Strings in JAVA

-> string is basically an object that represents sequence of char values.

-> strings are created and manipulated through the string class.

-> Once created, a string is immutable -- its value cannot be changed.

-> A class is a user-defined template for creating an object.

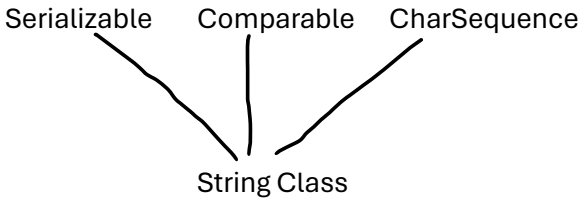
-> A string class is a user-defined template for creating and manipulating string objects, which are sequences of characters.

-> A string acts the same as an array of characters in Java.

Example:

```
char[] ch={'a','b','d','u','l','m','a','j','e','e','t','h'};
String s=new String(ch);
will be same
String name = "abdulmajeeth";
```

-> java.lang.String class implements Serializable, Comparable and CharSequence interfaces.



CharSequence Interface:

-> It's used to represent the sequence of characters.

-> String, StringBuffer and StringBuilder classes implement it.

-> It means, we can create strings in Java by using these three classes.

1.String:

-> String is an immutable class which means a constant and cannot be changed once created.

-> If wish to change, we need to create a new object

-> Even the functionality it provides like toupper, tolower, etc all these return a new object , its not modify the original object.

-> It is automatically thread safe.

Ways of Creating a String:

String literal:

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Example:

String str= "abdul";

Using new keyword:

-> JVM will create a new string object in normal (non-pool) heap memory and the literal “Welcome” will be placed in the string constant pool.

-> The object variable will refer to the object in the heap (non-pool)

Example:

String str= new String("abdul")

2.StringBuffer:

-> StringBuffer is a peer class of String, it is mutable in nature and it is thread safe class.

-> we can use it when we have multi threaded environment and shared object of string buffer i.e, used by mutiple thread.

-> As it is thread safe so there is extra overhead, so it is mainly used for multithreaded program.

Syntax:

StringBuffer demoString = new StringBuffer("abdulmajeeth");

Example:

```
public class StringBufferExample {
  public static void main(String[] args) {
    // Create a shared StringBuffer object
    StringBuffer sharedBuffer = new StringBuffer();

    // Create and start multiple threads
    Thread thread1 = new Thread(new StringAppendingTask(sharedBuffer, "Thread 1"));
    Thread thread2 = new Thread(new StringAppendingTask(sharedBuffer, "Thread 2"));

    thread1.start();
    thread2.start();

    // Wait for threads to complete
    try {
      thread1.join();
      thread2.join();
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print the final content of the StringBuffer
        System.out.println("Final content of StringBuffer: " + sharedBuffer.toString());
    }

    // Runnable task for appending strings to the shared StringBuffer
    static class StringAppendingTask implements Runnable {
        private StringBuffer sharedBuffer;
        private String threadName;

        public StringAppendingTask(StringBuffer sharedBuffer, String threadName) {
            this.sharedBuffer = sharedBuffer;
            this.threadName = threadName;
        }

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                sharedBuffer.append("Hello from " + threadName + "! ");
                try {
                    Thread.sleep(100); // Simulate some processing time
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Output:

Final content of StringBuffer: Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2!

Explanation:

1.We create a shared StringBuffer object named sharedBuffer.

2.We define a StringAppendingTask class that implements Runnable. Each instance of this class appends strings to the shared StringBuffer object in a loop.

3.We create two threads (thread1 and thread2), each running an instance of the StringAppendingTask class.

4.Both threads start simultaneously and append strings to the shared StringBuffer.

5.We wait for both threads to complete their tasks using join() to ensure the main thread waits for them to finish.

Finally, we print the content of the StringBuffer to see the final result.

3. StringBuilder:

-> It creates a mutable sequence of characters and it is not thread safe.

-> It is used only within the thread , so there is no extra overhead

-> so it is mainly used for single threaded program.

Syntax:

StringBuilder demoString = new StringBuilder("abdulmajeeth");

Example:

```
public class StringBuilderExample {
  public static void main(String[] args) {
    // Create a shared StringBuilder object
    StringBuilder sharedBuilder = new StringBuilder();

    // Create and start multiple threads
    Thread thread1 = new Thread(new StringAppendingTask(sharedBuilder, "Thread 1"));
    Thread thread2 = new Thread(new StringAppendingTask(sharedBuilder, "Thread 2"));

    thread1.start();
    thread2.start();

    // Wait for threads to complete
    try {
      thread1.join();
      thread2.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    // Print the final content of the StringBuilder
    System.out.println("Final content of StringBuilder: " + sharedBuilder.toString());
  }
```

```
    // Runnable task for appending strings to the shared StringBuilder
    static class StringAppendingTask implements Runnable {
        private StringBuilder sharedBuilder;
        private String threadName;

        public StringAppendingTask(StringBuilder sharedBuilder, String threadName) {
            this.sharedBuilder = sharedBuilder;
            this.threadName = threadName;
        }

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                sharedBuilder.append("Hello from " + threadName + "! ");
                try {
                    Thread.sleep(100); // Simulate some processing time
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Output:

Final content of StringBuilder: Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 1! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2! Hello from Thread 2!

Explanation:

1.We're using StringBuilder instead of StringBuffer.

2.We have the same structure as the previous example, with two threads appending strings concurrently to a shared StringBuilder.

3.Since StringBuilder is not synchronized, there is no guarantee that the operations performed by one thread won't interfere with those of another.

4.However, due to the lack of synchronization, the output might also contain inconsistencies or unexpected results. Multiple threads are accessing and modifying the same `StringBuilder` instance concurrently, which can lead to race conditions, data corruption, or other issues.

5. To make the `StringBuilder` thread-safe, you'd typically need to synchronize access to it manually using synchronization blocks or locks. Alternatively, you could use `StringBuffer`, which provides built-in thread safety through synchronization.

Immutable String in Java:

-> In Java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

-> Once a string object is created its data or state can’t be changed but a new string object is created.

Example:

```
import java.io.*;
class GFG {
  public static void main(String[] args)
  {
    String s = "Sachin";
    s.concat(" Tendulkar");
    System.out.println(s);
  }
}
```

Output:

Sachin

Explanation:

-> Here Sachin is not changed but a new object is created with “Sachin Tendulkar”. That is why a string is known as immutable.

-> two objects are created but s reference variable still refers to “Sachin” and not to “Sachin Tendulkar”.

-> But if we explicitly assign it to the reference variable, it will refer to the “Sachin Tendulkar” object.

Example:

```
import java.io.*;
class GFG {
  public static void main(String[] args)
  {
    String name = "Sachin";
    name = name.concat(" Tendulkar");
    System.out.println(name);
  }
}
```

Output:

Sachin Tendulkar

Memory Allotment of String:

-> Whenever a String Object is created as a literal, the object will be created in the String constant pool.

-> String constant pool allows JVM to optimize the initialization of String literal.

String demoString = "Geeks";

-> The string can also be declared using a new operator i.e. dynamically allocated.

-> In case of String are dynamically allocated they are assigned a new memory location in the heap.

-> This string will not be added to the String constant pool.

Example:

String demoString = new String("Geeks");

-> If you want to store this string in the constant pool then you will need to “intern” it.

Example:

String internedString = demoString.intern();

-> It is preferred to use String literals as it allows JVM to optimize memory allocation.

Example:

```
import java.io.*;
import java.lang.*;
class Test {
  public static void main(String[] args)
  {
    // Declare String without using new operator
    String name = "GeeksforGeeks";
    System.out.println("String name = " + name);
    // Declare String using new operator
    String newString = new String("GeeksforGeeks");
    System.out.println("String newString = " + newString);
  }
}
```

Output:

String name = GeeksforGeeks

String newString = GeeksforGeeks

equals() Vs == operator:

-> Essentially, equals() is a method, while == is an operator.

-> The == operator can be used for comparing references (addresses) and the .equals() method can be used to compare content.

-> To put it simply, == checks if the objects point to the same memory location, whereas .equals() compares the values of the objects.

StringTokenizer:

->It is a class in Java used to break a string into tokens, which are smaller strings based on a delimiter

->Delimiters can be specified by the programmer, or the default set of delimiters (which include spaces, tabs, and newlines) can be used.

->Initialization: You create a StringTokenizer object by passing the string to be tokenized as an argument to the constructor. Optionally, you can specify delimiters as well.

->Tokenization: Once the StringTokenizer object is created, you can retrieve tokens one by one using methods like nextToken(), hasMoreTokens(), or countTokens().

->Custom Delimiters: If you specify custom delimiters, the StringTokenizer breaks the string whenever it encounters any of those delimiters.

->Empty Tokens: By default, consecutive delimiters are treated as one delimiter. However, you can specify whether to return empty tokens or not.

Constructor	Description
StringTokenizer(String str)	Default delimiters like newline, space, tab, carriage return, and form feed.
StringTokenizer(String str, String delim)	delim is a set of delimiters that are used to tokenize the given string.
StringTokenizer(String str, String delim, boolean flag)	The first two parameters have the same meaning wherein The flag serves the following purpose.

Example:

```
import java.util.StringTokenizer;


public class StringTokenizerExample {
   public static void main(String[] args) {
      // Example string
      String sentence = "Java is a programming language";


      // Tokenize the string using default delimiters
```

```
        StringTokenizer tokenizer = new StringTokenizer(sentence);

        // Iterate through tokens
        while (tokenizer.hasMoreTokens()) {
            System.out.println(tokenizer.nextToken());
        }

        // Example with custom delimiter
        String data = "apple,banana,orange,grape";
        StringTokenizer tokenizer2 = new StringTokenizer(data, ",");

        // Iterate through tokens
        while (tokenizer2.hasMoreTokens()) {
            System.out.println(tokenizer2.nextToken());
        }

      //Example with custom delimiter with flag
              StringTokenizer st1 = new StringTokenizer(
              "JAVA : Code : String", " :", true);

        while (st1.hasMoreTokens())
            System.out.println(st3.nextToken());

StringTokenizer st2 = new StringTokenizer( "2+3-1*8/4", "+*-/");
while (st2.hasMoreTokens())
            System.out.println(st2.nextToken());



    }
}
```

Output:

```
Tokens from sentence:
Java
is
a
programming
```

language

Tokens from data with custom delimiter (,):
apple
banana
orange
grape

Tokens from 'JAVA : Code : String' with custom delimiter (:) and flag:
JAVA
:
Code
:
String

Tokens from '2+3-1*8/4' with multiple delimiters (+-*/):
2
3
1
8
4

String Conversion:

String str1 = "Abdul";
1.String to Char Array:
char[] str1charArray = str1.toCharArray();
2.Char array to String:
String str2=str1charArray.toString();
3.String to char
Char charValue=str1.charAt(0);
4.String to int
String str = "123";
int intValue = Integer.parseInt(str);