STRINGS

Introduction to String Handling

String is probably the most commonly used class in java library. String class is encapsulated under `java.lang` package. In java, every string that you create is actually an object of type **String**.

One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper class's objects are immutable.

Creating an Immutable class

```
public final class MyString

{

 final String str;

 MyString(String s)

 {

  this.str = s;

 }

 public String get()

 {

  return str;

 }

}
```

In this example **MyString** is an immutable class. **MyString**'s state cannot be changed once it is created.

Creating a String object

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)

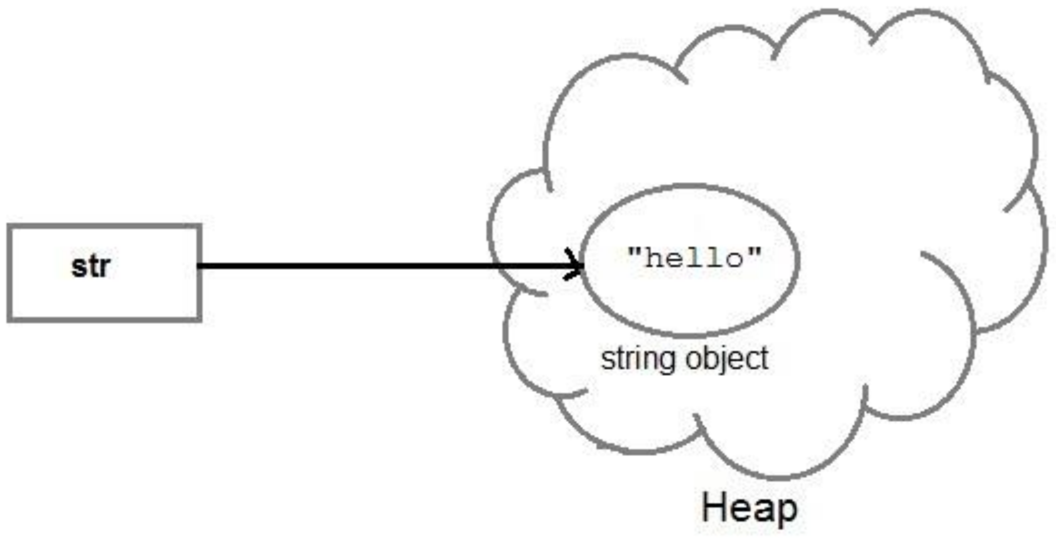
```
String str4 = str1 + str2;

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 object and How they are stored

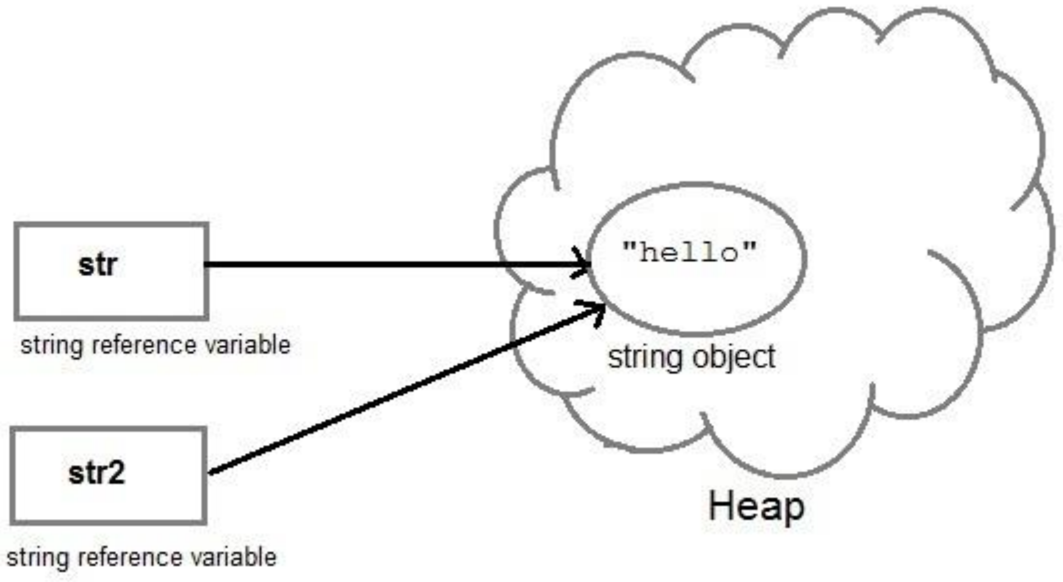
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

```
String str= "Hello";
```



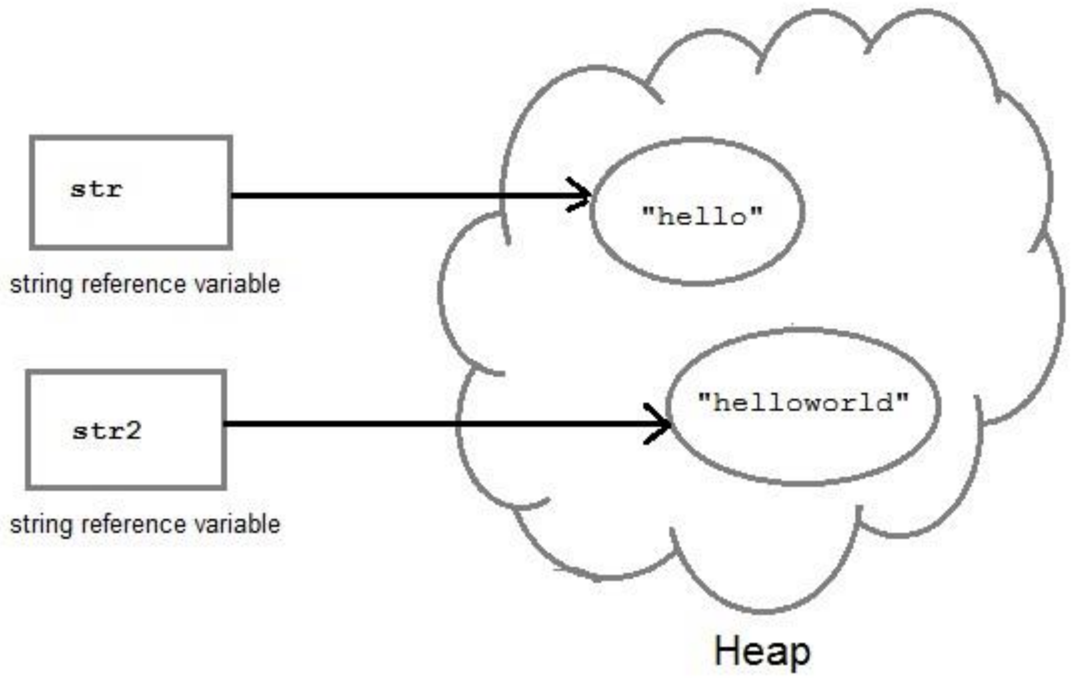
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

```
String str2=str;
```



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

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



Concatenating String

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using `+` operator

1) Using concat() method

```
string s = "Hello";

string str = "Java";

string str2 = s.concat(str);

String str1 = "Hello".concat("Java");     //works with string literals too.
```

2) Using + operator

```
string str = "Rahul";

string str1 = "Dravid";

string str2 = str + str1;

string st = "Rahul"+"Dravid";
```

String Comparison

String comparison can be done in 3 ways.

1. Using **equals()** method
2. Using `==` operator
3. By **CompareTo()** method

Using equals() method

equals() method compares two strings for equality. Its general syntax is,

```
boolean equals (Object str)
```

It compares the content of the strings. It will return **true** if string matches, else returns **false**.

```
String s = "Hell";

String s1 = "Hello";

String s2 = "Hello";

s1.equals(s2);     //true

s.equals(s1) ;    //false
```

Using == operator

`==` operator compares two object references to check whether they refer to same instance. This also, will return **true** on successful match.

```
String s1 = "Java";

String s2 = "Java";
String s3 = new string ("Java");
test(Sl == s2)     //true
test(s1 == s3)      //false
```

By compareTo() method

compareTo() method compares values and returns an int which tells if the string compared is less than, equal to or greater than th other string. Its general syntax is,

```
int compareTo(String str)
```

To use this function you must implement the **Comparable Interface**. compareTo() is the only function in Comparable Interface.

```
String s1 = "Abhi";
String s2 = "Viraaj";
String s3 = "Abhi";
s1.compareTo(S2);     //return -1 because s1 < s2
s1.compareTo(S3);     //return 0 because s1 == s3
s2.compareTo(s1);     //return 1 because s2 > s1
```

String class function

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

charAt()

charAt() function returns the character located at the specified index.

```
String str = "studytonight";
System.out.println(str.charAt(2));
```

```
Output : u
```

---

equalsIgnoreCase()

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

```
String str = "java";

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

```
Output : true
```

---

length()

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

```
String str = "Count me";

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

```
Output : 8
```

---

replace()

replace() method replaces occurances of character with a specified new character.

```
String str = "Change me";

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

```
Output : Change Me
```

---

substring()

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.

```
String str = "0123456789";

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

```
Output : 456789
```

```
System.out.println(str.substring(4,7));
```

```
Output : 456
```

---

toLowerCase()

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

```
String str = "ABCDEF";

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

```
Output : abcdef
```

---

valueOf()

Overloaded version of valueOf() method is present in String class for all primitive data types and for type Object.

NOTE : `valueOf()` function is used to convert **primitive data types** into Strings.

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

---

toString()

toString() method returns the string representation of the object used to invoke this method. **toString()** is used to represent any Java Object into a meaningful string representation. It is declared in the **Object class**, hence can be overrided by any java class. (Object class is super class of all java classes.)

```
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.

```
String str = "Hello World";
System.out.println(str.toString());
```

```
Output : Hello World
```

---

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.

```
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.

---

toUpperCase()

This method returns string with all lowercase character changed to uppercase.

```
String str = "abcdef";

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

```
Output : ABCDEF
```

---

trim()

This method returns a string from which any leading and trailing whitespaces has been removed.

```
String str = "   hello  ";

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

```
Output : hello
```

StringBuffer class

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. StringBuffer defines 4 constructors. They are,

1. **StringBuffer** ( )
2. **StringBuffer** (*int size*)
3. **StringBuffer** (*String str*)
4. **StringBuffer** (*charSequence [ ]ch*)

- `StringBuffer()` creates an empty string buffer and reserves room for 16 characters.
- `stringBuffer(int size)` creates an empty string and takes an integer argument to set capacity of the buffer.

Example showing difference between String and StringBuffer

```
class Test {
 public static void main(String args[])
 {
  String str = "study";
  str.concat("tonight");
  System.out.println(str);      // Output: study

  StringBuffer strB = new StringBuffer("study");
  strB.append("tonight");
  System.out.println(strB);    // Output: studytonight
 }
}
```

Important methods of StringBuffer class

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

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(String str)

StringBuffer append(int n)

StringBuffer append(Object obj)
```

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

```
StringBuffer str = new StringBuffer("test");
str.append(123);
```

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

```
Output : test123
```

---

insert()

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

```
StringBuffer insert(int index, String str)


StringBuffer insert(int index, int num)


StringBuffer insert(int index, 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.

```
StringBuffer str = new StringBuffer("test");

str.insert(4, 123);

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

```
Output : test123
```

---

reverse()

This method reverses the characters within a **StringBuffer** object.

```
StringBuffer str = new StringBuffer("Hello");

str.reverse();

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

```
Output : olleH
```

---

replace()

This method replaces the string from specified start index to the end index.

```
StringBuffer str = new StringBuffer("Hello World");

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

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

```
Output : Hello java
```

---

capacity()

This method returns the current capacity of **StringBuffer** object.

```
StringBuffer str = new StringBuffer();

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

```
Output : 16
```

---

ensureCapacity()

This method is used to ensure minimum capacity of **StringBuffer** object.

```
StringBuffer str = new StringBuffer("hello");

str.ensureCapacity(10);
```

StringBuilder class

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.

StringBuilder Constructors

1. **StringBuilder** ( ), creates an empty StringBuilder and reserves room for 16 characters.
2. **StringBuilder** (*int size*), create an empty string and takes an integer argument to set capacity of the buffer.
3. **StringBuilder** (*String str*), create a StringBuilder object and initialize it with string str.

Difference between StringBuffer and StringBuilder class

StringBuffer class	StringBuilder class
StringBuffer is synchronized.	StringBuilder is not synchronized.
Because of synchronisation, StringBuffer operation is slower than StringBuilder.	StringBuilder operates faster.

Example of StringBuilder

```
class Test {
 public static void main(String args[])
 {
  StringBuilder str = new StringBuilder("study");
  str.append( "tonight" );
  System.out.println(str);
  str.replace( 6, 13, "today");
  System.out.println(str);
  str.reverse();
  System.out.println(str);
  str.replace( 6, 13, "today");
 }
}
```

Output :

```
studytonight
studyttoday
yadottyduts
```