CHAPTER 8

STRINGS STORE SEQUENCES OF CHARACTER CODES

Computers perform many kinds of text processing tasks. Therefore, programming languages must have mechanisms to handle textual information. In Java, the basic mechanism for storing and handling text is an object of type **`String`**. Texts consist of sequences of characters, and a **`String`** object can contain a sequence of character codes.

`String` objects are like variables, but they are different from the variables that we have studied so far. **`String`** objects are actually instances of a class named **`String`**, and they can be manipulated with special string methods. In this chapter, we shall study the nature of **`String`** objects and string methods. Because methods and classes will be the subjects of the following chapters, it is possible that the last part of this chapter (string methods etc.) will be too difficult for you. So my suggestion is that, please, study the first three sections of this chapter, and, if the rest of the sections turn out to be too difficult, return to those sections after chapters 9 and 10. These "difficult" sections were included in this chapter because I wanted to have all string-related stuff presented in a single chapter.

8.1 "Variables" of type String

So far, we have studied programs that handle mostly numerical information. In addition to numerical information, programs often handle textual information. Textual information consists of sequences of letters, punctuation characters, and other special characters. Computers process textual information in the form of character codes (see the table on page 594). Every character that can be input from the keyboard has a unique character code. Uppercase and lowercase letters have different codes. Although character codes are nothing but binary numbers when they are stored in a computer's memory, programming languages have special features that enable them to handle character information in a different way from that of pure numerical information.

In Java, textual information is stored in strings which resemble both variables and arrays. A string can store a sequence of character codes that represent a text. Strings can be declared in the following way

```
String  name_from_keyboard ;
String  file_name ;
```

These declarations clearly resemble variable declarations. `String` is the name of a standard Java class. It is not a reserved keyword like the keywords `int`, `long`, `double`, etc. By studying program **Fullname.java**, you can find out that, like variables, strings can be assigned values and strings can be printed to the screen.

Although strings in some ways resemble variables, they are, however, in some ways different from the traditional variables. A string declaration like

```
String  first_name ;
```

specifies a reference, a name that can reference or refer to a `String` object. A `String` object can be created, for example, by invoking (calling) the method `nextLine()` for the keyboard object. For example, the statement

```
first_name  =  keyboard.nextLine() ;
```

creates a `String` object of those characters that are read by method `nextLine()`, and makes the name `first_name` reference the created object. A string name like `first_name` references a `String` object so that the address of the object is stored in the memory that is reserved in the declaration of the string. Figure 8-1 clarifies how string declarations and string object creations consume memory.

When a variable stores a numerical value, it stores the value as it is. A variable does not refer to a value stored elsewhere in the main memory. A string is different. A string references a `String` value (object) that is located in a different part of the main memory, the heap memory. For this reason, string can be said to be a reference type. Basic variable types such as `int`, `long`, `double`, etc., can be called value types to distinguish them from `String` and other reference types.

Here, two "variables" of type **String** are declared. These "variables" are not like the traditional variables although these declarations resemble the declarations of the variables of the basic types **int**, **long**, **short**, **double**, etc.

```
//  Fullname.java  (c)  Kari Laitinen

import  java.util.* ;

class Fullname
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      String  first_name ;
      String  last_name ;

      System.out.print( "\n Please, type in your first name: " ) ;
      first_name  =  keyboard.nextLine() ;
      System.out.print( "\n Please, type in your last name:  " ) ;
      last_name   =  keyboard.nextLine() ;

      System.out.print( "\n Your full name is "
                     +  first_name  +  " "  +  last_name  +  ".\n" ) ;
   }
}
```

Here we read values for the string "variables" from the keyboard by calling method **nextLine()** for the keyboard object. **nextLine()** reads a line of characters and constructs a **String** object of those characters. The **String** objects that are returned by the **nextLine()** method are assigned as values for the string "variables".

Strings can be joined with the string concatenation operator +. The **String** values (objects) which the names **first_name** and **last_name** reference are printed among the string literals that are given between double quotes.

Fullname.java - 1. The input/output of strings.

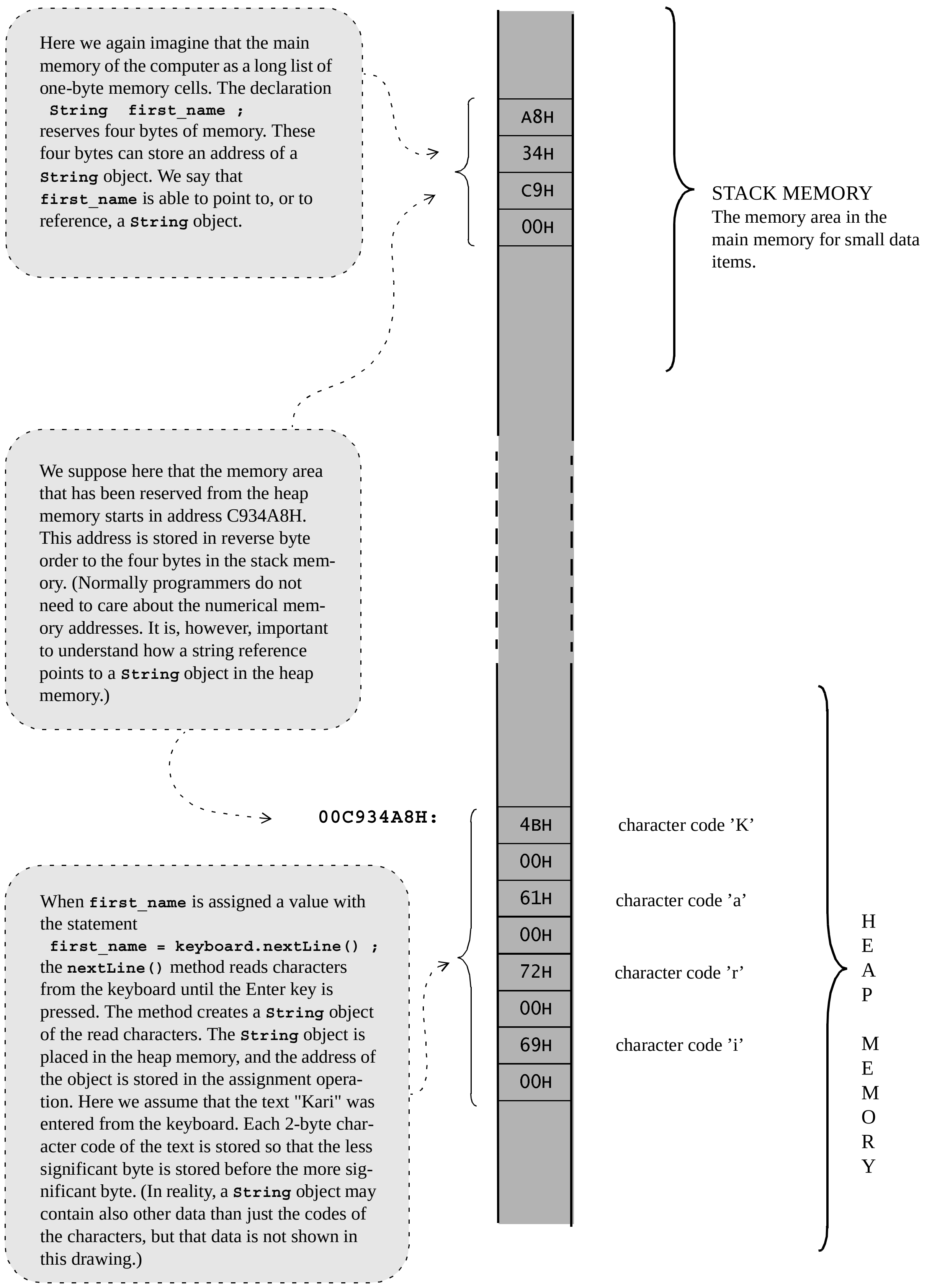
```
D:\javafiles2>java Fullname

 Please, type in your first name: Kari

 Please, type in your last name:  Laitinen

 Your full name is Kari Laitinen.
```

Fullname.java - X. A space is inserted between the first name and the last name in the output.



Here we again imagine that the main memory of the computer as a long list of one-byte memory cells. The declaration
 `String  first_name ;`
reserves four bytes of memory. These four bytes can store an address of a `String` object. We say that `first_name` is able to point to, or to reference, a `String` object.

The memory area in the main memory for small data items.

We suppose here that the memory area that has been reserved from the heap memory starts in address C934A8H. This address is stored in reverse byte order to the four bytes in the stack memory. (Normally programmers do not need to care about the numerical memory addresses. It is, however, important to understand how a string reference points to a `String` object in the heap memory.)

When `first_name` is assigned a value with the statement
 `first_name = keyboard.nextLine() ;`
the `nextLine()` method reads characters from the keyboard until the Enter key is pressed. The method creates a `String` object of the read characters. The `String` object is placed in the heap memory, and the address of the object is stored in the assignment operation. Here we assume that the text "Kari" was entered from the keyboard. Each 2-byte character code of the text is stored so that the less significant byte is stored before the more significant byte. (In reality, a `String` object may contain also other data than just the codes of the characters, but that data is not shown in this drawing.)

Figure 8-1. The text "Kari" stored in a String object.

8.2 String literals

Although it has not been explicitly mentioned, we have actually been using string information throughout this book. In most output statements there are constant texts inside double quotes. These constant texts are called string literals. They may also be called string constants. For example, the following statement outputs a string literal to the screen.

```
System.out.print( "I am a string literal." ) ;
```

String literals can be used to create `String` objects by assigning a string literal as a value to a declared string. For example, when a string like

```
String  some_string  ;
```

is declared, it can be assigned a value with a statement like

```
some_string  =  "I am a text inside a String object." ;
```

The above statement creates a `String` object of the characters that are given between the double quotes, and makes `some_string` reference (point to) the created `String` object. If necessary, the two statements above can be combined into the following single statement that both declares the string and assigns a value to it

```
String  some_string  = "I am a text inside a String object.";
```

A string literal is a data item of type `String`, and stored somewhere in the computer's main memory. String literals cannot be modified. They belong to the same category as integer, floating-point, or character literals. Integer literals are plain numbers (e.g. 4 or 1909). Floating-point literals are numbers with a decimal point (e.g. 25.106). Character literals are characters inside single quotes (e.g. 'A', 'n', or '+'). It is important to understand the difference between single-quoted character literals and double-quoted string literals. For example,

'A'	is a literal of type `char` and it means the character code of letter A (i.e. the numerical value 65 or 41H), but
"A"	means a string inside which the character code of letter A is stored.

The rule for writing string literals is that you include the characters of the string literal inside double quotes. A problem arises when you want to include the double quote character itself in a string literal. In Java, this problem has been solved so that, if you want to include a double quote character in a string literal, you simply have to add one backslash character \ before the double quote. A backslash preceding a double quote means that the double quote character is not the terminator of that string literal. A backslash in a string literal generally means that the character following the backslash will be interpreted in a special way. For example, the output statement

```
System.out.print( "\"C:\\TEMP\" is a directory. " ) ;
```

would produce the text

```
"C:\TEMP" is a directory.
```

on the screen. You can note that \" means a double quote and \\ means a backslash in the above string literal. Previously we have learned that \n means a newline in texts to be printed. Other characters that need to be printed with the help of a backslash are backspace \b, carriage return \r, the tabulator character \t, and single quote \'. In addition to string literals, the backslash sequences are applied in character literals. For example, '\n' means the character code for a newline and '\'' is the character code for a single quote. The characters that are written with a backslash are called escape sequence characters. The backslash character \ is an escape character with which we can escape from the general rules for writing string literals and character literals.

8.3 Accessing individual characters of a string

An object of type `String` contains zero or more character codes that represent a text. The character codes can be codes of letters (A, B, C, D, ..., a, b, c, d,...), codes of numerical digits (0, 1, 2, 3, ...), codes of special and punctuation characters (*, [, }, -, +, ., \, /, ...), or codes of "invisible" characters such as newlines (\n) or tabulators (\t). The character codes stored in a `String` object are arranged in such an order that the position of each character (e.g. the first character and the last character) can be identified. Actually, there is an array of character codes inside every `String` object. The individual characters stored in the array can be accessed with a special method named `charAt()`.

Programs **Widename.java** and **StringReverse.java** are examples that demonstrate how individual characters of `String` objects can be read and printed to the screen. By studying these programs you can see that a string can be processed in a loop in the same way as an array. A single character of a string can be read by using (calling) the `charAt()` method. An index expression is written between the parentheses when `charAt()` is called. Valid index values start counting from zero, and the last valid index value is one less than the length of the string.

For `String` objects there exists a method named `length()` that tells what is the length of the string in question, i.e., how many characters are stored in the string. The `length()` method can be called (invoked) for a `String` object by using the dot operator . in the following way

```
string_name.length()
```

Also the "invisible" characters like newlines and tabulators are counted as characters when the length of a string is determined. For example, the statements

```
String  short_text_lines  =  "\n aaa \n bbb \n ccc " ;

System.out.print( short_text_lines.length() ) ;
```

would print 18 to the screen because the string `short_text_lines` is made of 3 newlines, 6 space characters, and 9 letter characters. The newline character \n is a single character although we have to write it with two separate character symbols in our programs.

An empty string is a string that has been created, but that does not contain any characters. The length of an empty string is zero. You cannot access any characters of an empty string because all index values are illegal. The following is an example of the creation of an empty string

```
String  some_empty_string  =  "" ;
```

In general, the following facts apply to every string provided that the string is not empty:

- `string_name.charAt( 0 )` refers to the first character of the string.
- `string_name.charAt( string_name.length() - 1 )` refers to the last character of the string.
- `string_name.charAt(` *any valid index expression* `)` refers to a data item of type `char`.

A `String` object is an immutable entity once it has been created. This means that it is possible to read the characters of a `String` object, but it is not possible to modify them. Later on in this chapter we shall study `StringBuilder` objects which are mutable strings.

Method **nextLine()** reads a string from the keyboard. The program waits here until the user types in something and presses the Enter key. The newline character \n that represents the Enter key is not included in the read character string. After this statement has been executed, the input from the keyboard is stored in the **String** object referenced by **name_from_keyboard**.

In this **while** loop, the string **name_from_keyboard** is processed from the beginning to the end. The **while** loop stops when **character_index** reaches a value that is the length of the string. Method **length()** is used here to find out how many characters the **String** object contains.

```
//  Widename.java  (c) Kari Laitinen

import java.util.* ;

class Widename
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      String  name_from_keyboard ;
      int     character_index  =  0 ;

      System.out.print( "\n Please, type in your name: " ) ;
      name_from_keyboard  =  keyboard.nextLine() ;

      System.out.print( "\n Here is your name in a wider form: \n\n  " ) ;

      while ( character_index  <  name_from_keyboard.length() )
      {
         System.out.print(
             " "  +   name_from_keyboard.charAt( character_index ) ) ;
         character_index  ++  ;
      }
   }
}
```

To achieve widely spaced printing, this output statement prints one space character before each character from **name_from_keyboard**. Method **charAt()** returns a single character from the string. The value of **character_index** determines which character is currently being printed.

Widename.java - 1. Referring to individual characters of a string.

```
D:\javafiles2>java Widename

 Please, type in your name: Charles Babbage

 Here is your name in a wider form:

   C h a r l e s   B a b b a g e
```

Charles Babbage was a man who built mechanical computing machines more than 150 years a ago.

Widename.java - X. Here "wide" printing means spaces between characters.

Exercises related to strings

Exercise 8-1. Program **Widename.java** shows how a string can be printed in wide form, with a space between the characters of the string. Program **StringReverse.java** shows how the characters of a string can be printed in reverse order. Your task is to now write a program that does both these activities. The program should ask for a string from the keyboard, and print the characters of the string both in wide form and in reverse order. If string "Hello!" were entered from the keyboard, your program should print

```
! o l l e H
```

Exercise 8-2. Write a program that asks for a string from the keyboard, and explores each character in the given string and counts how many uppercase letters, lowercase letters, numbers, and other characters there are in the given string. By writing `keyboard.nextLine()` you can read the string from the keyboard. You can use normal integer variables to count different types of characters. When you use the `charAt()` method, you need an index variable (e.g. `character_index`). Studying program **Iffing.java** in Chapter 6 may help you in this exercise.

Exercise 8-3. Write a program that reads a string from the keyboard, and prints the character codes of the string in hexadecimal form. If string "Hello!" were entered from the keyboard, the program should print

```
48 65 6C 6C 6F 21
```

where 48 is the hexadecimal code for the uppercase letter H, 65 the hexadecimal code for the lowercase letter e, etc. When the `charAt()` method is called, it returns a value of type `char`. One possibility to print a value of type `char` in hexadecimal form is to first convert the `char` value to an `int` value, and then print the `int` value in hexadecimal form with the `printf()` method. For example, if `some_character` is a variable of type `char`, it can be printed in hexadecimal form with the statement

```
System.out.printf(  " %X",  (int) some_character ) ;
```

Exercise 8-4. Write a program that inputs a string from the keyboard, and prints the string in uppercase form. For example, if the string "Steven Jobs" were typed in from the keyboard, your program should print

```
STEVEN JOBS
```

Your program must find all lowercase letters in the given string and convert them to uppercase. A lowercase letter can be converted to uppercase by subtracting 20H from the lowercase character code. The following expression is true when `character_in_string` contains a lowercase letter

```
 ( character_in_string  >=  'a'  &&
   character_in_string  <=  'z' )
```

Another possibility to create this program is to use the string method `toUpperCase()`.

Exercise 8-5. Write a program that reads a string from the keyboard and checks whether the given string is a palindrome, a string that is the same were it read from left to right or from right to left. The following strings are examples of palindromes

```
aabbbccbbbaa
*xx12zzz21xx*
saippuakauppias
```

One possibility to make this program is to read the characters of the string with two indexes, one index incrementing from zero and the other index decrementing from the last character. The program should check whether corresponding letters at each end of the string are the same. Another possibility is to make a reversed copy from the original string, and use either the string method `compareTo()` or the string method `equals()` to check whether the strings are equal. Class `StringBuilder`, which will be studied later in this chapter, provides a `reverse()` method to reverse strings.