Classes `String` and `StringBuilder` 9

9.1 Methods for Obtaining Information from `String` Data

Java provides a variety of methods for extracting information from a `String` data. Since `String` is an object class, like `Scanner`, we execute a method on a `String` data by attaching a period, the name of the method, and the parameters:

```
STRING_DATA.METHOD_NAME( PARAMETERS )
```

A wide variety of methods are available for `String`. We can divide them roughly into four types:

1. the methods for obtaining information about the `String` data;
2. the methods for comparing the `String` data with another `String` data;
3. the methods for locating a pattern in the `String` data; and
4. the methods for producing a new `String` data from the `String` data.

Table 9.1 summarizes the `String` methods we will study. The table has two additional methods, `toCharArray` and `split`. We will study these methods later.

The first `String` method we lean is `length()`. This method belongs to the first category and returns the character length as `int`. The following code assigns the `String` literal `"the earth"` to a `String` variable named `message`. Next, the code obtains the character length of `message` and stores it in an `int` variable named `characterCount`. Finally, the code prints the values of `message` and `characterCount` using a `printf` statement.

```
String message = "the earth";
int characterCount = message.length();
System.out.printf( "\"\"%s\" has length %d%n", message, characterCount );
```

The output generated by the code is:

```
"the earth" has length 9
```

Another method in the first category is `charAt`. `charAt` receives an `int` value as a parameter, and returns the character of the `String` data at the character position represented by the parameter. In Java, a position count starts from 0, instead of from 1. We see this in `String` and in arrays (see

Table 9.1 A list of `String` methods

Name	Return type	Parameters	Action
`length`	`int`	None	Returns the character length of `s`
`charAt`	`char`	`int p`	Returns the character of `s` at position `p`
`toCharArray`	`char[]`	`void`	Returns the character array representation of `s`
`Split`	`String[]`	`String p`	Returns the `String` array generated by splitting with `s` as the delimiter
`equals`	`boolean`	`String o`	Returns whether or not `s` is equal to `o`
`compareTo`	`int`	`String o`	Returns the result of comparing `s` with `o`
`startsWith`	`boolean`	`String o`	Returns whether or not `s` starts with `o`
`endsWith`	`boolean`	`String o`	Returns whether or not `s` ends with `o`
`indexOf`	`int`	`String w`	Returns the lowest position where `w` occurs in `s`; if `w` does not occur in `s`, returns `-1`
`lastIndexOf`	`int`	`String w`	Returns the highest position where `w` occurs in `s`; if `w` does not occur in `s`, returns `-1`
`indexOf`	`int`	`String w, int p`	Returns the lowest position `>= p` where `w` occurs in `s`; if `w` does not occur in `s` at positions `>= p`, returns `-1`
`lastIndexOf`	`int`	`String w, int p`	Returns the highest position `< p` where `w` occurs in `s`; if `w` does not occur in `s` at positions `< p`, returns `-1`
`trim`	`String`	None	Returns a new `String` without the leading and trailing white space characters
`substring`	`String`	`int i`	Returns the suffix of `s` starting from position `i`
`substring`	`String`	`int i, int j`	Returns the substring of `s` between position `i` and position `j - 1`
`toUpperCase`	`String`	None	Returns a new `String` generated from `s` by converting all lowercase letters to uppercase
`toLowerCase`	`String`	None	Returns a new `String` generated from `s` by converting all uppercase letters to lowercase
`replace`	`String`	`String a, String b`	Returns a new `String` generated from `s` by substituting all occurrences of `a` by `b`
`replaceFirst`	`String`	`String a, String b`	Returns a new `String` generated from `s` by substituting the first occurrence of `a` by `b`

Here `s` represents the `String` data to which the methods appearing in the table are applied. For methods `indexOf` and `lastIndexOf`, a `char` value is also accepted as the first parameter

Chap. 12). For each `String` object having length `N`, its first character is at position 0, its second character is at position 1, and so on. The last character is at position `N - 1`. If `s` is a `String` data and `p` is an `int` value between 0 and `s.length() - 1`, then `s.charAt( p )` returns the `char` of `s` at position `p`.

Consider the following code fragment:

```
1  String message = "the sun";
2  System.out.println( "char at 1 is " + message.charAt( 1 ) );
3  System.out.println( "char at 5 is " + message.charAt( 5 ) );
4  System.out.println( "char at 6 is " + message.charAt( 6 ) );
```

This code produces the output:

```
char at 1 is h
char at 5 is u
char at 6 is n
```

If we call charAt with a position value outside the valid range, a run-time error, StringIndexOutOfBoundsException, occurs. The following program demonstrates the use of length and charAt, and generates StringIndexOutOfBoundsException. The program receives an input line from the user using the method nextLine of Scanner, and stores it in a String variable named line. The program then executes a for-loop that iterates over the sequence 0, ..., input.length() with an index variable named i. For each value of i, the program prints the value of i as well as the value of input.chatAt(i).

```java
import java.util.Scanner;
public class StringIndices
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter string: " );
    String input = keyboard.nextLine();
    for ( int i = 0; i <= input.length(); i ++ )
    {
      System.out.print( "position = " + i );
      System.out.println( " .. char is " + input.charAt( i ) );
    }
  }
}
```

Listing 9.1 An example of StringIndexOutOfBoundsException

Since the valid range of the parameter for input.chatAt is from 0 to input.length() - 1, the program makes an invalid method call at the last round of the iteration. At that round, the value of i is equal to input.length(). Here is how the program runs and how the error is generated.

```
Enter string: Hello, World!
position = 0 .. char is H
position = 1 .. char is e
position = 2 .. char is l
position = 3 .. char is l
position = 4 .. char is o
position = 5 .. char is ,
position = 6 .. char is
position = 7 .. char is W
position = 8 .. char is o
position = 9 .. char is r
position = 10 .. char is l
position = 11 .. char is d
position = 12 .. char is !
position = 13Exception in thread "main"
    java.lang.StringIndexOutOfBoundsException: String index out of range:
    13
  at java.lang.String.charAt(String.java:646)
  at StringIndices.main(StringIndices.java:12)
```

The error message starts immediately after the output position = 13.

9.2 Methods for Comparing String Data with Another

9.2.1 The Equality Test and the Comparison in Dictionary Order

As mentioned earlier, the mathematical equality and inequality tests work correctly only for primitive data types. `String` has two methods for content equality and content comparison. They are methods `equals` and `compareTo`. Let `s` and `t` be two `String` objects. `s.equals( t )` returns a `boolean` that represents whether or not `s` and `t` have the same character sequences. The relation computed by `equals` is symmetric and reflexive. In other words, `s.equals( t )` and `t.equals( s )` have the same values, and for all `s` that is not `null`, `s.equals( s )` is `true`.

The method `s.compareTo( t )` returns an `int` value representing the result of performing character-by-character comparison from start to end between the two `String` objects. The is based upon the indexes of the characters in the Unicode table. The comparison is terminated when either all the characters of either `s` or `t` have been examined or the character of `s` at the present position has been found to be different from the character of `t` at the same position.

In the former situation, there are three possible outcomes.

1. If all the characters have been complete examined for both `s` and `t`, the method returns 0.
2. If `s` has at least character remaining, it means that `t` is a proper prefix of `s` and the method returns a strictly positive integer.
3. If `t` has at least character remaining, it means that `s` is a proper prefix of `t` and the method returns a strictly negative integer.

In the latter situation, there are two possible outcomes.

1. If the character of `s` has a higher position than the character of `t` in the Unicode character indexes, he method returns a strictly positive integer.
2. If the character of `s` has a lower position than the character of `t` in the Unicode character indexes, the method returns a strictly negative integer.

The relation defined by the method `compareTo` of `String` data is transitive, in the sense that if `s.compareTo( t )` and `t.compareTo( u )` are both positive, then `s.compareTo( u )` is positive, and if `s.compareTo( t )` and `t.compareTo( u )` are both negative, then `s.compareTo( u )` is negative. Because of the transitive property, the method `compareTo` induces a complete ordering of all `String` values. This property is used in the method `sort` of class `Arrays` that we will see later in Sect. 13.1. Furthermore, `s.compareTo( t ) + t.compareTo( s )` is equal to 0.

It is practically impossible to remember the position of each character in the Unicode table, but the following information may be helpful:

- the numerals appear consecutively in ten positions, starting with `'0'` and ending with `'9'`;
- the uppercase letters appear consecutively in 26 positions, starting with `'A'` and ending with `'Z'`;
- the lowercase letters appear consecutively in 26 positions, starting with `'a'` and ending with `'z'`;
- the numerals precede the uppercase letters and the uppercase letters precede the lowercase letters.

The next program, `StringCompExample`, receives three `String` data, `text1`, `text2`, and `text3`, from the user and compares them. The program has two methods, `performEquals` and

```java
import java.util.*;
public class StringCompExample
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter #1: " );
    String text1 = keyboard.nextLine();
    System.out.print( "Enter #2: " );
    String text2 = keyboard.nextLine();
    System.out.print( "Enter #3: " );
    String text3 = keyboard.nextLine();

```

Listing 9.2 An example of String comparison (part 1). This section is for receiving the input data

performCompareTo. performEquals receives two String data, s and t, and reports the result of comparing s with t using equals. performCompareTo receives two String data, s and t, and reports the result of comparing s with t using compareTo. Both methods use printf for reporting the result. The method performEquals uses %s as the placeholder for printing the return value of the equals method, while the method performCompareTo uses %d as the placeholder for printing the return value of the compareTo. Using the two methods, the method main executes equals between text1 and text1 itself, between text1 and text2, and between text2 and text3. The method then executes comparesTo between text1 and text1 itself, and all distinct pairs (in this case, there are six of them).

```java
    performEquals( text1, text1 );
    performEquals( text1, text2 );
    performEquals( text2, text3 );

    performCompareTo( text1, text1 );
    performCompareTo( text1, text2 );
    performCompareTo( text2, text1 );
    performCompareTo( text2, text3 );
    performCompareTo( text3, text2 );
    performCompareTo( text3, text1 );
    performCompareTo( text1, text3 );
  }
  public static void performEquals( String s, String t )
  {
    boolean result = s.equals( t );
    System.out.printf( "\"%s\" equals \"%s\": %s%n", s, t, result );
  }
  public static void performCompareTo( String s, String t )
  {
    int result = s.compareTo( t );
    System.out.printf( "\"%s\" compareTo \"%s\": %d%n", s, t, result );
  }
}
```

Listing 9.3 An example of String comparison (part 2). This section is for executing comparisons

Here is an execution example of the program:

```
Enter #1: New Hampshire
Enter #2: New Mexico
Enter #3: New York
"New Hampshire" equals "New Hampshire": true
"New Hampshire" equals "New Mexico": false
"New Mexico" equals "New York": false
"New Hampshire" compareTo "New Hampshire": 0
"New Hampshire" compareTo "New Mexico": -5
"New Mexico" compareTo "New Hampshire": 5
"New Mexico" compareTo "New York": -12
"New York" compareTo "New Mexico": 12
"New York" compareTo "New Hampshire": 17
"New Hampshire" compareTo "New York": -17
```

The ordering among the three input data, from the smallest to the largest is:

`"New Hampshire"`, `"New Mexico""`, `"New York"`

Here is another example. This time, the input contains numerals:

```
Enter #1: ZeroOne
Enter #2: zeroone
Enter #3: 01
"ZeroOne" equals "ZeroOne": true
"ZeroOne" equals "zeroone": false
"zeroone" equals "01": false
"ZeroOne" compareTo "ZeroOne": 0
"ZeroOne" compareTo "zeroone": -32
"zeroone" compareTo "ZeroOne": 32
"zeroone" compareTo "01": 74
"01" compareTo "zeroone": -74
"01" compareTo "ZeroOne": -42
"ZeroOne" compareTo "01": 42
```

The ordering among the three inputs, from the smallest to the largest is:

`"01"`, `"ZeroOne"`, `"zeroone"`

9.2.2 The Prefix and Suffix Tests

The class `String` offers prefix and suffix tests, called `startsWith` and `startsWith`. Let `s` and `t` be `String` data.

- `s.startsWith( t )` returns a `boolean` value that represents whether or not `t` is a prefix of `s`.
- `s.endsWith( t )` returns a `boolean` value that represents whether or not `t` is a suffix of `s`.

The next code compares the `String` literal `"Computer Science"` with three other literals, `"Computer"`, `"Science"`, and `"Engineering"`, with respect to prefixes and suffices. The program uses methods, `prefixTest` and `suffixTest`, that receive two `String` parameters, performs either the prefix or the suffix tests, and reports the result.

```java
public class PrefixSuffix {
  public static void main( String[] args )
  {
    String cs = "Computer Science";
    String comp = "Computer";
    String sci = "Science";
    String eng = "Engineering";

    prefixTest( cs, comp );
    prefixTest( cs, sci );
    prefixTest( cs, eng );
    suffixTest( cs, comp );
    suffixTest( cs, sci );
    suffixTest( cs, eng );
  }
  public static void prefixTest( String line, String pattern )
  {
    String neg = "";
    if ( !line.startsWith( pattern ) )
    {
      neg = "not ";
    }
    System.out.printf( "\"%s\" is %sa prefix of \"%s\".%n",
        pattern, neg, line );
  }
  public static void suffixTest( String line, String pattern )
  {
    String neg = "";
    if ( !line.endsWith( pattern ) )
    {
      neg = "not ";
    }
    System.out.printf( "\"%s\" is %sa suffix of \"%s\".%n",
        pattern, neg, line );
  }
}
```

Listing 9.4 A program that demonstrates the use of `beginsWith` and `endsWith`

The execution of the code produces the following result:

```
Is "Computer" a prefix of "Computer Science"? true
Is "Science" a prefix of "Computer Science"? false
Is "Engineering" a prefix of "Computer Science"? false
Is "Computer" a suffix of "Computer Science"? false
Is "Science" a suffix of "Computer Science"? true
Is "Engineering" a suffix of "Computer Science"? false
```

9.3 Methods for Searching for a Pattern in a `String` Data

For two `String` data `s` and `t`, and an integer `q`, we say that `t` appears in `s` at position `q`, if the character sequence of `s` starting from position `q` has `t` as a prefix. More precisely, `t` appears in `s` at position `q` if `q + t.length() <= s.length()` and the character sequence `s.charAt( q ), ..., s.charAt( q + t.length() - 1 )` is equal to the characters of `t`. For instance, in `"Panama"`, `"a"` appears at 1, 3, and 5.

indexOf and lastIndexOf are methods for pattern search. Both methods receive the pattern to search for as a parameter. The pattern is either a String data or a char data. Both methods may take a second parameter. The second parameter, if present, is an int data and represents the region of search. The return type is int for both. The return value represents the position at the pattern appears. If the pattern does not appear, the return value is -1. The method indexOf returns the lowest position at which the pattern appears. If the second parameter is present, the method indexOf returns the lowest position at which the pattern appears among the positions greater than or equal to the value of the second parameter. The method lastIndexOf returns the highest position at which the pattern appears. If the second parameter is present, the method lastIndexOf returns the highest position at which the pattern appears among the positions less than the value of the second parameter. If there is no match in the specified region, both two-parameter versions return -1. Let seq be a String data that contains opening lines from the hymn "Swing Low, Sweet Chariot" in all lower case, as follows:

```
"swing low, sweet chariot, comin' for to carry me home"
```

We can present the characters of seq with their positions in a table-like format as follows:

```
char: swing low, sweet chariot, comin' for to carry me home
ten:  00000000000111111111112222222222233333333333444444444
one:  01234556789012345567890123455678901234556789012345567
```

In the table, the row starting with "char" shows the character sequence of seq, and the next two rows show their character positions. The row starting with "ten" represents the digit in the tens place, and the row starting with "one" represents the digit in the ones place. We can see that seq contains four occurrences of the letter 'e', at positions 12, 13, 43, and 47. The method call seq.indexOf('e') returns the smallest among the four, namely 12. The method call seq.indexOf('e', 20) returns the smallest among those that are greater than or equal to 20, namely 43. The method call seq.lastIndexOf('e') returns the largest among the four values, namely 47. The method call seq.lastIndexOf('e', 30) returns the largest among those that are strictly less than 30, namely 13. The method call seq.lastIndexOf('e', 12) returns -1 since none of the positions are smaller than 12. If the pattern to search for is "sw", the positions at which the pattern occurs are 0 and 11. Therefore, seq.indexOf("sw") returns 0, seq.indexOf("sw", 7) returns 11, seq.indexOf("sw", 12) returns −1, seq.lastIndexOf("sw", 7) returns 0, and seq.lastIndexOf("sw", 12) returns 11.

The next program, IndexOf, receives an input line, a pattern to search for in the input line, and a position that represents a search range, and then prints the result of executing the search methods. Here is the part for receiving the input from the user:

```java
import java.util.*;
public class IndexOf
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter the input: " );
    String input = keyboard.nextLine();
    System.out.print( "Enter the pattern: " );
    String pat = keyboard.nextLine();
    System.out.print( "Enter the position " );
    int pos = keyboard.nextInt();

```

Listing 9.5 A program that demonstrates the use of pattern search methods of String (part 1). The part that receives the input

The program presents the input and the character positions using the table-like format shown above. If the input from the user has length between 0 and 100, the decimal representation of each character position requires at most two digits. If a value between 0 and 99 is represented by an `int` variable `i`, its digit in the tens place is `i / 10` and its digit in the ones place is `i % 10`. Based upon this observation, the program uses the following code to produce the header.

```java
    System.out.println();
    for ( int i = 0; i <= input.length() - 1; i ++ )
    {
      System.out.print( ( i / 10 ) % 10 );
    }
    System.out.println();
    for ( int i = 0; i <= input.length() - 1; i ++ )
    {
      System.out.print( i % 10 );
    }
    System.out.println();
    System.out.println( input );
    for ( int i = 0; i <= input.length() - 1; i ++ )
    {
      System.out.print( '-' );
    }
    System.out.println();

```

Listing 9.6 A program that demonstrates the use of pattern search methods of String (part 2). The part that prints the header of the output

The program then executes pattern search. There are four different calls. The program announces the method it is about to call, and then prints the return value.

```java
    System.out.print( "indexOf(\"" + pat + "\"): " );
    System.out.println( input.indexOf( pat ) );
    System.out.print( "lastIndexOf(\"" + pat + "\"): " );
    System.out.println( input.lastIndexOf( pat ) );
    System.out.print( "indexOf(\"" + pat + "\"," + pos + "): " );
    System.out.println( input.indexOf( pat, pos ) );
    System.out.print( "lastIndexOf(\"" + pat + "\"," + pos + "): " );
    System.out.println( input.lastIndexOf( pat, pos ) );
  }
}
```

Listing 9.7 A program that demonstrates the use of pattern search methods of String (part 3). The part that calls the search methods and prints the results

Here is the result of executing the code with the two lines from “Swing Low, Sweet Chariot”:

```
Enter the input: swing low, sweet chariot, comin' for to carry me home
Enter the pattern: e
Enter the position 20

00000000001111111111222222222233333333334444444444555
01234567890123456789012345678901234567890123456789012
swing low, sweet chariot, comin' for to carry me home
```

```
------------------------------------------------------
indexOf("e"): 13
lastIndexOf("e"): 52
indexOf("e",20): 47
lastIndexOf("e",20): 14
```

Here is another example with the two lines from the second verse of the hymn "Jerusalem".

```
Enter the input: bring me my bow of burning gold! bring me my arrows of
    desire!
Enter the pattern: ow
Enter the position 10

00000000001111111111222222222233333333334444444444555555555566
01234567890123456789012345678901234567890123456789012345678901
bring me my bow of burning gold! bring me my arrows of desire!
---------------------------------------------------------------
indexOf("ow"): 13
lastIndexOf("ow"): 48
indexOf("ow",10): 13
lastIndexOf("ow",10): -1
```

9.4 Methods for Creating New `String` Data from Another

The final group of `String` methods that we learn consists of those that generate a new `String` data. Let `s` be a `String` object.

1. `s.toUpperCase()` returns a copy of `s` in which each lowercase letter is switched to its uppercase version; if no lowercase letter appears in `s`, the method returns the exact copy of `s`.
2. `s.toLowerCase()` returns a copy of `s` in which each uppercase letter is switched to its lowercase version; if no uppercase letter appears in `s`, the method returns the exact copy of `s`.
3. `s.substring( int startIndex )` returns a copy of `s` starting from position `startIndex`; if the index value is negative or greater than the length of `s`, the method produces a run-time error `StringIndexOutOfBoundsException`.
4. `s.substring( int startIndex, int endIndex )` returns a copy of `s` starting at position `startIndex` and ending at position `endIndex - 1`; if either index value is negative or greater than the length of `s`, the method produces a run-time error `StringIndexOutOfBoundsException`.
5. `s.replace( String x, String y )` returns a copy of `s` in which all the occurrences of `x` are replaced with `y`; if `x` does not occur in `s`, the method returns the exact copy of `s`. If there is only one occurrence of `x` in `s`, that occurrence is substituted with `y`. If there are multiple occurrences of `x` in `s` and if some consecutive occurrences overlap, the occurrences are chosen without overlap in a "greedy" fashion, as follows: The first occurrence chosen is the occurrence at the lowest position. From the second occurrence on, the occurrence chosen is the one at the lowest position that overlaps none of the previously chosen occurrences. For instance, `"abababa"` has three occurrences of `"aba"`, at positions 0, 2, and 4. The second occurrence overlap the first and the last. Given `"aba"` as the first parameter, `replace` chooses the positions 0 and 4.
6. `s.replaceFirst( String x, String y )` is a variant of `replace` where the substitution applies only to the first occurrence of `x`.
7. `s.trim()` returns a copy of `s` without all of its leading and trailing white space characters.

The next program demonstrates the use of the String generation methods. The method main of the program calls three methods names changesOne, changesTwo, and changesThree (Lines 6–8).

```
1  import java.util.*;
2  public class ModifyString
3  {
4    public static void main( String[] args )
5    {
6      changesOne();
7      changesTwo();
8      changesThree();
9    }
```

Listing 9.8 A program that demonstrates the use of String methods for generating new String data (part 1). The method main

The first method of three, changesOne, demonstrates the use of toLowerCase(), toUpperCase(), and trim(). The method receives an input line from the user, stores it in a variable named input (Lines 13 and 14), and then executes input.toLowerCase() (Lines 15 and 16), input.toUpperCase() (Lines 17 and 18), and input.trim() (Lines 19 and 20). The program uses a String variable named result to receive the return values of these methods.

```
10   public static void changesOne()
11   {
12     Scanner keyboard = new Scanner( System.in );
13     System.out.print( "Enter the input String: " );
14     String input = keyboard.nextLine();
15     String result = input.toLowerCase();
16     System.out.println( "lower: " + result );
17     result = input.toUpperCase();
18     System.out.println( "upper: " + result );
19     result = input.trim();
20     System.out.println( "trim: " + input.trim() );
21   }
```

Listing 9.9 A program that demonstrates the use of String methods for generating new String data (part 2). The method changesOne

In changesTwo, the program receives an input line (Lines 25 and 26) and two position values (Lines 27–30) from the user, and then calls the method substring. To announce the action that has been performed and its result, the program uses printf, with the format:

```
substring(%d)=%s%n
```

for the one-parameter version, and the format:

```
substring(%d,%d)=%s%n
```

for the two-parameter version. Each %d is the placeholder for the actual parameter, and each %s is the placeholder for the return value.

The nextLine appearing in Line 30 is necessary, for the following reason: After changesTwo, the method changesThree is called. The first action changesThree performs with a Scanner is to read an input line with nextLine. To receive the two position values, changesTwo uses nextInt. If there is no nextLine after the two calls of nextInt in changesTwo, the first nextLine in changesThree returns the sequence of characters entered between the last numeral of the second integer retrieved with nextInt in changesTwo and the return key that has been pressed to enter the numbers.

```
22    public static void changesTwo()
23    {
24      Scanner keyboard = new Scanner( System.in );
25      System.out.print( "Enter the input String: " );
26      String input = keyboard.nextLine();
27      System.out.print( "Enter start and end positions " );
28      int pos1 = keyboard.nextInt();
29      int pos2 = keyboard.nextInt();
30      keyboard.nextLine();
31      String result = input.substring( pos1 );
32      System.out.printf( "substring( %d ): %s%n", pos1, result );
33      result = input.substring( pos1, pos2 );
34      System.out.printf( "substring( %d, %d ): %s%n", pos1, pos2, result );
35    }
```

Listing 9.10 A program that demonstrates the use of String methods for generating new String data (part 3). The method changesTwo

In changesThree, the program receives an input line (Lines 39 and 40) and two additional lines representing the patterns (Lines 41–44) from the user, and then executes the pattern replacement (Lines 45–48).

```
36    public static void changesThree()
37    {
38      Scanner keyboard = new Scanner( System.in );
39      System.out.print( "Enter the input String: " );
40      String input = keyboard.nextLine();
41      System.out.print( "Enter pattern 1: " );
42      String pat1 = keyboard.nextLine();
43      System.out.print( "Enter pattern 2: " );
44      String pat2 = keyboard.nextLine();
45      String result = input.replaceFirst( pat1, pat2 );
46      System.out.printf( "replaceFirst( %s,%s ): %s%n",
47          pat1, pat2, result );
48      result = input.replace( pat1, pat2 ) ;
49      System.out.printf( "replace( %s,%s ): %s%n", pat1, pat2, result );
50    }
51  }
```

Listing 9.11 A program that demonstrates the use of String methods for generating new String data (part 4). The method changesThree

Here is one execution example of the code.

```
Enter the input String:    Sorry, Professor. My phone's alarm didn't
    work...
lower:    sorry, professor. my phone's alarm didn't work...
upper:    SORRY, PROFESSOR. MY PHONE'S ALARM DIDN'T WORK...
trim: Sorry, Professor. My phone's alarm didn't work...
Enter the input String: We have caught a possum resembling Fairway Frank.
Enter start and end positions 10 20
```

```
substring(10)=ught a possum resembling Fairway Frank.
substring(10,20)=ught a pos
Enter the input String: I've received an A in BIO101 and CHM101.
Enter pattern 1: A
Enter pattern 2: A+
replaceFirst(A,A+)=I've received an A+ in BIO101 and CHM101.
replaceF(A,A+)=I've received an A+ in BIO101 and CHM101.
```

9.4.1 String.format

There is one static `String` method that is used often. The method is `format`. We can use this method to mimic the action of `printf` and receive the result as a return value, instead of printing it on the screen. For example,

```
int x = 10;
double y = 1.7956;
String output = String.format( "x=%d,y=%.2f", x, y );
```

stores the character sequence `x=10,y=1.80` in the `String` variable `output` (because the rounding for `y` occurs at the third position after the decimal point).

9.5 Class StringBuilder

Quite often, we need to produce a long `String` output spreading over multiple lines, either on the screen or to some file. We can build such an output using `String` concatenation, by adding components one after another. A `StringBuilder` object can be used to build a `String` data through insertion, deletion, and concatenation. To create a `StringBuilder` object, we use a constructor, either with a `String` as its initial contents or without, as shown next:

```
StringBuilder builder1 = new StringBuilder();
StringBuilder builder2 = new StringBuilder( "Hello, World!" );
```

A `String` data that a `StringBuilder` object represents can be obtained by calling the method `toString`. Some methods of `String` are available for `StringBuilder` too. They include `length`, `charAt`, `indexOf`, `lastIndexOf`, and `substring`. The `StringBuilder` versions of these methods are applied to the `String` data that the `StringBuilder` object represents.

There are methods that are available in `StringBuilder` but not in `String`. They include `append`, `insert`, and `delete`:

- The method `append` receives one formal parameter and appends its value to the contents. The type of the parameter can be `boolean`, `char`, `double`, `float`, `int`, `long`, or `String`.
- The method `insert` receives two formal parameters. The first parameter is an `int` and specifies where, in the contents of the `StringBuilder` object, an insertion must be made. The second parameter specifies the actual data to insert. The type of the second parameter can be `boolean`, `char`, `double`, `float`, `int`, `long`, or `String`.
- The method `delete` receives two `int` parameters, `start` and `end`, and removes the characters at positions between `start` and `end - 1` from the contents. The use of an invalid index results in the run-time error of `StringIndexOutOfBoundsException` occurs. There is a one-parameter version of `delete`. This version removes all the characters starting from the position that the parameter specifies.

Here is a demonstration of how the methods of StringBuilder work. The program receives an input line from the user, and then collects all the lowercase letters appearing in the input line. The collected letters are simply connected without spacing in between and turned into a String data. The method main stores its input in a variable named input (Lines 7 and 8) and calls the method collect. The method stores the returned value in a variable named output (Line 9). The method then presents the two values (Line 11).

The method collect receives a String data as a formal parameter, and returns a String object (Line 13). The method instantiates a StringBuilder object (Line 15). It then goes through the characters of the input line one after another (Line 16). For each character encountered, if the character is a lowercase letter (Lines 18 and 19), the method appends the character to the builder (Line 21). After completing the examination, the method inserts the sequence -\n after every ten characters of the output. To accomplish this task, the method iterates the sequence builder.length(), ..., 1 with the variable i (Lines 24 and 25). At each round of the iteration, if the value of i is a multiple of 10 (Line 26), the method inserts the sequence at position i (Line 28). The values of i appear in decreasing order because an insertion changes the character positions of all the existing characters appearing after the position of insertion.

```
 1  import java.util.*;
 2  public class AlphabetCollection
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7      System.out.print( "Enter: " );
 8      String input = keyboard.nextLine();
 9      String output = collect( input );
10      System.out.println( "========" );
11      System.out.printf( "Input:%n%s%nhas become:%n%s%n", input, output );
12    }
13    public static String collect( String input )
14    {
15      StringBuilder builder = new StringBuilder();
16      for ( int i = 0; i < input.length(); i ++ )
17      {
18        char c = input.charAt( i );
19        if ( c >= 'a' && c <= 'z' )
20        {
21          builder.append( c );
22        }
23      }
24      int ell = builder.length();
25      for ( int i = ell; i >= 1; i -- ) {
26        if ( i % 10 == 0 )
27        {
28          builder.insert( i, "-\n" );
29        }
30      }
31      return builder.toString();
32    }
33  }
```

Listing 9.12 A program that demonstrates the use of StringBuilder objects

Here is an execution example of the code:

```
Enter: The title of this album is "Sgt. Pepper's Lonely Hearts Club Band"
========
Input:
The title of this album is "Sgt. Pepper's Lonely Hearts Club Band"
has become:
hetitleoft-
hisalbumis-
gtepperson-
elyeartslu-
band
```

Summary

- `String` has a wide variety of methods. None of them change the contents of the `String` to which the methods are applied.
- `StringBuilder` is a class for building a `String` object. Many methods of `String` can be applied to `StringBuilder` objects.
- `length` and `charAt` provide the character length of the `String` and the character at the specified position.
- The use of actual parameters outside the range for the `charAt` method produces `StringIndex OutOfBoundsException`.
- `indexOf` and `lastIndexOf` can be used to search for patterns in a `String` data and in a `StringBuilder` data. Both methods return `-1` if the pattern does not exist.
- `String` data and `StringBuilder` data can be compared using `compareTo`, `equals`, `startsWith`, and `endsWith`.
- `substring` generates a substring.
- `toUpperCase` and `toLowerCase` return a new `String` data after changing the cases.
- `trim` returns a new `String` data without leading and trailing white space characters.
- `replace` and `replaceFirst` return a new `String` data generated by substitution.
- `String.format` is a static method for generating the output of `System.out.printf` as a `String`.
- The methods available for `StringBuilder` but not for `String` include `append`, `delete`, and `insert`.

Exercises

1. **String arithmetic** Write a program named `ReceiveAndPrint` that receives a `String` value `s`, an `int` value `m`, and a `double` value `d` from the user, and then prints the following:
   - `s`
   - `m`
   - `d`
   - `m + d + s`
   - `m + s + d`
   - `s + m + d`

2. **Concept check**
   (a) Name the `String` method for comparing a `String` with another just for equality.
   (b) Name the `String` method for substituting all occurrences of one pattern with another.
   (c) State whether or not the following statement is true: For a `String` `word` having length 10, `word.substring( 1 )` and `w.substring( 1, 9 )` produce an identical result.
   (d) State whether or not the following statement is true: The `compareTo` method for `String` always produces `+1`, `0`, or `-1`.
   (e) Name the `String` method that returns, when applied to a `String` data, a new `String` without leading and trailing whitespace characters.
3. **Connecting `String` values** Write a public static method named `connect`. The method receives two `String` formal parameters: `word1` and `word2`. The method must return a `boolean`. The return value must represent whether or not the last character of `word1` is equal to the first character of `word2`, or the last character of `word2` is equal to the first character of `word1`. If either `word1` or `word2` has length 0, the method must return `false`.
4. **String methods** Let `word` be a `String` data whose value is `"School.of.Progressive.Rock"` (not including the quotation marks). State the return value of each of the following:
   (a) `word.length()`
   (b) `word.substring( 22 )`
   (c) `word.substring( 22, 24 )`
   (d) `word.indexOf( "oo" )`
   (e) `word.toUpperCase()`
   (f) `word.lastIndexOf( "o" )`
   (g) `word.indexOf( "ok" )`
5. **Understanding `String` methods** Suppose `String` variables `w` and `pat` are given as follows:

```
w = "Singin'_in_the_rain";
pat = "in";
```

   State the return value of each of the following:
   (a) `w.indexOf( pat )`
   (b) `w.indexOf( pat, 3 )`
   (c) `w.indexOf( pat, 6 )`
   (d) `w.lastIndexOf( pat )`
   (e) `w.length()`
   (f) `w.toUpperCase()`
   (g) `w.charAt( 0 )`
6. **Printing the letters of a `String` variable** Suppose `word` is a `String` variable. Using a for-loop, write a code that prints the letters of `word` from the start to the end, one letter per line with no indentation. For example, if the value of `word` is equal to the literal `"hurricanes"`, the output of the code is:

```
h
u
r
r
i
c
a
n
e
s
```

7. **Printing the suffixes of a `String` variable** Suppose `word` is a `String` variable. Using a for-loop, write a code that prints the suffixes of `word` starting from the longest to the shortest, one substring per line with no indentation. For example, if the value of `word` is equal to the literal `"hurricanes"`, the output of the code is:

```
hurricanes
urricanes
rricanes
ricanes
icanes
canes
anes
nes
es
s
```

8. **Concept check** Let `s` be a `String` variable whose value is the literal `"Mississippi"`. State the return value of each of the following:
(a) `s.length()`
(b) `s.indexOf( "si" )`
(c) `s.toUpperCase().indexOf( "si" )`
(d) `s.toLowerCase().indexOf( "si" )`
(e) `s.substring(0,s.indexOf( "i" ))`
(f) `s.substring( s.lastIndexOf( "i" ) )`
9. **Character order reversal** Write a program named `StringReverse` that receives a `String` data from the user, and then creates the reverse of the input. For example, the program may run as follows:

```
Enter an input String: Computer-Programming
The reverse of Computer-Programming is gnimmargorP-retupmoC.
```

The first line consists of the prompt (ending with `": "`) and the input. The second line is the output of the program after receiving the input. Try to use `printf` in producing the output.
10. **Sum of all digits** Write a program named `NumeralSum` that receives a `String` data from the user, and then computes the sum of the values of all the numerals appearing in it. For example, if the input is `BIO542L`, the sum is $5 + 4 + 2 = 11$.

Programming Projects

11. **Cyclic shifts** Suppose `w` is a `String` variable such that `w.length()` is greater than or equal to 1. For a positive integer k whose value is between 0 and `w.length()`, the k-th left cyclic shift of `w` is the `String` constructed from `w` by moving the first k characters of `w` after the last character of `w` while preserving the order of the k characters. For instance, if `w` has the value `"abcdefgh"` and k is 3, then the k-th left cyclic shift of `w` is `"efghabc"`. If the value of k is either equal to 0 or equal to `w.length()`, the k-th left cyclic shift produces a `String` value equal to the value of `w`. Write a program named `CyclicShift` that does the following: The program receives an input line from the user, and stores the input line in a variable `w`. The program receives a nonnegative integer from the user, and stores it in an `int` variable `k`. The program then constructs the k-th cyclic shift of `w` and prints it. Design your code so that the program is able to receive a line that contains the whitespace as input. If the value of `k` is out of range, the program reports that the value is invalid and stop. Here are execution examples of such a program.

```
Enter your input line: How are you?
Enter the shift value k: 4
The 4-th cyclic shift of
"How are you?"
is
"are you?How "
```

```
Enter your input line: How are you?
Enter the shift value k: 34
Invalid value for k
```

12. **Factor of a `String` data** The "minimum factor" of a character sequence w is the shortest prefix s of w such that w is a repetition of s. For example, the minimum factor of `"ababababab"` is `"ab"` and the minimum factor of `"abcdef"` is `"abcdef"`. Put differently, the length of the minimum factor of w is the smallest positive k such that the k-th left cyclic shift of w is equal to w.

    Consider computing the minimum factor of a given `String` object `w` using a for-loop that iterates over the sequence `1, ..., w.length()` with the variable `k`. At each round of the for-loop, the program checks if the `k`-th left cyclic shift of `w` is equal to `w`. The smallest value of `k` at which the shift produces the same `String` is the length of the factor.

    Write a program named `StringFactor` that receives an input line from the user, and then reports its minimum factor along with the length of the factor. Design the program so that it uses a method that returns the minimum factor.
13. **Enumerating all occurrences of a character pattern, part 1** Consider finding all occurrences of a pattern `pat` in a `String` data `input`. We can solve the problem by checking at each position of `input`, whether or not `pat` appears at the position. Write a program named `AllOccurrences` that receives the values for `pat` and `input` from the user, finds all the matching positions, and prints the total number of occurrences.

    Here is an example of how such a code may work.

```
Input some text: I'm singin' in the rain, I'm singin' in the rain
Input pattern: in
Found at position 5.
Found at position 8.
Found at position 12.
Found at position 21.
Found at position 30.
Found at position 33.
Found at position 37.
Found at position 46.
The number of occurrences is 8.
```

    The smallest position possible for `i` is 0.
14. **Enumerating all occurrences of a character pattern, part 2** Write another program named `AllOccurrencesAlt` that solves the previous problem with character-by-character comparisons between the input and the pattern. The task can be accomplished using a double for-loop.
15. **Switching between two neighbors** Write a program named `SwitchingBetweenNeighbors` that executes the following: The program receives a `String` value from the user. From the input, the program creates a new `String` value by switching between every pair of the characters in the input. The switching occurs between positions 0 and 1, between positions 2 and 3, and so. If the input the user provides has an odd length, the last character will remain in the same position. For example, the method should produce `"cseicne"` from `"science"`.

16. **Playing with `StringBuilder`** Write a program named `DoubleInsertion` that receives a `String` value from the user using `nextLine`, and then builds a new `String` using `StringBuilder` as follows:
    - Initially the builder is empty.
    - The program scans the characters of the input, from the beginning to the end, and executes the following:
      – If the position of the character is `2m` for some nonnegative integer `m`, the program inserts the character at position `m`.
      – If the position is `2m + 1` for some nonnegative integer `m`, the program appends the character at the very end.

    The program must print the input in one line, and then the output in the next line. For example, if `abcdefghijkl` is the input, the contents of the `StringBuilder` change as follows:

```
a
ab
acb
acbd
acebd
acebdf
acegbdf
acegbdfh
acegibdfh
acegibdfhj
acegikbdfhj
acegikbdfhjl
```

17. **All substrings** Write a program named `AllSubstrings` that receives a `String` from the user and produces all its nonempty substrings along with their ranges of character positions. Use a double for-loop to generate all pairs of index values `(i,j)` to provide the `substring` method as its actual parameters. Allocate three character spaces to each coordinate value. The output of the program may look like:

```
Enter your input string: karma
(  0,  1):k
(  0,  2):ka
(  0,  3):kar
(  0,  4):karm
(  1,  2):a
(  1,  3):ar
(  1,  4):arm
(  2,  3):r
(  2,  4):rm
(  3,  4):m
```

18. **All anti-substrings** Write a program named `AllAntiSubstrings` that receives a `String` from the user, and then produces all strings generated from the input by removing some substring. Use a double for-loop that iterates over all possible index pairs `(i,j)` such that `i < j`. For each pair, remove from the input the characters having indexes between `i` and `j - 1`. Allocate three character spaces to each coordinate value. Here is an example of how the code may work:

```
Enter your input string: walking
(  0,  1):alking
(  0,  2):lking
(  0,  3):king
(  0,  4):ing
(  0,  5):ng
(  0,  6):g
(  1,  2):wlking
(  1,  3):wking
(  1,  4):wing
(  1,  5):wng
(  1,  6):wg
(  2,  3):waking
(  2,  4):waing
(  2,  5):wang
(  2,  6):wag
(  3,  4):waling
(  3,  5):walng
(  3,  6):walg
(  4,  5):walkng
(  4,  6):walkg
(  5,  6):walkig
```

19. **All anti-substrings no.2** Write a program named `AllAntiSubstrings2` that receives a `String` from the user, and then produces all strings generated from the input by connecting two substrings. The substrings are generated using index triples `(i,j,k)` such that `i < j < k` and connecting `substring( 0, i )` with `substring( j, k )`. Allocate three character spaces to each coordinate value. Here is how the code may work.

```
Enter your input string: Davis
(  0,  1,  2):a
(  0,  1,  3):av
(  0,  1,  4):avi
(  0,  2,  3):v
(  0,  2,  4):vi
(  0,  3,  4):i
(  1,  2,  3):Dv
(  1,  2,  4):Dvi
(  1,  3,  4):Di
(  2,  3,  4):Dai
```

20. **Conversion to decimal** Write a program named `ToDecimal` that receives a binary integer from the user, and then prints its decimal representation. For example, 1111000 in binary is 120 in decimal and 11111011111 in binary is 2015 in decimal. Assume that the binary input is given as a `String` value.