The following constructor supports the extended Unicode character set:

String(int *codePoints*[ ], int *startIndex*, int *numChars*)

Here, *codePoints* is an array that contains Unicode code points. The resulting string is constructed from the range that begins at *startIndex* and runs for *numChars*.

There are also constructors that let you specify a **Charset**.

NOTE A discussion of Unicode code points and how they are handled by Java is found in Chapter 17.

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

int length( )

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the **+** operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length( )** method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the **+** to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now **s** contains the string "four: 4".

String Conversion and toString( )

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**. **valueOf( )** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf( )** calls the **toString( )** method on the object. We will look more closely at **valueOf( )** later in this chapter. Here, let's examine the **toString( )** method, because it is the means by which you can determine the string representation for objects of classes that you create.

Every class implements **toString( )** because it is defined by **Object**. However, the default implementation of **toString( )** is seldom sufficient. For most important classes that you create, you will want to override **toString( )** and provide your own string representations. Fortunately, this is easy to do. The **toString( )** method has this general form:

String toString( )

To implement **toString( )**, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class.

By overriding **toString( )** for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in **print( )** and **println( )** statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class:

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  public String toString() {
    return "Dimensions are " + width + " by " +
            depth + " by " + height + ".";
  }
}

class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

As you can see, **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Several are examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value **b** to **ch**.

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars( )**:

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

Here is the output of this program:

```
demo
```

getBytes( )

There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals( )** and **equalsIgnoreCase( )**:

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int *startIndex*, String *str2*,
                    int *str2StartIndex*, int *numChars*)

boolean regionMatches(boolean *ignoreCase*,
                                                int *startIndex*, String *str2*,
                                                int *str2StartIndex*, int *numChars*)

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )

String defines two methods that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)
boolean endsWith(String *str*)

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.

A second form of **startsWith( )**, shown here, lets you specify a starting point:

boolean startsWith(String *str*, int *startIndex*)

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

equals( ) Versus ==

It is important to understand that the **equals( )** method and the == operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

The variable **s1** refers to the **String** instance created by **"Hello"**. The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The method **compareTo( )** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements. It has this general form:

int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses **compareTo( )** to determine sort ordering for a bubble sort:

```
// A bubble sort for Strings.
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]);
    }
  }
}
```

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

As you can see from the output of this example, **compareTo( )** takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

int compareToIgnoreCase(String *str*)

This method returns the same results as **compareTo( )**, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

To search for the first occurrence of a character, use

int indexOf(int *ch*)

To search for the last occurrence of a character, use

int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String *str*)
int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*)
int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);
  }
}
```

The output from this program is shown here:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

```
String s = "   Hello World    ".trim();
```

This puts the string "Hello World" into **s**.

The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;

    System.out.println("Enter 'stop' to quit.");
    System.out.println("Enter State: ");
    do {
      str = br.readLine();
      str = str.trim(); // remove whitespace

      if(str.equals("Illinois"))
        System.out.println("Capital is Springfield.");
      else if(str.equals("Missouri"))
        System.out.println("Capital is Jefferson City.");
      else if(str.equals("California"))
        System.out.println("Capital is Sacramento.");
      else if(str.equals("Washington"))
        System.out.println("Capital is Olympia.");
      // ...
    } while(!str.equals("stop"));
  }
}
```

Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string. **valueOf( )** is also overloaded for type

Object, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

static String valueOf(double *num*)
static String valueOf(long *num*)
static String valueOf(Object *ob*)
static String valueOf(char *chars*[ ])

As discussed earlier, **valueOf( )** is called when a string representation of some other type of data is needed—for example, during concatenation operations. You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass to **valueOf( )** will return the result of a call to the object's **toString( )** method. In fact, you could just call **toString( )** directly and get the same result.

For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type. For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array. There is a special version of **valueOf( )** that allows you to specify a subset of a **char** array. It has this general form:

static String valueOf(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *chars* is the array that holds the characters, *startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

String toLowerCase( )
String toUpperCase( )

Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**. The default locale governs the conversion in both cases.

Here is an example that uses **toLowerCase( )** and **toUpperCase( )**:

```
// Demonstrate toUpperCase() and toLowerCase().

class ChangeCase {
  public static void main(String args[])
  {
    String s = "This is a test.";

    System.out.println("Original: " + s);

    String upper = s.toUpperCase();
    String lower = s.toLowerCase();
    System.out.println("Uppercase: " + upper);
    System.out.println("Lowercase: " + lower);
  }
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

One other point: Overloaded versions of **toLowerCase( )** and **toUpperCase( )** that let you specify a **Locale** object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

JDK 8 adds a new method to **String** called **join( )**. It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:

static String join(CharSequence *delim*, CharSequence . . . *strs*)

Here, *delim* specifies the delimiter used to separate the character sequences specified by *strs*. Because **String** implements the **CharSequence** interface, *strs* can be a list of strings. (See Chapter 17 for information on **CharSequence**.) The following program demonstrates this version of **join( )**:

```
// Demonstrate the join() method defined by String.
class StringJoinDemo {
  public static void main(String args[]) {

    String result = String.join(" ", "Alpha", "Beta", "Gamma");
    System.out.println(result);

    result = String.join(", ", "John", "ID#: 569",
                         "E-mail: John@HerbSchildt.com");
    System.out.println(result);
  }
}
```

The output is shown here:

```
Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com
```

In the first call to **join( )**, a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character.

The second form of **join( )** lets you join a list of strings obtained from an object that implements the **Iterable** interface. **Iterable** is implemented by the Collections Framework classes described in Chapter 18, among others. See Chapter 17 for information on **Iterable**.

Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods, including those summarized in the following table:

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 19 for details on formatting.)
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 19 for details on formatting.)
boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.

Method	Description
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **String**.

Notice that several of these methods work with regular expressions. Regular expressions are described in Chapter 30.

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

```
StringBuffer( )
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

length( ) and capacity( )

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity can be found through the **capacity( )** method. They have the following general forms:

```
int length( )
int capacity( )
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity( )

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**. **ensureCapacity( )** has this general form:

void ensureCapacity(int *minCapacity*)

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength( )

To set the length of the string within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

void setLength(int *len*)

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength( )** with a value less than the current value returned by **length( )**, then the characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength( )** to shorten a **StringBuffer**.

charAt( ) and setCharAt( )

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

char charAt(int *where*)
void setCharAt(int *where*, char *ch*)

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt( )** and **setCharAt( )**:

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " + sb.charAt(1));

    sb.setCharAt(1, 'i');
    sb.setLength(2);
    System.out.println("buffer after = " + sb);
    System.out.println("charAt(1) after = " + sb.charAt(1));
  }
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append( )

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)

The string representation of each parameter is obtained, often by calling **String.valueOf( )**. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append( )**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

The output of this example is shown here:

```
a = 42!
```

insert( )

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **String**s, **Object**s, and **CharSequence**s. Like **append( )**, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*)
StringBuffer insert(int *index*, char *ch*)
StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");
    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

The output of this example is shown here:

```
I like Java!
```

reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*)
StringBuffer deleteCharAt(int *loc*)

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete( )** and **deleteCharAt( )** methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");
    sb.delete(4, 7);
    System.out.println("After delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

Here is the output:

```
After replace: This was a test.
```

substring( )

You can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

String substring(int *startIndex*)
String substring(int *startIndex*, int *endIndex*)

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies several others, including those summarized in the following table:

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **StringBuffer**.
void trimToSize( )	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

```
class IndexOfDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("one two one");
    int i;

    i = sb.indexOf("one");
    System.out.println("First index: " + i);

    i = sb.lastIndexOf("one");
    System.out.println("Last index: " + i);
  }
}
```

The output is shown here:

```
First index: 0
Last index: 8
```

StringBuilder

Introduced by JDK 5, **StringBuilder** is a relatively recent addition to Java's string handling capabilities. **StringBuilder** is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

This page has been intentionally left blank

CHAPTER

17 Exploring java.lang

This chapter discusses those classes and interfaces defined by **java.lang**. As you know, **java.lang** is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java's most widely used package.

java.lang includes the following classes:

Boolean	Enum	Process	String
Byte	Float	ProcessBuilder	StringBuffer
Character	InheritableThreadLocal	ProcessBuilder.Redirect	StringBuilder
Character.Subset	Integer	Runtime	System
Character.UnicodeBlock	Long	RuntimePermission	Thread
Class	Math	SecurityManager	ThreadGroup
ClassLoader	Number	Short	ThreadLocal
ClassValue	Object	StackTraceElement	Throwable
Compiler	Package	StrictMath	Void
Double			

java.lang defines the following interfaces:

Appendable	Cloneable	Readable
AutoCloseable	Comparable	Runnable
CharSequence	Iterable	Thread.UncaughtExceptionHandler

Several of the classes contained in **java.lang** contain deprecated methods, most dating back to Java 1.0. These deprecated methods are still provided by Java to support an ever-shrinking pool of legacy code and are not recommended for new code. Because of this, the deprecated methods are not discussed here.

Primitive Type Wrappers

As mentioned in Part I of this book, Java uses primitive types, such as **int** and **char**, for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the *same instance* of an **int**. At times, you will need to create an object representation for one of these primitive types. For example, there are collection classes discussed in Chapter 18 that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class. To address this need, Java provides classes that correspond to each of the primitive types. In essence, these classes encapsulate, or *wrap*, the primitive types within a class. Thus, they are commonly referred to as *type wrappers*. The type wrappers were introduced in Chapter 12. They are examined in detail here.

Number

The abstract class **Number** defines a superclass that is implemented by the classes that wrap the numeric types **byte**, **short**, **int**, **long**, **float**, and **double**. **Number** has abstract methods that return the value of the object in each of the different number formats. For example, **doubleValue( )** returns the value as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

The values returned by these methods might be rounded, truncated, or result in a "garbage" value due to the effects of a narrowing conversion.

Number has concrete subclasses that hold explicit values of each primitive numeric type: **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long**.

Double and Float

Double and **Float** are wrappers for floating-point values of type **double** and **float**, respectively. The constructors for **Float** are shown here:

Float(double *num*)
Float(float *num*)
Float(String *str*) throws NumberFormatException

As you can see, **Float** objects can be constructed with values of type **float** or **double**. They can also be constructed from the string representation of a floating-point number.

The constructors for **Double** are shown here:

Double(double *num*)
Double(String *str*) throws NumberFormatException

Double objects can be constructed with a **double** value or a string containing a floating-point value.

The methods defined by **Float** include those shown in Table 17-1. The methods defined by **Double** include those shown in Table 17-2. Both **Float** and **Double** define the following constants:

BYTES	The width of a **float** or **double** in bytes (Added by JDK 8.)
MAX_EXPONENT	Maximum exponent
MAX_VALUE	Maximum positive value
MIN_EXPONENT	Minimum exponent
MIN_NORMAL	Minimum positive normal value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
NEGATIVE_INFINITY	Negative infinity
SIZE	The bit width of the wrapped value
TYPE	The **Class** object for **float** or **double**

Method	**Description**
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(float *num1*, float *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Float *f*)	Compares the numerical value of the invoking object with that of *f*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *FloatObj*)	Returns **true** if the invoking **Float** object is equivalent to *FloatObj*. Otherwise, it returns **false**.
static int floatToIntBits(float *num*)	Returns the IEEE-compatible, single-precision bit pattern that corresponds to *num*.
static int floatToRawIntBits(float *num*)	Returns the IEEE-compatible single-precision bit pattern that corresponds to *num*. A NaN value is preserved.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(float *num*)	Returns the hash code for *num*. (Added by JDK 8.)
static float intBitsToFloat(int *num*)	Returns **float** equivalent of the IEEE-compatible, single-precision bit pattern specified by *num*.

Table 17-1 The Methods Defined by **Float**

Method	Description
int intValue( )	Returns the value of the invoking object as an **int**.
static boolean isFinite(float *num*)	Returns **true** if *num* is not **NaN** and is not infinite. (Added by JDK 8.)
boolean isInfinite( )	Returns **true** if the invoking object contains an infinite value. Otherwise, it returns **false**.
static boolean isInfinite(float *num*)	Returns **true** if *num* specifies an infinite value. Otherwise, it returns **false**.
boolean isNaN( )	Returns **true** if the invoking object contains a value that is not a number. Otherwise, it returns **false**.
static boolean isNaN(float *num*)	Returns **true** if *num* specifies a value that is not a number. Otherwise, it returns **false**.
long longValue( )	Returns the value of the invoking object as a **long**.
static float max(float *val*, float *val2*)	Returns the maximum of *val* and *val2*. (Added by JDK 8.)
static float min(float *val*, float *val2*)	Returns the minimum of *val* and *val2*. (Added by JDK 8.)
static float parseFloat(String *str*) throws NumberFormatException	Returns the **float** equivalent of the number contained in the string specified by *str* using radix 10.
short shortValue( )	Returns the value of the invoking object as a **short**.
static float sum(float *val*, float *val2*)	Returns the result of *val* + *val2*. (Added by JDK 8.)
static String toHexString(float *num*)	Returns a string containing the value of *num* in hexadecimal format.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(float *num*)	Returns the string equivalent of the value specified by *num*.
static Float valueOf(float *num*)	Returns a **Float** object containing the value passed in *num*.
static Float valueOf(String *str*) throws NumberFormatException	Returns the **Float** object that contains the value specified by the string in *str*.

Table 17-1 The Methods Defined by **Float** *(continued)*

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(double *num1*, double *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.

Table 17-2 The Methods Defined by **Double**

Method	Description
int compareTo(Double *d*)	Compares the numerical value of the invoking object with that of *d*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static long doubleToLongBits(double *num*)	Returns the IEEE-compatible, double-precision bit pattern that corresponds to *num*.
static long doubleToRawLongBits(double *num*)	Returns the IEEE-compatible double-precision bit pattern that corresponds to *num*. A NaN value is preserved.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *DoubleObj*)	Returns **true** if the invoking **Double** object is equivalent to *DoubleObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashcode( )	Returns the hash code for the invoking object.
static int hashCode(double *num*)	Returns the hash code for *num*. (Added by JDK 8.)
int intValue( )	Returns the value of the invoking object as an **int**.
static boolean isFinite(double *num*)	Returns **true** if *num* is not **NaN** and is not infinite. (Added by JDK 8.)
boolean isInfinite( )	Returns **true** if the invoking object contains an infinite value. Otherwise, it returns **false**.
static boolean isInfinite(double *num*)	Returns **true** if *num* specifies an infinite value. Otherwise, it returns **false**.
boolean isNaN( )	Returns **true** if the invoking object contains a value that is not a number. Otherwise, it returns **false**.
static boolean isNaN(double *num*)	Returns **true** if *num* specifies a value that is not a number. Otherwise, it returns **false**.
static double longBitsToDouble(long *num*)	Returns **double** equivalent of the IEEE-compatible, double-precision bit pattern specified by *num*.
long longValue( )	Returns the value of the invoking object as a **long**.
static double max(double *val*, double *val2*)	Returns the maximum of *val* and *val2*. (Added by JDK 8.)
static double min(double *val*, double *val2*)	Returns the minimum of *val* and *val2*. (Added by JDK 8.)
static double parseDouble(String *str*) throws NumberFormatException	Returns the **double** equivalent of the number contained in the string specified by *str* using radix 10.

Table 17-2 The Methods Defined by **Double** *(continued)*

Method	Description
short shortValue( )	Returns the value of the invoking object as a **short**.
static double sum(double *val*, double *val2*)	Returns the result of *val* + *val2*. (Added by JDK 8.)
static String toHexString(double *num*)	Returns a string containing the value of *num* in hexadecimal format.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(double *num*)	Returns the string equivalent of the value specified by *num*.
static Double valueOf(double *num*)	Returns a **Double** object containing the value passed in *num*.
static Double valueOf(String *str*) throws NumberFormatException	Returns a **Double** object that contains the value specified by the string in *str*.

Table 17-2 The Methods Defined by **Double** *(continued)*

The following example creates two **Double** objects—one by using a **double** value and the other by passing a string that can be parsed as a **double**:

```
class DoubleDemo {
  public static void main(String args[]) {
    Double d1 = new Double(3.14159);
    Double d2 = new Double("314159E-5");

    System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
  }
}
```

As you can see from the following output, both constructors created identical **Double** instances, as shown by the **equals( )** method returning **true**:

```
   3.14159 = 3.14159 -> true
```

Understanding isInfinite( ) and isNaN( )

Float and **Double** provide the methods **isInfinite( )** and **isNaN( )**, which help when manipulating two special **double** and **float** values. These methods test for two unique values defined by the IEEE floating-point specification: infinity and NaN (not a number). **isInfinite( )** returns **true** if the value being tested is infinitely large or small in magnitude. **isNaN( )** returns **true** if the value being tested is not a number.

The following example creates two **Double** objects; one is infinite, and the other is not a number:

```
// Demonstrate isInfinite() and isNaN()
class InfNaN {
  public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2 = new Double(0/0.);

    System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
    System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
  }
}
```

This program generates the following output:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer, and Long

The **Byte**, **Short**, **Integer**, and **Long** classes are wrappers for **byte**, **short**, **int**, and **long** integer types, respectively. Their constructors are shown here:

Byte(byte *num*)
Byte(String *str*) throws NumberFormatException

Short(short *num*)
Short(String *str*) throws NumberFormatException

Integer(int *num*)
Integer(String *str*) throws NumberFormatException

Long(long *num*)
Long(String *str*) throws NumberFormatException

As you can see, these objects can be constructed from numeric values or from strings that contain valid whole number values.

The methods defined by these classes are shown in Tables 17-3 through 17-6. As you can see, they define methods for parsing integers from strings and converting strings back into integers. Variants of these methods allow you to specify the *radix*, or numeric base, for conversion. Common radixes are 2 for binary, 8 for octal, 10 for decimal, and 16 for hexadecimal.

The following constants are defined:

BYTES	The width of the integer type in bytes (Added by JDK 8.)
MIN_VALUE	Minimum value
MAX_VALUE	Maximum value
SIZE	The bit width of the wrapped value
TYPE	The **Class** object for **byte**, **short**, **int**, or **long**

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(byte *num1*, byte *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Byte *b*)	Compares the numerical value of the invoking object with that of *b*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static Byte decode(String *str*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *ByteObj*)	Returns **true** if the invoking **Byte** object is equivalent to *ByteObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(byte *num*)	Returns the hash code for *num*. (Added by JDK 8.)
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static byte parseByte(String *str*) throws NumberFormatException	Returns the **byte** equivalent of the number contained in the string specified by *str* using radix 10.
static byte parseByte(String *str*, int *radix*) throws NumberFormatException	Returns the **byte** equivalent of the number contained in the string specified by *str* using the specified radix.
short shortValue( )	Returns the value of the invoking object as a **short**.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(byte *num*)	Returns a string that contains the decimal equivalent of *num*.
static int toUnsignedInt(byte *val*)	Returns the value of *val* as an unsigned integer. (Added by JDK 8.)
static long toUnsignedLong(byte *val*)	Returns the value of *val* as an unsigned long integer. (Added by JDK 8.)
static Byte valueOf(byte *num*)	Returns a **Byte** object containing the value passed in *num*.
static Byte valueOf(String *str*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str*.
static Byte valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str* using the specified *radix*.

Table 17-3 The Methods Defined by **Byte**

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(short *num1*, short *num2*	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Short *s*)	Compares the numerical value of the invoking object with that of *s*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static Short decode(String *str*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *ShortObj*)	Returns **true** if the invoking **Short** object is equivalent to *ShortObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(short *num*)	Returns the hash code for *num*. (Added by JDK 8.)
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static short parseShort(String *str*) throws NumberFormatException	Returns the **short** equivalent of the number contained in the string specified by *str* using radix 10.
static short parseShort(String *str*, int *radix*) throws NumberFormatException	Returns the **short** equivalent of the number contained in the string specified by *str* using the specified *radix*.
static short reverseBytes(short *num*)	Exchanges the high- and low-order bytes of *num* and returns the result.
short shortValue( )	Returns the value of the invoking object as a **short**.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(short *num*)	Returns a string that contains the decimal equivalent of *num*.
static int toUnsignedInt(short *val*)	Returns the value of *val* as an unsigned integer. (Added by JDK 8.)
static long toUnsignedLong(short *val*)	Returns the value of *val* as an unsigned long integer. (Added by JDK 8.)
static Short valueOf(short *num*)	Returns a **Short** object containing the value passed in *num*.
static Short valueOf(String *str*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str* using radix 10.
static Short valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str* using the specified *radix*.

Table 17-4 The Methods Defined by **Short**

Method	Description
static int bitCount(int *num*)	Returns the number of set bits in *num*.
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(int *num1*, int *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Integer *i*)	Compares the numerical value of the invoking object with that of *i*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(int *num1*, int *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*. (Added by JDK 8.)
static Integer decode(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static int divideUnsigned(int *dividend*, int *divisor*)	Returns the result, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.)
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *IntegerObj*)	Returns **true** if the invoking **Integer** object is equivalent to *IntegerObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
static Integer getInteger(String *propertyName*)	Returns the value associated with the environmental property specified by *propertyName*. A **null** is returned on failure.
static Integer getInteger(String *propertyName*, int *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
static Integer getInteger(String *propertyName*, Integer *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(int *num*)	Returns the hash code for *num*. (Added by JDK 8.)
static int highestOneBit(int *num*)	Determines the position of the highest order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static int lowestOneBit(int *num*)	Determines the position of the lowest order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static int max(int *val*, int *val2*)	Returns the maximum of *val* and *val2*. (Added by JDK 8.)
static int min(int *val*, int *val2*)	Returns the minimum of *val* and *val2*. (Added by JDK 8.)
static int numberOfLeadingZeros(int *num*)	Returns the number of high-order zero bits that precede the first high-order set bit in *num*. If *num* is zero, 32 is returned.

Table 17-5 The Methods Defined by **Integer**

Method	Description
static int numberOfTrailingZeros(int *num*)	Returns the number of low-order zero bits that precede the first low-order set bit in *num*. If *num* is zero, 32 is returned.
static int parseInt(String *str*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using radix 10.
static int parseInt(String *str*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using the specified *radix*.
static int parseUnsignedInt(String *str*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10. (Added by JDK 8.)
static int parseUnsignedInt(String *str*, int *radix*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix specified by *radix*. (Added by JDK 8.)
static int remainderUnsigned(int *dividend*, int *divisor*)	Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.)
static int reverse(int *num*)	Reverses the order of the bits in *num* and returns the result.
static int reverseBytes(int *num*)	Reverses the order of the bytes in *num* and returns the result.
static int rotateLeft(int *num*, int *n*)	Returns the result of rotating *num* left *n* positions.
static int rotateRight(int *num*, int *n*)	Returns the result of rotating *num* right *n* positions.
short shortValue( )	Returns the value of the invoking object as a **short**.
static int signum(int *num*)	Returns –1 if *num* is negative, 0 if it is zero, and 1 if it is positive.
static int sum(int *val*, int *val2*)	Returns the result of *val* + *val2*. (Added by JDK 8.)
static String toBinaryString(int *num*)	Returns a string that contains the binary equivalent of *num*.
static String toHexString(int *num*)	Returns a string that contains the hexadecimal equivalent of *num*.
static String toOctalString(int *num*)	Returns a string that contains the octal equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(int *num*)	Returns a string that contains the decimal equivalent of *num*.
static String toString(int *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static long toUnsignedLong(int *val*)	Returns the value of *val* as an unsigned long integer. (Added by JDK 8.)
static String toUnsignedString(int *val*)	Returns a string that contains the decimal value of *val* as an unsigned integer. (Added by JDK 8.)
static String toUnsignedString(int *val*, int *radix*)	Returns a string that contains the value of *val* as an unsigned integer in the radix specified by *radix*. (Added by JDK 8.)
static Integer valueOf(int *num*)	Returns an **Integer** object containing the value passed in *num*.
static Integer valueOf(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static Integer valueOf(String *str*, int *radix*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str* using the specified *radix*.

Table 17-5 The Methods Defined by **Integer** *(continued)*

Method	Description
static int bitCount(long *num*)	Returns the number of set bits in *num*.
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(long *num1*, long *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Long *l*)	Compares the numerical value of the invoking object with that of *l*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(long *num1*, long *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*. (Added by JDK 8.)
static Long decode(String *str*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str*.
static long divideUnsigned(long *dividend*, long *divisor*)	Returns the result, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.)
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *LongObj*)	Returns **true** if the invoking **Long** object is equivalent to *LongObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
static Long getLong(String *propertyName*)	Returns the value associated with the environmental property specified by *propertyName*. A **null** is returned on failure.
static Long getLong(String *propertyName*, long *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
static Long getLong(String *propertyName*, Long *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(long *num*)	Returns the hash code for *num*. (Added by JDK 8.)
static long highestOneBit(long *num*)	Determines the position of the highest-order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static long lowestOneBit(long *num*)	Determines the position of the lowest-order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static long max(long *val*, long *val2*)	Returns the maximum of *val* and *val2*. (Added by JDK 8.)
static long min(long *val*, long *val2*)	Returns the minimum of *val* and *val2*. (Added by JDK 8.)
static int numberOfLeadingZeros(long *num*)	Returns the number of high-order zero bits that precede the first high-order set bit in *num*. If *num* is zero, 64 is returned.

Table 17-6 The Methods Defined by **Long**

Method	Description
static int numberOfTrailingZeros(long *num*)	Returns the number of low-order zero bits that precede the first low-order set bit in *num*. If *num* is zero, 64 is returned.
static long parseLong(String *str*) throws NumberFormatException	Returns the **long** equivalent of the number contained in the string specified by *str* using radix 10.
static long parseLong(String *str*, int *radix*) throws NumberFormatException	Returns the **long** equivalent of the number contained in the string specified by *str* using the specified *radix*.
static long parseUnsignedLong(String *str*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10. (Added by JDK 8.)
static long parseUnsignedLong(String *str*, int *radix*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix specified by *radix*. (Added by JDK 8.)
static long remainderUnsigned(long *dividend*, long *divisor*)	Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.)
static long reverse(long *num*)	Reverses the order of the bits in *num* and returns the result.
static long reverseBytes(long *num*)	Reverses the order of the bytes in *num* and returns the result.
static long rotateLeft(long *num*, int *n*)	Returns the result of rotating *num* left *n* positions.
static long rotateRight(long *num*, int *n*)	Returns the result of rotating *num* right *n* positions.
short shortValue( )	Returns the value of the invoking object as a **short**.
static int signum(long *num*)	Returns –1 if *num* is negative, 0 if it is zero, and 1 if it is positive.
static long sum(long *val*, long *val2*)	Returns the result of *val* + *val2*. (Added by JDK 8.)
static String toBinaryString(long *num*)	Returns a string that contains the binary equivalent of *num*.
static String toHexString(long *num*)	Returns a string that contains the hexadecimal equivalent of *num*.
static String toOctalString(long *num*)	Returns a string that contains the octal equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(long *num*)	Returns a string that contains the decimal equivalent of *num*.
static String toString(long *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static String toUnsignedString(long *val*)	Returns a string that contains the decimal value of *val* as an unsigned integer. (Added by JDK 8.)
static String toUnsignedString(long *val*, int *radix*)	Returns a string that contains the value of *val* as an unsigned integer in the radix specified by *radix*. (Added by JDK 8.)
static Long valueOf(long *num*)	Returns a **Long** object containing the value passed in *num*.
static Long valueOf(String *str*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str*.
static Long valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str* using the specified *radix*.

Table 17-6 The Methods Defined by **Long** *(continued)*

Converting Numbers to and from Strings

One of the most common programming chores is converting the string representation of a number into its internal, binary format. Fortunately, Java provides an easy way to accomplish this. The **Byte**, **Short**, **Integer**, and **Long** classes provide the **parseByte( )**, **parseShort( )**, **parseInt( )**, and **parseLong( )** methods, respectively. These methods return the **byte**, **short**, **int**, or **long** equivalent of the numeric string with which they are called. (Similar methods also exist for the **Float** and **Double** classes.)

The following program demonstrates **parseInt( )**. It sums a list of integers entered by the user. It reads the integers using **readLine( )** and uses **parseInt( )** to convert these strings into their **int** equivalents.

```
/* This program sums a list of numbers entered
   by the user.  It converts the string representation
   of each number into an int using parseInt().
*/

import java.io.*;

class ParseDemo {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;
    int i;
    int sum=0;

    System.out.println("Enter numbers, 0 to quit.");
    do {
      str = br.readLine();
      try {
        i = Integer.parseInt(str);
      } catch(NumberFormatException e) {
        System.out.println("Invalid format");
        i = 0;
      }
      sum += i;
      System.out.println("Current sum is: " + sum);
    } while(i != 0);
  }
}
```

To convert a whole number into a decimal string, use the versions of **toString( )** defined in the **Byte**, **Short**, **Integer**, or **Long** classes. The **Integer** and **Long** classes also provide the methods **toBinaryString( )**, **toHexString( )**, and **toOctalString( )**, which convert a value into a binary, hexadecimal, or octal string, respectively.

The following program demonstrates binary, hexadecimal, and octal conversion:

```
/* Convert an integer into binary, hexadecimal,
   and octal.
*/

class StringConversions {
  public static void main(String args[]) {
    int num = 19648;
    System.out.println(num + " in binary: " +
                       Integer.toBinaryString(num));

    System.out.println(num + " in octal: " +
                       Integer.toOctalString(num));

    System.out.println(num + " in hexadecimal: " +
                       Integer.toHexString(num));
  }
}
```

The output of this program is shown here:

```
    19648 in binary: 100110011000000
    19648 in octal: 46300
    19648 in hexadecimal: 4cc0
```

Character

Character is a simple wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the character.

The **Character** class defines several constants, including the following:

BYTES	The width of a **char** in bytes (Added by JDK 8.)
MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX_VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The **Class** object for **char**

Character includes several static methods that categorize characters and alter their case. A sampling is shown in Table 17-7. The following example demonstrates several of these methods:

```
// Demonstrate several Is... methods.

class IsDemo {
  public static void main(String args[]) {
    char a[] = {'a', 'b', '5', '?', 'A', ' '};

    for(int i=0; i<a.length; i++) {
      if(Character.isDigit(a[i]))
        System.out.println(a[i] + " is a digit.");
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + " is a letter.");
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + " is whitespace.");
      if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " is uppercase.");
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " is lowercase.");
    }
  }
}
```

The output from this program is shown here:

```
a is a letter.
a is lowercase.
b is a letter.
b is lowercase.
5 is a digit.
A is a letter.
A is uppercase.
  is whitespace.
```

Character defines two methods, **forDigit( )** and **digit( )**, that enable you to convert between integer values and the digits they represent. They are shown here:

static char forDigit(int *num*, int *radix*)
static int digit(char *digit*, int *radix*)

forDigit( ) returns the digit character associated with the value of *num*. The radix of the conversion is specified by *radix*. **digit( )** returns the integer value associated with the specified character (which is presumably a digit) according to the specified radix. (There is a second form of **digit( )** that takes a code point. See the following section for a discussion of code points.)

Another method defined by **Character** is **compareTo( )**, which has the following form:

int compareTo(Character *c*)

It returns zero if the invoking object and *c* have the same value. It returns a negative value if the invoking object has a lower value. Otherwise, it returns a positive value.

Method	**Description**
static boolean isDefined(char *ch*)	Returns **true** if *ch* is defined by Unicode. Otherwise, it returns **false**.
static boolean isDigit(char *ch*)	Returns **true** if *ch* is a digit. Otherwise, it returns **false**.
static boolean isIdentifierIgnorable(char *ch*)	Returns **true** if *ch* should be ignored in an identifier. Otherwise, it returns **false**.
static boolean isISOControl(char *ch*)	Returns **true** if *ch* is an ISO control character. Otherwise, it returns **false**.
static boolean isJavaIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Java identifier (other than the first character). Otherwise, it returns **false**.
static boolean isJavaIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Java identifier. Otherwise, it returns **false**.
static boolean isLetter(char *ch*)	Returns **true** if *ch* is a letter. Otherwise, it returns false.
static boolean isLetterOrDigit(char *ch*)	Returns **true** if *ch* is a letter or a digit. Otherwise, it returns **false**.
static boolean isLowerCase(char *ch*)	Returns **true** if *ch* is a lowercase letter. Otherwise, it returns **false**.
static boolean isMirrored(char *ch*)	Returns **true** if *ch* is a mirrored Unicode character. A mirrored character is one that is reversed for text that is displayed right-to-left.
static boolean isSpaceChar(char *ch*)	Returns **true** if *ch* is a Unicode space character. Otherwise, it returns **false**.
static boolean isTitleCase(char *ch*)	Returns **true** if *ch* is a Unicode titlecase character. Otherwise, it returns **false**.
static boolean isUnicodeIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Unicode identifier (other than the first character). Otherwise, it returns **false**.
static Boolean isUnicodeIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Unicode identifier. Otherwise, it returns **false**.
static boolean isUpperCase(char *ch*)	Returns **true** if *ch* is an uppercase letter. Otherwise, it returns **false**.
static boolean isWhitespace(char *ch*)	Returns **true** if *ch* is whitespace. Otherwise, it returns **false**.
static char toLowerCase(char *ch*)	Returns lowercase equivalent of *ch*.
static char toTitleCase(char *ch*)	Returns titlecase equivalent of *ch*.
static char toUpperCase(char *ch*)	Returns uppercase equivalent of *ch*.

Table 17-7 Various **Character** Methods

Character includes a method called **getDirectionality( )** which can be used to determine the direction of a character. Several constants are defined that describe directionality. Most programs will not need to use character directionality.

Character also overrides the **equals( )** and **hashCode( )** methods.

Two other character-related classes are **Character.Subset**, used to describe a subset of Unicode, and **Character.UnicodeBlock**, which contains Unicode character blocks.

Additions to Character for Unicode Code Point Support

Relatively recently, major additions were made to **Character**. Beginning with JDK 5, the **Character** class has included support for 32-bit Unicode characters. In the past, all Unicode characters could be held by 16 bits, which is the size of a **char** (and the size of the value encapsulated within a **Character**), because those values ranged from 0 to FFFF. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to 10FFFF.

Here are three important terms. A *code point* is a character in the range 0 to 10FFFF. Characters that have values greater than FFFF are called *supplemental characters*. The *basic multilingual plane (BMP)* are those characters between 0 and FFFF.

The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a **char** can hold, some means of handling the supplemental characters was needed. Java addressed this problem in two ways. First, Java uses two **chars** to represent a supplemental character. The first **char** is called the *high surrogate*, and the second is called the *low surrogate*. New methods, such as **codePointAt( )**, were provided to translate between code points and supplemental characters.

Secondly, Java overloaded several preexisting methods in the **Character** class. The overloaded forms use **int** rather than **char** data. Because an **int** is large enough to hold any character as a single value, it can be used to store any character. For example, all of the methods in Table 17-7 have overloaded forms that operate on **int**. Here is a sampling:

static boolean isDigit(int *cp*)
static boolean isLetter(int *cp*)
static int toLowerCase(int *cp*)

In addition to the methods overloaded to accept code points, **Character** adds methods that provide additional support for code points. A sampling is shown in Table 17-8.

Boolean

Boolean is a very thin wrapper around **boolean** values, which is useful mostly when you want to pass a **boolean** variable by reference. It contains the constants **TRUE** and **FALSE**, which define true and false **Boolean** objects. **Boolean** also defines the **TYPE** field, which is the **Class** object for **boolean**. **Boolean** defines these constructors:

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

Method	Description
static int charCount(int *cp*)	Returns 1 if *cp* can be represented by a single **char**. It returns 2 if two **char**s are needed.
static int codePointAt(CharSequence *chars*, int *loc*)	Returns the code point at the location specified by *loc*.
static int codePointAt(char *chars*[ ], int *loc*)	Returns the code point at the location specified by *loc*.
static int codePointBefore(CharSequence *chars*, int *loc*)	Returns the code point at the location that precedes that specified by *loc*.
static int codePointBefore(char *chars*[ ], int *loc*)	Returns the code point at the location that precedes that specified by *loc*.
static boolean isBmpCodePoint(int *cp*)	Returns **true** if *cp* is part of the basic multilingual plane and **false** otherwise.
static boolean isHighSurrogate(char *ch*)	Returns **true** if *ch* contains a valid high surrogate character.
static boolean isLowSurrogate(char *ch*)	Returns **true** if *ch* contains a valid low surrogate character.
static boolean isSupplementaryCodePoint(int *cp*)	Returns **true** if *cp* contains a supplemental character.
static boolean isSurrogatePair(char *highCh*, char *lowCh*)	Returns **true** if *highCh* and *lowCh* form a valid surrogate pair.
static boolean isValidCodePoint(int *cp*)	Returns **true** if *cp* contains a valid code point.
static char[ ] toChars(int *cp*)	Converts the code point in *cp* into its **char** equivalent, which might require two **char**s. An array holding the result is returned.
static int toChars(int *cp*, char *target*[ ], int *loc*)	Converts the code point in *cp* into its **char** equivalent, storing the result in *target*, beginning at *loc*. Returns 1 if *cp* can be represented by a single **char**. It returns 2 otherwise.
static int toCodePoint(char *highCh*, char *lowCh*)	Converts *highCh* and *lowCh* into their equivalent code point.

Table 17-8 A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be **true**. Otherwise, it will be **false**.

Boolean defines the methods shown in Table 17-9.

Method	Description
boolean booleanValue( )	Returns **boolean** equivalent.
static int compare(boolean *b1*, boolean *b2*)	Returns zero if *b1* and *b2* contain the same value. Returns a positive value if *b1* is **true** and *b2* is **false**. Otherwise, returns a negative value.
int compareTo(Boolean *b*)	Returns zero if the invoking object and *b* contain the same value. Returns a positive value if the invoking object is **true** and *b* is **false**. Otherwise, returns a negative value.
boolean equals(Object *boolObj*)	Returns **true** if the invoking object is equivalent to *boolObj*. Otherwise, it returns **false**.
static Boolean getBoolean(String *propertyName*)	Returns **true** if the system property specified by *propertyName* is **true**. Otherwise, it returns **false**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(boolean *boolVal*)	Returns the hash code for *boolVal*. (Added by JDK 8.)
static boolean logicalAnd(boolean *op1*, boolean *op2*)	Performs a logical AND of *op1* and *op2* and returns the result. (Added by JDK 8.)
static boolean logicalOr(boolean *op1*, boolean *op2*)	Performs a logical OR of *op1* and *op2* and returns the result. (Added by JDK 8.)
static boolean logicalXor(boolean *op1*, boolean *op2*)	Performs a logical XOR of *op1* and *op2* and returns the result. (Added by JDK 8.)
static boolean parseBoolean(String *str*)	Returns **true** if *str* contains the string "true". Case is not significant. Otherwise, returns **false**.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(boolean *boolVal*)	Returns the string equivalent of *boolVal*.
static Boolean valueOf(boolean *boolVal*)	Returns the **Boolean** equivalent of *boolVal*.
static Boolean valueOf(String *boolString*)	Returns **true** if *boolString* contains the string "true" (in uppercase or lowercase). Otherwise, it returns **false**.

Table 17-9 The Methods Defined by **Boolean**

Void

The **Void** class has one field, **TYPE**, which holds a reference to the **Class** object for type **void**. You do not create instances of this class.

Process

The abstract **Process** class encapsulates a *process*—that is, an executing program. It is used primarily as a superclass for the type of objects created by **exec( )** in the **Runtime** class, or by **start( )** in the **ProcessBuilder** class. **Process** contains the methods shown in Table 17-10.

Method	Description
void destroy( )	Terminates the process.
Process destroyForcibly( )	Forces termination of the invoking process. Returns a reference to the process. (Added by JDK 8.)
int exitValue( )	Returns an exit code obtained from a subprocess.
InputStream getErrorStream( )	Returns an input stream that reads input from the process' **err** output stream.
InputStream getInputStream( )	Returns an input stream that reads input from the process' **out** output stream.
OutputStream getOutputStream( )	Returns an output stream that writes output to the process' **in** input stream.
boolean isAlive( )	Returns **true** if the invoking process is still active. Otherwise, returns **false**. (Added by JDK 8.)
int waitFor( ) throws InterruptedException	Returns the exit code returned by the process. This method does not return until the process on which it is called terminates.
boolean waitFor(long *waitTime*, TimeUnit *timeUnit*) throws InterruptedException	Waits for the invoking process to end. The amount of time to wait is specified by *waitTime* in the units specified by *timeUnit*. Returns **true** if the process has ended and **false** if the wait time runs out. (Added by JDK 8.)

Table 17-10 The Methods Defined by **Process**

Runtime

The **Runtime** class encapsulates the run-time environment. You cannot instantiate a **Runtime** object. However, you can get a reference to the current **Runtime** object by calling the static method **Runtime.getRuntime( )**. Once you obtain a reference to the current **Runtime** object, you can call several methods that control the state and behavior of the Java Virtual Machine. Applets and other untrusted code typically cannot call any of the **Runtime** methods without raising a **SecurityException**. Several commonly used methods defined by **Runtime** are shown in Table 17-11.

Method	Description
void addShutdownHook(Thread *thrd*)	Registers *thrd* as a thread to be run when the Java Virtual Machine terminates.
Process exec(String *progName*) throws IOException	Executes the program specified by *progName* as a separate process. An object of type **Process** is returned that describes the new process.
Process exec(String *progName*, String *environment*[ ]) throws IOException	Executes the program specified by *progName* as a separate process with the environment specified by *environment*. An object of type **Process** is returned that describes the new process.
Process exec(String *comLineArray*[ ]) throws IOException	Executes the command line specified by the strings in *comLineArray* as a separate process. An object of type **Process** is returned that describes the new process.

Table 17-11 A Sampling of Methods Defined by **Runtime**

Method	Description
Process exec(String *comLineArray*[ ], String *environment*[ ]) throws IOException	Executes the command line specified by the strings in *comLineArray* as a separate process with the environment specified by *environment*. An object of type **Process** is returned that describes the new process.
void exit(int *exitCode*)	Halts execution and returns the value of *exitCode* to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error.
long freeMemory( )	Returns the approximate number of bytes of free memory available to the Java run-time system.
void gc( )	Initiates garbage collection.
static Runtime getRuntime( )	Returns the current **Runtime** object.
void halt(int *code*)	Immediately terminates the Java Virtual Machine. No termination threads or finalizers are run. The value of *code* is returned to the invoking process.
void load(String *libraryFileName*)	Loads the dynamic library whose file is specified by *libraryFileName*, which must specify its complete path.
void loadLibrary(String *libraryName*)	Loads the dynamic library whose name is associated with *libraryName*.
Boolean removeShutdownHook(Thread *thrd*)	Removes *thrd* from the list of threads to run when the Java Virtual Machine terminates. It returns **true** if successful—that is, if the thread was removed.
void runFinalization( )	Initiates calls to the **finalize( )** methods of unused but not yet recycled objects.
long totalMemory( )	Returns the total number of bytes of memory available to the program.
void traceInstructions(boolean *traceOn*)	Turns on or off instruction tracing, depending upon the value of *traceOn*. If *traceOn* is **true**, the trace is displayed. If it is **false**, tracing is turned off.
void traceMethodCalls(boolean *traceOn*)	Turns on or off method call tracing, depending upon the value of *traceOn*. If *traceOn* is **true**, the trace is displayed. If it is **false**, tracing is turned off.

Table 17-11 A Sampling of Methods Defined by **Runtime** *(continued)*

Let's look at two of the most common uses of the **Runtime** class: memory management and executing additional processes.

Memory Management

Although Java provides automatic garbage collection, sometimes you will want to know how large the object heap is and how much of it is left. You can use this information, for example, to check your code for efficiency or to approximate how many more objects of a certain type can be instantiated. To obtain these values, use the **totalMemory( )** and **freeMemory( )** methods.

As mentioned in Part I, Java's garbage collector runs periodically to recycle unused objects. However, sometimes you will want to collect discarded objects prior to the collector's next appointed rounds. You can run the garbage collector on demand by calling the **gc( )** method. A good thing to try is to call **gc( )** and then call **freeMemory( )** to get a baseline memory usage. Next, execute your code and call **freeMemory( )** again to see how much memory it is allocating. The following program illustrates this idea:

```
// Demonstrate totalMemory(), freeMemory() and gc().

class MemoryDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    long mem1, mem2;
    Integer someints[] = new Integer[1000];

    System.out.println("Total memory is: " +
                       r.totalMemory());
    mem1 = r.freeMemory();
    System.out.println("Initial free memory: " + mem1);
    r.gc();
    mem1 = r.freeMemory();
    System.out.println("Free memory after garbage collection: "
                       + mem1);

    for(int i=0; i<1000; i++)
      someints[i] = new Integer(i); // allocate integers

    mem2 = r.freeMemory();
    System.out.println("Free memory after allocation: "
                       + mem2);
    System.out.println("Memory used by allocation: "
                       + (mem1-mem2));

    // discard Integers
    for(int i=0; i<1000; i++) someints[i] = null;

    r.gc(); // request garbage collection

    mem2 = r.freeMemory();
    System.out.println("Free memory after collecting" +
                       " discarded Integers: " + mem2);

  }
}
```

Sample output from this program is shown here (of course, your actual results may vary):

```
Total memory is: 1048568
Initial free memory: 751392
Free memory after garbage collection: 841424
Free memory after allocation: 824000
Memory used by allocation: 17424
Free memory after collecting discarded Integers: 842640
```

Executing Other Programs

In safe environments, you can use Java to execute other heavyweight processes (that is, programs) on your multitasking operating system. Several forms of the **exec( )** method allow you to name the program you want to run as well as its input parameters. The **exec( )** method returns a **Process** object, which can then be used to control how your Java program interacts with this new running process. Because Java can run on a variety of platforms and under a variety of operating systems, **exec( )** is inherently environment-dependent.

The following example uses **exec( )** to launch **notepad**, Windows' simple text editor. Obviously, this example must be run under the Windows operating system.

```
// Demonstrate exec().
class ExecDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
  }
}
```

There are several alternative forms of **exec( )**, but the one shown in the example is the most common. The **Process** object returned by **exec( )** can be manipulated by **Process**' methods after the new program starts running. You can kill the subprocess with the **destroy( )** method. The **waitFor( )** method causes your program to wait until the subprocess finishes. The **exitValue( )** method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur. Here is the preceding **exec( )** example modified to wait for the running process to exit:

```
// Wait until notepad is terminated.
class ExecDemoFini {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
      p.waitFor();
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
    System.out.println("Notepad returned " + p.exitValue());
  }
}
```

While a subprocess is running, you can write to and read from its standard input and output. The **getOutputStream( )** and **getInputStream( )** methods return the handles to standard **in** and **out** of the subprocess. (I/O is examined in detail in Chapter 20.)

ProcessBuilder

ProcessBuilder provides another way to start and manage processes (that is, programs). As explained earlier, all processes are represented by the **Process** class, and a process can be started by **Runtime.exec( )**. **ProcessBuilder** offers more control over the processes. For example, you can set the current working directory.

ProcessBuilder defines these constructors:

```
ProcessBuilder(List<String> args)
ProcccessBuilder(String ... args)
```

Here, *args* is a list of arguments that specify the name of the program to be executed along with any required command-line arguments. In the first constructor, the arguments are passed in a **List**. In the second, they are specified through a varargs parameter. Table 17-12 describes the methods defined by **ProcessBuilder**.

In Table 17-12, notice the methods that use the **ProcessBuilder.Redirect** class. This abstract class encapsulates an I/O source or target linked to a subprocess. Among other things, these methods enable you to redirect the source or target of I/O operations. For example, you can redirect to a file by calling **to( )**, redirect from a file by calling **from( )**, and append to a file by calling **appendTo( )**. A **File** object linked to the file can be obtained by calling **file( )**. These methods are shown here:

```
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file( )
```

Another method supported by **ProcessBuilder.Redirect** is **type( )**, which returns a value of the enumeration type **ProcessBuilder.Redirect.Type**. This enumeration describes the type of the redirection. It defines these values: **APPEND**, **INHERIT**, **PIPE**, **READ**, or **WRITE**. **ProcessBuilder.Redirect** also defines the constants **INHERIT** and **PIPE**.

Method	Description
List<String> command( )	Returns a reference to a **List** that contains the name of the program and its arguments. Changes to this list affect the invoking object.
ProcessBuilder command(List<String> *args*)	Sets the name of the program and its arguments to those specified by *args*. Changes to this list affect the invoking object. Returns a reference to the invoking object.
ProcessBuilder command(String ... *args*)	Sets the name of the program and its arguments to those specified by *args*. Returns a reference to the invoking object.
File directory( )	Returns the current working directory of the invoking object. This value will be **null** if the directory is the same as that of the Java program that started the process.

Table 17-12 The Methods Defined by **ProcessBuilder**