STRINGS, STRING BUFFER & STRING BUILDER

Table of Contents

STRINGS,STRINGBUFFER & STRINGBUILDER

1.0 Purpose with Problem Scenario

Strings receive special treatment in Java, because they are used frequently in a program. Hence, efficiency (in terms of computation and storage) is crucial.

The designers of Java decided to retain primitive types in an object-oriented language, instead of making everything an object, so as to improve the performance of the language. Primitives are stored in the call stack, which require less storage spaces and are cheaper to manipulate. On the other hand, objects are stored in the program heap, which require complex memory management and more storage spaces.

For performance reason, Java's String is designed to be in between a primitive and a class.

1.2 Introduction:

A Java string is a series of characters gathered together, like the word "Hello", or the phrase "practice makes perfect".

The String class is commonly used for holding and manipulating strings of text in Java programs.

In its simplest form, you use the String class by typing some text surrounded by double quotes. This is called a String literal.

Example: "This is a Java String"

Anywhere you need a String object, you can type a String literal as in:

```
System.out.println("This is a Java String");
```

String is associated with string literal in the form of double-quoted texts such as "Hello, world!".

1.3 Detailed Explanation:

1.3.1 Strings are Objects in Java:

In Java, Strings are objects and not a primitive type. Even string constants are actually String objects.

For example

System.out.println ("Welcome to Java World");

The string "Welcome to Java World" is a **String constant** or **String literal** or **String object.**

Whenever it encounters a string literal/object in your code, the compiler creates a String object with its value—in this case "Welcome to Java World".

String represents combinations of character literals that are enclosed in double quotes, i.e.) any character(s) inside double quotes. Each character in a string is a 16-bit Unicode character, to provide a rich, international set of characters.

String is a class built into the Java language defined in the java.lang package. String is one of the widely used java classes. It is special in java as it has some special characteristics than a usual java class.

1.3.2 String Construction:

String can be created in number of ways, here are a few ways of creating string object.

1.*Using a String literal:* String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
String str1 = "Hello";
```

2.*Using another String object:*

```
String str2 = new String(str1);
```

3.*Using new Keyword:*

```
String str3 = new String("Java");
```

4.Using + operator (Concatenation):

String str4 = str1 + str2;

or,

```
String str5 = "hello"+"Java";
```

Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as String constant pool inside the heap memory

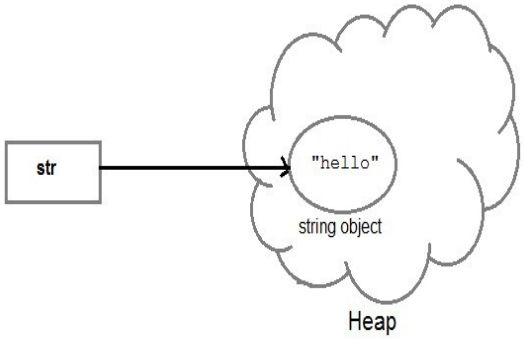
String pool (String intern pool) is a special storage area in Java heap. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

Any good programming language is to make efficient use of memory, As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the universe of String literals for a program.

To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool."When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.)

1.3.3 String Object and how they are stored:

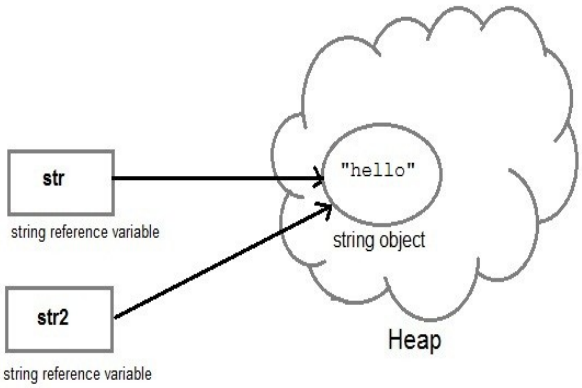
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already



String str= "Hello";

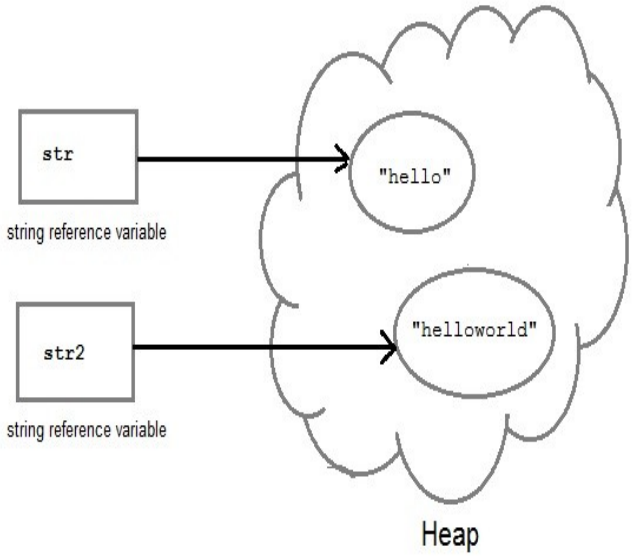
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

String str2=str;



But if we change the new string, its reference gets modified.

```
str2=str2.concat("world");
```

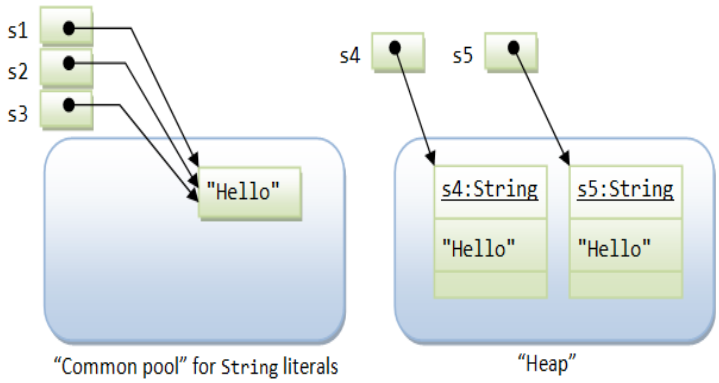


1.3.4 String Literal vs. String Object:

Java has provided a special mechanism for keeping the String literals - in a so-called string common pool. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to conserve storage for frequently-used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.

For Example:

```
String s1="Hello";          // String literal

String s2="Hello";          // String literal

String s3 = s1;                  // same reference

String s4 = new String("Hello"); // String object

String s5 = new String("Hello"); // String object
```



1.3.5 String is immutable:

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable.

Because String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas).

For example,

// inefficient codes

```
String str = "Hello";
for (int i = 1; i < 1000; ++i) {
    str = str + i;
```

If the contents of a String have to be modified frequently, use the **StringBuffer** or **StringBuilder** class instead. Let us explore in details by considering the below code Snippet:

public class ImmutableStringObjects {

public static void main(String[] args) {

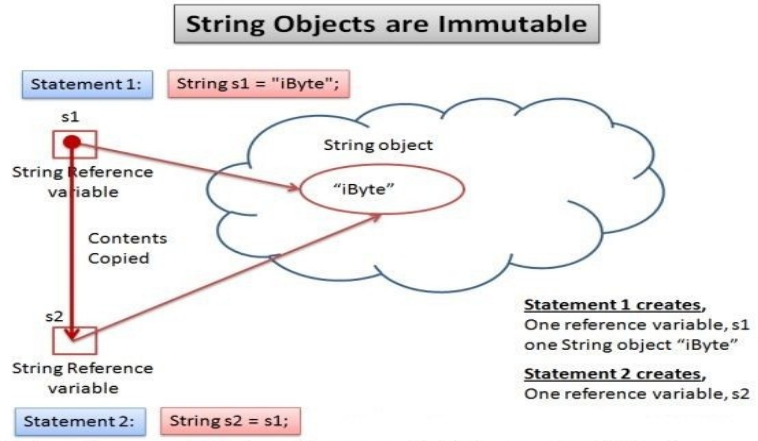
```
String s1 = "iByte"; //Statement 1
String s2 = s1; //Statement 2
s2 = s1.concat("Code's"); //Statement 3
s2 = s2 + " JavaWorld"; //Statement 4
System.out.println("s1 = " + s1);
```

```
System.out.println("s2 = " + s2);

  }

}
```

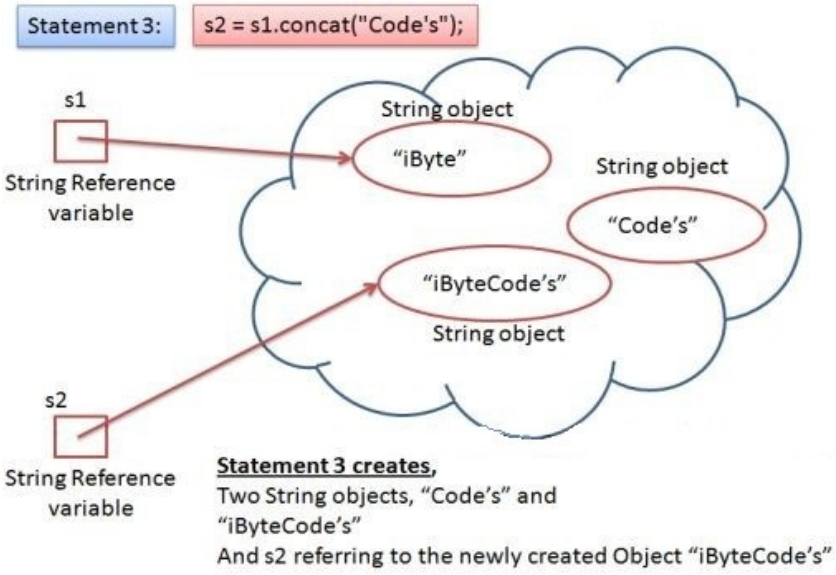
How many String objects and reference variables are created?

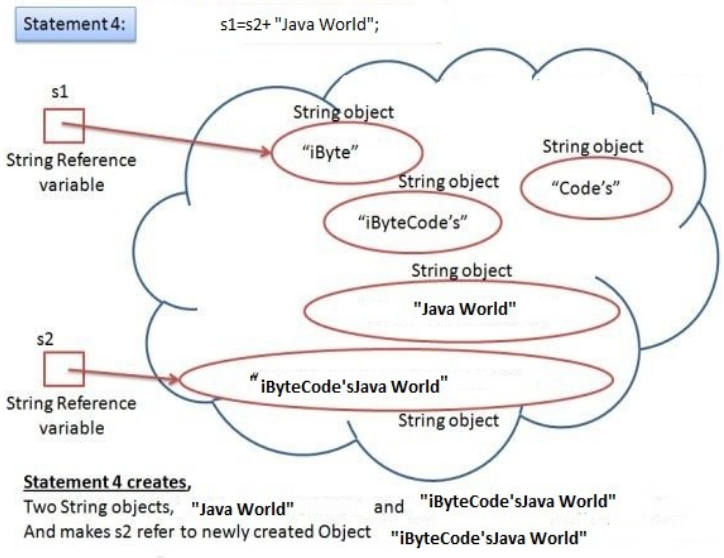
Two Reference Variables: s1 and s2

Five String Objects/literals:“iByte”,“Code’s”,“iByteCode’s”,“Java World”,“iByteCode’sJava World”



String reference variables have bit-pattern (kind of pointer but NOT pointer) defining how to get to the String Object in the memory





Totally two references variables and five objects are created.

Why Strings are Immutable:

Normally immutability in java is achieved through following steps :

a. Don't provide mutator methods for any field

b. Make all fields final and private

c. Don’t allow subclasses by declaring the class final itself

d. Return deep cloned objects with copied content for all mutable fields in class

Note:Immutable objects have a very compelling list of positive qualities, they are among the simplest and most robust kinds of classes you can possibly build. When you create immutable classes, entire categories of problems simply disappear.

Java also has its share of immutable classes which are primarily **String class and wrapper classes.** In this post, we will understand the need of immutability for String class.

1) Security : The first and undeniably most important reason is security. Well, its not only about your application, but even for JDK itself. Java class loading mechanism works on class names passed as parameters, then these classes are searched in class path. Imagine for a minute, Strings were mutable, then anybody could have injected its own class-loading mechanism with very little effort and destroyed or hacked in any application in a minute.

2) Performance : As already discussed Java uses string pool for achieving better performance of Strings. But at the same time if string is not immutable, changing the string with one reference will lead to the wrong value for the other references which could lead to erroneous results. Hence indirectly immutability is required to achieve better performance.

3) Thread safety: The word thread safe, means multiple threads trying to change the properties of a single object, To make this happen with String(which is immutable) is highly impossible. You can't change the properties of string object once it is created. If you want to do any operations over String, it will create a new string. So strings are thread safe(Immutable objects are safe when shared between multiple threads in mulch-threaded applications).

1.3.6 String Functions:

The following methods are some of the most commonly used methods of String class:

a. **charAt(int index)** :It

a. returns the character at the specified index.

b. An index ranges from 0 to length() - 1.

c.The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

Example: String str = "Strings are Immutable";

System.out.println(str.charAt(2));

Output: r

b. **equalsIgnoreCase()** : determines the equality of two Strings, ignoring thier case (upper or lower case doesn't matters with this function).

Example: String str = "java";

System.out.println(str.equalsIgnoreCase("JAVA"));

Output: true

c. **length() :** function returns the number of characters in a String.

Example : String str = "Count me";

System.out.println(str.length());

Output : 8

d. **replace**(char oldChar, char newChar): returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

Example : String str = "Change me";

System.out.println(str.replace('m','M'));

Output: Change Me

e. **substring():** method returns a part of the string. **substring()** method has two forms,

public String substring(int begin);

public String substring(int begin, int end);

The first argument represents the starting point of the subtring. If the substring() method is called with only one argument, the subtring returned, will contain characters from specified starting point to the end of original string.

But, if the call to substring() method has two arguments, the second argument specify the end point of substring.

Note:If end parameter is missing, then the extraction starts from the starting position to the rest of the string.

Example : String str = "0123456789";

```
System.out.println(str.substring(4));
```

Output: 456789

Example-1 : System.out.println(str.substring(4,7));

Output : 456

f.**toLowerCase()** : method returns string with all uppercase characters converted to lowercase.

Example : String str = "ABCDEF";

```
System.out.println(str.toLowerCase());
```

Output: abcdef

g. **valueOf() :** function is used to convert **primitive data types** into Strings. Overloaded version of valueOf() method is present in String class for all primitive data types and for type Object

But for objects, valueOf() method calls **toString()** function.

*The java **toString()** method is used when we need a string representation of an object. It is defined in Object class. This method can be overridden to customize the String representation of the Object.*

Example :

```
public class Car {
    public static void main(String args[])
{
    Car c=new Car();
    System.out.println(c);
    }
   public String toString()
{
    return "This is my car object";
    }
}
```

Output : This is my car object

Whenever we will try to print any object of class Car, its toString() function will be called. toString() can also be used with normal string objects.

Example :String str = "Hello World";

System.out.println(str.toString());

Output : Hello World

h. **toString() with Concatenation**: Whenever we concatenate any other primitive data type, or object of other classes with a String object, **toString()** function or **valueOf()** function is called automatically to change the other object or primitive type into string, for successful concatenation.

Example: int age = 10;

String str = "He is" + age + "years old.";

In above case **10** will be automatically converted into string for concatenation using **valueOf()** function.

I. **toUpperCase()** : This method returns string with all lowercase character changed to uppercase.

Example : String str = "abcdef";

System.out.println(str.toLowerCase());

Output : ABCDEF

j. **trim()** : This method returns a string from which any leading and trailing white spaces has been removed.

Example: String str = " hello ";

System.out.println(str.trim());

Output : hello

k. equals() - The **equals()** method compares two objects for equality and returns **true** if they are equal and the result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

public boolean **equals**(Object anObject)

Example:

```
String obj1 = new String("xyz");
String obj2 = new String("xyz");
if(obj1.equals(obj2))
  System.out.printlln("obj1==obj2 is TRUE");
else
  System.out.println("obj1==obj2 is FALSE")
```

Output: obj1==obj2 is TRUE

"==" operator:

In Java, when the "==" operator is used to compare 2 objects, it checks to see if the objects refer to the same place in memory. In other words, it checks to see if the 2 object names are basically references to the same memory location. A very simple example will help clarify this:

Example:

```
String obj1 = new String("xyz");
String obj2 = new String("xyz");
if(obj1 == obj2)
  System.out.println("obj1==obj2 is TRUE");
else
  System.out.println("obj1==obj2 is FALSE");
```

Output: obj1==obj2 is FALSE

Conclusion: **== Operator** compares the references. Though the objects, obj1 and obj2 are same internally, they differ on using this operation as we compare references. **equals()** on the other hand compares the values. Hence, the comparison between obj1 and obj2 would pass.

l. compareTo() - Compares two strings lexicographically(dictionary order).

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.

public int **compareTo**(String anotherString)
public int **compareToIgnoreCase**(String str)

Example :Suppose s1 and s2 are two string variables. If:

s1 == s2 :0

s1 > s2 :positive value

s1 < s2 :negative value

```
class SimpleExample{

 public static void main(String args[]){

     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 )

 }

}
```

Output:0 1 -1

1.3.7 String Constructors:

String class provides many constructors to create a new String object from character array, byte array, String Buffer, String Builder, etc.

Constructor	Purpose
String()	Creates an empty string.
String(String)	Creates a string from the specified string.
String(char[])	Creates a string from an array of characters.
String(char[], int offset, int count)	Creates a string from the specified subset of characters in an array.
String (byte[] byte)	Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String (byte[] byte, int offset, int len)	Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length, String charsetName)	Constructs a new String by decoding the specified subarray of bytes using the specified charset.
String(byte[] bytes, String charsetName)	Constructs a new String by decoding the specified array of bytes using the specified charset.
String(StringBuffer)	Creates a string from StringBuffer argument.
String(StringBuilder)	Creates a string from StringBuilder argument.

1.4 Understanding StringBuffer and StringBuilder:

As explained earlier, Strings are immutable because String literals with same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side-effects to other Strings sharing the same storage.

JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package java.lang) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side-effect to other objects.

StringBuilder class was introduced in JDK 1.5. It is the same as StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations. However, for single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

1.StringBuffer : StringBuffer class is used to create a **mutable** string object. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with **String** objects, we will end up with a lot of memory leak.

So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously.

1.4.1 StringBuffer Constructors:

StringBuffer defines 4 constructors. They are

1.**StringBuffer()**:creates an empty string buffer and reserves room for 16 characters

2.**StringBuffer(int size)**: creates an empty string and takes an integer argument to set capacity of the buffer

3.**StringBuffer(String str):** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that takes place.

4.**StringBuffer(charSequence ch[]) :** Constructs a string buffer that contains the same characters as the specified CharSequence.

Example showing difference between String and StringBuffer

```
class Test {

 public static void main(String args[])

 {

  String str = "Java";
```

```
  str.concat("World");
  System.out.println(str);     // Output: Java
  StringBuffer strB = new StringBuffer("Java");
  strB.append("World");
  System.out.println(strB);   // Output: JavaWorld
 }
}
```

1.4.2 StringBuffer Functions:

The following methods are some most commonly used methods of StringBuffer class.

a. **append()** :This method will concatenate the string representation of any type of data to the end of the invoking **StringBuffer** object.

append() method has several overloaded forms.

StringBuffer append(int n)

StringBuffer append(String str)

StringBuffer append(Object obj)

The string representation of each parameter is appended to **StringBuffer** object.

Example:
```
StringBuffer str = new StringBuffer("test");
str.append(123);
System.out.println(str);
```

Output : test123

b. **insert()** :This method inserts one string into another. Here are few forms of insert() method.

StringBuffer insert(int n)

StringBuffer insert(String str)

StringBuffer insert(Object obj)

Here the first parameter gives the index at which position the string will be inserted and string representation of second parameter is inserted into **StringBuffer** object.

Example : StringBuffer str = new StringBuffer("test");

str.insert(4, 123);

System.out.println(str);

Output: test123

c. **reverse()** :This method reverses the characters within a **StringBuffer** object.

Example : StringBuffer str = new StringBuffer("Hello");

str.reverse();

System.out.println(str);

Output : olleH

d. **replace():** This method replaces the string from specified start index to the end index.

Example: StringBuffer str = new StringBuffer("Hello World");

str.replace(6, 11, "java");

System.out.println(str);

Output : Hello java

e. **capacity()** :This method returns the current capacity of StringBuffer object.

Example : StringBuffer str = new StringBuffer();

System.out.println(str.capacity());

Output: 16

f. **ensureCapacity()** : This method is used to ensure minimum capacity of **StringBuffer** object.

Example : StringBuffer str = new StringBuffer("hello");

str.ensureCapacity(10);

1.4.3 StringBuilder:

StringBuilder is identical to StringBuffer except for one important difference it is not synchronized, which means it is not thread safe. Its because StringBuilder methods are not synchronised.

StringBuffer class is a mutable class unlike the String class which is immutable. Both the capacity and character string of a StringBuffer Class. StringBuffer can be changed dynamically. String buffers are preferred when heavy modification of character strings is involved (appending, inserting, deleting, modifying etc).

In other words, if multiple threads are accessing a StringBuilder instance at the same time, its integrity cannot be guaranteed. However, for a single-thread program (most commonly), doing away with the overhead of synchronization makes the StringBuilder faster.

StringBuilder is API-compatible with the StringBuffer class, i.e., having the same set of constructors and methods, but with no guarantee of synchronization. It can be a drop-in replacement for StringBuffer under a single-thread environment.

1.4.4 StringBuilder Constructors:

I. **StringBuilder( )** Creates an empty string builder with a capacity of 16 (16 empty elements).

II. **StringBuilder(int size)** Creates an empty string builder with the specified initial capacity.

III. **StringBuilder (String str)** Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

IV. **StringBuilder(CharSequence cs)** Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.

1.4.5 StringBuilder Functions:

StringBuilder is API-compatible with the StringBuffer class, i.e., having the same set of constructors and methods, but with no guarantee of synchronization. It can be a drop-in replacement for StringBuffer under a single-thread environment.

The StringBuilder class has some methods related to length and capacity that the String class does not have:

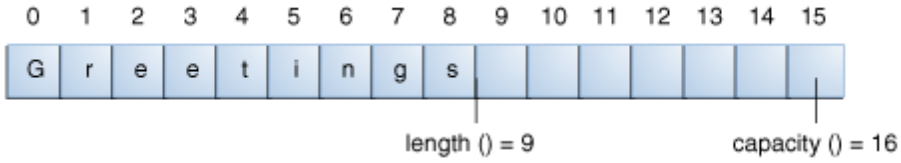
Method	Description
void setLength(int newLength)	Sets the length of the character sequence. If newLengthis less than length(), the last characters in the character sequence are truncated. If newLength is greater than length(), null characters are added at the end of the character sequence.
void ensureCapacity(int minCapacity)	Ensures that the capacity is at least equal to the specified minimum.

A number of operations (for example, append(), insert(), or setLength()) can increase the length of the character sequence in the string builder so that the resultant length() would be greater than the current capacity(). When this happens, the capacity is automatically increased.

For example, the following code

```
// creates empty builder, capacity 16
StringBuilder sb = new StringBuilder();
// adds 9 character string at beginning
sb.append("Greetings");
```

will produce a string builder with a length of 9 and a capacity of 16:



1.4.6 Difference between **StringBuffer** and **StringBuilder** class

StringBuffer Class	**StringBuffer Class**
StringBuffer is synchronized	StringBuilder is not synchronized
Because of synchronisation, StringBuffer operation is slower than StringBuilder	StringBuilder operates faster

Example:
```
public class StringBuilderDemo {
        public static void main(String[] args) {
            String palindrome = "Dot saw I was Tod";
            StringBuilder sb = new StringBuilder(palindrome);
            sb.reverse();  // reverse it
            System.out.println(sb);
    }
}
```

Output: doT saw I was toD

1.5 Choose your string class wisely:

Choosing the wrong string class can cause undesirable results. For example, you can easily append the letter X to a String object: String s = s + "X". But this concatenation operation inside a loop is inefficient. That is because the String class happens to create a StringBuilder (or StringBuffer) object to do the concatenation work. Creating this object every time is costly.

A better choice for this example is the StringBuilder class. For example, we can simply append the letter X to the StringBuilder object: sbd.append('X'). This is much faster than the previous example. In sample testing (200,000 iterations), the previous example took almost one minute to complete while StringBuilder took less than a second.

If you add a requirement for multiple threads, then StringBuffer becomes the better choice. But in sample testing (one million iterations), StringBuffer was five times slower than StringBuilder. However StringBuffer guarantees thread safety, whereas StringBuilder does not.

In other words, you can get unpredictable results when using StringBuilder objects in multithreaded programs.

1.6 Code Snippets:

Benchmarking String/StringBuffer/StringBuilder:

Try it Yourself

The following program compare the times taken to reverse a long string via a String object and a StringBuffer.

```
 // Reversing a long String via a String vs. a StringBuffer
public class StringsBenchMark {
  public static void main(String[] args) {
    long beginTime, elapsedTime;
    // Build a long string
    String str = "";
    int size = 16536;
    char ch = 'a';
    beginTime = System.nanoTime();   // Reference time in nanoseconds
    for (int count = 0; count < size; ++count) {
      str += ch;
      ++ch;
      if (ch > 'z') {
        ch = 'a';
```

```
        }
    }
    elapsedTime = System.nanoTime() - beginTime;
    System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Build String)");
// Reverse a String by creating a StringBuffer with the given String and invoke its reverse()
    beginTime = System.nanoTime();
    StringBuffer sBufferReverseMethod = new StringBuffer(str);
    sBufferReverseMethod.reverse();     // use reverse() method
    elapsedTime = System.nanoTime() - beginTime;
      System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuffer's reverse() method)");


// Reverse a String by creating a StringBuilder with the given String and invoke its reverse()
    beginTime = System.nanoTime();
    StringBuffer sBuilderReverseMethod = new StringBuffer(str);
    sBuilderReverseMethod.reverse();
    elapsedTime = System.nanoTime() - beginTime;
     System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuidler's reverse() method)");
  }
}
```

Output: Elapsed Time is 332100 usec (Build String)

Elapsed Time is 847 usec (Using StringBuffer's reverse() method)

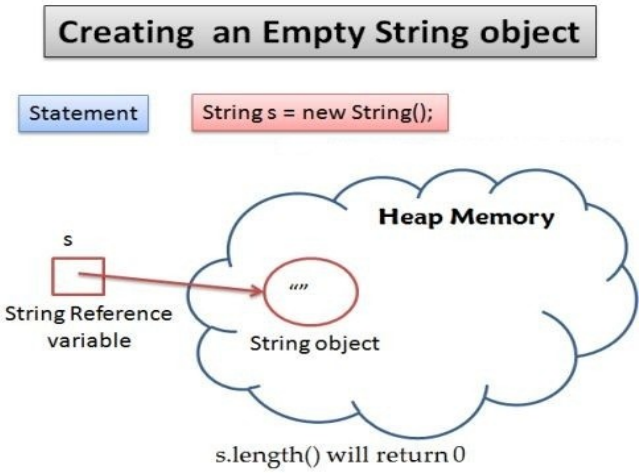
Elapsed Time is 836 usec (Using StringBuidler's reverse() method)

Try it Yourself

I. Creating Empty String:

```
public class CreatingStringsUsingConstrutcor {
    public static void main(String[] args) {
        String s = new String();
        System.out.println("Length = " + s.length()); } }
```

Output: Length=0



II. Creating a String object with value:You can create a new String object that contains the same character sequence as another String object

```
String str = new String(" Java World");

System.out.println("str = " + str);
```

Output: Java World

III. Char array to String:You can create a String object from character array. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

```
public class CreatingStringsUsingConstrutcor {

   public static void main(String[] args) {

      char chars[] = { 'i', 'B', 'y', 't', 'e', 'C', 'o', 'd', 'e'};

      String s2 = new String(chars);

      String s3 = new String(chars, 1, 4);

      System.out.println("s2 = " + s2);

      System.out.println("s3 = " + s3);

   }

}
```

Output:

```
s2 = iByteCode

s3 = Byte
```

IV. byte array to String:You can create a String object from byte array.

```
byte byteAscii[] = {65, 66, 67, 68, 69, 70 };
String strAscii = new String(byteAscii);
System.out.println("strAscii = " + strAscii);
```

Output: strAscii = ABCDEF

V. Passing byte as parameter for String creation :

Scenario I:byte bytes[] = { 'w', 'o', 'r', 'l', 'd' };

```
String s4 = new String(bytes);
System.out.println("s4 = " + s4);
```

s4 = world

Output: s4=world

VI. Scenario II:byte bytes[] = { 'w', 'o', 'r', 'l', 'd' };

```
String s5 = new String(bytes, 1, 3);
System.out.println("s5 = " + s5);
```

Output :s5 = orl

VII. Scenario III byte bytes[] = { 'w', 'o', 'r', 'l', 'd' };

```
try {
   String s7 = new String(bytes, 1, 3, "UTF-8");
   System.out.println("s7 = " + s7);
```

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

Output: s7 = orl

VIII. Constructing String from StringBuffer

You can create a new String from a string buffer. The contents of the string buffer are copied; subsequent modification of the string buffer does not affect the newly created string.

```
StringBuffer sb = new StringBuffer("Welcome to Java World");
String s8 = new String(sb);
System.out.println("s8 = " + s8);
```

Output : s8 = Welcome to Java World

IX. Constructing String from StringBuilder

You can create a new String from a StringBuilder. The contents of the StringBuilder are copied to String and subsequent modification of the string builder does not affect the newly created string.

```
    StringBuilder builder = new StringBuilder("Welcome to Java World");
  String s9 = new String(builder);
 System.out.println("s9 = " + s9);
```

Output : s9 = Welcome to Java World

1.7 Check Your Self:

1. What is an immutable object ?

A. an object whose state can be changed after it is created

B. an object whose state cannot be changed after it is created.

C.an object which cannot be casted to another type.

D. an object which cannot be cloned.

2. Individual character inside a String is accessed by using the characterAt method?

A. True B. False

3.The following code will assign the String "are" to str2.

String str = "How are you?";

String str2 = str.substring(5, 7);

A. True B. False

4.The following two sets of code will assign the same value to str2.

(i)String str = "Hello";

String str2 = str + "Java";

(ii)StringBuffer strBuf = new StringBuffer("Hello");

String str2 = strBuf.append("Java").toString( );

A. True B. False

5.Which of the following is false?

A. String is a primitive data type.

B. char is a primitive data type.

C. The charAt method returns a char value.

D. StringBuffer is not a primitive data type.

6.Which of the following is not a valid way to create String object ?

A. String str2 = new String(new char[]{'a','b','c'});

B. String str = new String("abc");

C. String str1 = "abc";

D. String str3 = 'a'+'b'+'c';

7.String objects are stored in a special memory area known as ?

A. Heap B. Stack C.String Constant Pool D. Method Area

8.Which method is used to remove leading and trailing whitespaces ?

A. substring() B. replace() C. removeSpace() D. trim()

9.Which class is used to create mutable and non-synchronized string ?

A. StringBuffer B. StringBuilder C. String D. StringTokenizer

10.What will be the output of following code ?

```
String str = "Java was developed by James Ghosling";
System.out.println(str.substring(19));
```

1.8 Additional Information:

The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner. StringTokenizer implements the Enumeration interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using StringTokenizer.

To use StringTokenizer, you specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, ",;:" sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.

StringTokenizer Constructors:

I. **StringTokenizer(String , String, boolean)** Constructs a StringTokenizer on the specified String, using the specified delimiter set.

II. **StringTokenizer(String, String)** Constructs a StringTokenizer on the specified String, using the specified delimiter set.

III. **StringTokenizer(String)** Constructs a StringTokenizer on the specified String, using the default delimiter set (which is " \t\n\r").

Following are the methods available in StringTokenizer Class

a.countTokens() Returns the next number of tokens in the String using the current deliminter set.

b.hasMoreElements() Returns true if the Enumeration has more elements.

c. hasMoreTokens() Returns true if more tokens exist.

d.nextElement() Returns the next element in the Enumeration.

e. nextToken() Returns the next token of the String.

f. nextToken(String) Returns the next token, after switching to the new delimiter set.

Example usage:

```
      String s = "this is a test";
StringTokenizer st = new StringTokenizer(s);

while (st.hasMoreTokens()) {

      println(st.nextToken());

}
```

Prints the following on the console:

this

is

a

test

Note: StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split() method of String or the java.util.regex package instead.

1.9 Best Practice's :

- Create strings as literals instead of creating String objects using 'new' key word whenever possible

- + operator gives best performance for String concatenation if Strings resolve at compile time

- StringBuffer with proper initial size gives best performance for String concatenation if Strings resolve at run time.

- When performing String transformation operations such as removing, inserting, replacing or appending characters, concatenating or splitting Strings use either the StringBuilder or the StringBuffer class. The StringBuilder class is introduced in Java 1.5 and is the non–synchronized counterpart of the StringBuffer class. Thus if only one Thread will be performing the String transformation operations then favor the StringBuilder class because is the best performer

- The Java String, StringBuilder and StringBuffer classes are not interchangeable. Pick the string class that addresses your needs, or else your programs will be inefficient or even incorrect.

- Favor the creation of literal strings and string–valued constant expressions rather than creating new String Objects using one of the String constructor methods

- Utilizing character arrays to perform String transformation operations yields the best performance results but is the less flexible approach

1.10 Take Away Points:

- The String class is commonly used for holding and manipulating strings of text in Java programs. It is found in the standard java.lang package which is automatically imported, so you don't need to do anything special to use it.

- The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

- Strings are more efficient if they are not modified (because they are shared in the string common pool). However, if you have to modify the content of a string frequently (such as a status message), you should use the StringBuffer class (or the StringBuilder described below) instead.

- For performance reason, Java's String is designed to be in between a primitive and a class.

- String is immutable, it is not efficient to use String if you need to modify your string frequently

- String class provides many constructors to create a new String object from character array, byte array, StringBuffer, StringBuilder, etc.
- StringBuffer class is used to create a mutable string object.
- StringBuffer class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously.
- StringBuilder is not synchronized

1.11 Related Topics:

StringTokenizer

1.12 References:

http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String
http://docs.oracle.com/cd/A97688_16/generic.903/bp/java.htm#1007796