a", 512.0);
```

produces this output:

```
  0x1.0p9
```

Formatting Time and Date

One of the more powerful conversion specifiers is **%t**. It lets you format time and date information. The **%t** specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in Table 19-14. For example, to display minutes, you would use **%tM**, where **M** indicates minutes in a two-character field. The argument corresponding to the **%t** specifier must be of type **Calendar**, **Date**, **Long**, or **long**.

Here is a program that demonstrates several of the formats:

```
// Formatting time and date.
import java.util.*;

class TimeDateFormat {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    // Display standard 12-hour time format.
    fmt.format("%tr", cal);
    System.out.println(fmt);
    fmt.close();

    // Display complete time and date information.
    fmt = new Formatter();
    fmt.format("%tc", cal);
    System.out.println(fmt);
    fmt.close();

    // Display just hour and minute.
    fmt = new Formatter();
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);
    fmt.close();

    // Display month by name and number.
    fmt = new Formatter();
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Suffix	Replaced By
a	Abbreviated weekday name
A	Full weekday name
b	Abbreviated month name
B	Full month name
c	Standard date and time string formatted as *day month date hh::mm:ss tzone year*
C	First two digits of year
d	Day of month as a decimal (01—31)
D	month/day/year
e	Day of month as a decimal (1—31)
F	year-month-day
h	Abbreviated month name
H	Hour (00 to 23)
I	Hour (01 to 12)
j	Day of year as a decimal (001 to 366)
k	Hour (0 to 23)
l	Hour (1 to 12)
L	Millisecond (000 to 999)
m	Month as decimal (01 to 13)
M	Minute as decimal (00 to 59)
N	Nanosecond (000000000 to 999999999)
p	Locale's equivalent of AM or PM in lowercase
Q	Milliseconds from 1/1/1970
r	*hh:mm:ss* (12-hour format)
R	*hh:mm* (24-hour format)
S	Seconds (00 to 60)
s	Seconds from 1/1/1970 UTC
T	*hh:mm:ss* (24-hour format)
y	Year in decimal without century (00 to 99)
Y	Year in decimal including century (0001 to 9999)
z	Offset from UTC
Z	Time zone name

Table 19-14 The Time and Date Format Suffixes

Sample output is shown here:

```
03:15:34 PM
Wed Jan 01 15:15:34 CST 2014
3:15
January Jan 01
```

The %n and %% Specifiers

The **%n** and**%%** format specifiers differ from the others in that they do not match an argument. Instead, they are simply escape sequences that insert a character into the output sequence. The **%n** inserts a newline. The **%%** inserts a percent sign. Neither of these characters can be entered directly into the format string. Of course, you can also use the standard escape sequence **\n** to embed a newline character.

Here is an example that demonstrates the **%n** and **%%** format specifiers:

```
// Demonstrate the %n and %% format specifiers.
import java.util.*;

class FormatDemo3 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Copying file%nTransfer is %d%% complete", 88);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It displays the following output:

```
Copying file
Transfer is 88% complete
```

Specifying a Minimum Field Width

An integer placed between the **%** sign and the format conversion code acts as a *minimum field-width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you want to pad with 0's, place a 0 before the field-width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is five. The field-width specifier can be used with all format specifiers except **%n**.

The following program demonstrates the minimum field-width specifier by applying it to the **%f** conversion:

```
// Demonstrate a field-width specifier.
import java.util.*;

class FormatDemo4 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    fmt.format("|%f|%n|%12f|%n|%012f|",
               10.12345, 10.12345, 10.12345);

    System.out.println(fmt);
    fmt.close();

  }
}
```

This program produces the following output:

```
|10.123450|
|   10.123450|
|00010.123450|
```

The first line displays the number 10.12345 in its default width. The second line displays that value in a 12-character field. The third line displays the value in a 12-character field, padded with leading zeros.

The minimum field-width modifier is often used to produce tables in which the columns line up. For example, the next program produces a table of squares and cubes for the numbers between 1 and 10:

```
// Create a table of squares and cubes.
import java.util.*;

class FieldWidthDemo {
  public static void main(String args[]) {
    Formatter fmt;

    for(int i=1; i <= 10; i++) {
      fmt = new Formatter();
      fmt.format("%4d %4d %4d", i, i*i, i*i*i);
      System.out.println(fmt);
      fmt.close();
    }

  }
}
```

Its output is shown here:

```
   1    1    1
   2    4    8
   3    9   27
   4   16   64
   5   25  125
   6   36  216
   7   49  343
   8   64  512
   9   81  729
  10  100 1000
```

Specifying Precision

A *precision specifier* can be applied to the **%f**, **%e**, **%g**, and **%s** format specifiers. It follows the minimum field-width specifier (if there is one) and consists of a period followed by an integer. Its exact meaning depends upon the type of data to which it is applied.

When you apply the precision specifier to floating-point data using the **%f** or **%e** specifiers, it determines the number of decimal places displayed. For example, **%10.4f** displays a number at least ten characters wide with four decimal places. When using **%g**, the precision determines the number of significant digits. The default precision is 6.

Applied to strings, the precision specifier specifies the maximum field length. For example, **%5.7s** displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

The following program illustrates the precision specifier:

```
// Demonstrate the precision modifier.
import java.util.*;

class PrecisionDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Format 4 decimal places.
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Format to 2 decimal places in a 16 character field
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Display at most 15 characters in a string.
    fmt = new Formatter();
    fmt.format("%.15s", "Formatting with Java is now easy.");
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
123.1235
        1.23e+02
Formatting with
```

Using the Format Flags

Formatter recognizes a set of format *flags* that lets you control various aspects of a conversion. All format flags are single characters, and a format flag follows the **%** in a format specification. The flags are shown here:

Flag	Effect
–	Left justification
#	Alternate conversion format
0	Output is padded with zeros rather than spaces
space	Positive numeric output is preceded by a space
+	Positive numeric output is preceded by a + sign
,	Numeric values include grouping separators
(	Negative numeric values are enclosed within parentheses

Not all flags apply to all format specifiers. The following sections explain each in detail.

Justifying Output

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For instance, **%–10.2f** left-justifies a floating-point number with two decimal places in a 10-character field. For example, consider this program:

```
// Demonstrate left justification.
import java.util.*;

class LeftJustify {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Right justify by default
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();

    // Now, left justify.
    fmt = new Formatter();
    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
|    123.12|
|123.12    |
```

As you can see, the second line is left-justified within a 10-character field.