String in Java

String is not a primitive data type in java. Primitive data types are the most basic data type in a language like int, double etc.
String is a class in Java. When we create a string variable in Java we are actually creating the object of Java.Lang.String class. String class contains a number of useful built-in methods to work with.

Example 1: A simple String object

```
1  String obj1 = "abc";
2  String obj2 = "wxyz";
3  int len = obj1.length();
4  System.out.println(len);
5  char ch = obj2.charAt(1);
6  System.out.println(ch);
```

Output:

```
3
x
```

Explanation:

In this example we have created two string objects. We are then calling methods of String class on these objects to get total number of characters and character at position 1 respectively.

Empty and Null Strings

An object with null reference means it is not pointing to any value. A String variable (an object) can hold null reference. A String with null reference do not point to any value. If we try to invoke any method on null reference we'll get a NullPointerException. On the other hand a String initialized with "" (empty double-quotes) is an actual string though it contains no characters. Do not confuse between null String and empty String (""). An empty String will not generate any NullPointerException if a method is invoked on it.

Example 2.1: Empty and null Strings

```
1  String obj = "";
2  String str = null;
3  int len = obj.length();
4  System.out.println(len);
5  System.out.println(str);
```

Output:

```
0
null
```

Explanation: An empty String have 0 length (because it contains no characters). However, it still points to valid String value with 0 characters. Hence, the method invoked through its object will not generate any NullPointerException. Object str have null value assigned to it. Hence, printing str will give null as output.

String Exceptions

We deal with majorly 2 exceptions related to array:

- NullPointerException
- StringIndexOutOfBoundsException

Example 3.1: NullPointerException with null Strings

```
1  String obj = null;
2  int len = obj.length();
3  System.out.println(len);
```

Output:

```
NullPointerException at line 2
```

Explanation:

A null String (pointing to nothing) will generate NullPointerException if a method is invoked through it. The code however will not generate any compile time error because our String object is initialized and we are invoking a method on an initialized object.

Example 3.2: index out of range

```
1  String str = "Try";
2  System.out.println(str.length());
3  System.out.println(str.charAt(3));
```

Output:

```
3
StringIndexOutOfBoundsException at line 3
```

Explanation: String indexes range from 0 to length – 1. If we try to access any index outside this range we'll get StringIndexOutOfBoundsException at runtime. Hence, any value less than 0 and greater than length – 1 will throw exception at runtime.

Example 3.3: Special Case - String of size 0 (empty string)

```
1  String str = "";
2  System.out.println(str.length());
3  System.out.println(str.charAt(0));
```

Output:

```
0
```

```
StringIndexOutOfBoundsException at line 3
```

Explanation: Empty string contain no character. Hence, accessing any index will throw runtime exception.

String class constructor

String is a special class in Java. So far we have assigned direct values (a String literal) to define/initialize our String class objects. However, we can also call String class constructor using new operator to do the same. Syntax is similar to how we initialize object of any class.

Example 4: Using String class constructor

```
1   String obj = new String("Trisect");
2   int len = obj.length();
3   System.out.println(obj);
4   System.out.println(len);
```

Output:

```
Trisect
7
```

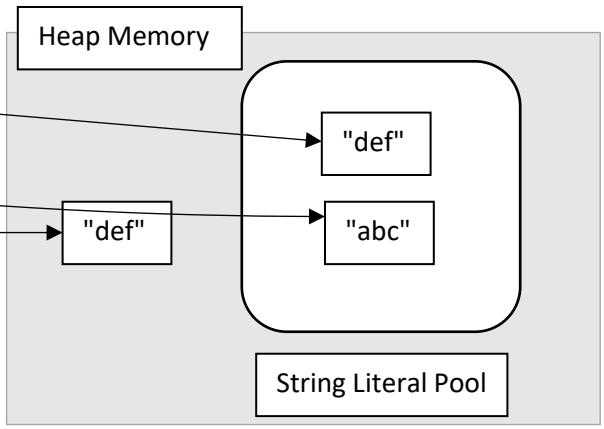
Explanation: As you can see from the example above the string object/variables initialized using String class constructor show similar behavior.

Heap Memory vs String Literal Pool

Strings are treated specially in Java. Whenever we create an object of any class the object gets stored in Java heap memory. The variable then contains the reference to the object in the heap memory. String Literal Pool is a special area inside heap memory. Strings, depending on how they are initialized, are stored in either heap memory or String literal pool. When we initialize a String object using new operator the object is stored in heap memory and its reference is returned. However, when we directly initialize a String object using string literal it gets stored in String literal pool and its reference is returned.

Example 5: Heap Memory vs String Literal Pool

```
1   String obj1 = "abc";
2   String obj2 = new String("def");
```



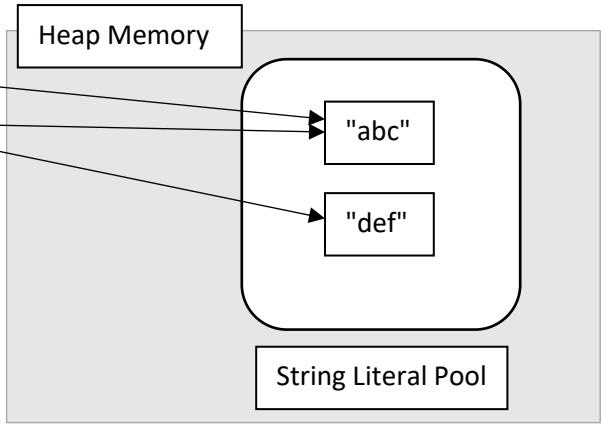
Explanation: As you can see from the diagram above obj1 points to the String "abc" created in String literal pool and obj2 points to String "def" which got created heap memory (because we used new operator to initialize it). String literal "def" itself got initialized in String literal pool.

Benefit of String Literal Pool

Any String literal gets created inside String literal pool. The String literal pool however do not allow any duplicates. Thus if two or more string objects are initialized with the same literal value then all objects will point to the same String literal (i.e. all will have same reference). This helps in reducing the overall memory required. Since, we don't need to allocate new memory space for same String literals multiple times.

Example 6: Benefit of String literal Pool

```
1   String obj1 = "abc";
2   String obj2 = "def";
3   String obj3 = "abc";
```



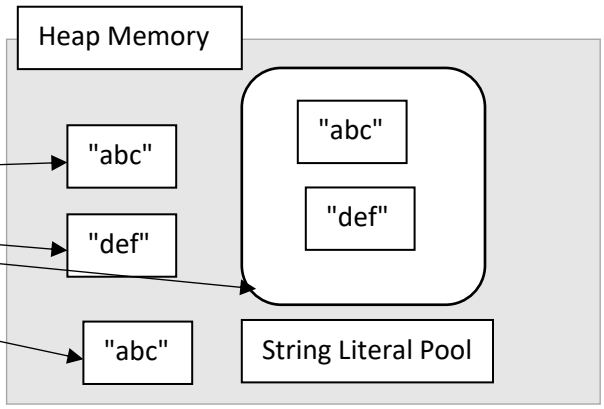
Explanation: As you can see from the diagram above obj1 and obj3 point to the same String literal from String pool. When a new String object is created by String literal the value is first searched in String Pool if it is present there then the same reference is returned.

Disadvantage of Heap Memory Allocation for Strings

When we create a String object using new keyword the string thus created is stored in heap memory. Any String literal passed as parameter to the constructor of String class however is stored in String Pool. Unlike a Sting Pool if we create multiple objects using the same value with new operator a new object will be created each time. Thus new operator wastes a lot of memory space and hence should be avoided.

Example 7: Memory wastage using new operator

```
1   String obj1 = new String("abc");
2   String obj2 = new String("def");
3   String obj3 = new String("abc");
```



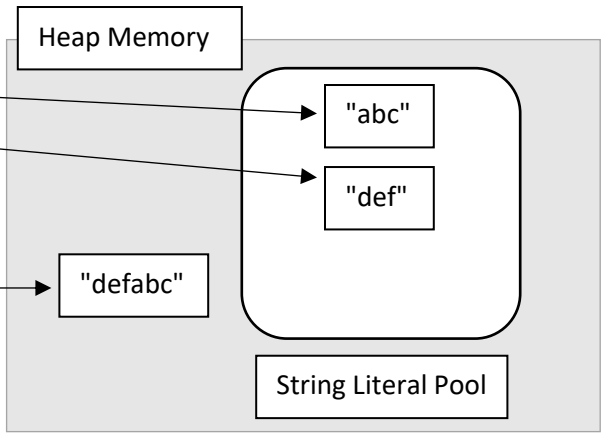
Explanation: Each object created using the new operator creates a new object inside heap memory. The String literal passed as arguments to our String class constructor are created inside String Pool.

Strings are immutable

Strings are immutable i.e. once created we cannot change the value of String. We need to understand this concept in a better way. We can assign or concatenate new values to a String variable/object. However, with each new assignment or concatenation we are creating a new String in memory. The original String object in memory remains intact, we just point to the new value.

Example 8: String immutability

```
1   String str1 = "abc";
2   str1 = "def";
3   str1 = str1 + "abc";
```



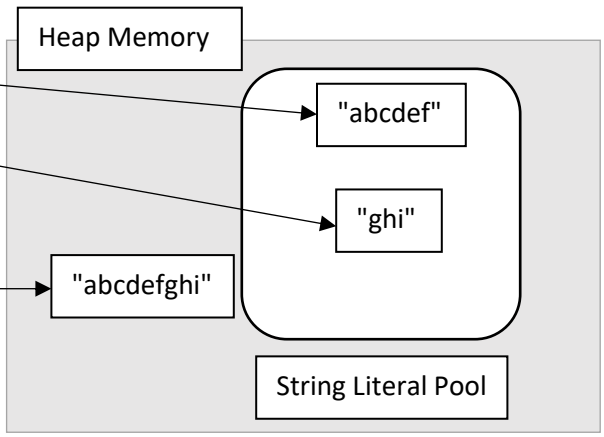
Explanation: Each new assignment or concatenation creates a new object in memory. You can notice that concatenation's result is stored inside heap memory instead of String pool.

Concatenation of string literals

The concatenation of string literals will always be placed in string pool. However, concatenation of a string with any variable will always create object in heap memory.

Example 9: Concatenation of string literals

```
1   String str1 = "abc" + "def";
2   String str2 = str1 + "ghi";
```



Explanation: Java compiler will combine the two string literals and put the final result in string pool. However, if the values are generated by appending to another variable the result is stored in heap memory. Java compiler does a bit of optimization at compile time. The string literals (values inside

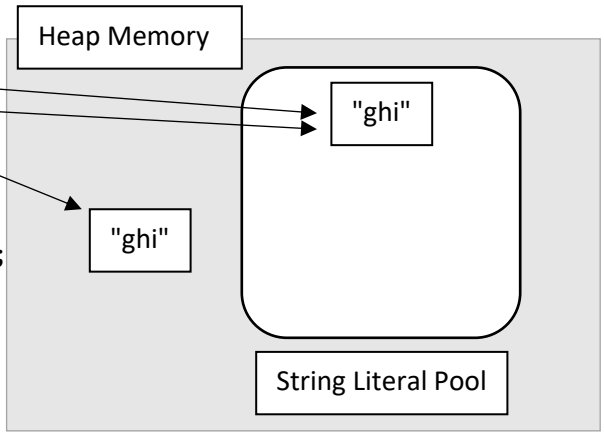
double quotes) are treated as constants. For example, "hello", "world", "trisect" are all string constants. When we concatenate two constant strings (constant expression) the resultant string is computed at compile time itself and treated as a string literal. In the above program str1 = "abc" + "def"; is replaced with str1 = "abcdef". However, any concatenation involving variables are computed at runtime.

String Equality using equals() method

We always use equals() method to check if two strings are same. The equals() method compares the two string character by character and if any character does not match then it returns false.

Example 10.1: Using equals() method

```
1   String str1 = "ghi";
2   String str2 = "g" + "hi";
3   String str3 = new String("ghi");
4
5   System.out.println(str1.equals(str2));
6   System.out.println(str3.equals("ghi"));
7   System.out.println(str2.equals("ghii"));
```



Output:

```
true
true
false
```

Explanation: equals() method always compare the string values irrespective of their memory location. Hence, str3, "ghi", str2 and str1 are all same because all of them contain same characters (in order and frequency).

Example 10.2: Comparing with null

```
1   String str1 = "abc";
2   boolean bool = str1.equals(null);
3   System.out.println(bool);
```

Output:

```
false
```

Example 10.3: Issues with null comparison

```
1   String str1 = null;
2   boolean bool = str1.equals(null);
3   System.out.println(bool);
```

Output:

```
NullPointerException at line 2
```

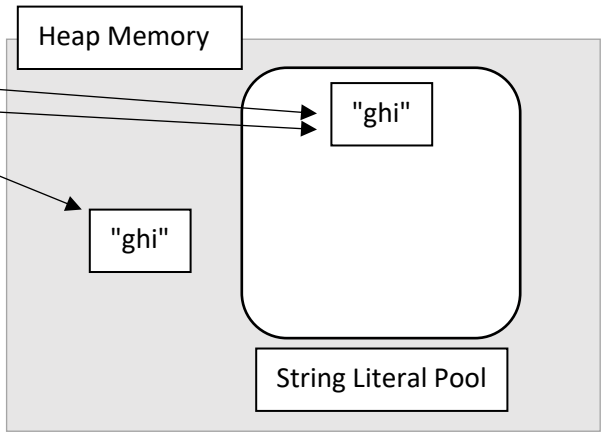
Explanation: Here str1 contains null reference and we are trying to invoke a method on null reference which will generate null pointer exception.

== operator vs equals()

== operator checks the references of two object. If used with String objects == will check if both the objects contains same reference (points to same object in memory). This is true only when the String is present in String pool. However, if we try to compare two strings (that are same in value) one from heap and another from string pool then == will always return false. Hence, result of == cannot be a reliable indicator to check equality of two strings.

Example 11.1: Using ==

```
1   String str1 = "ghi";
2   String str2 = "ghi";
3   String str3 = new String("ghi");
4
5   System.out.println(str1==str2);
6   System.out.println(str2==str3);
```



Output:

```
true
false
```

Explanation: str1 and str2 both points to same string hence == gives true in this case. However, comparison between str2 and str3 generates false, despite both of them containing same value. This happens because both str2 and str3 points to different objects and == operator compares only the object references.

Example 11.2: == can be used to check if string is null

```
1   String str1 = null;
2   boolean bool = (str1 == null);
3   System.out.println(bool);
```

Output:

```
true
```

Explanation: Comparison with null is not possible with equals() method because it is not a valid string value. So we always use == for this purpose.

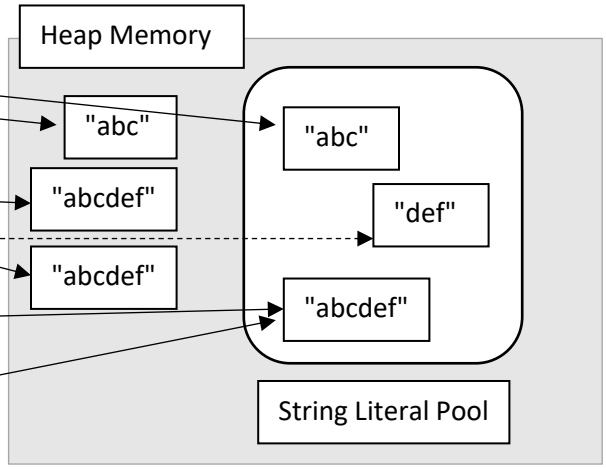
intern() method: Moving objects from heap to string pool

We can force creation of a string object inside string pool using intern() method of class. Note all strings initialized with string literals are by default interned. Whenever we call intern() method on a string object, if the pool already contains a string equal to this String object, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is

returned. This method always return a String that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Example 12: Using intern() method

```
1  String str1 = "abc";
2  String str2 = new String("abc");
3  System.out.println(str1==str2);
4
5  String str3 = str1 + "def";
6  String str4 = str1 + "def";
7  System.out.println(str3==str4);
8
9  String str5 = str3.intern();
10 System.out.println(str3==str4);
11
12 String str6 = str4.intern();
13 System.out.println(str5==str6);
```



Output:

```
false
false
false
true
```

Explanation: At line 9 when we invoke intern() method on str3 it searches for a string in string pool with same value. Since, the pool do not contain similar value a new object is created in string pool and its reference is returned. Next at line 12 when we invoke intern again this time reference of same object from string pool is returned (new object isn't created).

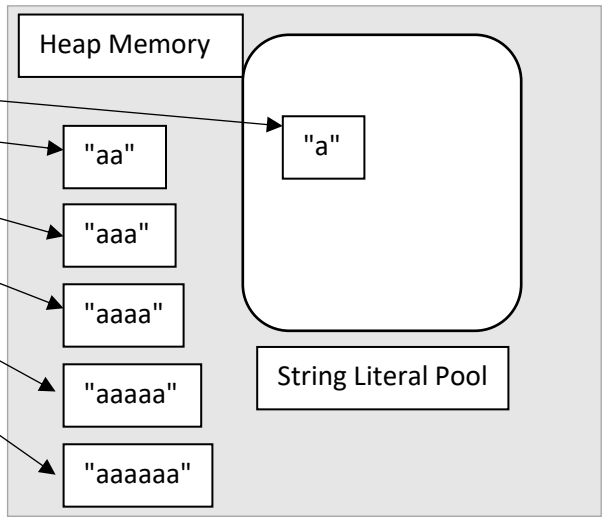
String vs StringBuilder

Concatenation (appending/joining) in normal String object is fine as long as we are concatenating few set of values. However, it gets very expensive if say we try to concatenate 100 times (or 10,000) because normal String objects are immutable and hence for each new operation a new object will be created in memory. This will create a lot of garbage and can quickly deplete the available memory.

StringBuilder on the other hand are mutable in nature. So for if lots of changes are required then we generally use StringBuilder (methods like append(), insert(), delete() etc are provided). When you append values to StringBuilder you are just editing the objects value rather than creating new objects.

Example 13.1: Without StringBuilder

```
1  String str = "a";
2  str = str + "a";
3  str = str + "a";
4  str = str + "a";
5  str = str + "a";
6  str = str + "a";
7  System.out.println(str);
```



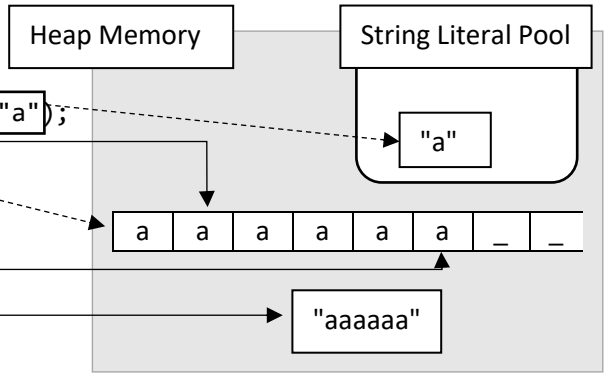
Output:

```
aaaaaa
```

Explanation: The above program will create a one "a" in string literal pool. However, due to its immutable nature each consequent string concatenation operation will create a new string inside heap memory. Since we are concatenating 5 times the resultant objects in memory will be 5. Consider the number of objects created if the concatenation operation is executed 100 times.

Example 13.2: Using StringBuilder

```
1  StringBuilder sb = new StringBuilder("a");
2  sb.append("a");
3  sb.append("a");
4  sb.append("a");
5  sb.append("a");
6  sb.append("a");
7  String str = sb.toString();
8  System.out.println(str);
```



Output:

```
aaaaaa
```

Explanation: The above program will create a single StringBuilder object and a single object "a" in string literal pool. As you can see that a StringBuilder object behaves more like a character array. Subsequent concatenation operation keep on filling the same array. Finally to get the String value from StringBuilder object we can use toString() method which returns the string equivalent of StringBuilder object. Thus only 3 objects are there in memory. Even if we append 100 times the above program will only create 3 objects. The same would have created 100 new objects if we used only String.

String vs StringBuilder vs StringBuffer

String are immutable. StringBuilder and StringBuffer are both mutable in nature. So in comparison to String they perform when lots of changes in data are required.

Concatenation operator '+' internally uses StringBuffer or StringBuilder.

StringBuffer is synchronized and hence thread-safe. But, the drawback is they due to this they are resource intensive in comparison to StringBuilder.

StringBuffer was the only option of string manipulation in earlier versions of Java. However, recent version support StringBuilder in addition to StringBuffer. StringBuilder being non-synchronized (no thread safety) performs a lot better than the other two in single thread environment. Since, most of the applications are not multithreaded in nature hence StringBuilder is preferred.

Problem 1	**In how many ways can we create a String object? Give example for each.**

Problem 2	**Why are String object in Java called as immutable? Explain with example.**

Problem 3	**Explain the concept of String Literal Pool. Does it provide any benefit?**

Problem 4	**How many objects will be created in each of the following segments? Explain with a diagram for each**

Code 4.1

```
1  String str1 = "Hello";
2  String str2 = "World";
```

Code 4.2

```
1  String str1 = "Hello";
2  String str2 = "Hello";
```

Code 4.3

```
1  String str2 = new String("World");
```

Code 4.4

```
1  String str1 = "Hello";
2  String str2 = str1 + ":World";
```

Code 4.5

```
1  String str1 = new String("Hello");
2  String str2 = str1 + "World";
3  String str3 = str2.intern();
```

Code 4.6

```
1  String str1 = "Hello" + "World";
2  String str2 = "HelloWorld"
```

Problem 5	**Explain the difference between equals() and == with an example.**

Problem 6	Give output for each of the following code segments. Explain your answer.

Code 6.1

```
1  String str = "";
2  int len = str.length();
3  System.out.println(len);
```

Code 6.2

```
1  String str = ""
2  char ch = str.charAt(0);
3  System.out.println(ch);
```

Code 6.3

```
1  String obj = null;
2  System.out.println(obj);
```

Code 6.4

```
1  String str1 = null;
2  int len = str1.length();
3  System.out.println(len);
```

Code 6.5

```
1  String str1 = "Trisect";
2  String str2 = new String("Trisect");
3  String str3 = "Tri" + "sect";
4
5  System.out.println(str1==str2);
6  System.out.println(str1==str3);
7  System.out.println(str2==str3);
8
9  System.out.println(str1.equals(str2));
10 System.out.println(str1.equals(str3));
11 System.out.println(str2.equals(str3));
```

Code 6.6

```
1  String str1 = null;
2  System.out.println(str1==null);
3  System.out.println(str1.equals(null));
```

Problem 7	When a lot of changes are required in data, which one should be used: String or StringBuilder? Explain your answer.

Problem 8	Explain the difference between StringBuilder and StringBuffer.