Java String

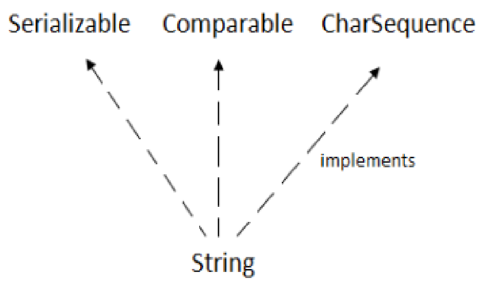
In Java, String is an object that represents a sequence of character values. An array of characters work in the same way as java string. Example:

char[] arr = {'S', 'o', 'p', 'h', 'i', 'a'};
String result = new String(arr);

Is the same as

String result = "Sophia";

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



CharSequence Interface

The CharSequence interface is 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 3 classes.

The java string is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

How to create a string Object

There are two ways to create String object:

1. String Literal
2. By new keyword

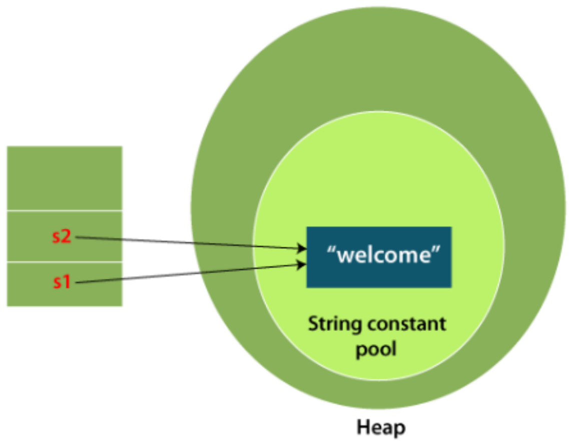
1) String Literal

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

String s = “Welcome”;

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn’t exist in the pool, a new string instance is created and placed in the pool. For example:

String s1 = “Welcome”;
String s2 = “Welcome”; // It doesn’t create a new instance



Note: String constant pool is the memory area where string objects are stored.

Java uses the concept of string literal to make java memory more efficient(because no new objects are created if it already exists in the string constant pool);

2) By New Keyword

String s = new String(“Welcome”); // creates two objects and one reference variable

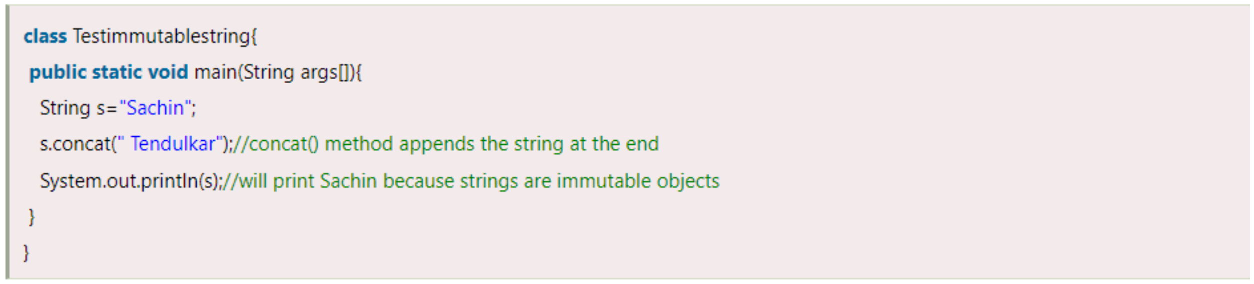
Here, JVM will create a new string object in normal(non-pool) heap memory, and the literal “Welcome” will be placed in the string constant pool. The variables will refer to the object in the heap(non-pool);

Immutable String in Java

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

Once string object is created, its data or state can't be changed but a new String Object is created.

Example:

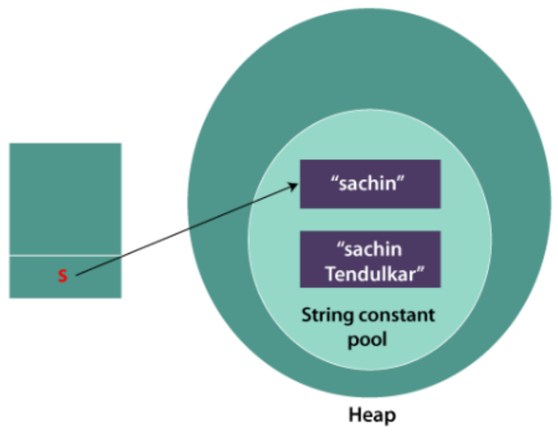


```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Output:

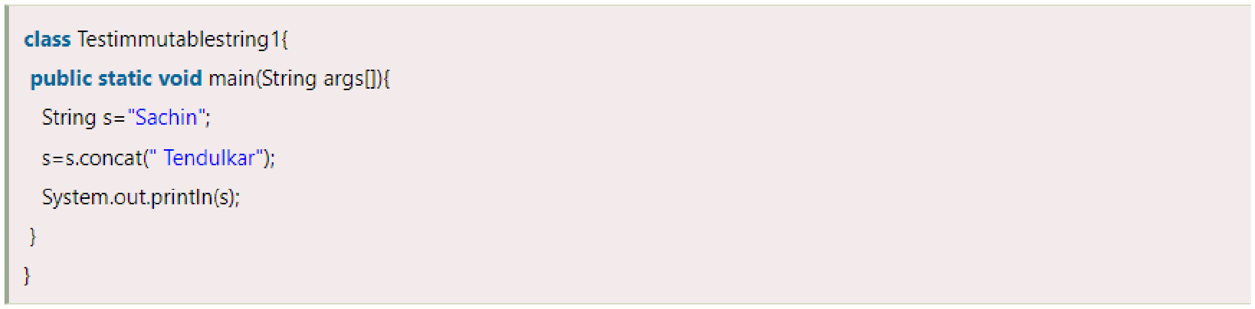
Sachin

Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



Above you can see that the two objects are created but the s reference variable still refers to Sachin not to Sachin Tendulkar.

But if we explicitly assign it to the reference variable, it will refer to the Sachin Tendulkar object. Example:



```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Output

Sachin Tendulkar

In this case, s points to Sachin Tendulkar but the Sachin object is not modified.

Why String Objects are Immutable in Java

1. Class Loader:

A class loader in java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different. To avoid this kind of misinterpretation, string is immutable.

2. Thread safe:

As the string object is immutable, we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security:

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space:
The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

String Methods

- **int length()**: Returns the number of characters in the String.
  "GeeksforGeeks".length(); // returns 13

- **Char charAt(int i)**: Returns the character at ith index.
  "GeeksforGeeks".charAt(3); // returns 'k'

- **String substring (int i):** Return the substring from the ith index character to end.
  "GeeksforGeeks".substring(3); // returns "GeeksforGeeks"

- **String substring (int i, int j):** Returns the substring from i to j-1 index.
  "GeeksforGeeks".substring(2, 5); // returns "eks"

- **String concat(String str):** Concatenates specified string to the end of this string.
  String s1 = "Geeks";
  String s2 = "forGeeks";

  String output = s1.concat(s2); // returns "GeeksforGeeks"

- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.
  String s = "Learn Share Learn";

  int output = s.indexOf("Share"); // returns 6

- **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.
  String s = ”Learn Share Learn”;

  int output = s.indexOf("ea",3);// returns 13

- **Int lastIndexOf(String s):** Returns the index within the string of the last occurrence of the specified string.
  String s = ”Learn Share Learn”;

  int output = s.lastIndexOf("a"); // returns 14

- **boolean equals(Object otherObj):** Compares this string to the specified object.
  Boolean out = “Geeks”.equals(“Geeks”); // returns true
  Boolean out = “Geeks”.equals(“geeks”); // returns false

- **boolean equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.
  Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true
  Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true

- **int compareTo(String anotherString):** Compares two string lexicographically.
  int out = s1.compareTo(s2);
  // where s1 and s2 are
  // strings to be compared

  This returns the difference s1-s2. If :

  out < 0 // s1 comes before s2

  out = 0 // s1 and s2 are equal.

  out > 0   // s1 comes after s2.

- **int compareToIgnoreCase(String anotherString):** Compares two strings lexicographically, ignoring case considerations.
  int out = s1.compareToIgnoreCase(s2);

// where s1 and s2 are

// strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2

out = 0   // s1 and s2 are equal.

out > 0   // s1 comes after s2.

Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

- **String toLowerCase():** Converts all the characters in the String to lower case.
  String word1 = "HeLLo";
  String word3 = word1.toLowerCase(); // returns "hello"
- **String toUpperCase():** Converts all the characters in the String to upper case.
  String word1 = "HeLLo";
  String word2 = word1.toUpperCase(); // returns "HELLO"
- **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
  String word1 = " Learn Share Learn ";
  String word2 = word1.trim(); // returns "Learn Share Learn"
- **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of oldChar with newChar.
  String s1 = "feeksforfeeks";
  String s2 = "feeksforfeeks".replace('f' ,'g'); // returns "geeksgorgeeks"
  Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks
- **String intern():** Creates an exact copy of a String object in the heap memory and stores it in the String constant pool. When the intern method is invoked, if the pool already contains a String equal to this String object as determined by the

equals(Object) method, 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.

```
String s = new String(“Sachin”);
String s2 = s.intern();
System.out.println(s2); //returns “Sachin”
```

String Buffer Class in Java

StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer class is used to create mutable (modifiable) strings. The StringBuffer class in java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer class

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

Methods of StringBuffer Class

1) append() method

The append() method concatenates the given argument with this string.

```java
import java.io.* ;

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

2) insert() method

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

Example:

```java
import java.io.* ;

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

3) replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex-1.

Example:

```java
import java.io.* ;

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

4) delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex-1.

Example

```
import java.io.* ;

class A{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello");
        sb.delete(1,3);
        System.out.println(sb);
    }
}
```

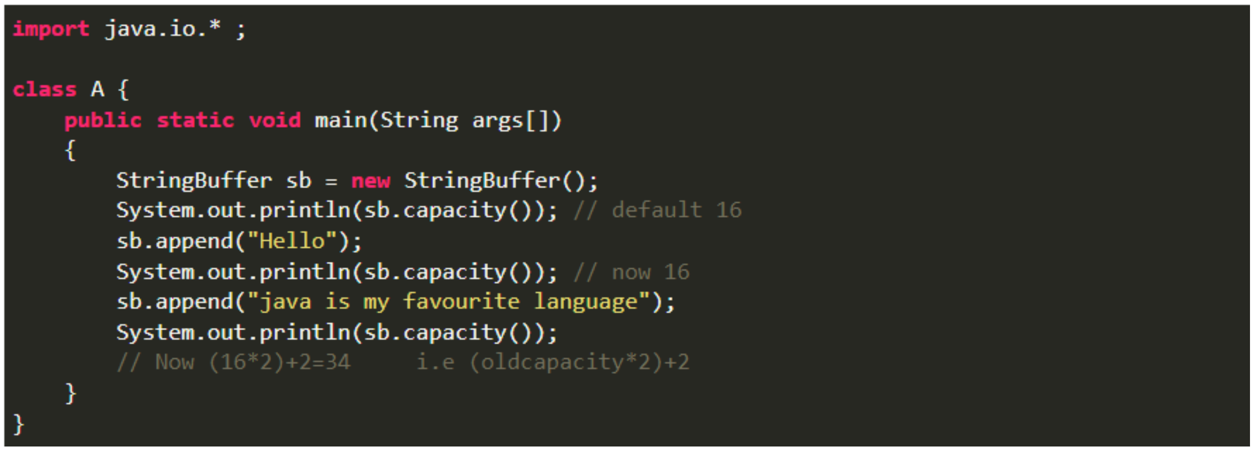
5) The reverse() method

The reverse() method of StringBuilder class reverses the current string. Example:

```
import java.io.* ;

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

6) capacity() method

- The capacity() method of 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.
- e.g:



```
import java.io.* ;

class A {
    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
    }
}
```

Constructors of StringBuffer Class

1. StringBuffer(): It reserves room for 16 characters without reallocation

stringBuffer s = new stringBuffer();

2. StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.

stringBuffer s = new stringBuffer(20);

3. StringBuffer(String str): It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

stringBuffer s = new stringBuffer("GeeksforGeeks");

Methods of StringBuffer class

Methods	Action Performed
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length() method
capacity()	The total allocated capacity can be found by the length() method
charAt()	This method returns the char value in this sequence at the specified index

delete()	Deletes a sequence of characters from the invoking object
deleteCharAt()	Deletes the character at the index specified by loc
ensureCapacity()	Ensures capacity is at least equal to the given minimum
insert()	Inserts text at the specified index position
length()	Returns length of the string
reverse()	Reverse the characters within a StringBuffer object
replace()	Replace one set of characters with another set inside a StringBuffer object

StringBuilder Class in Java

StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters. The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. However, the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does. Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer 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. Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used. String Builder is not thread-safe and high in performance compared to String buffer.

Syntax:

public final class stringBuilder

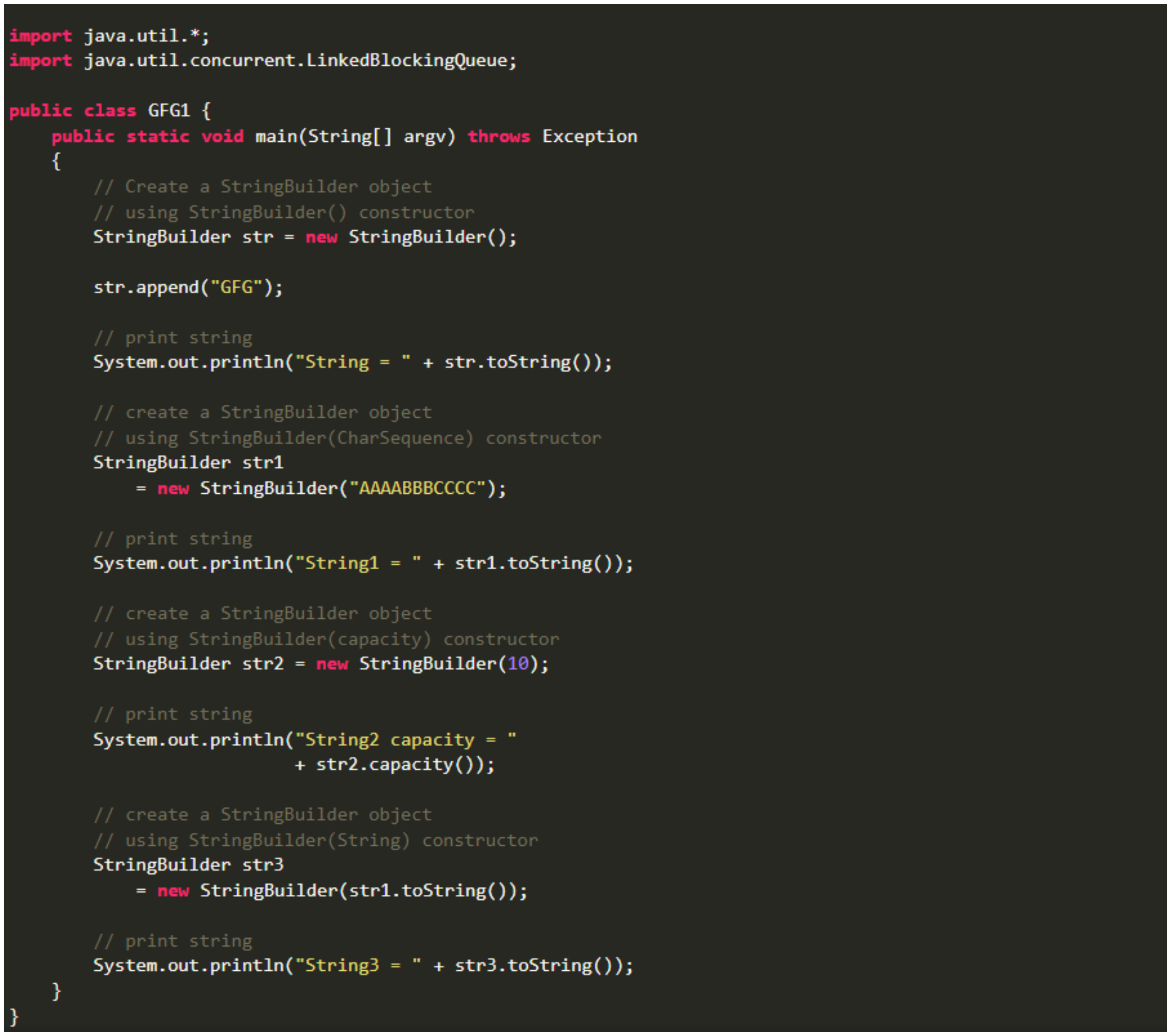
extends Object

Implements Serializable, CharSequnece

Constructors in Java StringBuilder Class

- **StringBuilder():** Constructs a string builder with no characters in it and an initial capacity of 16 characters.
- **StringBuilder(int capacity):** Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
- **StringBuilder(CharSequence seq):** Constructs a string builder that contains the same characters as the specified CharSequence.
- **StringBuilder(String str):** Constructs a string builder initialized to the contents of the specified string.

Below is a sample program to illustrate StringBuilder in Java.



```java
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;

public class GFG1 {
    public static void main(String[] argv) throws Exception
    {
        // Create a StringBuilder object
        // using StringBuilder() constructor
        StringBuilder str = new StringBuilder();

        str.append("GFG");

        // print string
        System.out.println("String = " + str.toString());

        // create a StringBuilder object
        // using StringBuilder(CharSequence) constructor
        StringBuilder str1
            = new StringBuilder("AAAABBBCCCC");

        // print string
        System.out.println("String1 = " + str1.toString());

        // create a StringBuilder object
        // using StringBuilder(capacity) constructor
        StringBuilder str2 = new StringBuilder(10);

        // print string
        System.out.println("String2 capacity = "
                           + str2.capacity());

        // create a StringBuilder object
        // using StringBuilder(String) constructor
        StringBuilder str3
            = new StringBuilder(str1.toString());

        // print string
        System.out.println("String3 = " + str3.toString());
    }
}
```

Output

```
String = GFG
String1 = AAAABBBCCCC
String2 capacity = 10
String3 = AAAABBBCCCC
```

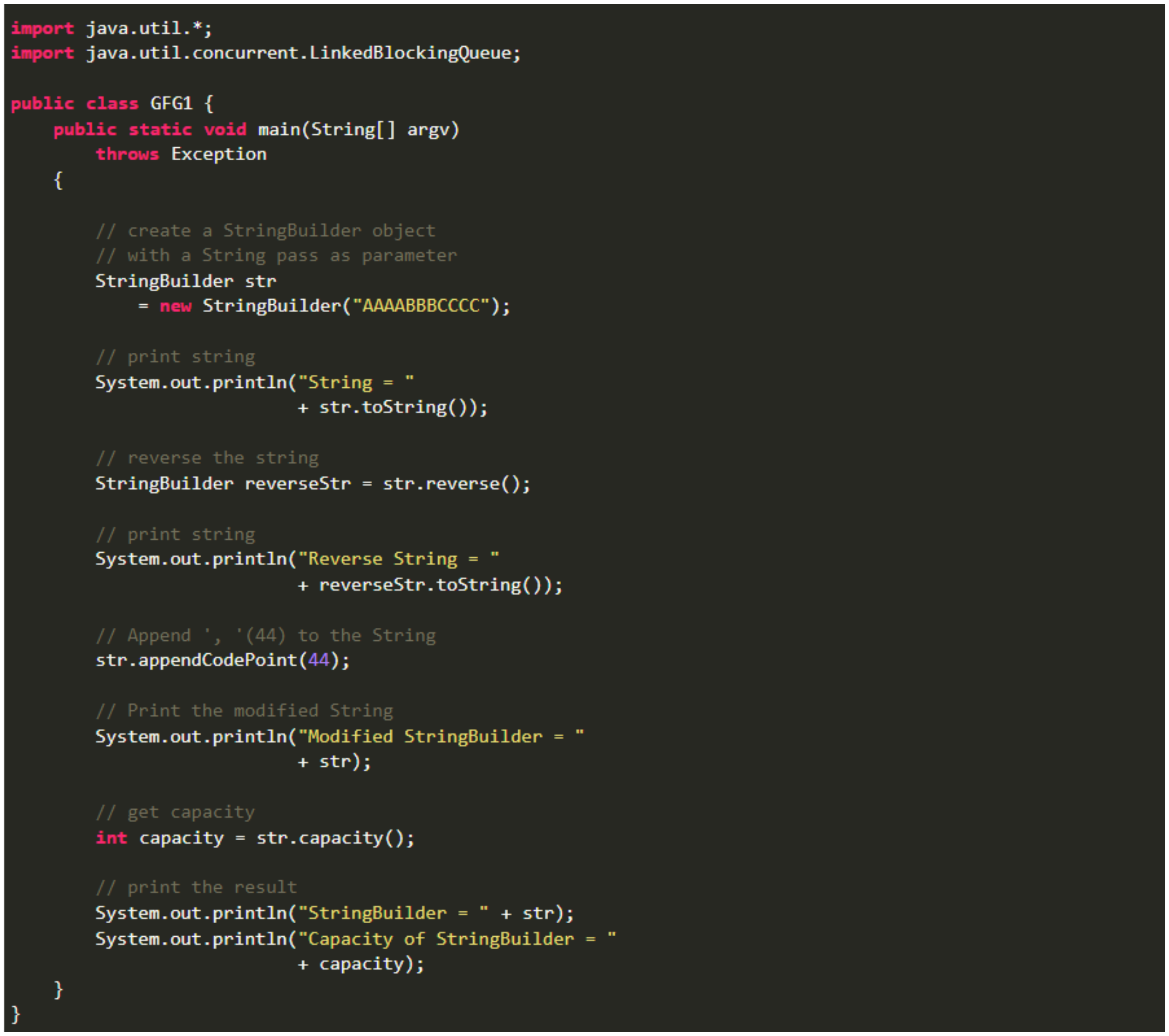
Methods in Java StringBuilder

1. StringBuilder append(X x): This method appends the string representation of the X type argument to the sequence.

2. StringBuilder appendCodePoint(int codePoint): This method appends the string representation of the codePoint argument to this sequence.
3. int capacity(): This method returns the current capacity.
4. char charAt(int index): This method returns the char value in this sequence at the specified index.
5. IntStream chars(): This method returns a stream of int zero-extending the char values from this sequence.
6. int codePointAt(int index): This method returns the character (Unicode code point) at the specified index.
7. int codePointBefore(int index): This method returns the character (Unicode code point) before the specified index.
8. int codePointCount(int beginIndex, int endIndex): This method returns the number of Unicode code points in the specified text range of this sequence.
9. IntStream codePoints(): This method returns a stream of code point values from this sequence.
10. StringBuilder delete(int start, int end): This method removes the characters in a substring of this sequence.
11. StringBuilder deleteCharAt(int index): This method removes the char at the specified position in this sequence.
12. void ensureCapacity(int minimumCapacity): This method ensures that the capacity is at least equal to the specified minimum.
13. void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin): This method characters are copied from this sequence into the destination character array dst.
14. int indexOf(): This method returns the index within this string of the first occurrence of the specified substring.

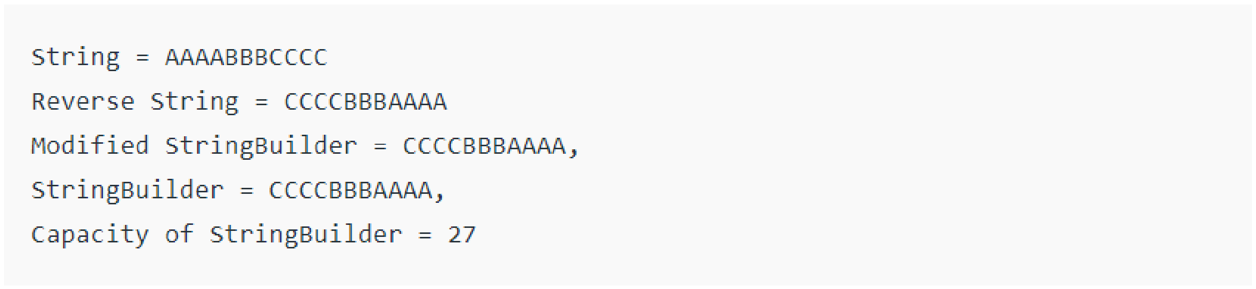
15. StringBuilder insert(int offset, boolean b): This method inserts the string representation of the booalternatelean argument into this sequence.
16. StringBuilder insert(): This method inserts the string representation of the char argument into this sequence.
17. int lastIndexOf(): This method returns the index within this string of the last occurrence of the specified substring.
18. int length(): This method returns the length (character count).
19. int offsetByCodePoints(int index, int codePointOffset): This method returns the index within this sequence that is offset from the given index by codePointOffset code points.
20. StringBuilder replace(int start, int end, String str): This method replaces the characters in a substring of this sequence with characters in the specified String.
21. StringBuilder reverse(): This method causes this character sequence to be replaced by the reverse of the sequence.
22. void setCharAt(int index, char ch): In this method, the character at the specified index is set to ch.
23. void setLength(int newLength): This method sets the length of the character sequence.
24. CharSequence subSequence(int start, int end): This method returns a new character sequence that is a subsequence of this sequence.
25. String substring(): This method returns a new String that contains a subsequence of characters currently contained in this character sequence.
26. String toString(): This method returns a string representing the data in this sequence.
27. void trimToSize(): This method attempts to reduce storage used for the character sequence.

Example:



```java
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;

public class GFG1 {
    public static void main(String[] argv)
        throws Exception
    {

        // create a StringBuilder object
        // with a String pass as parameter
        StringBuilder str
            = new StringBuilder("AAAABBBCCCC");

        // print string
        System.out.println("String = "
                           + str.toString());

        // reverse the string
        StringBuilder reverseStr = str.reverse();

        // print string
        System.out.println("Reverse String = "
                           + reverseStr.toString());

        // Append ', '(44) to the String
        str.appendCodePoint(44);

        // Print the modified String
        System.out.println("Modified StringBuilder = "
                           + str);

        // get capacity
        int capacity = str.capacity();

        // print the result
        System.out.println("StringBuilder = " + str);
        System.out.println("Capacity of StringBuilder = "
                           + capacity);
    }
}
```

Output:



```
String = AAAABBBCCCC
Reverse String = CCCCBBBAAAA
Modified StringBuilder = CCCCBBBAAAA,
StringBuilder = CCCCBBBAAAA,
Capacity of StringBuilder = 27
```

Java toString() Method

If you want to represent any object as a string, the toString() method comes into existence.

The toString() method returns the String representation of the object.

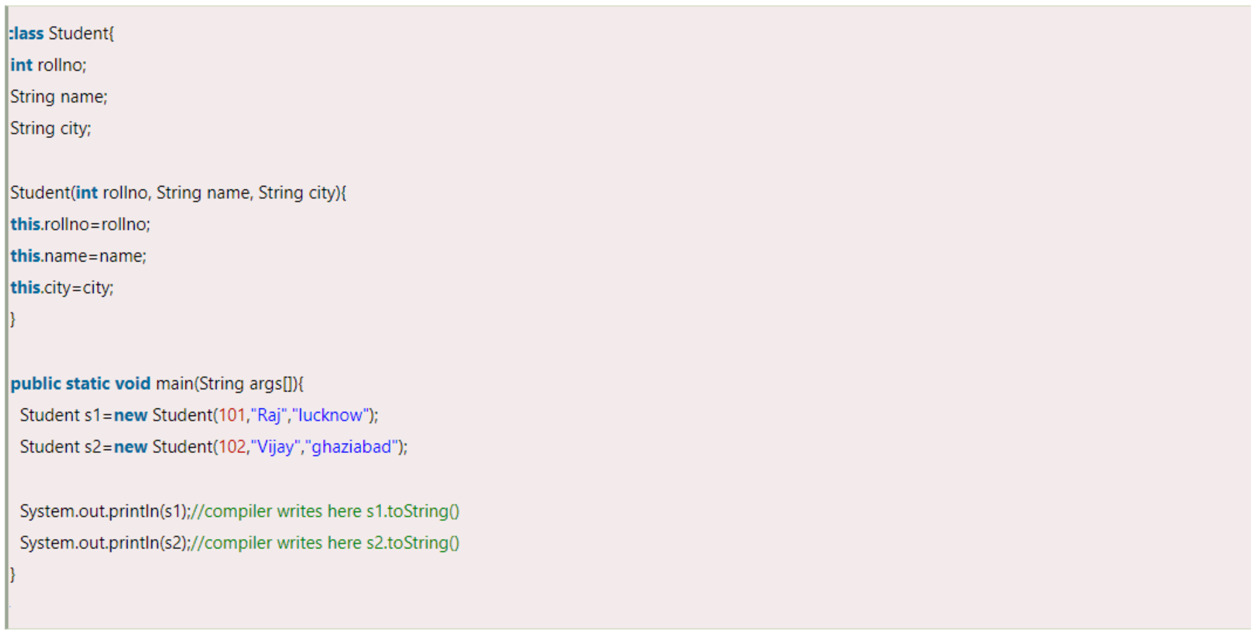
If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

Advantages of Java toString() method

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.

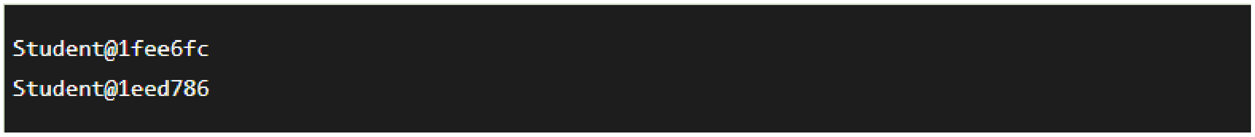
Understanding problem without toString() method

Let's see the simple code that prints reference.



```
:lass Student{
int rollno;
String name;
String city;

Student(int rollno, String name, String city){
this.rollno=rollno;
this.name=name;
this.city=city;
}

public static void main(String args[]){
  Student s1=new Student(101,"Raj","lucknow");
  Student s2=new Student(102,"Vijay","ghaziabad");

  System.out.println(s1);//compiler writes here s1.toString()
  System.out.println(s2);//compiler writes here s2.toString()
}
```

Output:



```
Student@1fee6fc
Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since Java compiler internally calls the toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

Output

```
101 Raj lucknow
102 Vijay ghaziabad
```

In the above program, Java compiler internally calls the toString() method, overriding this method will return the specified values of s1 and s2 objects of Student class.