String Builder in Java

Author: Nitish k

Java String Builder Class

Java StringBuilder class is used to create mutable string. The StringBuilder class is same as StringBuffer class except that it is non synchronized.

Important Constructors of StringBuilder class

Constructor	Description
StringBuilder()	It creates an empty String Builder with the initial capacity of 16.
StringBuilder(String str)	It creates a String Builder with the specified string.
StringBuilder(int length)	It creates an empty String Builder with the specified capacity as length.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	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.

public StringBuilder insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	It is used to reverse the string.
public int capacity()	It is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	It is used to return the character at the specified position.
public int length()	It is used to return the length of the string i.e. total number of characters.

public String substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

Examples

1) For append() method

```
package stringbuilder;
public class Example {

	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder("Hello");
		sb.append(" World");
		System.out.println(sb);
	}
}
```

2) For insert() method

Inserts a string at given position

Eg:

```
package stringbuilder;

public class Example2 {

	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder("Hello");
		sb.insert(5, " World!!");
		System.out.println(sb);
	}
}
```

3) For replace() method

The StringBuilder replace() method replaces the given string from the specified beginindex and endindex

Eg:

```
package stringbuilder;
```

```
public class Example3 {

	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder("hello");
		sb.replace(1, 3, "Java");
		System.out.println(sb);
	}
}
```

4) For delete() method
The delete() method of StringBuilder class deletes the string from the specified beginindex to endindex.
Eg:

```
package stringbuilder;
public class Example4 {
	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder("hello");
		sb.delete(1, 3);
		System.out.println(sb);
	}
}
```

5) For reverse() method
The reverse() method of StringBuilder class reverses the current string.

Case 1:
To reverse the entire string of input
Eg:

```
package stringbuilder;
public class Example5 {
	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder("This is a Sentence");
		sb.reverse();
		System.out.println(sb);
	}
}
```

Case 2:
To reverse the string word by word
Steps:

i. Tokenize each word using String.split() method.
ii. Loop through string array and use StringBuilder.reverse() method to reverse each word.
iii. Join all reversed word to get back the word

```
package stringbuilder;
import java.io.*;
public class Example5 {

	public static void main(String[] args) throws IOException{
		// TODO Auto-generated method stub
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		System.out.println("Enter a String");
		String str = br.readLine();
		System.out.println("Initial String is:");
		System.out.println(str);
		StringBuilder sb = new StringBuilder();
		String [] words = str.split(" ");
		for(String word : words)
		{
			String reverseWord = new
StringBuilder(word).reverse().toString();
			sb.append(reverseWord+" ");
		}
		System.out.println("Reversed String is: ");
		System.out.println(sb);
	}
}
```

6) StringBuilder capacity() method
The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2.
Eg:

```
class StringBuilderExample6{
	public static void main(String args[]){
		StringBuilder sb=new StringBuilder();
		System.out.println(sb.capacity());//default 16
```

```
                sb.append("Hello");
                System.out.println(sb.capacity());//now 16
                sb.append("Java is my favourite language");
                System.out.println(sb.capacity());//now (16*2)+2=34 i.e
                (oldcapacity*2)+2
        }
}
```

7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2.

Eg:

```
class StringBuilderExample7{
        public static void main(String args[]){
                StringBuilder sb=new StringBuilder();
                System.out.println(sb.capacity());//default 16
                sb.append("Hello");
                System.out.println(sb.capacity());//now 16
                sb.append("Java is my favourite language");
                System.out.println(sb.capacity());//now (16*2)+2=34 i.e
                (oldcapacity*2)+2
                sb.ensureCapacity(10);//now no change
                System.out.println(sb.capacity());//now 34
                sb.ensureCapacity(50);//now (34*2)+2
                System.out.println(sb.capacity());//now 70
        }
}
```

Java StringBuffer Class

Java StringBuffer is used to create mutable String objects. The StringBuffer class in java is the same as String class except it is mutable.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	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.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

1) Using append method

   The append() method concatenates the given argument with this String

   Eg:

```
package stringbuilder;

public class Example6 {

	public static void main(String[] args) {
		StringBuffer sb = new StringBuffer("Hello");
		sb.append(" World");
		System.out.println(sb);
	}

}
```

2) Using insert() method

The insert() method inserts the String with this string at the given position.

Eg:

```
class StringBufferExample2{
	public static void main(String args[]){
		StringBuffer sb=new StringBuffer("Hello ");
		sb.insert(1,"Java");//now original string is changed
		System.out.println(sb);//prints HJavaello
	}
}
```

3) Using replace() method:

The replace() method replaces the given String from the specified beginIndex and endIndex

Eg:

```
class StringBufferExample3{
	public static void main(String args[]){
		StringBuffer sb=new StringBuffer("Hello");
		sb.replace(1,3,"Java");
		System.out.println(sb);//prints HJavalo
	}
}
```

4) Using delete() method:
The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.
Eg:

```
class StringBufferExample4{
	public static void main(String args[]){
```

```
                StringBuffer sb=new StringBuffer("Hello");
                sb.delete(1,3);
                System.out.println(sb);//prints Hlo
        }
}
```

5) Using reverse() method:
The reverse() method of the StringBuilder class reverses the current String.

Eg:

```
class StringBufferExample5{
        public static void main(String args[]){
                StringBuffer sb=new StringBuffer("Hello");
                sb.reverse();
                System.out.println(sb);//prints olleH
        }
}
```

6) Using capacity method
The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

Eg:

```
class StringBufferExample6{
        public static void main(String args[]){
                StringBuffer sb=new StringBuffer();
                System.out.println(sb.capacity());//default 16
                sb.append("Hello");
                System.out.println(sb.capacity());//now 16
                sb.append("java is my favourite language");
                System.out.println(sb.capacity());//now (16*2)+2=34 i.e
                (oldcapacity*2)+2
        }
}
```

7) Using ensureCapacity() method
The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

Eg:

```
class StringBufferExample7{
```

```
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer();
        System.out.println(sb.capacity());//default 16
        sb.append("Hello");
        System.out.println(sb.capacity());//now 16
        sb.append("java is my favourite language");
        System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
        sb.ensureCapacity(10);//now no change
        System.out.println(sb.capacity());//now 34
        sb.ensureCapacity(50);//now (34*2)+2
        System.out.println(sb.capacity());//now 70
    }
}
```

Immutable Classes in Java

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. All the wrapper classes and String class is immutable. We can create immutable classes by creating **final** class that have final data members

Eg:

```
package practice;
class Employeer
{
    final String PanCardNumber;
    public Employeer(String PanCardNumber)
    {
            this.PanCardNumber = PanCardNumber;
    }
    public void getPancard()
```

```
    {
        System.out.println(PanCardNumber);
    }
}
public class Employee {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Employeer obj = new Employeer("ABC12345");
        obj.getPancard();
    }
}
```

Important Points to note from above example:

1) The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
2) The class is final so we cannot create the subclass.
3) There is no setter methods i.e. we have no option to change the value of instance variable.

Java toString() Method

If u want to represent any object as a string, toString() method comes into existence. The toString() method returns the String representation of the object. If you print any object, Java compiler internally invokes the toString() method on the object. SO overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on implementation.

Advantage

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

Eg:

```
package practice;

public class Student {

	String Name;
	String city;
	int age;

	public Student(String Name,String city,int age)
	{
		this.Name = Name;
		this.city = city;
		this.age = age;
	}
	public String toString()
	{
		return Name+" "+city+" "+age;
	}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Student s1 =  new Student("Raj","Bangalore",21);
		Student s2 = new Student("Verma","Bombay",22);
		System.out.println(s1);
		System.out.println(s2);
	}
}
```

Memory Management & String Interning
There are majorly two types of memory associated with a program, that are stack memory and heap memory. You should keep in mind that only primitive data-types: Byte, Short, Int, Long, Double, Float, Char and Boolean are stored directly in stack memory.

Since, strings are non-primitive data types in Java, hence they are not stored directly in stack memory. Instead, they are created in the heap memory and just a reference to this location is held by the stack memory.

But, there is a catch here! There is a special area or segment of heap memory known as intern pool. Strings in Java are created in this area known as intern pool

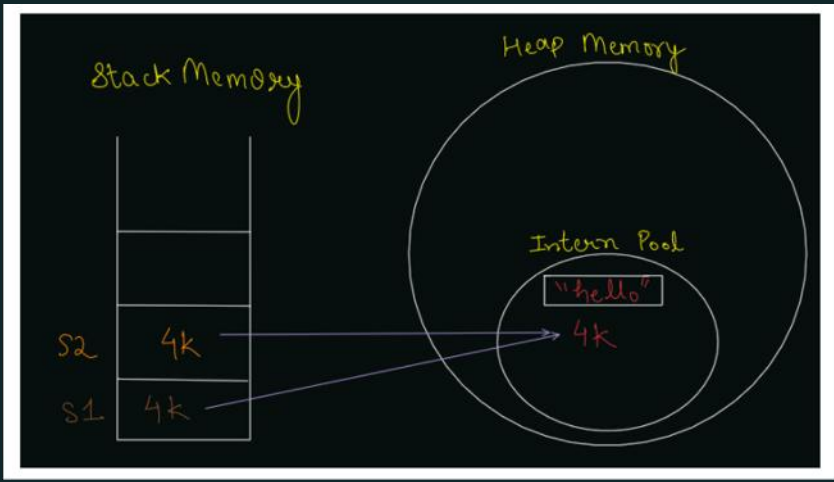
Let us take an example of three strings declared in the following manner:

```
1.        String s1 = "hello";
2.        String s2 = "hello";
3.        String s3 = new String("hello");
```

Statement 1, will make a string of 5 characters "hello" in the intern pool, let us suppose, at starting address = 4k. And this address (4k) is stored in the stack memory with s1 holding the address.

Now, Statement 2, will make another string of 5 characters "hello" in the intern pool...! No

First, a character array {'h','e','l','l','o'} is searched in the intern pool. If it already exists (which in this case, is yes as string at location 4k = "hello" is already present), then JVM (Java Virtual Machine) will not create another string "hello" in the intern pool. Instead, it will store the reference of the same 4k location for the s2 string as well. Hence, now 2 strings s1 and s2 point to the same "hello" literal in the intern pool.



If it would have been not present, then JVM would have created another string literal and allocated a new memory location to the String.

This process of searching for an identical string literal in the intern pool, and if present, sharing the same memory of string literal for different strings is known as String Interning in Java.

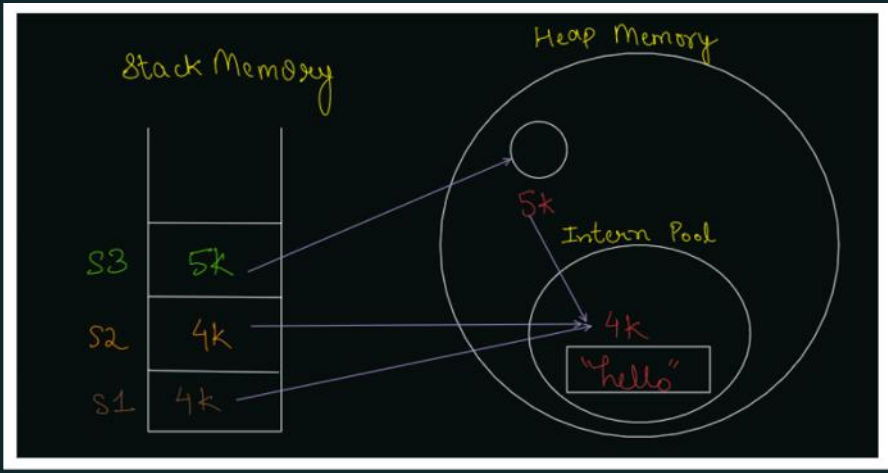
We have covered WHAT is interning and HOW it takes place, but WHY interning takes place? What is the need or it's advantages?

Is there a way to avoid String Interning?

Yes, we can create the strings using the new keyword. This will create a new shell object outside of the intern pool (with a new address) for the string.

But, there is still a problem. If there will be an identical character array, then this shell object will also point to the same character array.

For statement 3, a new shell object will be created at location, say 5k, but, in this case also, the shell object will point to the same character array at location 4k, since it contains the string "hello".



So, we have seen WHAT, HOW & WHY of Interning. But what are the impacts of doing String Interning? Two major impacts are the unexpected result in String comparisons and it makes Strings Immutable.

Comparison operator (==) vs equals() method

We should avoid using == operator for comparing strings. Let us know why?

What do you think will be output of two comparison operations:

Is s1 == s2?
Is s1 == s3?

You might have guessed that those operations will give the result 'true'. But, only the first operation will meet your expectations. The result of s1 == s3 will be 'false'.

It is because, in stack memory, s1 is pointing to 4k memory location whereas s3 is pointing to a different memory location (5k), now JVM will not check whether the shell object of s3 at 5k is pointing to the same character array or not. It will just return false.

Hence, even after s1 = "hello" and s3 = "hello", the result will be false as s3 is initialized using the new keyword.

To compare strings in java, we should make a habit to always use equals() method. s1.equals(s2) will be directly returned true as both point to the same (4k) memory location.

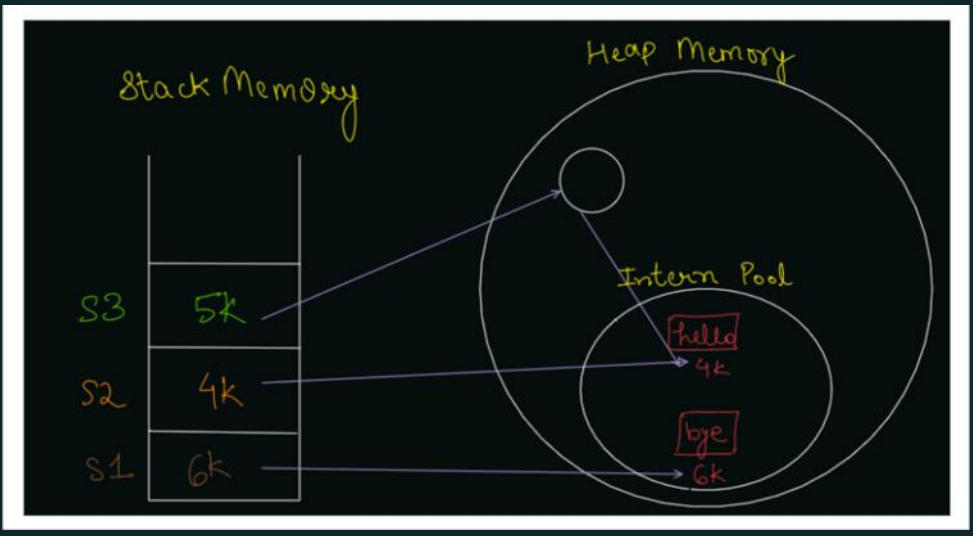
Now, s1.equals(s3) will not be returned false even if both point to different memory locations, instead they will now be checked whether they are equal character-by-character or not. Since both are "hello", it will also return true.

Q) What is IMMUTABILITY of Strings?
If you look for the meaning of immutability in English dictionary, you will find it means 'which cannot be changed/modified'. Does it mean that Strings cannot be changed in Java? Well, a proper answer is:

'Reference is mutable, instance is not.' Once a string object is created, its data or state can't be changed but the string can point to another character array.

If we try to modify s1 as : s1 = "bye", then what will happen? Will the character array {h, e, l, l, o} which is not only being pointed by s1 but also by s2, change?

No, but a new object will be created with a character array {b, y, e} and s1 will now start pointing to this new address location.



Hence, the reference of string (in stack memory) is mutable, but the instance (character array in intern pool) is immutable.

Q) But, why does Immutability occur in the first place?
Let us assume, if you were allowed to modify the instance of string, i.e. modify the character array {h, e, l, l, o} in the intern pool, then what would have been the consequences?

Since the character array at 4k location is pointed by more than 1 string, modifying the character array itself would have led to modification of all the strings which are pointing it.

For example), if we would have changed s1's character at 3rd index to say 'd', then character array would have been modified to {h, e, d, l, o}, hence the character at 3rd index for the string s2 would also have changed to 'd'.

This can be a surprise to the developer who does not know about the existence of string s1, but is working on a module which uses string s2. He would wonder why s2 has become "hedlo", but he himself initialized it to "hello".

Thus, to avoid such surprises, it is not allowed to modify the instance of strings in Java. Strings are immutable in java as an immediate cause of string interning.

Q) What is the impact of Immutability?
Immutability can lead to low performance issues, both in terms of space and time. Consider the following example:

```
Scanner scn = new Scanner(System.in);
int n = scn.nextInt();
String s = "";
for(int i=0; i<n; i++){
    s += i;
}
System.out.println(s);
```

What do you think will be the time complexity of the above code? Will it be O(n), as we are adding only 1 character n times?

No, the time complexity of the above piece of code is as bad as O(n2). As we have discussed earlier, that instance of string cannot be modified, hence adding a character to the end of string does not mean it will get appended in the end of the character array.

Instead, even if a single character is appended, firstly JVM will create a new string object, copy all of the characters present to the new string, and then append the single character to the newly created string object. Hence we are first copying the entire string which is O(n) work even for appending a single character.

And since we are appending single character for n times, the overall time complexity will turn out to be O(n * n) = O(n2).

For example) to add character 'd' to string s = "hello", we have to first make a new string object, then copy all characters {h, e, l, l, o} to the new string and append 'd' to the new string. Hence we are not operating on 1 character, but s.length() + 1 characters.

So, consider if the current string is taking 2GB space in memory, then adding a single character will lead to creation of a copy of the string of entire 2GB and appending 1 character (of few byte(s)) to it.

Sources : Pepcodingp on Memory Management & String Interning