Chapter 4

Mathematical Functions, Characters and Strings

Objectives

- To solve mathematical problems by using the methods in the **Math** class
- To represent characters using the **char** type
- To encode characters using ASCII and Unicode
- To represent special characters using the escape sequences
- To compare and test characters using the static methods in the **Character** class
- To represent strings using the **String** object, and using some of the common String methods for obtaining string length, for accessing characters in the string, for concatenating strings, for converting a string to upper or lowercases, for trimming a string, for comparing strings and to obtain substrings.
- To introduce objects and instance methods
- To read a character and strings from the console
- To format output using the **System.out.printf** method

Common Mathematical Functions

- *Java provides many useful methods in the* **Math** *class for performing common mathematical functions.*
- A method is a group of statements that performs a specific task. You have already used the **pow(a, b)** method to compute a^b in Section 2.9.4, and the **random()** method for generating a random number in Section 3.7.
- This section introduces other useful methods in the **Math** class. In particular, the *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.

Rounding Methods

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

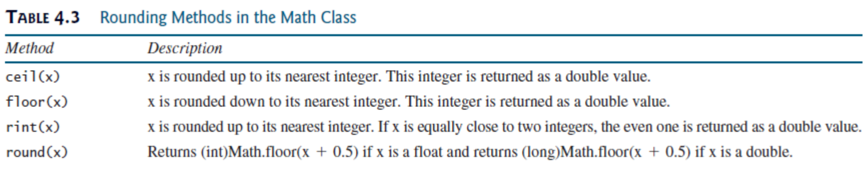


TABLE 4.3 Rounding Methods in the Math Class

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 up 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.

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.max(**4.4**, **5.0**) returns **5.0**
  - Math.min(**2.5**, **4.6**) returns **2.5**
  - Math.abs(**-2**) returns **2**
  - Math.abs(**-2.1**) returns **2.1**

The **random** Method

- You have 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**).
- You can use it to write a simple expression to generate random numbers in any range. In general,

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

- For example,

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

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

Character Data Type and Operations

- 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';
```

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

Unicode

- A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character.
- Java supports *Unicode*, originally designed as a 16-bit character encoding. The primitive data type **char** was intended to take advantage of this design by providing a simple data type that could hold any character.
- However, it turned out that the 65,536 characters possible in a 16-bit encoding are not sufficient to represent all the characters in the world.
- The Unicode standard therefore has been extended to allow up to 1,112,064 characters.
- Those characters that go beyond the original 16-bit limit are called *supplementary characters*.
- For simplicity, this book considers only the original 16-bit Unicode characters. These characters can be stored in a **char** type variable.
- A 16-bit Unicode takes two bytes, preceded by **\u**, expressed in four hexadecimal digits that run from **\u0000** to **\uFFFF**.

ASCII code

- Most computers use *ASCII* (*American Standard Code for Information Interchange*), an 8-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters.
- Unicode includes ASCII code, with **\u0000** to **\u007F** corresponding to the 128 ASCII characters.
- Table 4.4 shows the ASCII code for some commonly used characters. Appendix B, 'The ASCII Character Set,' gives a complete list of ASCII characters and their decimal and hexadecimal codes.

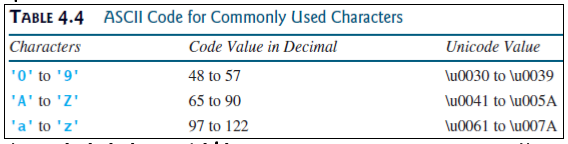


TABLE 4.4 ASCII Code for Commonly Used Characters

Characters	*Code Value in Decimal*	*Unicode Value*
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

- You can use ASCII characters such as **'X'**, **'1'**, and **'$'** in a Java program as well as Unicodes. Thus, for example, the following statements are equivalent:
  - **char** letter = **'A'**;
  - **char** letter = **'\u0041'**; // Character A's Unicode is 0041

Casting between **char** and Numeric Types

- When a floating-point value is cast into a **char**, the floating-point value is first cast into an **int**, which is then cast into a **char**.

  char ch = (**char**)**65.25**; // Decimal 65 is assigned to ch

  System.out.println(ch); // ch is character A

- When a **char** is cast into a numeric type, the character's Unicode is cast into the specified numeric type.

  int i = (**int**)**'A'**; // The Unicode of character A is assigned to i

  System.out.println(i); // i is 65

Casting between **char** and Numeric Types

- Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of **'a'** is **97**, which is within the range of a byte, these implicit castings are fine:

```
byte b = 'a';
int i = 'a';
```

- But the following casting is incorrect, because the Unicode **\uFFF4** cannot fit into a byte:

```
byte b = '\uFFF4';
```

- Any positive integer between **0** and **FFFF** in hexadecimal can be cast into a character implicitly.

Escape Sequences for Special Characters

- The following statement has a compile error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of characters.

```
System.out.println("He said "Java is fun"");
```

- To overcome this problem, Java uses a special notation to represent special characters. This special notation, called an *escape sequence*, consists of a backslash (****) followed by a character or a combination of digits.
- So, now you can print the quoted message using the following statement: `System.out.println("He said \"Java is fun\"");`

Escape Sequences

- The backslash \ is called an *escape character*. It is a special character. To display this character, you have to use an escape sequence \\. For example, the following code

  System.out.println(**"\\\\t is a tab character"**);

  displays

  \t is a tab character

TABLE 4.5 Escape Sequences

Escape Sequence	*Name*	*Unicode Code*
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000A
\f	Formfeed	\u000C
\r	Carriage Return	\u000D
\\\\	Backslash	\u005C
\\"	Double Quote	\u0022

Comparing and Testing Characters

- Two characters can be compared using the relational operators just like comparing two numbers. This is done by comparing the Unicodes of the two characters. For example,
  - **'a'** < **'b'** is true because the Unicode for **'a'** (**97**) is less than the Unicode for **'b'** (**98**).
  - **'a'** < **'A'** is false because the Unicode for **'a'** (**97**) is greater than the Unicode for **'A'** (**65**).
  - **'1'** < **'8'** is true because the Unicode for **'1'** (**49**) is less than the Unicode for **'8'** (**56**).
- For convenience, Java provides the following methods in the **Character** class for testing characters as shown in Table 4.6.

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.
isLetterOfDigit(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**'));

displays
isDigit('a') is false
isLetter('a') is true

The String Type

- 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"**;

- The **String** type is not a primitive type. It is known as a *reference type*. Reference data types will be discussed in detail in Chapter 9, Objects and Classes.
- Table 4.7 lists the **String** methods for obtaining string length, for accessing characters in the string, for concatenating strings, for converting a string to upper 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.

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

the index is between **0** and **string.length()–1**. For example, **message.charAt(0)** returns the character **W** and **s.charAt(s.length())** would cause a **StringIndexOutOfBoundsException**

Concatenating Strings

- You can use the **concat** method to concatenate two strings. For example: String s3 = s1.concat(s2);
- You can use the plus (**+**) operator to concatenate two strings, so the previous statement is equivalent to String s3 = s1 + s2;
- Recall that the **+** operator can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated.
- Note that at least one of the operands must be a string in order for concatenation to take place.
- The augmented **+=** operator can also be used for string concatenation. For example, message += **" and Java is fun"**;
- If **i = 1** and **j = 2**, what is the output of the following statement?

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

- 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));

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**.
- The **trim()** method returns a new string by eliminating whitespace characters from both ends of the string.
- The characters **' '**, **\t**, **\f**, **\r**, or **\n** are known as *whitespace characters*. For example, **"\t Good Night \n".trim()** returns a new string **Good Night**.

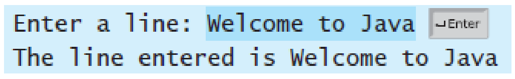
instance method vs static method

- Strings are objects in Java. The methods in Table 4.7 can only be invoked from a specific string instance. For this reason, these methods are called *instance methods*.
- A non-instance 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 **reference-variable.methodName(arguments)**.
- A method may have many arguments or no arguments. For example, the **charAt(index)** method has one argument, but the **length()** method has no 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)**
- **Are the methods in the Character class, as discussed in Table 4.6 (slide 12) static methods or instance methods?**

Reading a String from the Console

- To read a string from the console, invoke the **next()** method on a **Scanner** object.
- The **next()** method reads a string that ends with a whitespace character.
- You 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.

```
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 ↵Enter
The line entered is Welcome to Java
```

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);
```

Comparing Strings

- The **String** class contains the methods as shown in Table 4.8 for comparing two strings.

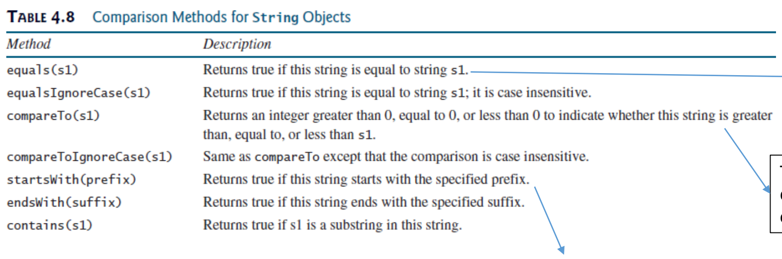


TABLE 4.8 Comparison Methods for `String` Objects

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.

String s1 = **"Welcome to Java"**;
String s2 = **"Welcome to Java"**;
System.out.println(s1.equals(s2)); // true

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.

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

String class: equals method

- How do you compare the contents of two strings? You 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 you whether they have the same contents.
- Therefore, you cannot use the **==** operator to find out whether two string variables have the same contents. Instead, you should use the **equals** method.
- The following code, for instance, can be used to compare two strings:

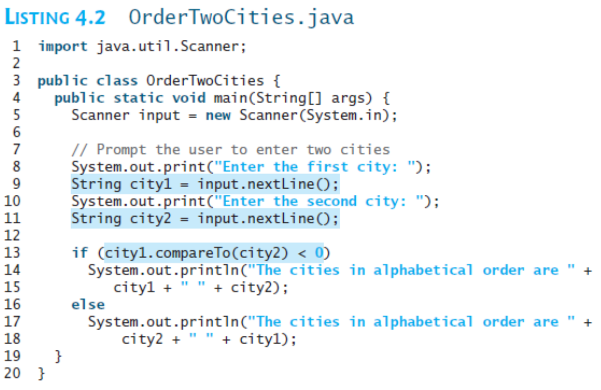
```
if (string1.equals(string2))
   System.out.println("string1 and string2 have the same contents");
else
   System.out.println("string1 and string2 are not equal");
```

String class: compareTo method

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

Comparing Strings: an example

- Listing 4.2 gives a program that prompts the user to enter two cities and displays them in alphabetical order.
- 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, 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

```
 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 }
```

Obtaining Substrings

- You can obtain a single character from a string using the **charAt** method. You can also obtain a substring from a string using the **substring** method in the **String** class, as shown in Table 4.9.

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 that the character at `endIndex` is not part of the substring.

- For example,

```
String message = "Welcome to Java";
String message = message.substring(0, 11) + "HTML";
```

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

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

Finding a Character or a Substring in a String

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

TABLE 4.10 The String class contains the methods for finding substrings.

Method	*Description*
`index(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.

Finding a Character or a Substring in a String - examples

- **For example,**
  - **"Welcome to Java".indexOf('o', 5)** returns **9**.
  - **"Welcome to Java".indexOf("come")** returns **3**.
  - **"Welcome to Java".indexOf("java", 5)** returns **-1**.
  - **"Welcome to Java".lastIndexOf('o')** returns **9**.
- Suppose a string **s** contains the first name and last name separated by a space. You 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);
```

Conversion between Strings and Numbers

- You can convert a numeric string into a number. To convert a string into an **int** value, 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, use the **Double.parseDouble** method, as follows:

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

  where **doubleString** is a numeric string such as **"123.45"**.
- If the string is not a numeric string, the conversion would cause a runtime error.
- The **Integer** and **Double** classes are both included in the **java.lang** package, and thus they are automatically imported.
- You can convert a number into a string, simply use the string concatenating operator as follows: `String s = number + "";`

Formatting Console Output

- *You can use the* **System.out.printf** *method to display formatted output on the console.*
- 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);
```

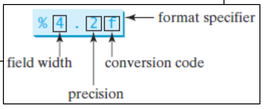
and the output is “Interest is $16.404674”

```
System.out.println("Interest is $“ + (int)(interest * 100) / 100.0);
```

and the output is “Interest is $16.4”

```
System.out.printf("Interest is $%4.2f", interest);
```

and the output is “Interest is $16.40”



Some Common Format specifier

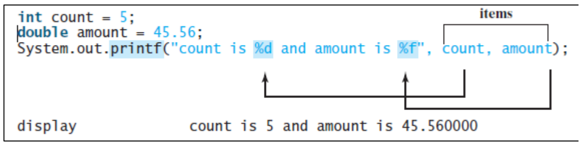
- The syntax to invoke **System.out.printf** *method* is

  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 displayed. 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”



```
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
```

Format specifier: specifying width and precision

- By default, a floating-point value is displayed with six digits after the decimal point.
- You can specify the width and precision in a format specifier, as shown in the examples in Table 4.12.
- If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

  System.out.printf(**"%3d#%2s#%4.2f\n"**, **1234**, **"Java"**, **51.6653**);

  displays

  1234#Java#51.67

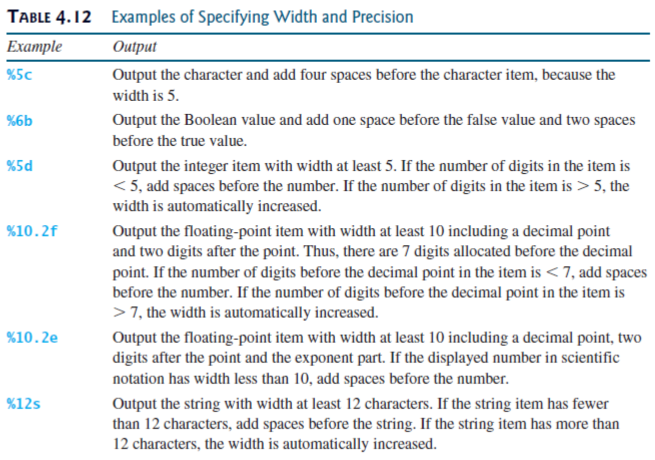


TABLE 4.12 Examples of Specifying Width and Precision

Example	*Output*
`%5c`	Output the character and add four spaces before the character item, because the width is 5.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%5d`	Output the integer item with width at least 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits in the item is > 5, the width is automatically increased.
`%10.2f`	Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus, there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is < 7, add spaces before the number. If the number of digits before the decimal point in the item is > 7, the width is automatically increased.
`%10.2e`	Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.
`%12s`	Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

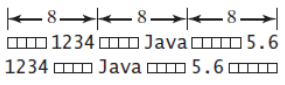
right justified vs left justified

- By default, the output is right justified. You can put the minus sign (**-**) in the format specifier to specify that the item is left justified in the output within the specified field.
- For example, the following statements

  System.out.printf(**"%8d%8s%8.1f\n"**, **1234**, **"Java"**, **5.63**);

  System.out.printf(**"%-8d%-8s%-8.1f \n"**, **1234**, **"Java"**, **5.63**);

  display



```
□□□□1234□□□□Java□□□□□5.6
1234□□□□Java□□□□5.6□□□□□
```

  where the square box (n) denotes a blank space.

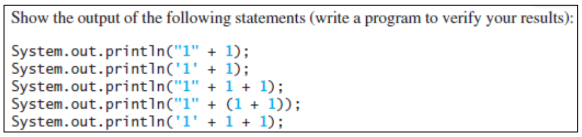
Some notes regarding the use of format specifiers

- The items must match the format specifiers in exact type. The item for the format specifier **%f** or **%e** must be a floating-point type value such as **40.0**, not **40**.
- Thus, an **int** variable cannot match **%f** or **%e**.
- The **%** sign denotes a format specifier. To output a literal **%** in the format string, use **%%**.

Chapter Summary

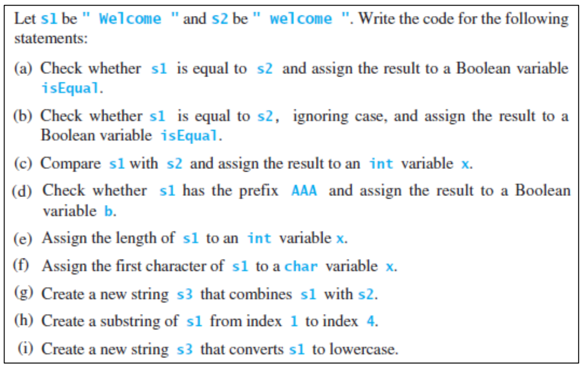
- Java provides the mathematical methods **pow**, **sqrt**, **cell**, **floor**, **rint**, **round**, **min**, **max**, **abs**, and **random** in the **Math** class for performing mathematical functions.
- The character type **char** represents a single character.
- An escape sequence consists of a backslash (****) followed by a character or a combination of digits.
- The character **** is called the escape character.
- A *string* is a sequence of characters. A string value is enclosed in matching double quotes (**"**). A character value is enclosed in matching single quotes (**'**).
- Strings are objects in Java. A method that can only be invoked from a specific object is called an *instance method*. A non-instance method is called a static method, which can be invoked without using an object.
- The **printf** method can be used to display a formatted output using format specifiers.

Ideas for further practice



Show the output of the following statements (write a program to verify your results):

```
System.out.println("1" + 1);
System.out.println('1' + 1);
System.out.println("1" + 1 + 1);
System.out.println("1" + (1 + 1));
System.out.println('1' + 1 + 1);
```



Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

(a) Check whether `s1` is equal to `s2` and assign the result to a Boolean variable `isEqual`.

(b) Check whether `s1` is equal to `s2`, ignoring case, and assign the result to a Boolean variable `isEqual`.

(c) Compare `s1` with `s2` and assign the result to an `int` variable `x`.

(d) Check whether `s1` has the prefix `AAA` and assign the result to a Boolean variable `b`.

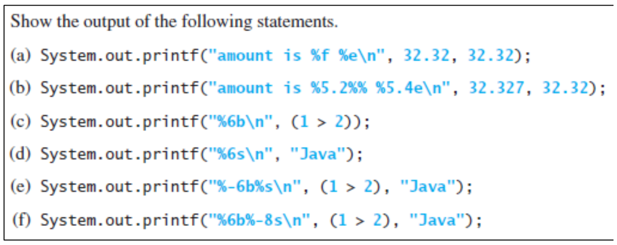
(e) Assign the length of `s1` to an `int` variable `x`.

(f) Assign the first character of `s1` to a `char` variable `x`.

(g) Create a new string `s3` that combines `s1` with `s2`.

(h) Create a substring of `s1` from index `1` to index `4`.

(i) Create a new string `s3` that converts `s1` to lowercase.



Show the output of the following statements.

(a) `System.out.printf("amount is %f %e\n", 32.32, 32.32);`

(b) `System.out.printf("amount is %5.2%% %5.4e\n", 32.327, 32.32);`

(c) `System.out.printf("%6b\n", (1 > 2));`

(d) `System.out.printf("%6s\n", "Java");`

(e) `System.out.printf("%-6b%s\n", (1 > 2), "Java");`

(f) `System.out.printf("%6b%-8s\n", (1 > 2), "Java");`