CHAPTER 9: STRINGS

9.1 Declaration and Initialization

Strings are sequences of characters but in Java strings are implemented as objects.

There are **two** ways of creating a string in Java:

1. Using string literal - A new string object is created by using double quotation marks like

```
String str1 = “Hello world!” ;
String empty_str = “”;  #this is an empty
string
```

Using a string literal to create a string is more memory efficient for Java since it checks the string constant pool after a string object is created. If the string is already present in the string constant pool, then only the reference to the string is returned and if the string does not exist only then a new string object is created. You can consider the string constant pool as a special memory location.

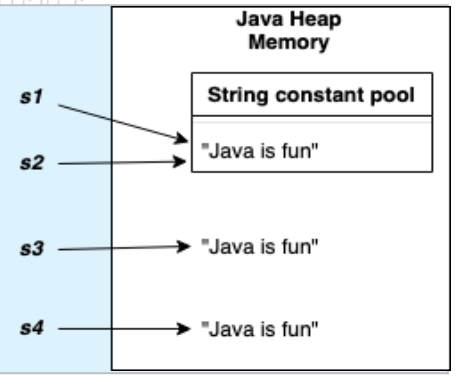
2. Using the new keyword, i.e., using the String class constructor - A new string object is created by using the String class constructors. There are 13 constructors in the String class which can be used to create a string object. The most common ones are

```
String str2 = new String(“Hello world!”);
String empty_str = new String();  #this is an empty string
char [] character_array = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’,
‘!’};
String str3 = new String(character_array);
```

Using the new keyword creates a new String object each time in the heap memory.

Let’s write these 4 lines of code and see how the string objects are stored in the diagram below.

```
String s1 = “Java is fun”;
String s2 = “Java is fun”;
String s3 = new String(“Java is fun”);
String s4 = new String(“Java is fun”);
```



When s1 is created, there is no String literal “Java is fun” in the string constant pool and so it is added there. When s2 is created, the string “Java is fun” is already present in the pool and hence its reference is only returned to s2, and that is why s1 and s2 both point at the same String object. But s3 and s4 are created using the String constructor and so both created new String objects which reside in the heap and both have separate locations.

9.2 String Input

We need to import the Scanner class in order to take a string input from the user. Let’s see an example.

```
import java.util.Scanner;
class Main {
        public static void main(String[] args) {
                Scanner sc = new
Scanner(System.in);
                String user_input = sc.nextLine();
                System.out.println(user_input);

        }
}
```

Notice that we used sc.nextLine() to take a string input because nextLine() takes the input until a new line is encountered i.e. a \n is encountered.

9.3 String Escape Sequence

If we want to create strings which would incorporate single or double quotation marks then we need an escape sequence to create such strings i.e. \. Let's see an example:

```
String s = "I love \'Java\' very
much";
System.out.println(s);
```

The output of the above code is "I love 'Java' very much. Try the above code by replacing the single quotations with double quotes.

9.4 String Length

We can find the size of a string i.e. we can find how many characters are present in a string using the **length()** method like:


```
String s = "I love Java";
System.out.println(s.length())
;
```

The output of the above code is 11 since there are 11 characters in str5 including the spaces.

9.5 String Indexing

Each character in a string can be accessed using its position in the string. These positions are called index numbers and are numbered from 0 to length of the string -1.

String s = "Programming is fun";

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	P	r	o	g	r	a	m	m	i	n	g		i	s		f	u	n

It can be seen that the first element can be found in index=0 and the last element can be found in index= length-1.

In order to access a particular element from the string, we can use the method **charAt().** This method takes an index number as an argument and returns the particular element in that position. Let's extract the characters 'P' at index=0, 'n' at index=17 and 'g' at index 10.

```
String s = “Programming is fun”;
char first_character = s.charAt(0);
char last_character = s.charAt(s.length() -
1);
char random= s.charAt(10);
System.out.println(first_character);
System.out.println(last_character);
System.out.println(random);
```

9.6 Character Array

We know that strings are a sequence of characters and each character has an index number defining its position in the string. A character in Java is represented using single quotation marks. A string can be converted into an array of its characters using the **toCharArray()** method like :

```
String s = “This is a string” ;
char [] char_array =
s.toCharArray();
System.out.println(char_array);
```

The above code will convert the string s to a character array.

9.7 Mutability of Strings

In the previous section 9.3, we saw how we can access the elements in a string. However, we cannot modify a string because in Java, strings are ***immutable***. This means that we cannot modify the elements which make up the string. If we want to modify a string then a new string needs to be created which would contain the modifications, leaving the original string unchanged.

9.8 String Concatenation

We can connect multiple strings together by using the + operator between them. This process is called concatenation where a new string is created by chaining one/more strings together.

```
String s1 = “I love Java ” ;
String s2 = “programming ” ;
String s3 = s1 + s2 + “very
much”;
System.out.println(s3);
```

The output of the above code is: *I love Java programming very much*. The output is a new string which is stored in s3. Let’s see another example:

```
String s1 = “CSE”;
int num = 110;
String output = s1 + num
System.out.println(output)
;
```

Here, notice that we are concatenating an integer to a string using the + operator. In this case the output is a string *CSE110*. This means that, wherever an operand of the + operator will be a string, the Java compiler will automatically convert the other operand to a string representation and then contact it to create a new string.

Let’s see another example:

```
String semester = “Fall ” ;
int y1 = 20;
String output = semester + y1 +
23;
System.out.println(output);
```

Here, the output is: *Fall2023*. Note how operator precedence causes the concatenation of the string “Fall” to the integer 20 at first which creates the string “Fall20” and then this string is concatenated to the integer 23 which creates the output string “Fall2023”.

Had we written: **String output = y1 + 23 + semester**, then the operator precedence would cause the integer in y1 and the integer 23 to be added using the + operator to produce an integer 43 and then this integer would be concatenated to the string “Fall” to produce a string “43Fall”. Try this out yourself.

9.9 Comparing Strings

- **equals()** - This method checks whether the contents of two strings are identical or not. It returns true if two strings contain the same elements and false otherwise. Let’s see an example:

```
String s1 = “Let us code”;
String s2 = “Java”;
String s3 = “Let us code”;
System.out.println(s1.equals(s2))
;
System.out.println(s1.equals(s3))
;
```

Here, the output of the first print statement is ***false*** since the elements of s1 and s2 are identical and the output of the second print statement is ***true*** since the elements of s1 and s3 are identical.

Note: equals() method is case sensitive and so the strings “Let us code” and “Let us Code” will yield false. However, another method **equalsIgnoreCase()** will ignore the case differences and will yield true.

- **== operator** - The == operator in Java checks whether two string objects have the same reference/location or not. Let us see an example:

```
String s1 = new String(“Let us
code”);
String s2 = new String(“Let us
code”);
System.out.println(s1.equals(s2));
System.out.println(s1 == s2);
```

Here, the output of the first print statement is ***true*** since the elements of s1 and s2 are identical and the output of the second print statement is ***false*** since s1 and s2 refer to different string objects stored in different memory locations (Refer to the heap memory where string objects are stored when created using the new keyword).

- **compareTo()** - This method compares two strings lexicographically. It returns 0 if both strings are equal with identical characters in the same positions. It returns a positive integer if the first string is lexicographically greater than the second string and it returns a negative integer if the first string is lexicographically smaller than the second string. Let’s see a few examples.

```
String s1 = “Java”;
String s2 = “Java”;
String s3 = “Jade”;
String s4 = “Lava”;
System.out.println(s1.compareTo(s2))
;
System.out.println(s1.compareTo(s3))
;
System.out.println(s1.compareTo(s4))
;
```

The first output is 0 since both s1 and s2 are identical. The second output is 18 since “v” is 18 more than “d” lexicographically. The third output is -2 since “J” is 2 less than “L” lexicographically.

9.10 Searching a Character in a String

Sometimes we need to find the index number of a character in a string. There are 2 methods which help us to do this:

- **indexOf()** - This method takes a character/substring as an argument and finds the position of the first occurrence of the character/substring in the string.
- **lastIndexOf()** - This method takes a character/substring as an argument and finds the position of the last occurrence of the character/substring in the string.

Let us see an example:

```
String s1 = “random string”;
System.out.println(s1.indexOf(‘n’));
System.out.println(s1.lastIndexOf(‘n’))
;
System.out.println(s1.indexOf(“ring”))
;
```

The output of the first print statement is 2 since the first occurance of the character ‘n’ is in index=2 and the output of the second print statement is 11 since the last occurance of the character ‘n’ is in index=11. The 3rd output is 9 since the substring “ring” is found from index=9.

9.11 String Slicing

The **substring()** method helps us to slice a string in order to create a substring. If we mention only the starting position *start_index*, then it produces a new substring starting at the *start_index* to the end of the string. If we mention both the *start_index* and the *end_index*, then it produces a new substring starting at the *start_index* and ending at the *end_index*. Let us see an example:

```
String s1 = “Java Programming is
fun”;
String sub1 = s1.substring(5);
String sub2 = s1.substring(5,12);
System.out.println(sub1);
System.out.println(sub2);
```

The output of the first print statement is *Programming is fun* and the output of the second print statement is *program*. Notice that the *end_index* is exclusive which means the substring will stop at the index before the *end_index* and not include the character at the *end_index* itself.

9.12 Modifying a String

We know that strings are immutable but we have been using different string methods to produce new strings which contain the modifications. If we want to replace a character or a substring with a different character or substring then we can use the **replace()** method. This method creates a new string which contains the replaced character or the substring. Let's see an example:

```
String s1 = "marathon";
String new_s1 = s1.replace("a", "e");
String new_s2 = s1.replace("mara",
"py");
System.out.println(new_s1);
System.out.println(new_s2);
```

The output of the first print statement is *merethon.* We can see that all the occurrences of "a" have been replaced by "e". The output of the second print statement is *python*. We can see that the substring "mara" has been replaced by the substring "py".

9.13 Trimming and Splitting a String

- **trim()** - This method removes any leading and trailing whitespaces and returns a new string as a result. Let's see an example :

```
String s1 = "   Java  ";
String s2 = s1.trim();
System.out.println(s2);
```

The output of the above code is *Java* without the white spaces before and after the word "Java".

- **split()** - This method converts a string to a string array on the basis of a given parameter as delimiter. Let's see an example :

```
String s1 = "Java is fun";
String [] s1_array = s1.split("
");

String s2 = "Java,is,,fun";
String [] s2_array =
s2.split(",");
```

9.14 Case Conversion

We can convert uppercase alphabetical characters to lowercase and vice versa using two methods.

- **toLowerCase()** - This method converts all uppercase letters to lowercase in a string and returns a new modified string.

```
String s1 = "Java Programming is
FUN";
String lower_s1 = s1.toLowerCase();
System.out.println(lower_s1);
```

The output of the above code is *java programming is fun* where all characters are presented in lowercase.

- **toUpperCase()** - This method converts all lowercase letters to uppercase in a string and returns a new modified string.

```
String s2 = "i love to CODE";
String upper_s2=
s2.toUpperCase();
```

```
System.out.println(upper_s2);
```

The output of the above code is *I LOVE TO CODE* where all characters are presented in uppercase.

9.15 Data Conversion in Strings

- **valueOf()** - This method of String class is used to convert other data types to String type.

```
int digit = 85;
float decimal = 54.643F; //F represents
float
char letter = 'A';

String s1 = String.valueOf(digit);
System.out.println(s1);

String s2 = String.valueOf(decimal);
System.out.println(s2);

String s3 = String.valueOf(letter);
System.out.println(s3);
```

The variable s1 will contain a string representation of 85 ("85"), the variable s2 will contain a string representation of 54.643 ("54.643") and the variable s3 will contain a string representation of 'A' ("A").

9.16 ASCII Values

All keyboard characters have unique Unicode values. Refer to this ASCII code table to see the unicode values corresponding to each character: https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html.

Sometimes we need to work with ASCII values of strings or characters. In that case we need to convert an element in a string into its corresponding Unicode value. We can use the **codePointAt()** method which takes an index of a string as a parameter and returns the Unicode value of the element in the given index.

```
String s1 = "Hello World!";
int unicode_val =
s1.codePointAt(0);
System.out.println(unicode_val);
```

The output of the above code is *72* which is the Unicode value of index=0 element, that is, "H". Try to find the Unicode value of "W" by yourself.

We can also use these Unicode values to convert uppercase letters to lowercase and vice versa. Visit the link above and check the Unicode value of uppercase "A". You'll see that it's 65. Now, check the Unicode value of lowercase "a". You'll see that it's 97. The difference between the two values is 32. That means if we add 32 to the Unicode value of "A" we will get the Unicode value of "a" and if we subtract 32 from the Unicode value of "a" we will get the unique value of "A". If we cast the Unicode value using (char) then we will get the corresponding character. This will follow for all alphabets from "A" to "Z" and from "a" to "z". Let's see an example:

```
String upper_s1 = "CODE";
int unicode_upper =
upper_s1.codePointAt(2);
int unicode_lower = unicode_upper + 32;
char lower_letter= (char)unicode_lower;
System.out.println(lower_letter);
```

The output of the above code is the character *d.* The Unicode at index=2(uppercase 'D') of "CODE" is 68 and if we add 32 to it, we will get 100 which is the Unicode value of lowercase 'd' and hence converting the value to character using char casting gives us the letter 'd'.

9.17 Checking Substring Existence

- **contains()** - This method checks whether a substring is present in a string or not. It returns *true* if the substring exists in the string and *false* otherwise.

```
String s1 = "I love Java programming";
Boolean res1 = s1.contains("Java");
Boolean res2 = s1.contains("love Java
");
Boolean res3 = s1.contains("p");
Boolean res4 = s1.contains("java");
System.out.println(res1);
System.out.println(res2);
System.out.println(res3);
System.out.println(res4);
```

The output of the first three prints are *true* since “Java”, “love Java “ and “p” all these substrings exist in s1 but the output of the last print is *false* since “java” with lowercase “j” does not exist in s1.

9.18 Worksheet

A. Write a Java program that takes a string as input and reverses it.

Sample input 1:
Programming
Sample output 1:
gnimmargorP

B. Write a Java program that takes a string as input and counts the number of vowels (a, e, i, o, u) present in the string.

Sample input 1:
Programming
Sample output 1:
3

C. Write a Java program that takes a string as input and checks if it is a palindrome. A palindrome is a word, phrase, number, or other sequence of characters that reads the same backward as forward. The program should output "Palindrome" if the string is a palindrome and "Not a Palindrome" otherwise.

Sample input 1:
ABCDCBA
Sample output 1:
Palindrome
Sample input 2:
Nepal
Sample output 2:
Not a Palindrome

D. Write a program which takes a string as input and returns the number of characters in the string.

Sample input 1:
Programming
Sample output 1:
11

E. Input a word into a String. Print each character on a line by itself.

Sample input 1:
Program
Sample output 1:
P
r
o
g
r
a
m

F. Your task is to input a word into a String. Then print code for each character in the String using the 2nd method discussed above.

Sample input 1:
Pro
Sample output 1:
P : 80
r : 114
o : 111

G. Print the statistics of occurrence of each character on a line by itself. Assume that user will only give CAPITAL letters. So, you will have to count values of CAPITAL letters only.

Sample input 1:
BALL

Sample output 1:
A which is 65 was found 1 time(s)
B which is 66 was found 1 time(s)
L which is 76 was found 2 time(s)

H. Input a word into a String.
Print the word.
Print the word again after adding "==THE END==" at the end of the word.
Then print the word again.
Your whole program may contain the word "String" at most two times.
Sample input 1:
Programming
Sample output 1:
Programming
Programming==THE END==
Programming

I. Answer the following theoretical questions:

- What is string immutability in Java? Explain why strings are immutable and discuss the advantages and disadvantages of immutability.
- Discuss the significance of the "equals()" method versus the "==" operator when comparing strings in Java.
- What is the purpose of the "substring()" method in Java? Explain how it can be used to extract substrings from a larger string.
- Describe the process of converting a string to a numeric value in Java. Explain the use of the "parseInt()" and "valueOf()" methods for this purpose.
- What are some common methods available in the String class in Java? Provide examples of how these methods can be used.