Strings, Characters and Regular Expressions

Java How to Program, 8/e

16.1 Introduction

- This chapter discusses class `String`, class `StringBuilder` and class `Character` from the `java.lang` package.
- These classes provide the foundation for string and character manipulation in Java.
- The chapter also discusses regular expressions that provide applications with the capability to validate input.

16.3.1 `String` Constructors

- No-argument constructor creates a `String` that contains no characters (i.e., the empty string, which can also be represented as `""`) and has a length of 0.
- Constructor that takes a `String` object copies the argument into the new `String`.
- Constructor that takes a `char` array creates a `String` containing a copy of the characters in the array.
- Constructor that takes a `char` array and two integers creates a `String` containing the specified portion of the array.

```
1  // Fig. 16.1: StringConstructors.java
2  // String class constructors.
3
4  public class StringConstructors
5  {
6     public static void main( String[] args )
7     {
8        char[] charArray = { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
9        String s = new String( "hello" );
10
11       // use String constructors
12       String s1 = new String();
13       String s2 = new String( s );
14       String s3 = new String( charArray );
15       String s4 = new String( charArray, 6, 3 );
16
17       System.out.printf(
18          "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
19          s1, s2, s3, s4 ); // display strings
20    } // end main
21 } // end cl
```

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

Fig. 16.1 | String class constructors. (Part 2 of 2.)

16.3.2 `String` Methods `length`, `charAt` and `getChars`

- `String` method `length` determines the number of characters in a string.
- `String` method `charAt` returns the character at a specific position in the `String`.
- `String` method `getChars` copies the characters of a `String` into a character array.
  - The first argument is the starting index in the `String` from which characters are to be copied.
  - The second argument is the index that is one past the last character to be copied from the `String`.
  - The third argument is the character array into which the characters are to be copied.
  - The last argument is the starting index where the copied characters are placed in the target character array.

```
1  // Fig. 16.2: StringMiscellaneous.java
2  // This application demonstrates the length, charAt and getChars
3  // methods of the String class.
4
5  public class StringMiscellaneous
6  {
7     public static void main( String[] args )
8     {
9        String s1 = "hello there";
10       char[] charArray = new char[ 5 ];
11
12       System.out.printf( "s1: %s", s1 );
13
14       // test length method
15       System.out.printf( "\nLength of s1: %d", s1.length() );
16
17       // loop through characters in s1 with charAt and display reversed
18       System.out.print( "\nThe string reversed is: " );
19
20       for ( int count = s1.length() - 1; count >= 0; count-- )
21          System.out.printf( "%c ", s1.charAt( count ) );
22
```

Fig. 16.2 | `String` class character-manipulation methods. (Part 1 of 2.)

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

```
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. 16.2 | `String` class character-manipulation methods. (Part 2 of 2.)

16.3.3 Comparing Strings

- Strings are compared using the numeric codes of the characters in the strings.
- Figure 16.3 demonstrates `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches` and using the equality operator == to compare `String` objects.

```
1  // Fig. 16.3: StringCompare.java
2  // String methods equals, equalsIgnoreCase, compareTo and regionMatches.
3
4  public class StringCompare
5  {
6     public static void main( String[] args )
7     {
8        String s1 = new String( "hello" ); // s1 is a copy of "hello"
9        String s2 = "goodbye";
10       String s3 = "Happy Birthday";
11       String s4 = "happy birthday";
12
13       System.out.printf(
14          "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );
15
16       // test for equality
17       if ( s1.equals( "hello" ) )  // true
18          System.out.println( "s1 equals \"hello\"" );
19       else
20          System.out.println( "s1 does not equal \"hello\"" );
21
```

Fig. 16.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 1 of 4.)

```
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          else
26             System.out.println( "s1 is not the same object as \"hello\"" );
27
28          // test for equality (ignore case)
29          if ( s3.equalsIgnoreCase( s4 ) )  // true
30             System.out.printf( "%s equals %s with case ignored\n", s3, s4 );
31          else
32             System.out.println( "s3 does not equal s4" );
33
34          // test compareTo
35          System.out.printf(
36             "\ns1.compareTo( s2 ) is %d", s1.compareTo( s2 ) );
37          System.out.printf(
38             "\ns2.compareTo( s1 ) is %d", s2.compareTo( s1 ) );
39          System.out.printf(
40             "\ns1.compareTo( s1 ) is %d", s1.compareTo( s1 ) );
41          System.out.printf(
42             "\ns3.compareTo( s4 ) is %d", s3.compareTo( s4 ) );
43          System.out.printf(
44             "\ns4.compareTo( s3 ) is %d\n\n", s4.compareTo( s3 ) );
```

Fig. 16.3 | `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`. (Part 2 of 4.)

```
45
46          // test regionMatches (case sensitive)
47          if ( s3.regionMatches( 0, s4, 0, 5 ) )
48             System.out.println( "First 5 characters of s3 and s4 match" );
49          else
50             System.out.println(
51                "First 5 characters of s3 and s4 do not match" );
52
53          // test regionMatches (ignore case)
54          if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
55             System.out.println(
56                "First 5 characters of s3 and s4 match with case ignored" );
57          else
58             System.out.println(
59                "First 5 characters of s3 and s4 do not match" );
60       } // end main
61    } // end class St
```

```
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. 16.3 | `String` metho
`regionMatches`. (Part 3 of 4

16.3.3 Comparing Strings (cont.)

- `String` methods `startsWith` and `endsWith` determine whether strings start with or end with a particular set of characters

```
1  // Fig. 16.4: StringStartEnd.java
2  // String methods startsWith and endsWith.
3
4  public class StringStartEnd
5  {
6     public static void main( String[] args )
7     {
8        String[] strings = { "started", "starting", "ended", "ending" };
9
10       // test method startsWith
11       for ( String string : strings )
12       {
13          if ( string.startsWith( "st" ) )
14             System.out.printf( "\"%s\" starts with \"st\"\n", string );
15       } // end for
16
17       System.out.println();
18
```

Fig. 16.4 | `String` methods `startsWith` and `endsWith`. (Part 1 of 3.)

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

Fig. 16.4 | String

```
"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. 16.4 | `String` methods `startsWith` and `endsWith`. (Part 3 of 3.)

16.3.4 Locating Characters and Substrings in Strings

- Figure 16.5 demonstrates the many versions of `String` methods `indexOf` and `lastIndexOf` that search for a specified character or substring in a `String`.

```
1  // Fig. 16.5: StringIndexMethods.java
2  // String searching methods indexOf and lastIndexOf.
3
4  public class StringIndexMethods
5  {
6     public static void main( String[] args )
7     {
8        String letters = "abcdefghijklmabcdefghijklm";
9
10       // test indexOf to locate a character in a string
11       System.out.printf(
12          "'c' is located at index %d\n", letters.indexOf( 'c' ) );
13       System.out.printf(
14          "'a' is located at index %d\n", letters.indexOf( 'a', 1 ) );
15       System.out.printf(
16          "'$' is located at index %d\n\n", letters.indexOf( '$' ) );
17
18       // test lastIndexOf to find a character in a string
19       System.out.printf( "Last 'c' is located at index %d\n",
20          letters.lastIndexOf( 'c' ) );
21       System.out.printf( "Last 'a' is located at index %d\n",
22          letters.lastIndexOf( 'a', 25 ) );
23       System.out.printf( "Last '$' is located at index %d\n\n",
24          letters.lastIndexOf( '$' ) );
```

Fig. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 1 of 3.)

```
25
26          // test indexOf to locate a substring in a string
27          System.out.printf( "\"def\" is located at index %d\n",
28             letters.indexOf( "def" ) );
29          System.out.printf( "\"def\" is located at index %d\n",
30             letters.indexOf( "def", 7 ) );
31          System.out.printf( "\"hello\" is located at index %d\n\n",
32             letters.indexOf( "hello" ) );
33
34          // test lastIndexOf to find a substring in a string
35          System.out.printf( "Last \"def\" is located at index %d\n",
36             letters.lastIndexOf( "def" ) );
37          System.out.printf( "Last \"def\" is located at index %d\n",
38             letters.lastIndexOf( "def", 25 ) );
39          System.out.printf( "Last \"hello\" is located at index %d\n",
40             letters.lastIndexOf( "hello" ) );
41       } // end main
42    } // end class StringIndexMethods
```

Fig. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 2 of 3.)

```
'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. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 3 of 3.)

16.3.5 Extracting Substrings from Strings

- Class `String` provides two `substring` methods to enable a new `String` object to be created by copying part of an existing `String` object. Each method returns a new `String` object.
- The version that takes one integer argument specifies the starting index in the original `String` from which characters are to be copied.
- The version that takes two integer arguments receives the starting index from which to copy characters in the original `String` and the index one beyond the last character to copy.

```
1  // Fig. 16.6: SubString.java
2  // String class substring methods.
3
4  public class SubString
5  {
6     public static void main( String[] args )
7     {
8        String letters = "abcdefghijklmabcdefghijklm";
9
10       // test substring methods
11       System.out.printf( "Substring from index 20 to end is \"%s\"\n",
12          letters.substring( 20 ) );
13       System.out.printf( "%s \"%s\"\n",
14          "Substring from index 3 up to, but not including 6 is",
15          letters.substring( 3, 6 ) );
16    } // end main
17 } // end class SubString
```

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

Fig. 16.6 | `String` class `substring` methods.

16.3.6 Concatenating Strings

- `String` method `concat` concatenates two `String` objects and returns a new `String` object containing the characters from both original `String`s.
- The original `String`s to which `s1` and `s2` refer are not modified.

```java
// Fig. 16.7: 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 );
   } // end main
} // end class StringConcatenation
```

```
s1 = Happy
s2 = Birthday

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

Fig. 16.7 | `String` method `concat`.

16.3.7 Miscellaneous `String` Methods

- Method `replace` return a new `String` object in which every occurrence of the first `char` argument is replaced with the second.
  - An overloaded version enables you to replace substrings rather than individual characters.
- Method `toUpperCase` generates a new `String` with uppercase letters.
- Method `toLowerCase` returns a new `String` object with lowercase letters.
- Method `trim` generates a new `String` object that removes all whitespace characters that appear at the beginning or end of the `String` on which `trim` operates.
- Method `toCharArray` creates a new character array containing a copy of the characters in the `String`.

```
1  // Fig. 16.8: StringMiscellaneous2.java
2  // String methods replace, toLowerCase, toUpperCase, trim and toCharArray.
3
4  public class StringMiscellaneous2
5  {
6     public static void main( String[] args )
7     {
8        String s1 = "hello";
9        String s2 = "GOODBYE";
10       String s3 = "   spaces   ";
11
12       System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\n\n", s1, s2, s3 );
13
14       // test method replace
15       System.out.printf(
16          "Replace 'l' with 'L' in s1: %s\n\n", s1.replace( 'l', 'L' ) );
17
18       // test toLowerCase and toUpperCase
19       System.out.printf( "s1.toUpperCase() = %s\n", s1.toUpperCase() );
20       System.out.printf( "s2.toLowerCase() = %s\n\n", s2.toLowerCase() );
21
```

Fig. 16.8 | `String` methods `replace`, `toLowerCase`, `toUpperCase`, `trim` and `toCharArray`. (Part 1 of 3.)

```
22         // test trim method
23         System.out.printf( "s3 after trim = \"%s\"\n\n", s3.trim() );
24
25         // test toCharArray method
26         char[] charArray = s1.toCharArray();
27         System.out.print( "s1 as a character array = " );
28
29         for ( char character : charArray )
30            System.out.print( character );
31
32         System.out.println();
33      } // end main
34   } // end class StringMiscellaneous2
```

Fig. 16.8 | Str
toCharArray. (Pa

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

16.6 Tokenizing `Strings`

- When you read a sentence, your mind breaks it into tokens—individual words and punctuation marks that convey meaning.
- Compilers also perform tokenization.
- `String` method `split` breaks a `String` into its component tokens and returns an array of `String`s.
- Tokens are separated by delimiters
  - Typically white-space characters such as space, tab, newline and carriage return.
  - Other characters can also be used as delimiters to separate tokens.

```
1  // Fig. 16.18: TokenTest.java
2  // StringTokenizer object used to tokenize strings.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class TokenTest
7  {
8     // execute application
9     public static void main( String[] args )
10    {
11       // get sentence
12       Scanner scanner = new Scanner( System.in );
13       System.out.println( "Enter a sentence and press Enter" );
14       String sentence = scanner.nextLine();
15
16       // process user sentence
17       String[] tokens = sentence.split( " " );
18       System.out.printf( "Number of elements: %d\nThe tokens are:\n",
19          tokens.length );
20
21       for ( String token : tokens )
22          System.out.println( token );
23    } // end main
24 } // end class TokenTest
```

Fig. 16.18 | `StringTokenizer` object used to tokenize strings. (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. 16.18 | `StringTokenizer` object used to tokenize strings. (Part 2 of 2.)

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

- A regular expression is a specially formatted `String` that describes a search pattern for matching characters in other `String`s.
- Useful for validating input and ensuring that data is in a particular format.
- One application of regular expressions is to facilitate the construction of a compiler.
  - Often, a large and complex regular expression is used to validate the syntax of a program.
  - If the program code does not match the regular expression, the compiler knows that there is a syntax error within the code.

```
1  // Fig. 16.18: TokenTest.java
2  // StringTokenizer object used to tokenize strings.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class TokenTest
7  {
8     // execute application
9     public static void main( String[] args )
10    {
11       // get sentence
12       Scanner scanner = new Scanner( System.in );
13       System.out.println( "Enter a sentence and press Enter" );
14       String sentence = scanner.nextLine();
15
16       // process user sentence
17       String[] tokens = sentence.split( " " );
18       System.out.printf( "Number of elements: %d\nThe tokens are:\n",
19          tokens.length );
20
21       for ( String token : tokens )
22          System.out.println( token );
23    } // end main
24 } // end class TokenTest
```

Fig. 16.18 | `StringTokenizer` object used to tokenize strings. (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. 16.18 | `StringTokenizer` object used to tokenize strings. (Part 2 of 2.)

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- `String` method `matches` receives a `String` that specifies the regular expression and matches the contents of the `String` object on which it's called to the regular expression.
  - The method returns a `boolean` indicating whether the match succeeded.
- A regular expression consists of literal characters and special symbols.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- Figure 16.19 specifies some predefined character classes that can be used with regular expressions.
- A character class is an escape sequence that represents a group of characters.
- A digit is any numeric character.
- A word character is any letter (uppercase or lowercase), any digit or the underscore character.
- A white-space character is a space, a tab, a carriage return, a newline or a form feed.
- Each character class matches a single character in the `String` we're attempting to match with the regular expression.
- Regular expressions are not limited to predefined character classes.
- The expressions employ various operators and other forms of notation to match complex patterns.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- To match a set of characters that does not have a predefined character class, use square brackets, `[]`.
  - The pattern `"[aeiou]"` matches a single character that's a vowel.
- Character ranges are represented by placing a dash (`-`) between two characters.
  - `"[A-Z]"` matches a single uppercase letter.
- If the first character in the brackets is `"^"`, the expression accepts any character other than those indicated.
  - `"[^Z]"` is not the same as `"[A-Y]"`, which matches uppercase letters A–Y—`"[^Z]"` matches any character other than capital Z, including lowercase letters and nonletters such as the newline character.

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 nonwhite-space character

Fig. 16.19 | Predefined character classes.

```
1  // Fig. 16.20: ValidateInput.java
2  // Validate user information using regular expressions.
3
4  public class ValidateInput
5  {
6     // validate first name
7     public static boolean validateFirstName( String firstName )
8     {
9        return firstName.matches( "[A-Z][a-zA-Z]*" );
10    } // end method validateFirstName
11
12    // validate last name
13    public static boolean validateLastName( String lastName )
14    {
15       return lastName.matches( "[a-zA-z]+([ '-][a-zA-Z]+)*" );
16    } // end method validateLastName
17
18    // validate address
19    public static boolean validateAddress( String address )
20    {
21       return address.matches(
22          "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
23    } // end method validateAddress
24
```

Fig. 16.20 | Validating user information using regular expressions. (Part 1 of 2.)

```
25      // validate city
26      public static boolean validateCity( String city )
27      {
28         return city.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
29      } // end method validateCity
30
31      // validate state
32      public static boolean validateState( String state )
33      {
34         return state.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) ;
35      } // end method validateState
36
37      // validate zip
38      public static boolean validateZip( String zip )
39      {
40         return zip.matches( "\\d{5}" );
41      } // end method validateZip
42
43      // validate phone
44      public static boolean validatePhone( String phone )
45      {
46         return phone.matches( "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
47      } // end method validatePhone
48   } // end class ValidateInput
```

Fig. 16.20 | Validating user information using regular expressions. (Part 2 of 2.)

```
1  // Fig. 16.21: Validate.java
2  // Validate user information using regular expressions.
3  import java.util.Scanner;
4
5  public class Validate
6  {
7     public static void main( String[] args )
8     {
9        // get user input
10       Scanner scanner = new Scanner( System.in );
11       System.out.println( "Please enter first name:" );
12       String firstName = scanner.nextLine();
13       System.out.println( "Please enter last name:" );
14       String lastName = scanner.nextLine();
15       System.out.println( "Please enter address:" );
16       String address = scanner.nextLine();
17       System.out.println( "Please enter city:" );
18       String city = scanner.nextLine();
19       System.out.println( "Please enter state:" );
20       String state = scanner.nextLine();
21       System.out.println( "Please enter zip:" );
22       String zip = scanner.nextLine();
```

Fig. 16.21 | Inputs and validates data from user using the `ValidateInput` class. (Part 1 of 4.)

```
23          System.out.println( "Please enter phone:" );
24          String phone = scanner.nextLine();
25
26          // validate user input and display error message
27          System.out.println( "\nValidate Result:" );
28
29          if ( !ValidateInput.validateFirstName( firstName ) )
30             System.out.println( "Invalid first name" );
31          else if ( !ValidateInput.validateLastName( lastName ) )
32             System.out.println( "Invalid last name" );
33          else if ( !ValidateInput.validateAddress( address ) )
34             System.out.println( "Invalid address" );
35          else if ( !ValidateInput.validateCity( city ) )
36             System.out.println( "Invalid city" );
37          else if ( !ValidateInput.validateState( state ) )
38             System.out.println( "Invalid state" );
39          else if ( !ValidateInput.validateZip( zip ) )
40             System.out.println( "Invalid zip code" );
41          else if ( !ValidateInput.validatePhone( phone ) )
42             System.out.println( "Invalid phone number" );
43          else
44             System.out.println( "Valid input.  Thank you." );
45       } // end main
46    } // end class Validate
```

Fig. 16.21 | Inputs and validates data from user using the `ValidateInput` class. (Part 2 of 4.)

```
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. 16.21 | Inputs and validates data from user using the `ValidateInput` class. (Part 3 of 4.)

```
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. 16.21 | Inputs and validates data from user using the `ValidateInput` class. (Part 4 of 4.)

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- Ranges in character classes are determined by the letters' integer values.
  - `"[A-Za-z]"` matches all uppercase and lowercase letters.
- The range `"[A-z]"` matches all letters and also matches those characters (such as [and \) with an integer value between uppercase Z and lowercase a.
- Like predefined character classes, character classes delimited by square brackets match a single character in the search object.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- When the regular-expression operator `"*"` appears in a regular expression, the application attempts to match zero or more occurrences of the subexpression immediately preceding the `"*"`.
- Operator `"+"` attempts to match one or more occurrences of the subexpression immediately preceding `"+"`.
- The character `"|"` matches the expression to its left or to its right.
  - `"Hi (John|Jane)"` matches both `"Hi John"` and `"Hi Jane"`.
- Parentheses are used to group parts of the regular expression.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- The asterisk (*) and plus (+) are formally called quantifiers.
- Figure 16.22 lists all the quantifiers.
- A quantifier affects only the subexpression immediately preceding the quantifier.
- Quantifier question mark (?) matches zero or one occurrences of the expression that it quantifies.
- A set of braces containing one number (`{n}`) *matches exactly n occurrences of the expression it quantifies.*
- Including a comma after the number enclosed in braces matches at least *n occurrences of the quantified expression.*
- A set of braces containing two numbers (`{n,m}`), *matches between n and m occurrences of the expression that it qualifies.*

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. 16.22 | Quantifiers used in regular expressions.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.
- All of the quantifiers are greedy.
  - They match as many occurrences as they can as long as the match is still successful.
- If a quantifier is followed by a question mark (`?`), the quantifier becomes reluctant (sometimes called lazy).
  - It will match as few occurrences as possible as long as the match is still successful.
- `String` Method `matches` checks whether an entire `String` conforms to a regular expression.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- Sometimes it's useful to replace parts of a string or to split a string into pieces. For this purpose, class `String` provides methods `replaceAll`, `replaceFirst` and `split`.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- `String` method `replaceAll` replaces text in a `String` with new text (the second argument) wherever the original `String` matches a regular expression (the first argument).
- Escaping a special regular-expression character with \ instructs the matching engine to find the actual character.
- `String` method `replaceFirst` replaces the first occurrence of a pattern match.

```
1  // Fig. 16.23: RegexSubstitution.java
2  // String methods replaceFirst, replaceAll and split.
3  import java.util.Arrays;
4
5  public class RegexSubstitution
6  {
7     public static void main( String[] args )
8     {
9        String firstString = "This sentence ends in 5 stars *****";
10       String secondString = "1, 2, 3, 4, 5, 6, 7, 8";
11
12       System.out.printf( "Original String 1: %s\n", firstString );
13
14       // replace '*' with '^'
15       firstString = firstString.replaceAll( "\\*", "^" );
16
17       System.out.printf( "^ substituted for *: %s\n", firstString );
18
19       // replace 'stars' with 'carets'
20       firstString = firstString.replaceAll( "stars", "carets" );
21
22       System.out.printf(
23          "\"carets\" substituted for \"stars\": %s\n", firstString );
24
```

Fig. 16.23 | `String` methods `replaceFirst`, `replaceAll` & `split`. (Part 1 of 3.)

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

Fig. 16.23 | `String` methods `replaceFirst`, `replaceAll` & `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. 16.23 | `String` methods `replaceFirst`, `replaceAll` & `split`. (Part 3 of 3.)

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- In addition to the regular-expression capabilities of class `String`, Java provides other classes in package `java.util.regex` that help developers manipulate regular expressions.
- Class `Pattern` represents a regular expression.
- Class `Matcher` contains both a regular-expression pattern and a `CharSequence` in which to search for the pattern.
- `CharSequence` (package `java.lang`) is an interface that allows read access to a sequence of characters.
- The interface requires that the methods `charAt`, `length`, `subSequence` and `toString` be declared.
- Both `String` and `StringBuilder` implement interface `CharSequence`, so an instance of either of these classes can be used with class `Matcher`.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- If a regular expression will be used only once, `static` `Pattern` method `matches` can be used.
  - Takes a `String` that specifies the regular expression and a `CharSequence` on which to perform the match.
  - Returns a `boolean` indicating whether the search object (the second argument) matches the regular expression.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- If a regular expression will be used more than once, it's more efficient to use `static` `Pattern` method `compile` to create a specific `Pattern` object for that regular expression.
  - Receives a `String` representing the pattern and returns a new `Pattern` object, which can then be used to call method `matcher`
  - Method `matcher` receives a `CharSequence` to search and returns a `Matcher` object.
- `Matcher` method `matches` performs the same task as `Pattern` method `matches`, but receives no arguments—the search pattern and search object are encapsulated in the `Matcher` object.
- Class `Matcher` provides other methods, including `find`, `lookingAt`, `replaceFirst` and `replaceAll`.

```java
// Fig. 16.24: RegexMatches.java
// Classes Pattern and Matcher.
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
   public static void main( String[] args )
   {
      // create regular expression
      Pattern expression =
         Pattern.compile( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );

      String string1 = "Jane's Birthday is 05-12-75\n" +
         "Dave's Birthday is 11-04-68\n" +
         "John's Birthday is 04-28-73\n" +
         "Joe's Birthday is 12-17-77";

      // match regular expression to string and print matches
      Matcher matcher = expression.matcher( string1 );

      while ( matcher.find() )
         System.out.println( matcher.group() );
   } // end main
} // end class RegexMatches
```

Fig. 16.24 | Classes `Pattern` and `Matcher`. (Part 1 of 2.)

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

Fig. 16.24 | Classes `Pattern` and `Matcher`. (Part 2 of 2.)

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- The dot character "`.`" in a regular expression matches any single character except a newline character.
- `Matcher` method `find` attempts to match a piece of the search object to the search pattern.
  - Each call to this method starts at the point where the last call ended, so multiple matches can be found.
- `Matcher` method `lookingAt` performs the same way, except that it always starts from the beginning of the search object and will always find the first match if there is one.

16.7 Regular Expressions, Class `Pattern` and Class `Matcher` (cont.)

- `Matcher` method `group` returns the `String` from the search object that matches the search pattern.
  - The `String` that is returned is the one that was last matched by a call to `find` or `lookingAt`.