4.3.5 Can the following conversions involving casting be allowed? If so, find the converted result.

```
char c = 'A';
int i = (int)c;

float f = 1000.34f;
int i = (int)f;

double d = 1000.34;
int i = (int)d;

int i = 97;
char c = (char)i;
```

4.3.6 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    char x = 'a';
    char y = 'c';
    System.out.println(++x);
    System.out.println(y++);
    System.out.println(x - y);
  }
}
```

4.3.7 Write the code that generates a random lowercase letter.

4.3.8 Show the output of the following statements:

```
System.out.println('a' < 'b');
System.out.println('a' <= 'A');
System.out.println('a' > 'b');
System.out.println('a' >= 'A');
System.out.println('a' == 'a');
System.out.println('a' != 'b');
```

4.4 The String Type

Key Point

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";
```

VideoNote
Introduce strings and objects

`String` is a predefined class in the Java library, just like the classes `System` and `Scanner`. The `String` type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. The variable declared by a reference type is known as a reference variable that references an object. Here, `message` is a reference variable that references a string object with contents `Welcome to Java`.

Reference data types will be discussed in detail in Chapter 9, Objects and Classes. For the time being, you need to know only how to declare a `String` variable, how to assign a string to the variable, and how to use the methods in the `String` class. More details on using strings will be covered in Chapter 10.

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.

instance method
static method

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 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 **`referenceVariable.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)`**.

4.4.1 Getting String Length

You 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

string literal

empty string

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`**.

4.4.2 Getting Characters from a String

`charAt(index)`

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`**.

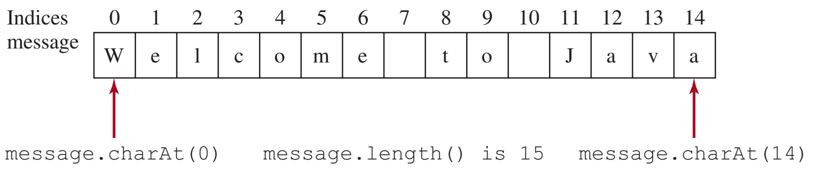


FIGURE 4.1 The characters in a **`String`** object can be accessed using its index.

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`.

string index range

4.4.3 Concatenating Strings

You 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);
```

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;
```

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 or a character with a string. In this case, the number or character 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:

concatenate strings and numbers

```
// 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?

```
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));
```

4.4.4 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`.

toLowerCase()
toUpperCase()

whitespace character

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,

trim()

"\t Good Night \n".trim() returns a new string **Good Night**.

4.4.5 Reading a String from the Console

read strings

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 ↵Enter
s1 is Welcome
s2 is to
s3 is Java
```

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

token-based input

line-based input

For convenience, we call the input using the methods **next()**, **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()**, and **nextDouble()** the token-based input, because they read individual elements separated by whitespace characters rather than an entire line. The **nextLine()** method is called a line-based input.

avoid input errors

Important Caution

To *avoid input errors*, do not use a line-based input after a token-based input in the program. The reasons will be explained in Section 12.11.4, "How Does **Scanner** Work?"

4.4.6 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);
```

4.4.7 Comparing Strings

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.

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:

`string1.equals(string2)`

```
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)`

```
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. You 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.

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  }
```

input `city1`

input `city2`

compare two cities

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

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`.

4.4.8 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 (see Figure 4.2) in the `String` class, as given in Table 4.9.

For example,

```
String message = "Welcome to Java";
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.

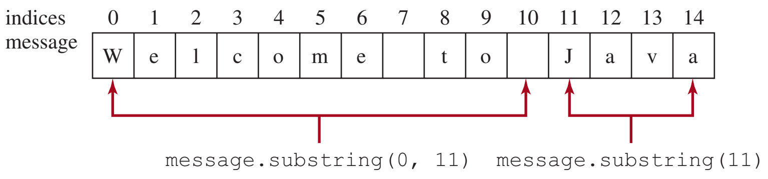


FIGURE 4.2 The `substring` method obtains a substring from a string.

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

`beginIndex <= endIndex`

4.4.9 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 listed in Table 4.10.

TABLE 4.10 The `String` Class Contains the Methods for Finding Substrings

Method	*Description*
`indexOf(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,

IndexOf

```
"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.
```

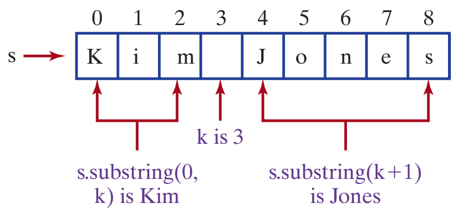
lastIndexOf

```
"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. 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);
```

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



4.4.10 Conversion between Strings and Numbers

Integer.parseInt method

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"**.

Double.parseDouble method

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:

number to string

```
String s = number + "";
```

Check Point

4.4.1 Suppose **s1**, **s2**, and **s3** are three strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = "Programming is fun";
String s3 = "Welcome to Java";
```

What are the results of the following expressions?

(a) `s1 == s2`
(b) `s2 == s3`
(c) `s1.equals(s2)`
(d) `s1.equals(s3)`
(e) `s1.compareTo(s2)`
(f) `s2.compareTo(s3)`
(g) `s2.compareTo(s2)`
(h) `s1.charAt(0)`
(i) `s1.indexOf('j')`
(j) `s1.indexOf("to")`
(k) `s1.lastIndexOf('a')`
(l) `s1.lastIndexOf("o", 15)`
(m) `s1.length()`
(n) `s1.substring(5)`
(o) `s1.substring(5, 11)`
(p) `s1.startsWith("Wel")`
(q) `s1.endsWith("Java")`
(r) `s1.toLowerCase()`
(s) `s1.toUpperCase()`
(t) `s1.concat(s2)`
(u) `s1.contain(s2)`
(v) `"\t Wel \t".trim()`

4.4.2 Suppose `s1` and `s2` are two strings. Which of the following statements or expressions are incorrect?

```
String s = "Welcome to Java";
String s3 = s1 + s2;
s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
int i = s1.length();
char c = s1(0);
char c = s1.charAt(s1.length());
```

4.4.3 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);
```

4.4.4 Evaluate the following expressions (write a program to verify your results):

```
1 + "Welcome " + 1 + 1
1 + "Welcome " + (1 + 1)
1 + "Welcome " + ('\u0001' + 1)
1 + "Welcome " + 'a' + 1
```

4.4.5 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) Compare `s1` with `s2`, ignoring case, and assign the result to an `int` variable `x`.

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

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

(g) Assign the length of **s1** to an **int** variable **x**.
(h) Assign the first character of **s1** to a **char** variable **x**.
(i) Create a new string **s3** that combines **s1** with **s2**.
(j) Create a substring of **s1** starting from index **1**.
(k) Create a substring of **s1** from index **1** to index **4**.
(l) Create a new string **s3** that converts **s1** to lowercase.
(m) Create a new string **s3** that converts **s1** to uppercase.
(n) Create a new string **s3** that trims whitespaces on both ends of **s1**.
(o) Assign the index of the first occurrence of the character **e** in **s1** to an **int** variable **x**.
(p) Assign the index of the last occurrence of the string **abc** in **s1** to an **int** variable **x**.

4.4.6 Write one statement to return the number of digits in an integer **i**.

4.4.7 Write one statement to return the number of digits in a double value **d**.

4.5 Case Studies

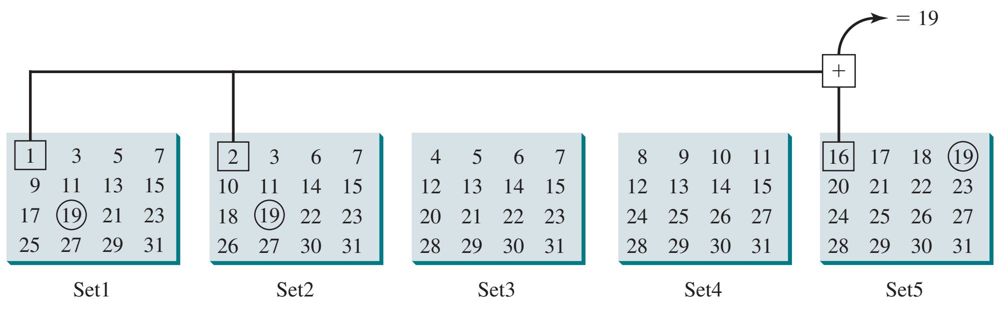
Strings are fundamental in programming. The ability to write programs using strings is essential in learning Java programming.

Key Point

You will frequently use strings to write useful programs. This section presents three examples of solving problems using strings.

4.5.1 Case Study: Guessing Birthdays

You can find out the date of the month when your friend was born by asking five questions. Each question asks whether the day is in one of the five sets of numbers.



The birthday is the sum of the first numbers in the sets where the day appears. For example, if the birthday is **19**, it appears in Set1, Set2, and Set5. The first numbers in these three sets are **1**, **2**, and **16**. Their sum is **19**.

Listing 4.3 gives a program that prompts the user to answer whether the day is in Set1 (lines 41–44), in Set2 (lines 50–53), in Set3 (lines 59–62), in Set4 (lines 68–71), and in Set5 (lines 77–80). If the number is in the set, the program adds the first number in the set to **day** (lines 47, 56, 65, 74, and 83).

LISTING 4.3 GuessBirthday.java

```
1  import java.util.Scanner;
2
3  public class GuessBirthday {
4    public static void main(String[] args) {
5      String set1 =
```