Day5: String, StringBuilder, StringBuffer, Scanner

String class:

In Java, a string is an object that represents a sequence of characters. there is a predefined class in java called **String**, which belongs to **java.lang** package, this String class is used to create a string object.

We use **double quotes** to represent a string object in Java, for example

```
String message = "Welcome to Masai";
```

Here the **message** is a variable of String type which is initialized with the string object

“Welcome to Masai”. this message variable is not a primitive type variable like int, char, byte, etc.

Instead, this message variable is a reference variable of type String class.

There are two ways to create a String object in Java:

1. By string literal

   String s = “Welcome”;

2. By new keyword

   String s = new String(”Welcome”);

   .

1. By string literal:

Java String literal is created by using double-quotes. For Example:

String s="Welcome";

In Java, the JVM maintains a **string pool** to store all of its strings inside the memory. The string pool helps in reusing the strings.

Here, we are directly providing the value “Welcome”. Hence, the compiler first checks the string pool to see if this string already exists.

- **If this string already exists**, the new string is not created. Instead, the new reference points to the already existing string “Welcome”.
- **If the string doesn't exist**, the new string “Welcome” is created inside this string pool area.

Example:

```
String s1 = "Welcome";
String s2 = "Welcome";

if(s1 == s2)
  System.out.println("same");
else
  System.out.println("not same");


output:
  same
```

1. By new keyword:

```
String s1 = new String("Welcome");
```

Here the value of the string is not directly provided, hence a new “Welcome” string is created inside the heap memory even though the “Welcome” string object is already present inside the string pool area.

Example:

```
String s1 = new String("Welcome");
String s2 = new String("Welcome");


if(s1 == s2)
  System.out.println("same");
else
  System.out.println("not same");


output:
  not same
```

String class provides various constructors to create a new string object:

some of them are:

1. **String(byte[] bytes)** – Construct a new String by decoding the *byte array*

   example :

   .

   ```
   byte[] bytes = {100,101,102};
   String str =new String(bytes);


   output:
   def
   ```

2. **String(char[] chars)** – Allocates a new String from the given *Character array*

   example:

```
char chars[] = {'M', 'a', 's', 'a', 'i'};
String s = new String(chars);

output:
Masai
```

3. **String(StringBuffer sbuffer)** – Allocates a new string from the string in StringBuffer

   example:

```
StringBuffer sbuffer = new StringBuffer("Masai");
String s = new String(sbuffer); //Masai
```

4. **String(StringBuilder sbuilder)** – Allocates a new string from the string in StringBuilder

   example:

```
StringBuilder sbuilder = new StringBuilder("Masai");
String s = new String(sbuilder); //Masai
```

Java String Operations:

The Java language provides special support for the string concatenation operator (+), and for the conversion of other objects to strings.

If one operand of this (+) operator is a string then the entire expression with result a string object

Example1:

```
    String name = "Java ";
    System.out.println("Student Name is : " + name);
```

Example:

```
System.out.println("Hello" + 10 + 20 + "Welcome");

output: Hello1020Welcome
```

Java String class provides various methods to perform different operations on strings.

some of them are:

1. **int length():** Returns the number of characters in the String
2. **Char charAt(int i):** Returns the character at ith index.
3. **String substring (int i):** Return the substring from the ith index character to end
4. **String substring (int i, int j):** Returns the substring from i to j-1 index
5. **String concat(String str):** Concatenates specified string to the end of this string.
6. **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

7. **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.
8. **Int lastIndexOf(String s):** Returns the index within the string of the last occurrence of the specified string.
9. **boolean equals(Object otherObj):** Compares this string to the specified object.
10. **boolean equalsIgnoreCase(String anotherString):** Compares string to another string, ignoring case considerations.
11. **int compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.
12. **String toUpperCase():** Converts all the characters in the String to upper case.
13. **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
14. **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*
15. char[] toCharArray(): return the char array from an string object.

Java Strings are Immutable

In Java, strings are **immutable**. This means, once we create a string, we cannot change that string.

example1:

```
String message = "Welcome";
message.concat(" user");
System.out.println(message);

output:
Welcome
```

Reason: Here, we are using the concat() method to add another string to the previous string. since string objects are immutable, if we call any method to do any modification in the string object, that method will return a new string object with modified content.

example2

```
String message = "Welcome";
String newMessage = message.concat(" user");
System.out.println(message);
System.out.println(newMessage);

output:
Welcome
Welcome user
```

Why String objects are immutable in Java?

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Masai". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Note: If we want to create a mutable (modifiable) String object, we should use either StringBuffer or StringBuilder classes. both classes belong to java.lang package also.

StringBuilder

```
public final classStringBuilder
extendsObject
implementsSerializable,CharSequence
```

A mutable sequence of characters. This class provides an API compatible with StringBuffer

, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer

in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer

as it will be faster under most implementations.

The principal operations on a `StringBuilder` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string builder. The `append` method always adds these characters at the end of the builder; the `insert` method adds the characters at a specified point.

```
StringBuilder z= new StringBuilder("start");
z.append("le");
```

For example, if `z` refers to a string builder object whose current contents are "`start`", then the method call `z.append("le")` would cause the string builder to contain "`startle`".

String vs StringBuilder Comparison Table

The table below summarizes the comparisons between String vs StringBuilder**:**

Parameter	String	StringBuilder
Performance during concatenation	Slow as compared to StringBuilder during frequent updates and concatening.	Fast as compared to String. Like it is faster than String while concatenating many strings together in a loop.
Mutability	The string created using the String class is immutable.	The string created using StringBuilder is mutable.
Memory Usage during concatenation	Requires more memory to create a new instance if operations performed on a string change its value.	It requires less memory as it updates the existing instance.
Nature of Object	An object of String is read-only in nature.	An object of StringBuilder is dynamic in nature.

~~*StringBuffer [To Be Discussed After Multithreading]*~~

A thread-safe, mutable sequence of characters. A string buffer is like a String

, but can be modified. String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

Thread-safe code is code that will work even if many Threads are executing it simultaneously.
More on this after the multithreading lecture.

Difference between StringBuffer and StringBuilder class

1. *Most of the methods of StringBuffer is synchronized i.e. thread-safe. whereas most of the methods of StringBuilder is non-synchronized i.e. not thread-safe. It means two threads can call the methods of StringBuilder simultaneously.*

2. *StringBuffer is less efficient than StringBuilder. whereas StringBuilder is more efficient than StringBuffer.*

These StringBuffer and StringBuilder class provides many methods to perform modification in the same string object. example

1. *public **StringBuilder append**(String anotherString): It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.*
2. *public StringBuilder **reverse**(): is used to reverse the string.*
3. *public StringBuilder replace(int startIndex, int endIndex, String str): It is used to replace the string from specified startIndex and endIndex.*

Example:

```
StringBuilder message = new StringBuilder("Welcome");
message.append(" to Masai");

System.out.println(message);

output:
Welcome to Masai;
```

Java String compare:

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1.**By Using equals() Method**

The equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring the case.

2. **By Using == operator**

   The == operator compares references not values.

   example

```
String s1="Hello";
String s2="Hello";
String s3=new String(Hello");
```

```
System.out.println(s1==s2);//true (because both refer to same instance)
System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```

3. **String compare by compareTo() method**

   Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

   Suppose s1 and s2 are two String objects. If:

   - **s1 == s2** : The method returns 0.
   - **s1 > s2** : The method returns a positive value.
   - **s1 < s2** : The method returns a negative value.

   example:

```
String s1="Sachin";
String s2="Sachin";
String s3="Ratan";
System.out.println(s1.compareTo(s2));//0
System.out.println(s1.compareTo(s3));//1(because s1>s3)
System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
```

Scanner class in Java:

The Scanner class is used to read the input data from different sources like input streams, users, files, etc. This Scanner class belongs to **java.util** package.

Example:

```
Scanner input = new Scanner(System.in);
```

The `System.in` parameter is used to take input from the standard input. It works just like taking inputs from the keyboard.

To use the Scanner class inside our application, we need to import this class.

```
import java.util.Scanner;
```

Java Scanner Methods to Take Input

The `Scanner` class provides various methods that allow us to read inputs of different types.

`nextInt()` reads an `int` value from the user.

`nextFloat()` reads a `float` value form the user.

`nextBoolean()` reads a `boolean` value from the user

`nextLine()` reads a line of text from the user

`next()` reads a word from the user

`nextLine()` reads a whole line

`nextByte()` reads a `byte` value from the user

`nextDouble()` reads a `doubl`e value from the user

`nextShort()` reads a `short` value from the user.

`nextLong()` reads a `long` value from the user

Example 1: Read a Line of Text Using Scanner

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter your name: ");

    // takes input from the keyboard
    String name = scanner.nextLine();

    // prints the name
    System.out.println("My name is " + name);


  }
}
```

Reading an integer from the user input:

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates a Scanner object
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter number 1: ")
    int num1 = scanner.nextInt();

    System.out.println("Enter number 2: ")
    int num2 = scanner.nextInt();

    System.out.println("Result is : " + (num1+num2));

  }
}
```

Difference between next and nextLine method:

example next() method:

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {

    // creates an object of Scanner
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter your name: ");

    // reads the entire word
    String value = scanner.next();
    System.out.println("Using next(): " + value);

    scanner.close();
  }
}

Output:
Enter your name: Ram Kumar
Using next(): Ram
```

In the above example, we have used the `next()` method to read a string from the user.

Here, we have provided the full name. However, the `next()` method only reads the first name.

This is because the `next()` method reads input up to the **whitespace** character. Once the **whitespace** is encountered, it returns the string (excluding the whitespace).

example nextLine() method:

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
```

```
    // creates an object of Scanner
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter your name: ");

    // reads the entire word
    String value = scanner.nextLine();
    System.out.println("Using next(): " + value);

    scanner.close();
  }
}

Output:
Enter your name: Ram Kumar
Using nextLine(): Ram Kumar
```

Unlike `next()` the `nextLine()` method reads the entire line of input including spaces. The method is terminated when it encounters a next line character, `\n`

.

References:

https://www.javatpoint.com/

https://www.geeksforgeeks.org/

https://www.programiz.com/java-programming/