Chapter 14
Strings, Characters and Regular Expressions

Java How to Program, 11/e, Global Edition
Questions? E-mail paul.deitel@deitel.com

Strings, Characters and Regular Expressions

OBJECTIVES

In this chapter you'll:

- Create and manipulate immutable character-string objects of class `String`.
- Create and manipulate mutable character-string objects of class `StringBuilder`.
- Create and manipulate objects of class `Character`.
- Break a `String` object into tokens using `String` method `split`.
- Use regular expressions to validate `String` data entered into an application.

OUTLINE

OUTLINE (cont.)

14.4 Class `StringBuilder`

- 14.4.1 `StringBuilder` Constructors
- 14.4.2 `StringBuilder` Methods `length`, `capacity`, `setLength` and `ensureCapacity`
- 14.4.3 `StringBuilder` Methods `charAt`, `setCharAt`, `getChars` and `reverse`
- 14.4.4 `StringBuilder` `append` Methods
- 14.4.5 `StringBuilder` Insertion and Deletion Methods

OUTLINE (cont.)

14.5 Class `Character`

14.6 Tokenizing `String`s

14.7 Regular Expressions, Class `Pattern` and Class `Matcher`

- 14.7.1 Replacing Substrings and Splitting Strings
- 14.7.2 Classes `Pattern` and `Matcher`

14.8 Wrap-Up

Performance Tip 14.1

To conserve memory, Java treats all string literals with the same contents as a single `String` object that has many references to it.

Performance Tip 14.2

As of Java SE 9, Java uses a more compact `String` representation. This significantly reduces the amount of memory used to store `String`s containing only Latin-1 characters—that is, those with the character codes 0–255. For more information, see JEP 254's proposal at `http://openjdk.java.net/jeps/254`.

```
1  // Fig. 14.1: StringConstructors.java
2  // String class constructors.
3
4  public class StringConstructors {
5     public static void main(String[] args) {
6        char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};
7        String s = new String("hello");
8
9        // use String constructors
10       String s1 = new String();
11       String s2 = new String(s);
12       String s3 = new String(charArray);
13       String s4 = new String(charArray, 6, 3);
14
15       System.out.printf(
16          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
17    }
18 }
```

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

Fig. 14.1 | `String` class constructors.

Performance Tip 14.3

It's not necessary to copy an existing `String` object. `String` objects are immutable, because class `String` does not provide methods that allow the contents of a `String` object to be modified after it is created. In fact, it's rarely necessary to call `String` constructors.

```
1  // Fig. 14.2: StringMiscellaneous.java
2  // This application demonstrates the length, charAt and getChars
3  // methods of the String class.
4
5  public class StringMiscellaneous {
6     public static void main(String[] args) {
7        String s1 = "hello there";
8        char[] charArray = new char[5];
9
10       System.out.printf("s1: %s", s1);
11
12       // test length method
13       System.out.printf("%nLength of s1: %d", s1.length());
14
15       // loop through characters in s1 with charAt and display reversed
16       System.out.printf("%nThe string reversed is: ");
17
18       for (int count = s1.length() - 1; count >= 0; count--) {
19          System.out.printf("%c ", s1.charAt(count));
20       }
```

Fig. 14.2 | `String` methods `length`, `charAt` and `getChars`. (Part 1 of 2.)


```
21
22       // copy characters from string into charArray
23       s1.getChars(0, 5, charArray, 0);
24       System.out.printf("%nThe character array is: ");
25
26       for (char character : charArray) {
27          System.out.print(character);
28       }
29
30       System.out.println();
31    }
32 }
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

Fig. 14.2 | `String` methods `length`, `charAt` and `getChars`. (Part 2 of 2.)

```
1  // Fig. 14.3: StringCompare.java
2  // String methods equals, equalsIgnoreCase, compareTo and regionMatches.
3
4  public class StringCompare {
5     public static void main(String[] args) {
6        String s1 = new String("hello"); // s1 is a copy of "hello"
7        String s2 = "goodbye";
8        String s3 = "Happy Birthday";
9        String s4 = "happy birthday";
10
11       System.out.printf(
12          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);
13
```

Fig. 14.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 1 of 5.)


```
14       // test for equality
15       if (s1.equals("hello")) {  // true
16          System.out.println("s1 equals \"hello\"");
17       }
18       else {
19          System.out.println("s1 does not equal \"hello\"");
20       }
21
22       // test for equality with ==
23       if (s1 == "hello") { // false; they are not the same object
24          System.out.println("s1 is the same object as \"hello\"");
25       }
26       else {
27          System.out.println("s1 is not the same object as \"hello\"");
28       }
29
```

Fig. 14.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 2 of 5.)

```
30         // test for equality (ignore case)
31         if (s3.equalsIgnoreCase(s4)) { // true
32            System.out.printf("%s equals %s with case ignored%n", s3, s4);
33         }
34         else {
35            System.out.println("s3 does not equal s4");
36         }
37
38         // test compareTo
39         System.out.printf(
40            "%ns1.compareTo(s2) is %d", s1.compareTo(s2));
41         System.out.printf(
42            "%ns2.compareTo(s1) is %d", s2.compareTo(s1));
43         System.out.printf(
44            "%ns1.compareTo(s1) is %d", s1.compareTo(s1));
45         System.out.printf(
46            "%ns3.compareTo(s4) is %d", s3.compareTo(s4));
47         System.out.printf(
48            "%ns4.compareTo(s3) is %d%n%n", s4.compareTo(s3));
```

Fig. 14.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 3 of 5.)


```
49
50         // test regionMatches (case sensitive)
51         if (s3.regionMatches(0, s4, 0, 5)) {
52            System.out.println("First 5 characters of s3 and s4 match");
53         }
54         else {
55            System.out.println(
56               "First 5 characters of s3 and s4 do not match");
57         }
58
59         // test regionMatches (ignore case)
60         if (s3.regionMatches(true, 0, s4, 0, 5)) {
61            System.out.println(
62               "First 5 characters of s3 and s4 match with case ignored");
63         }
64         else {
65            System.out.println(
66               "First 5 characters of s3 and s4 do not match");
67         }
68      }
69   }
```

Fig. 14.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 4 of 5.)

```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1 equals "hello"
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo(s2) is 1
s2.compareTo(s1) is -1
s1.compareTo(s1) is 0
s3.compareTo(s4) is -32
s4.compareTo(s3) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

Fig. 14.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 5 of 5.)

Common Programming Error 14.1

Comparing references with == can lead to logic errors, because == compares the references to determine whether they refer to the same object, not whether two objects have the same contents. When two separate objects that contain the same values are compared with ==, the result will be `false`. When comparing objects to determine whether they have the same contents, use method `equals`.

```
1  // Fig. 14.4: StringStartEnd.java
2  // String methods startsWith and endsWith.
3
4  public class StringStartEnd {
5     public static void main(String[] args) {
6        String[] strings = {"started", "starting", "ended", "ending"};
7
8        // test method startsWith
9        for (String string : strings) {
10          if (string.startsWith("st")) {
11             System.out.printf("\"%s\" starts with \"st\"%n", string);
12          }
13       }
14
15       System.out.println();
16
```

Fig. 14.4 | `String` methods `startsWith` and `endsWith`. (Part 1 of 3.)


```
17       // test method startsWith starting from position 2 of string
18       for (String string : strings) {
19          if (string.startsWith("art", 2)) {
20             System.out.printf(
21                "\"%s\" starts with \"art\" at position 2%n", string);
22          }
23       }
24
25       System.out.println();
26
27       // test method endsWith
28       for (String string : strings) {
29          if (string.endsWith("ed")) {
30             System.out.printf("\"%s\" ends with \"ed\"%n", string);
31          }
32       }
33    }
34 }
```

Fig. 14.4 | `String` methods `startsWith` and `endsWith`. (Part 2 of 3.)

```
"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

Fig. 14.4 | `String` methods `startsWith` and `endsWith`. (Part 3 of 3.)


```
1  // Fig. 14.5: StringIndexMethods.java
2  // String searching methods indexOf and lastIndexOf.
3
4  public class StringIndexMethods {
5     public static void main(String[] args) {
6        String letters = "abcdefghijklmabcdefghijklm";
7
8        // test indexOf to locate a character in a string
9        System.out.printf(
10          "'c' is located at index %d%n", letters.indexOf('c'));
11       System.out.printf(
12          "'a' is located at index %d%n", letters.indexOf('a', 1));
13       System.out.printf(
14          "'$' is located at index %d%n%n", letters.indexOf('$'));
15
```

Fig. 14.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 1 of 4.)

```
16          // test lastIndexOf to find a character in a string
17          System.out.printf("Last 'c' is located at index %d%n",
18             letters.lastIndexOf('c'));
19          System.out.printf("Last 'a' is located at index %d%n",
20             letters.lastIndexOf('a', 25));
21          System.out.printf("Last '$' is located at index %d%n%n",
22             letters.lastIndexOf('$'));
23
24          // test indexOf to locate a substring in a string
25          System.out.printf("\"def\" is located at index %d%n",
26             letters.indexOf("def"));
27          System.out.printf("\"def\" is located at index %d%n",
28             letters.indexOf("def", 7));
29          System.out.printf("\"hello\" is located at index %d%n%n",
30             letters.indexOf("hello"));
```

Fig. 14.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 2 of 4.)


```
31
32          // test lastIndexOf to find a substring in a string
33          System.out.printf("Last \"def\" is located at index %d%n",
34             letters.lastIndexOf("def"));
35          System.out.printf("Last \"def\" is located at index %d%n",
36             letters.lastIndexOf("def", 25));
37          System.out.printf("Last \"hello\" is located at index %d%n",
38             letters.lastIndexOf("hello"));
39       }
40    }
```

Fig. 14.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 3 of 4.)

```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1

Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1

"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

Fig. 14.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 4 of 4.)


```
1  // Fig. 14.6: SubString.java
2  // String class substring methods.
3
4  public class SubString {
5     public static void main(String[] args) {
6        String letters = "abcdefghijklmabcdefghijklm";
7
8        // test substring methods
9        System.out.printf("Substring from index 20 to end is \"%s\"%n",
10          letters.substring(20));
11       System.out.printf("%s \"%s\"%n",
12          "Substring from index 3 up to, but not including, 6 is",
13          letters.substring(3, 6));
14    }
15 }
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including, 6 is "def"
```

Fig. 14.6 | `String` class `substring` methods.

```java
// Fig. 14.7: StringConcatenation.java
// String method concat.

public class StringConcatenation {
   public static void main(String[] args) {
      String s1 = "Happy ";
      String s2 = "Birthday";

      System.out.printf("s1 = %s%ns2 = %s%n%n",s1, s2);
      System.out.printf(
         "Result of s1.concat(s2) = %s%n", s1.concat(s2));
      System.out.printf("s1 after concatenation = %s%n", s1);
   }
}
```

```
s1 = Happy
s2 = Birthday

Result of s1.concat(s2) = Happy Birthday
s1 after concatenation = Happy
```

Fig. 14.7 | String method concat.


```java
// Fig. 14.8: StringMiscellaneous2.java
// String methods replace, toLowerCase, toUpperCase, trim and toCharArray.

public class StringMiscellaneous2 {
   public static void main(String[] args) {
      String s1 = "hello";
      String s2 = "GOODBYE";
      String s3 = "   spaces   ";

      System.out.printf("s1 = %s%ns2 = %s%ns3 = %s%n%n", s1, s2, s3);

      // test method replace
      System.out.printf(
         "Replace 'l' with 'L' in s1: %s%n%n", s1.replace('l', 'L'));

      // test toLowerCase and toUpperCase
      System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
      System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
```

Fig. 14.8 | String methods replace, toLowerCase, toUpperCase, trim and toCharArray. (Part 1 of 3.)

```
19
20          // test trim method
21          System.out.printf("s3 after trim = \"%s\"%n%n", s3.trim());
22
23          // test toCharArray method
24          char[] charArray = s1.toCharArray();
25          System.out.print("s1 as a character array = ");
26
27          for (char character : charArray) {
28             System.out.print(character);
29          }
30
31          System.out.println();
32       }
33    }
```

Fig. 14.8 | `String` methods `replace`, `toLowerCase`, `toUpperCase`, `trim` and `toCharArray`. (Part 2 of 3.)


```
s1 = hello
s2 = GOODBYE
s3 =    spaces

Replace 'l' with 'L' in s1: heLLo

s1.toUpperCase() = HELLO
s2.toLowerCase() = goodbye

s3 after trim = "spaces"

s1 as a character array = hello
```

Fig. 14.8 | `String` methods `replace`, `toLowerCase`, `toUpperCase`, `trim` and `toCharArray`. (Part 3 of 3.)

```
1  // Fig. 14.9: StringValueOf.java
2  // String valueOf methods.
3
4  public class StringValueOf {
5     public static void main(String[] args) {
6        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
7        boolean booleanValue = true;
8        char characterValue = 'Z';
9        int integerValue = 7;
10       long longValue = 10000000000L; // L suffix indicates long
11       float floatValue = 2.5f; // f indicates that 2.5 is a float
12       double doubleValue = 33.333; // no suffix, double is default
13       Object objectRef = "hello"; // assign string to an Object reference
14
15       System.out.printf(
16          "char array = %s%n", String.valueOf(charArray));
17       System.out.printf("part of char array = %s%n",
18          String.valueOf(charArray, 3, 3));
```

Fig. 14.9 | String valueOf methods. (Part 1 of 3.)


```
19       System.out.printf(
20          "boolean = %s%n", String.valueOf(booleanValue));
21       System.out.printf(
22          "char = %s%n", String.valueOf(characterValue));
23       System.out.printf("int = %s%n", String.valueOf(integerValue));
24       System.out.printf("long = %s%n", String.valueOf(longValue));
25       System.out.printf("float = %s%n", String.valueOf(floatValue));
26       System.out.printf(
27          "double = %s%n", String.valueOf(doubleValue));
28       System.out.printf("Object = %s", String.valueOf(objectRef));
29    }
30 }
```

Fig. 14.9 | String valueOf methods. (Part 2 of 3.)

```
char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello
```

Fig. 14.9 | `String valueOf` methods. (Part 3 of 3.)

Performance Tip 14.4

Java can perform certain optimizations involving `String` objects (such as referring to one `String` object from multiple variables) because it knows these objects will not change. `String`s (not `StringBuilder`s) should be used if the data will not change.

Performance Tip 14.5

In programs that frequently perform string concatenation, or other string modifications, it's often more efficient to implement the modifications with class `StringBuilder`.

Software Engineering Observation 14.1

`StringBuilder`s are not thread safe. If multiple threads require access to the same dynamic string information, use class **StringBuffer** in your code. Classes `StringBuilder` and `StringBuffer` provide identical capabilities, but class `StringBuffer` is thread safe. For more details on threading, see Chapter 23.

```
1  // Fig. 14.10: StringBuilderConstructors.java
2  // StringBuilder constructors.
3
4  public class StringBuilderConstructors {
5     public static void main(String[] args) {
6        StringBuilder buffer1 = new StringBuilder();
7        StringBuilder buffer2 = new StringBuilder(10);
8        StringBuilder buffer3 = new StringBuilder("hello");
9
10       System.out.printf("buffer1 = \"%s\"%n", buffer1);
11       System.out.printf("buffer2 = \"%s\"%n", buffer2);
12       System.out.printf("buffer3 = \"%s\"%n", buffer3);
13    }
14 }
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

Fig. 14.10 | StringBuilder constructors.


```
1  // Fig. 14.11: StringBuilderCapLen.java
2  // StringBuilder length, setLength, capacity and ensureCapacity methods.
3
4  public class StringBuilderCapLen {
5     public static void main(String[] args) {
6        StringBuilder buffer = new StringBuilder("Hello, how are you?");
7
8        System.out.printf("buffer = %s%nlength = %d%ncapacity = %d%n%n",
9           buffer.toString(), buffer.length(), buffer.capacity());
10
11       buffer.ensureCapacity(75);
12       System.out.printf("New capacity = %d%n%n", buffer.capacity());
13
14       buffer.setLength(10));
15       System.out.printf("New length = %d%nbuffer = %s%n",
16          buffer.length(), buffer.toString());
17    }
18 }
```

Fig. 14.11 | StringBuilder length, setLength, capacity and ensureCapacity methods. (Part 1 of 2.)

```
buffer = Hello, how are you?
length = 19
capacity = 35

New capacity = 75

New length = 10
buffer = Hello, how
```

Fig. 14.11 | `StringBuilder` `length`, `setLength`, `capacity` and `ensureCapacity` methods. (Part 2 of 2.)

Performance Tip 14.6

Dynamically increasing the capacity of a `StringBuilder` can take a relatively long time. Executing a large number of these operations can degrade the performance of an application. If a `StringBuilder` is going to increase greatly in size, possibly multiple times, setting its capacity high at the beginning will increase performance.

```
1  // Fig. 14.12: StringBuilderChars.java
2  // StringBuilder methods charAt, setCharAt, getChars and reverse.
3
4  public class StringBuilderChars {
5     public static void main(String[] args) {
6        StringBuilder buffer = new StringBuilder("hello there");
7
8        System.out.printf("buffer = %s%n", buffer.toString());
9        System.out.printf("Character at 0: %s%nCharacter at 4: %s%n%n",
10          buffer.charAt(0), buffer.charAt(4));
11
12       char[] charArray = new char[buffer.length()];
13       buffer.getChars(0, buffer.length(), charArray, 0);
14       System.out.print("The characters are: ");
15
```

Fig. 14.12 | StringBuilder methods charAt, setCharAt, getChars and reverse. (Part 1 of 3.)


```
16       for (char character : charArray) {
17          System.out.print(character);
18       }
19
20       buffer.setCharAt(0, 'H');
21       buffer.setCharAt(6, 'T');
22       System.out.printf("%n%nbuffer = %s", buffer.toString());
23
24       buffer.reverse();
25       System.out.printf("%n%nbuffer = %s%n", buffer.toString());
26    }
27 }
```

Fig. 14.12 | StringBuilder methods charAt, setCharAt, getChars and reverse. (Part 2 of 3.)

```
buffer = hello there
Character at 0: h
Character at 4: o

The characters are: hello there

buffer = Hello There

buffer = erehT olleH
```

Fig. 14.12 | StringBuilder methods charAt, setCharAt, getChars and reverse. (Part 3 of 3.)


```
1  // Fig. 14.13: StringBuilderAppend.java
2  // StringBuilder append methods.
3
4  public class StringBuilderAppend
5  {
6     public static void main(String[] args)
7     {
8        Object objectRef = "hello";
9        String string = "goodbye";
10       char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
11       boolean booleanValue = true;
12       char characterValue = 'Z';
13       int integerValue = 7;
14       long longValue = 10000000000L;
15       float floatValue = 2.5f;
16       double doubleValue = 33.333;
17
18       StringBuilder lastBuffer = new StringBuilder("last buffer");
19       StringBuilder buffer = new StringBuilder();
20
21       buffer.append(objectRef)
22             .append(System.getProperty("line.separator"))
23             .append(string)
```

Fig. 14.13 | StringBuilder append methods. (Part 1 of 3.)

```
24              .append(System.getProperty("line.separator"))
25              .append(charArray)
26              .append(System.getProperty("line.separator"))
27              .append(charArray, 0, 3)
28              .append(System.getProperty("line.separator"))
29              .append(booleanValue)
30              .append(System.getProperty("line.separator"))
31              .append(characterValue);
32              .append(System.getProperty("line.separator"))
33              .append(integerValue)
34              .append(System.getProperty("line.separator"))
35              .append(longValue)
36              .append(System.getProperty("line.separator"))
37              .append(floatValue)
38              .append(System.getProperty("line.separator"))
39              .append(doubleValue)
40              .append(System.getProperty("line.separator"))
41              .append(lastBuffer);
42
43          System.out.printf("buffer contains%n%s%n", buffer.toString());
44      }
45  }
```

Fig. 14.13 | StringBuilder append methods. (Part 2 of 3.)


```
buffer contains
hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

Fig. 14.13 | StringBuilder append methods. (Part 3 of 3.)

```
1  // Fig. 14.14: StringBuilderInsertDelete.java
2  // StringBuilder methods insert, delete and deleteCharAt.
3
4  public class StringBuilderInsertDelete {
5     public static void main(String[] args) {
6        Object objectRef = "hello";
7        String string = "goodbye";
8        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
9        boolean booleanValue = true;
10       char characterValue = 'K';
11       int integerValue = 7;
12       long longValue = 10000000;
13       float floatValue = 2.5f; // f suffix indicates that 2.5 is a float
14       double doubleValue = 33.333;
15
16       StringBuilder buffer = new StringBuilder();
17
18       buffer.insert(0, objectRef);
19       buffer.insert(0, "  "); // each of these contains two spaces
20       buffer.insert(0, string);
21       buffer.insert(0, "  ");
22       buffer.insert(0, charArray);
23       buffer.insert(0, "  ");
```

Fig. 14.14 | StringBuilder methods insert, delete and deleteCharAt. (Part 1 of 3.)


```
24       buffer.insert(0, charArray, 3, 3);
25       buffer.insert(0, "  ");
26       buffer.insert(0, booleanValue);
27       buffer.insert(0, "  ");
28       buffer.insert(0, characterValue);
29       buffer.insert(0, "  ");
30       buffer.insert(0, integerValue);
31       buffer.insert(0, "  ");
32       buffer.insert(0, longValue);
33       buffer.insert(0, "  ");
34       buffer.insert(0, floatValue);
35       buffer.insert(0, "  ");
36       buffer.insert(0, doubleValue);
37
38       System.out.printf(
39          "buffer after inserts:%n%s%n%n", buffer.toString());
40
41       buffer.deleteCharAt(10); // delete 5 in 2.5
42       buffer.delete(2, 6); // delete .333 in 33.333
43
44       System.out.printf(
45          "buffer after deletes:%n%s%n", buffer.toString());
46    }
47 }
```

Fig. 14.14 | StringBuilder methods insert, delete and deleteCharAt. (Part 2 of 3.)

```
buffer after inserts:
33.333  2.5  10000000  7  K  true  def  abcdef  goodbye  hello

buffer after deletes:
33  2.  10000000  7  K  true  def  abcdef  goodbye  hello
```

Fig. 14.14 | `StringBuilder` methods `insert`, `delete` and `deleteCharAt`. (Part 3 of 3.)


```
1  // Fig. 14.15: StaticCharMethods.java
2  // Character static methods for testing characters and converting case.
3  import java.util.Scanner;
4
5  public class StaticCharMethods {
6     public static void main(String[] args) {
7        Scanner scanner = new Scanner(System.in); // create scanner
8        System.out.println("Enter a character and press Enter");
9        String input = scanner.next();
10       char c = input.charAt(0); // get input character
11
12       // display character info
13       System.out.printf("is defined: %b%n", Character.isDefined(c));
14       System.out.printf("is digit: %b%n", Character.isDigit(c));
15       System.out.printf("is first character in a Java identifier: %b%n",
16          Character.isJavaIdentifierStart(c));
```

Fig. 14.15 | `Character` `static` methods for testing characters and converting case. (Part 1 of 5.)

```
17          System.out.printf("is part of a Java identifier: %b%n",
18             Character.isJavaIdentifierPart(c));
19          System.out.printf("is letter: %b%n", Character.isLetter(c));
20          System.out.printf(
21             "is letter or digit: %b%n", Character.isLetterOrDigit(c));
22          System.out.printf(
23             "is lower case: %b%n", Character.isLowerCase(c));
24          System.out.printf(
25             "is upper case: %b%n", Character.isUpperCase(c));
26          System.out.printf(
27             "to upper case: %s%n", Character.toUpperCase(c));
28          System.out.printf(
29             "to lower case: %s%n", Character.toLowerCase(c));
30       }
31    }
```

Fig. 14.15 | `Character` static methods for testing characters and converting case. (Part 2 of 5.)


```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

Fig. 14.15 | `Character` static methods for testing characters and converting case. (Part 3 of 5.)

```
Enter a character and press Enter
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

Fig. 14.15 | Character static methods for testing characters and converting case. (Part 4 of 5.)


```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

Fig. 14.15 | Character static methods for testing characters and converting case. (Part 5 of 5.)

```
1  // Fig. 14.16: StaticCharMethods2.java
2  // Character class static conversion methods.
3  import java.util.Scanner;
4
5  public class StaticCharMethods2 {
6     public static void main(String[] args) {
7        Scanner scanner = new Scanner(System.in);
8
9        // get radix
10       System.out.println("Please enter a radix:");
11       int radix = scanner.nextInt();
12
13       // get user choice
14       System.out.printf("Please choose one:%n1 -- %s%n2 -- %s%n",
15          "Convert digit to character", "Convert character to digit");
16       int choice = scanner.nextInt();
17
```

Fig. 14.16 | Character class static conversion methods. (Part 1 of 3.)


```
18       // process request
19       switch (choice) {
20          case 1: // convert digit to character
21             System.out.println("Enter a digit:");
22             int digit = scanner.nextInt();
23             System.out.printf("Convert digit to character: %s%n",
24                Character.forDigit(digit, radix));
25             break;
26          case 2: // convert character to digit
27             System.out.println("Enter a character:");
28             char character = scanner.next().charAt(0);
29             System.out.printf("Convert character to digit: %s%n",
30                Character.digit(character, radix));
31             break;
32       }
33    }
34 }
```

Fig. 14.16 | Character class static conversion methods. (Part 2 of 3.)

```
Please enter a radix:
16
Please choose one:
1 -- Convert digit to character
2 -- Convert character to digit
2
Enter a character:
A
Convert character to digit: 10
```

```
Please enter a radix:
16
Please choose one:
1 -- Convert digit to character
2 -- Convert character to digit
1
Enter a digit:
13
Convert digit to character: d
```

Fig. 14.16 | `Character` class `static` conversion methods. (Part 3 of 3.)


```
1  // Fig. 14.17: OtherCharMethods.java
2  // Character class instance methods.
3  public class OtherCharMethods {
4     public static void main(String[] args) {
5        Character c1 = 'A';
6        Character c2 = 'a';
7
8        System.out.printf(
9           "c1 = %s%nc2 = %s%n%n", c1.charValue(), c2.toString());
10
11       if (c1.equals(c2)) {
12          System.out.println("c1 and c2 are equal%n");
13       }
14       else {
15          System.out.println("c1 and c2 are not equal%n");
16       }
17    }
18 }
```

Fig. 14.17 | `Character` class instance methods. (Part 1 of 2.)

```
c1 = A
c2 = a

c1 and c2 are not equal
```

Fig. 14.17 | `Character` class instance methods. (Part 2 of 2.)


```
1  // Fig. 14.18: TokenTest.java
2  // Tokenizing with String method split
3  import java.util.Scanner;
4
5  public class TokenTest {
6     // execute application
7     public static void main(String[] args) {
8        // get sentence
9        Scanner scanner = new Scanner(System.in);
10       System.out.println("Enter a sentence and press Enter");
11       String sentence = scanner.nextLine();
12
13       // process user sentence
14       String[] tokens = sentence.split(" ");
15       System.out.printf("Number of elements: %d%nThe tokens are:%n",
16          tokens.length);
17
18       for (String token : tokens) {
19          System.out.println(token);
20       }
21    }
22 }
```

Fig. 14.18 | Tokenizing with `String` method `split`. (Part 1 of 2.)

```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of elements: 7
The tokens are:
This
is
a
sentence
with
seven
tokens
```

Fig. 14.18 | Tokenizing with `String` method `split`. (Part 2 of 2.)

Character	Matches	Character	Matches
`\d`	any digit	`\D`	any nondigit
`\w`	any word character	`\W`	any nonword character
`\s`	any white-space character	`\S`	any non-whitespace character

Fig. 14.19 | Predefined character classes.

```
1  // Fig. 14.20: ValidateInput.java
2  // Validating user information using regular expressions.
3
4  public class ValidateInput {
5     // validate first name
6     public static boolean validateFirstName(String firstName) {
7        return firstName.matches("[A-Z][a-zA-Z]*");
8     }
9
10    // validate last name
11    public static boolean validateLastName(String lastName) {
12       return lastName.matches("[a-zA-z]+(['-][a-zA-Z]+)*");
13    }
14
15    // validate address
16    public static boolean validateAddress(String address) {
17       return address.matches(
18          "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
19    }
```

Fig. 14.20 | Validating user information using regular expressions. (Part 1 of 2.)


```
20
21    // validate city
22    public static boolean validateCity(String city) {
23       return city.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
24    }
25
26    // validate state
27    public static boolean validateState(String state) {
28       return state.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
29    }
30
31    // validate zip
32    public static boolean validateZip(String zip) {
33       return zip.matches("\\d{5}");
34    }
35
36    // validate phone
37    public static boolean validatePhone(String phone) {
38       return phone.matches("[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}");
39    }
40 }
```

Fig. 14.20 | Validating user information using regular expressions. (Part 2 of 2.)

```
1  // Fig. 14.21: Validate.java
2  // Input and validate data from user using the ValidateInput class.
3  import java.util.Scanner;
4
5  public class Validate {
6     public static void main(String[] args) {
7        // get user input
8        Scanner scanner = new Scanner(System.in);
9        System.out.println("Please enter first name:");
10       String firstName = scanner.nextLine();
11       System.out.println("Please enter last name:");
12       String lastName = scanner.nextLine();
13       System.out.println("Please enter address:");
14       String address = scanner.nextLine();
15       System.out.println("Please enter city:");
16       String city = scanner.nextLine();
17       System.out.println("Please enter state:");
18       String state = scanner.nextLine();
19       System.out.println("Please enter zip:");
20       String zip = scanner.nextLine();
21       System.out.println("Please enter phone:");
22       String phone = scanner.nextLine();
```

Fig. 14.21 | Input and validate data from user using the `ValidateInput` class. (Part 1 of 5.)


```
23
24       // validate user input and display error message
25       System.out.printf("%nValidate Result:");
26
27       if (!ValidateInput.validateFirstName(firstName)) {
28          System.out.println("Invalid first name");
29       }
30       else if (!ValidateInput.validateLastName(lastName)) {
31          System.out.println("Invalid last name");
32       }
33       else if (!ValidateInput.validateAddress(address)) {
34          System.out.println("Invalid address");
35       }
36       else if (!ValidateInput.validateCity(city)) {
37          System.out.println("Invalid city");
38       }
```

Fig. 14.21 | Input and validate data from user using the `ValidateInput` class. (Part 2 of 5.)

```
39          else if (!ValidateInput.validateState(state)) {
40             System.out.println("Invalid state");
41          }
42          else if (!ValidateInput.validateZip(zip)) {
43             System.out.println("Invalid zip code");
44          }
45          else if (!ValidateInput.validatePhone(phone)) {
46             System.out.println("Invalid phone number");
47          }
48          else {
49             System.out.println("Valid input.  Thank you.");
50          }
51       }
52    }
```

Fig. 14.21 | Input and validate data from user using the `ValidateInput` class. (Part 3 of 5.)


```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
123
Please enter phone:
123-456-7890

Validate Result:
Invalid zip code
```

Fig. 14.21 | Input and validate data from user using the `ValidateInput` class. (Part 4 of 5.)

```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
12345
Please enter phone:
123-456-7890

Validate Result:
Valid input.  Thank you.
```

Fig. 14.21 | Input and validate data from user using the `ValidateInput` class. (Part 5 of 5.)

Quantifier	Matches
*	Matches zero or more occurrences of the pattern.
+	Matches one or more occurrences of the pattern.
?	Matches zero or one occurrences of the pattern.
{*n*}	Matches exactly *n* occurrences.
{*n*,}	Matches at least *n* occurrences.
{*n*,*m*}	Matches between *n* and *m* (inclusive) occurrences.

Fig. 14.22 | Quantifiers used in regular expressions.

```
1  // Fig. 14.23: RegexSubstitution.java
2  // String methods replaceFirst, replaceAll and split.
3  import java.util.Arrays;
4
5  public class RegexSubstitution {
6     public static void main(String[] args) {
7        String firstString = "This sentence ends in 5 stars *****";
8        String secondString = "1, 2, 3, 4, 5, 6, 7, 8";
9
10       System.out.printf("Original String 1: %s%n", firstString);
11
12       // replace '*' with '^'
13       firstString = firstString.replaceAll("\\*", "^");
14
15       System.out.printf("^ substituted for *: %s%n", firstString);
16
17       // replace 'stars' with 'carets'
18       firstString = firstString.replaceAll("stars", "carets");
19
20       System.out.printf(
21          "\"carets\" substituted for \"stars\": %s%n", firstString);
```

Fig. 14.23 | `String` methods `replaceFirst`, `replaceAll` and `split`. (Part 1 of 3.)


```
22
23       // replace words with 'word'
24       System.out.printf("Every word replaced by \"word\": %s%n%n",
25          firstString.replaceAll("\\w+", "word"));
26
27       System.out.printf("Original String 2: %s%n", secondString);
28
29       // replace first three digits with 'digit'
30       for (int i = 0; i < 3; i++) {
31          secondString = secondString.replaceFirst("\\d", "digit");
32       }
33
34       System.out.printf(
35          "First 3 digits replaced by \"digit\" : %s%n", secondString);
36
37       System.out.print("String split at commas: ");
38       String[] results = secondString.split(",\\s*"); // split on commas
39       System.out.println(Arrays.toString(results));
40    }
41 }
```

Fig. 14.23 | `String` methods `replaceFirst`, `replaceAll` and `split`. (Part 2 of 3.)

```
Original String 1: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original String 2: 1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" : digit, digit, digit, 4, 5, 6, 7, 8
String split at commas: [digit, digit, digit, 4, 5, 6, 7, 8]
```

Fig. 14.23 | `String` methods `replaceFirst`, `replaceAll` and `split`. (Part 3 of 3.)

Common Programming Error 14.2

A regular expression can be tested against an object of any class that implements interface `CharSequence`, but the regular expression must be a `String`. Attempting to create a regular expression as a `StringBuilder` is an error.

```
1  // Fig. 14.24: RegexMatches.java
2  // Classes Pattern and Matcher.
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class RegexMatches {
7     public static void main(String[] args) {
8        // create regular expression
9        Pattern expression =
10          Pattern.compile("J.*\\d[0-35-9]-\\d\\d-\\d\\d");
11
12       String string1 = "Jane's Birthday is 05-12-75\n" +
13          "Dave's Birthday is 11-04-68\n" +
14          "John's Birthday is 04-28-73\n" +
15          "Joe's Birthday is 12-17-77";
```

Fig. 14.24 | Classes Pattern and Matcher. (Part 1 of 2.)


```
16
17       // match regular expression to string and print matches
18       Matcher matcher = expression.matcher(string1);
19
20       while (matcher.find()) {
21          System.out.println(matcher.group());
22       }
23    }
24 }
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 14.24 | Classes Pattern and Matcher. (Part 2 of 2.)

Common Programming Error 14.3

Method `matches` (from class `String`, `Pattern` or `Matcher`) will return `true` only if the entire search object matches the regular expression. Methods `find` and `lookingAt` (from class `Matcher`) will return `true` if a portion of the search object matches the regular expression.