Chapter 4: Mathematical Functions, Characters, and Strings

This chapter introduces methods for performing common mathematical operations.

1. Common Mathematical Functions

Java provides many useful methods in the **Math** class for performing common mathematical functions. This section introduces other useful methods in the **Math** class. They can be categorized as *trigonometric methods*, *exponent methods*, and *service methods*. Service methods include the rounding, min, max, absolute, and random methods.

In addition to methods, the **Math** class provides two useful **double** constants, **PI** and **E** (the base of natural logarithms). You can use these constants as **Math.PI** and **Math.E** in any program.

1.1 Trigonometric Methods

The **Math** class contains the following methods as listed in Table 4.1 for performing trigonometric functions:

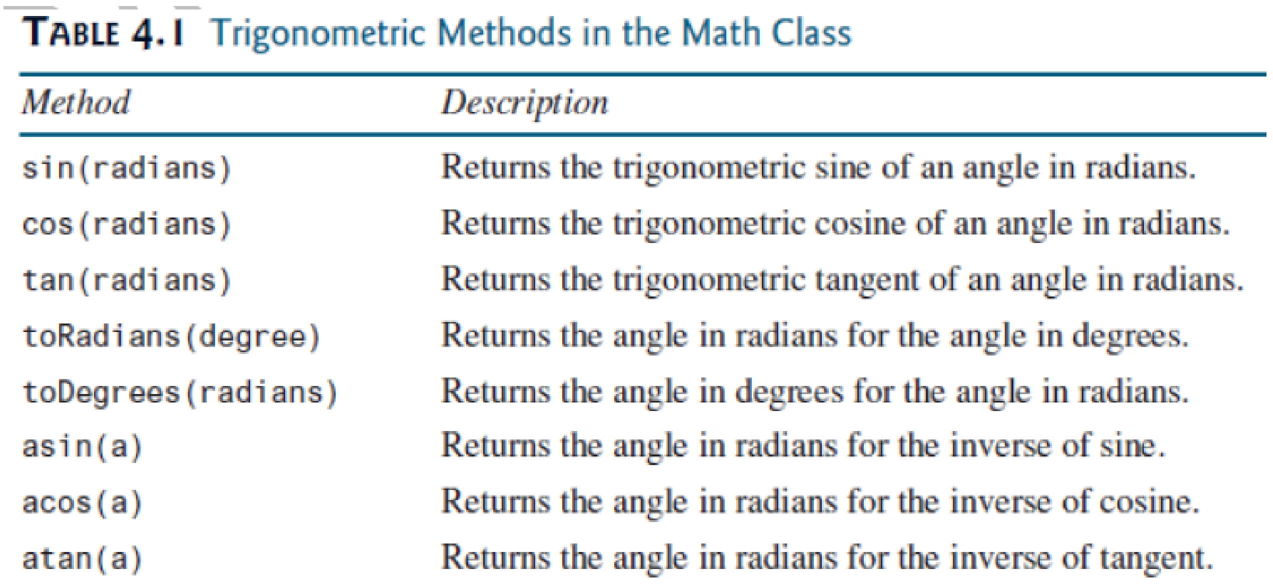


TABLE 4.1 Trigonometric Methods in the Math Class

Method	*Description*
`sin(radians)`	Returns the trigonometric sine of an angle in radians.
`cos(radians)`	Returns the trigonometric cosine of an angle in radians.
`tan(radians)`	Returns the trigonometric tangent of an angle in radians.
`toRadians(degree)`	Returns the angle in radians for the angle in degrees.
`toDegrees(radians)`	Returns the angle in degrees for the angle in radians.
`asin(a)`	Returns the angle in radians for the inverse of sine.
`acos(a)`	Returns the angle in radians for the inverse of cosine.
`atan(a)`	Returns the angle in radians for the inverse of tangent.

Examples:

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns 0.5236 (same as π/6)
Math.sin(0) returns 0.0
Math.sin(Math.toRadians(270)) returns -1.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns 0.866
Math.cos(Math.PI / 2) returns 0
Math.asin(0.5) returns 0.523598333 (same as π/6)
Math.acos(0.5) returns 1.0472 (same as π/3)
Math.atan(1.0) returns 0.785398 (same as π/4)
```

1.2 Exponent Methods

There are five methods related to exponents in the **Math** class as listed in Table 4.2.

TABLE 4.2 Exponent Methods in the Math Class

Method	*Description*
`exp(x)`	Returns e raised to power of x (e^x).
`log(x)`	Returns the natural logarithm of x ($\ln(x) = \log_e(x)$).
`log10(x)`	Returns the base 10 logarithm of x ($\log_{10}(x)$).
`pow(a, b)`	Returns a raised to the power of b (a^b).
`sqrt(x)`	Returns the square root of x ($\sqrt{x}$) for $x >= 0$.

For example,

$e^{3.5}$ is `Math.exp(3.5)`, which returns `33.11545`
$\ln(3.5)$ is `Math.log(3.5)`, which returns `1.25276`
$\log_{10}(3.5)$ is `Math.log10(3.5)`, which returns `0.544`
2^3 is `Math.pow(2, 3)`, which returns `8.0`
3^2 is `Math.pow(3, 2)`, which returns `9.0`
$4.5^{2.5}$ is `Math.pow(4.5, 2.5)`, which returns `42.9567`
$\sqrt{4}$ is `Math.sqrt(4)`, which returns `2.0`
$\sqrt{10.5}$ is `Math.sqrt(10.5)`, which returns `3.24`

1.3 The Rounding Methods

The **Math** class contains four rounding methods as listed in Table 4.3.

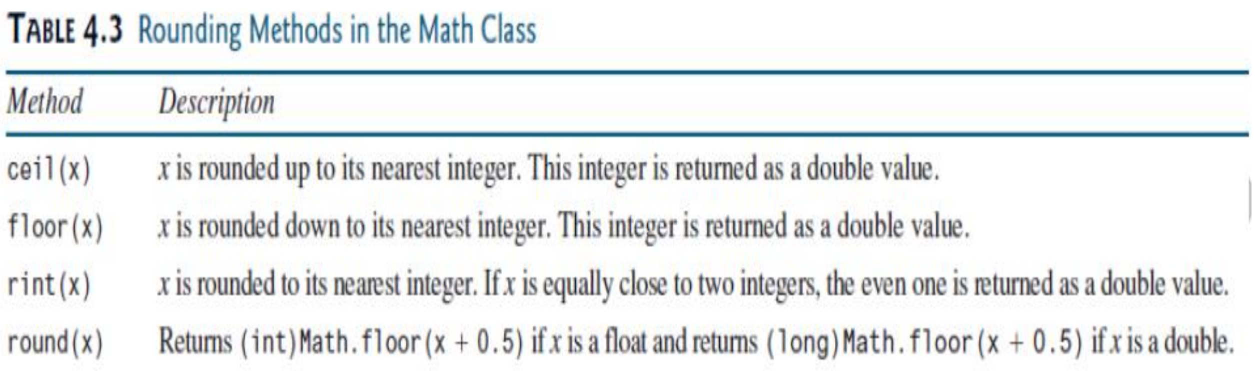
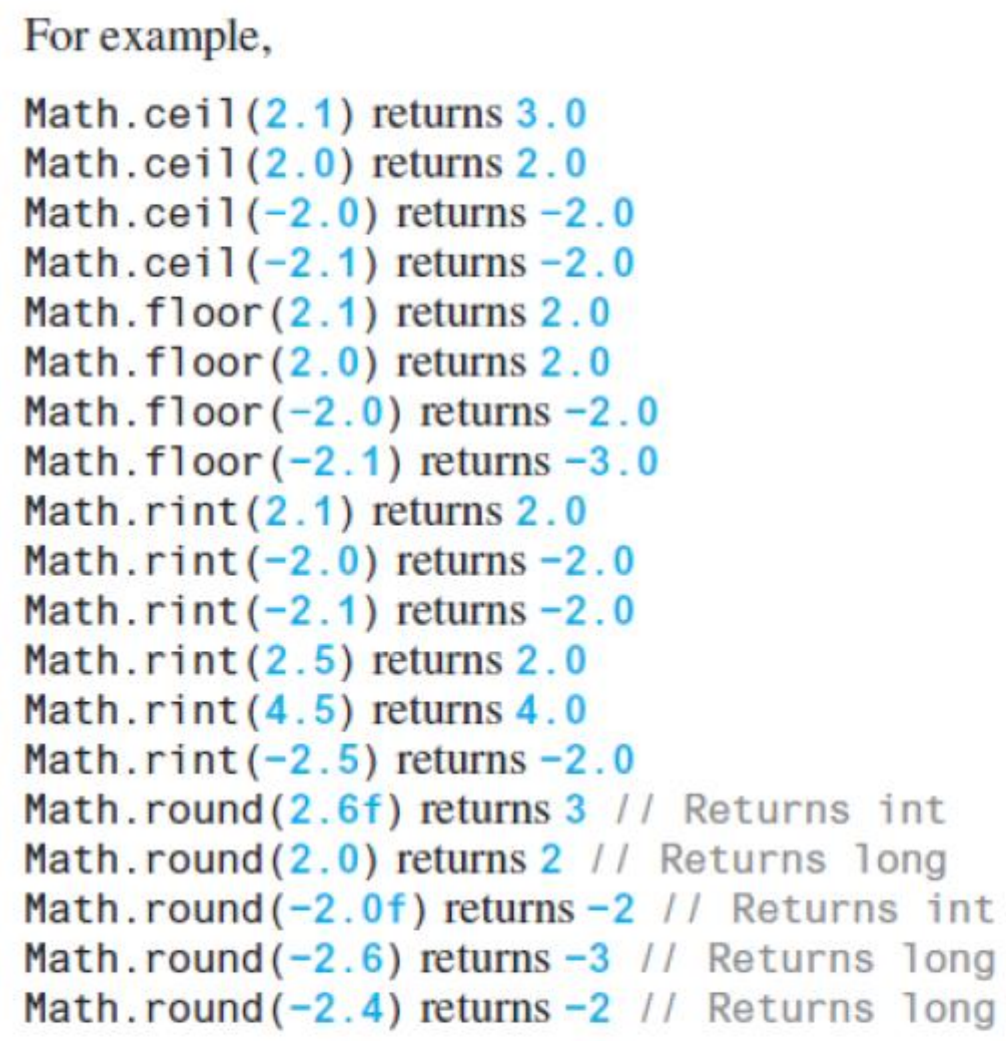


TABLE 4.3 Rounding Methods in the Math Class

Method	*Description*
`ceil(x)`	*x* is rounded up to its nearest integer. This integer is returned as a double value.
`floor(x)`	*x* is rounded down to its nearest integer. This integer is returned as a double value.
`rint(x)`	*x* is rounded to its nearest integer. If *x* is equally close to two integers, the even one is returned as a double value.
`round(x)`	Returns `(int)Math.floor(x + 0.5)` if *x* is a float and returns `(long)Math.floor(x + 0.5)` if *x* is a double.



For example,

```
Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1) returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5) returns 2.0
Math.rint(4.5) returns 4.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3 // Returns int
Math.round(2.0) returns 2 // Returns long
Math.round(-2.0f) returns -2 // Returns int
Math.round(-2.6) returns -3 // Returns long
Math.round(-2.4) returns -2 // Returns long
```

1.4 The min, max, and abs Methods

The **min** and **max** methods return the minimum and maximum numbers of two numbers (**int**, **long**, **float**, or **double**).
The **abs** method returns the absolute value of the number (**int**, **long**, **float**, or **double**). For example:

Math.max(**2**, **3**) returns **3**

Math.min(**2.5**, **4.6**) returns **2.5**

Math.max(Math.max(**2.5**, **4.6**), Math.min(**3**, **5.6**)) returns **4.6**

Math.abs(−**2**) returns **2**

Math.abs(−**2.1**) returns **2.1**

1.5 The random Method

We used the **random()** method in the preceding chapter. This method generates a random **double** value greater than or equal to 0.0 and less than 1.0 (**0** <= **Math.random()** < **1.0**). We can use it to write a simple expression to generate random numbers in any range. For example:

`(int)(Math.random() * 10)` ⟶ Returns a random integer between 0 and 9.

`50 + (int)(Math.random() * 50)` ⟶ Returns a random integer between 50 and 99.

In general,

`a + Math.random() * b` ⟶ Returns a random number between a and a + b, excluding a + b.

2. Character Data Type and Operations

In addition to processing numeric values, we can process characters in Java. The character data type, **char**, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```
char letter = 'A';
char numChar = '4';
```

The first statement assigns character **A** to the **char** variable **letter**. The second statement assigns digit character **4** to the **char** variable **numChar**.

Caution

A string literal must be enclosed in double quotation marks (" "). A character literal is a single character enclosed in single quotation marks (' '). Therefore, "A" is a string, but 'A' is a character.

For example, the following code tests whether a character **ch** is an uppercase letter, a lowercase letter, or a digital character:

```
if (ch >= 'A' && ch <= 'Z')
    System.out.println(ch + " is an uppercase letter");
else if (ch >= 'a' && ch <= 'z')
    System.out.println(ch + " is a lowercase letter");
else if (ch >= '0' && ch <= '9')
    System.out.println(ch + " is a numeric character");
```

For convenience, Java provides the following methods in the **Character** class for testing characters as listed in Table 4.6. The **Character** class is defined in the **java.lang** package.

TABLE 4.6 Methods in the Character Class

Method	*Description*
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOrDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

For example,

```
System.out.println("isDigit('a') is " + Character.isDigit('a'));
System.out.println("isLetter('a') is " + Character.isLetter('a'))
System.out.println("isLowerCase('a') is "
  + Character.isLowerCase('a'));
System.out.println("isUpperCase('a') is "
  + Character.isUpperCase('a'));
System.out.println("toLowerCase('T') is "
  + Character.toLowerCase('T'));
System.out.println("toUpperCase('q') is "
  + Character.toUpperCase('q'));
```

displays

```
isDigit('a') is false
isLetter('a') is true
isLowerCase('a') is true
isUpperCase('a') is false
toLowerCase('T') is t
toUpperCase('q') is Q
```

Note

The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character. For example, the following statements display character b:

```
char ch = 'a';
System.out.println(++ch);
```

3. The String Type

A string is a sequence of characters. The **char** type represents only one character. To represent a string of characters, use the data type called **String**. For example, the following code declares **message** to be a string with the value **"Welcome to Java"**.

String message = **"Welcome to Java"**;

String is a predefined class in the Java library, just like the classes **System** and **Scanner**. Here, **message** is a reference variable that references a string object with contents **Welcome to Java**.

Table 4.7 lists the **String** methods for obtaining string length, for accessing characters in the string, for concatenating string, for converting string to uppercases or lowercases, and for trimming a string.

TABLE 4.7 Simple Methods for `String` Objects

Method	*Description*
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string `s1`.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase.
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

Strings are objects in Java. The methods listed in Table 4.7 can only be invoked from a specific string instance. For this reason, these methods are called *instance methods*. A noninstance method is called a *static method*. A static method can be invoked without using an object. All the methods defined in the **Math** class are static methods. They are not tied to a specific object instance.

The syntax to invoke an instance method is **referenceVariable.methodName(arguments)**.

Recall that the syntax to invoke a static method is **ClassName.methodName(arguments)**. For example, the **pow** method in the **Math** class can be invoked using **Math.pow(2, 2.5)**.

3.1 Getting String Length

We can use the **length()** method to return the number of characters in a string. For example, the following code:

```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is " + message.length());
```

Displays:

```
The length of Welcome to Java is 15
```

Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the *string literal* to refer directly to strings without creating new variables. Thus, `"Welcome to Java".length()` is correct and returns `15`. Note that `""` denotes an *empty string* and `"".length()` is `0`.

3.2 Getting Characters from a String

The **s.charAt(index)** method can be used to retrieve a specific character in a string **s**, where the index is between **0** and **s.length()–1**. For example, **message.charAt(0)** returns the character **W**, as shown in Figure 4.1. Note that the index for the first character in the string is **0**.

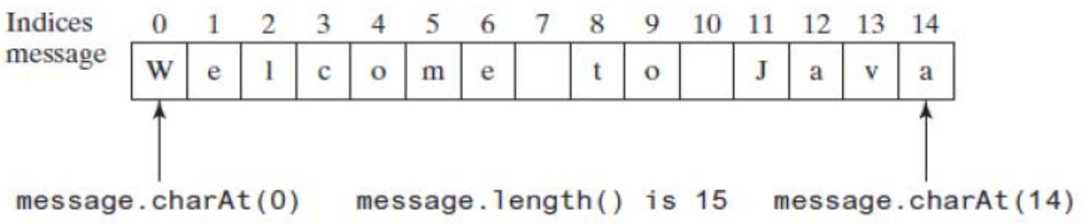


FIGURE 4.1 The characters in a `String` object can be accessed using its index.

Caution

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length()–1`. For example, `s.charAt(s.length())` would cause a `StringIndexOutOfBoundsException`.

3.3 Concatenating Strings

We can use the **concat** method to concatenate two strings. The statement given below, for example, concatenates strings **s1** and **s2** into **s3**:

String s3 = s1.concat(s2);

Because string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (+) operator to concatenate two strings, so the previous statement is equivalent to

String s3 = s1 + s2;

The following code combines the strings **message**, **" and "**, and **"HTML"** into one string:

String myString = message + **" and "** + **"HTML"**;

Recall that the **+** operator can also concatenate a number with a string. In this case, the number is converted into a string then concatenated. Note at least one of the operands must be a string in order for concatenation to take place. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. Here are some examples:

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

If neither of the operands is a string, the plus sign (+) is the addition operator that adds two numbers.

The augmented **+=** operator can also be used for string concatenation. For example, the following code appends the string **" and Java is fun"** with the string **"Welcome to Java"** in **message**.

```
message += " and Java is fun";
```

So the new `message` is `"Welcome to Java and Java is fun."`
If `i = 1` and `j = 2`, what is the output of the following statement?

The output is **"i + j is 12"** because **"i + j is"** is concatenated with the value of **i** first. To force **i + j** to be executed first, enclose **i + j** in the parentheses, as follows:

System.out.println(**"i + j is "** + (i + j));

3.4 Converting Strings

The **toLowerCase()** method returns a new string with all lowercase letters, and the **toUpperCase()** method returns a new string with all uppercase letters. For example:

"Welcome".toLowerCase() returns a new string **welcome**.
"Welcome".toUpperCase() returns a new string **WELCOME**.

3.5 Reading a String from the Console

To read a string from the console, invoke the **next()** method on a **Scanner** object. For example, the following code reads three strings from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

```
Enter three words separated by spaces: Welcome to Java
s1 is Welcome
s2 is to
s3 is Java
```

The **next()** method reads a string that ends with a whitespace character. We can use the **nextLine()** method to read an entire line of text. The **nextLine()** method reads a string that ends with the *Enter* key pressed. For example, the following statements read a line of text:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a line: ");
String s = input.nextLine();
System.out.println("The line entered is " + s);
```

```
Enter a line: Welcome to Java
The line entered is Welcome to Java
```

3. 6 Reading a Character from the Console

To read a character from the console, use the **nextLine()** method to read a string and then invoke the **charAt(0)** method on the string to return a character. For example, the following code reads a character from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

3.7 Comparing Strings

How do we compare the contents of two strings? We might attempt to use the = = operator, as follows:

```
 if (string1 = = string2)
       System.out.println("string1 and string2 are the same object");
else
      System.out.println("string1 and string2 are different objects");
```

However, the = = operator checks only whether **string1** and **string2** refer to the same object; it does not tell us whether they have the same contents. Therefore, we cannot use the = = operator to find out whether two string variables have the same contents.

In Java, the **String** class contains the methods, as listed in Table 4.8, for comparing two strings.

TABLE 4.8 Comparison Methods for `String` Objects

Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.
`contains(s1)`	Returns true if `s1` is a substring in this string.

The following code, for instance, can be used to compare two strings:

```
if (string1.equals(string2))
      System.out.println("string1 and string2 have the same contents");
else
     System.out.println("string1 and string2 are not equal");
```

For example, the following statements display **true** then **false**:

```
String s1 = "Welcome to Java";
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The **compareTo** method can also be used to compare two strings. For example, consider the following code:

s1.compareTo(s2)

The method returns the value **0** if **s1** is equal to **s2**, a value less than **0** if **s1** is lexicographically (i.e., in terms of Unicode ordering) less than **s2**, and a value greater than **0** if **s1** is lexicographically greater than **s2**.

The actual value returned from the **compareTo** method depends on the offset of the first two distinct characters in **s1** and **s2** from left to right. For example, suppose **s1** is **abc** and **s2** is **abg**, and **s1.compareTo(s2)** returns **−4**. The first two characters (**a** vs. **a**) from **s1** and **s2** are compared. Because they are equal, the second two characters (**b** vs. **b**) are compared.

Because they are also equal, the third two characters (**c** vs. **g**) are compared. Since the character **c** is **4** less than **g**, the comparison returns **−4**.

Caution
Syntax errors will occur if you compare strings by using relational operators >, >=, <, or <=. Instead, you have to use s1.compareTo(s2).

Note
The equals method returns true if two strings are equal, and false if they are not. The compareTo method returns 0, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The **String** class also provides the **equalsIgnoreCase** and **compareToIgnoreCase** methods for comparing strings. The **equalsIgnoreCase** and **compareToIgnoreCase** methods ignore the case of the letters when comparing two strings. We can also use **str.startsWith(prefix)** to check whether string **str** starts with a specified prefix, **str.endsWith(suffix)** to check whether string **str** ends with a specified suffix, and **str.contains(s1)** to check whether string **str** contains string **s1**. For example:

"Welcome to Java".startsWith("We") returns **true**.
"Welcome to Java".startsWith("we") returns **false**.
"Welcome to Java".endsWith("va") returns **true**.
"Welcome to Java".endsWith("v") returns **false**.
"Welcome to Java".contains("to") returns **true**.
"Welcome to Java".contains("To") returns **false**.

Listing 4.2 gives a program that prompts the user to enter two cities and displays them in alphabetical order. The program reads two strings for two cities (lines 9 and 11). If

input.nextLine() is replaced by **input.next()** (line 9), you cannot enter a string with spaces for **city1**. Since a city name may contain multiple words separated by spaces, the program uses the **nextLine** method to read a string (lines 9 and 11). Invoking **city1.compareTo(city2)** compares two strings **city1** with **city2** (line 13). A negative return value indicates that **city1** is less than **city2**.

LISTING 4.2 OrderTwoCities.java

```java
 1  import java.util.Scanner;
 2
 3  public class OrderTwoCities {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6
 7      // Prompt the user to enter two cities
 8      System.out.print("Enter the first city: ");
 9      String city1 = input.nextLine();
10      System.out.print("Enter the second city: ");
11      String city2 = input.nextLine();
12
13      if (city1.compareTo(city2) < 0)
14        System.out.println("The cities in alphabetical order are " +
15          city1 + " " + city2);
16      else
17        System.out.println("The cities in alphabetical order are " +
18          city2 + " " + city1);
19    }
20  }
```

```
Enter the first city: New York ↵Enter
Enter the second city: Boston ↵Enter
The cities in alphabetical order are Boston New York
```

3. 8 Obtaining Substrings

We can obtain a single character from a string using the **charAt** method. We can also obtain a substring from a string using the **substring** method (see Figure 4.2) in the **String** class, as given in Table 4.9.

For example,

String message = **"Welcome to Java"**;

String message = message.substring(**0**,**11**) + **"HTML"**;

The string **message** now becomes **Welcome to HTML**.

TABLE 4.9 The String Class Contains the Methods for Obtaining Substrings

Method	*Description*
substring(beginIndex)	Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string, as shown in Figure 4.2.
substring(beginIndex, endIndex)	Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1, as shown in Figure 4.2. Note the character at endIndex is not part of the substring.

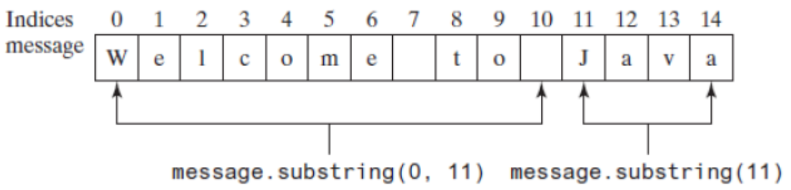


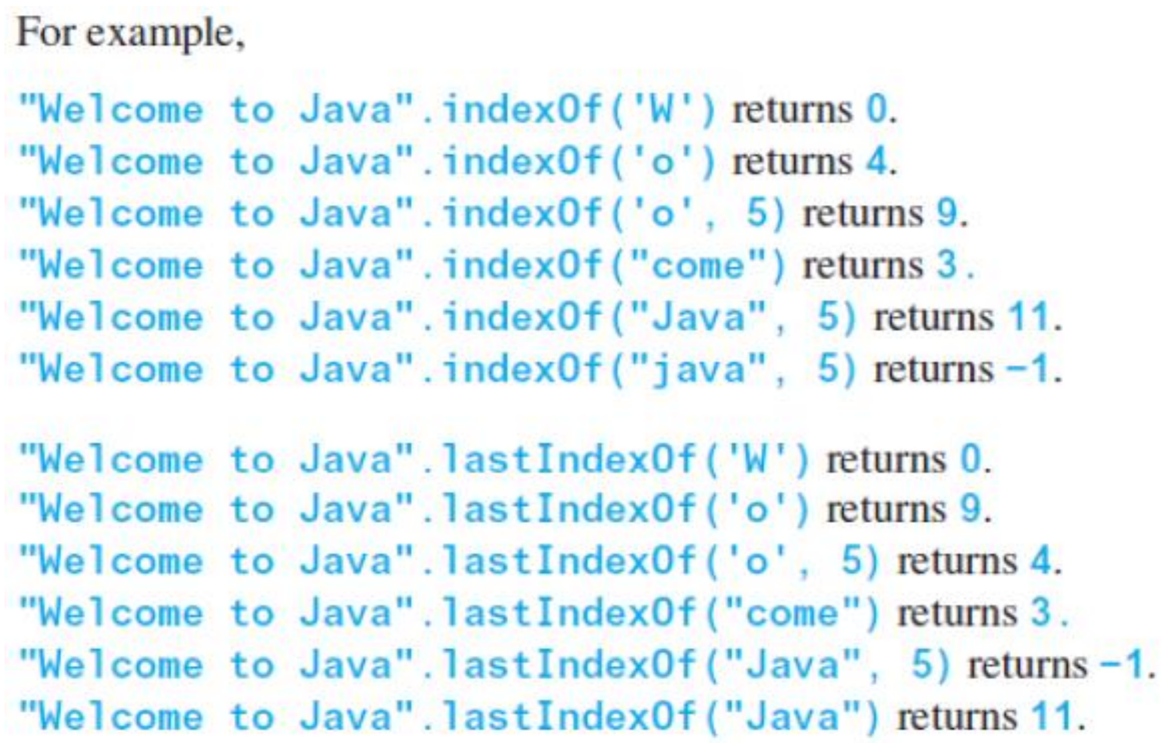
FIGURE 4.2 The substring method obtains a substring from a string.

Note
If beginIndex is endIndex, substring(beginIndex, endIndex) returns an empty string with length 0. If beginIndex > endIndex, it would be a runtime error.

The **String** class provides several versions of **indexOf** and **lastIndexOf** methods to find a character or a substring in a string, as listed in Table 4.10.

TABLE 4.10 The String Class Contains the Methods for Finding Substrings

Method	*Description*
index Of (ch)	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
indexOf(ch, fromIndex)	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
indexOf(s)	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
indexOf(s, fromIndex)	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
lastIndexOf(ch)	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
lastIndexOf(ch, fromIndex)	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
lastIndexOf(s)	Returns the index of the last occurrence of string s. Returns -1 if not matched.
lastIndexOf(s, fromIndex)	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.



For example,

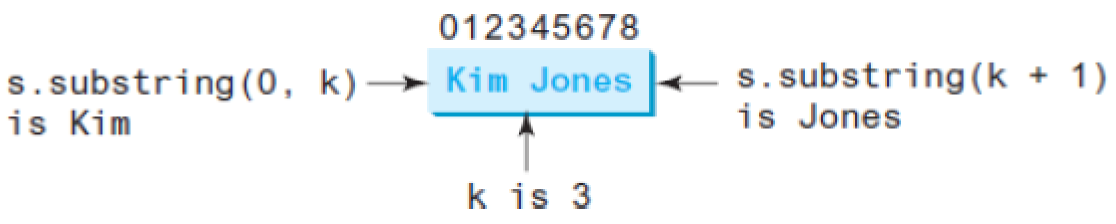
```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('o') returns 4.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.

"Welcome to Java".lastIndexOf('W') returns 0.
"Welcome to Java".lastIndexOf('o') returns 9.
"Welcome to Java".lastIndexOf('o', 5) returns 4.
"Welcome to Java".lastIndexOf("come") returns 3.
"Welcome to Java".lastIndexOf("Java", 5) returns -1.
"Welcome to Java".lastIndexOf("Java") returns 11.
```

Suppose that a string **s** contains the first name and last name separated by a space. We can use the following code to extract the first name and last name from the string:

```
int k = s.indexOf(' ');
    String firstName = s.substring(0, k);
    String lastName = s.substring(k + 1);
```

For example, if **s** is **Kim Jones**, the following diagram illustrates how the first name and last name are extracted.



3.9 Conversion between Strings and Numbers

We can convert a numeric string into a number. To convert a string into an **int** value, we use the **Integer.parseInt** method, as follows:

```
int intValue = Integer.parseInt(intString);
```

where **intString** is a numeric string such as **"123"**.

To convert a string into a **double** value, we use the **Double.parseDouble** method, as follows:

```
double doubleValue = Double.parseDouble(doubleString);
```

where **doubleString** is a numeric string such as **"123.45"**.

4. Formatting Console Output

Often, it is desirable to display numbers in a certain format. For example, the following code computes interest, given the amount and the annual interest rate:

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is $" + interest);
```

```
Interest is $16.404674
```

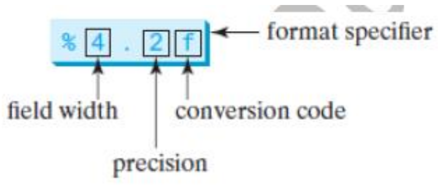
Because the interest amount is currency, it is desirable to display only two digits after the decimal point. To do this, you can write the code as follows:

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is $"
  + (int)(interest * 100) / 100.0);
```

```
Interest is $16.4
```

However, the format is still not correct. There should be two digits after the decimal point: **16.40** rather than **16.4**. We can fix it by using the **printf** method, as follows:

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.printf("Interest is $%4.2f",
  interest);
```



```
Interest is $16.40
```

The **f** in the **printf** stands for formatted, implying that the method prints an item in some format. The syntax to invoke this method is

System.out.printf(format, item1, item2, ..., item*k*);

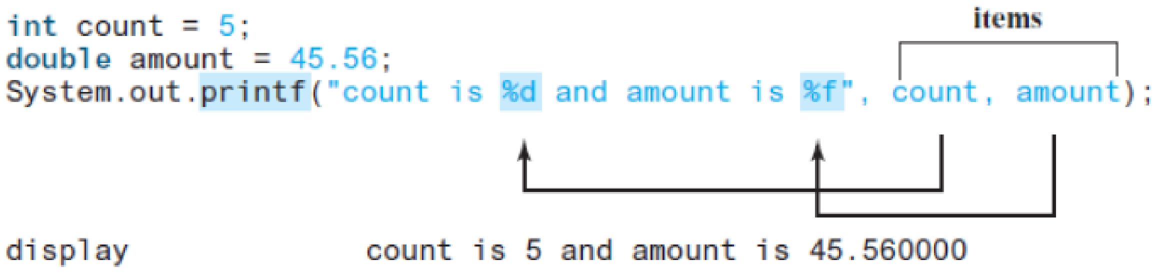
where **format** is a string that may consist of substrings and format specifiers.

A *format specifier* specifies how an item should be formatted. An item may be a numeric value, a character, a Boolean value, or a string. A simple format specifier consists of a percent sign (**%**) followed by a conversion code. Table 4.11 lists some frequently used simple format specifiers.

TABLE 4.11 Frequently Used Format Specifiers

Format Specifier	*Output*	*Example*
`%b`	A Boolean value	True or false
`%c`	A character	'a'
`%d`	A decimal integer	200
`%f`	A floating-point number	45.460000
`%e`	A number in standard scientific notation	4.556000e+01
`%s`	A string	"Java is cool"

Here is an example:



```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display              count is 5 and amount is 45.560000
```