java.lang Package

1.. Introduction

The following are some of important classes present in java.lang package:

- Object class
- String class
- StringBuffer class
- StringBuilder class (1.5 v)
- Wrapper Classes
- Autoboxing and Auto unboxing (1.5 v)

For writing any java program the most commonly required classes and interfaces are encapsulated in the separate package which is nothing but java.lang package. It is not required to import java.lang package in our program because it is available by default to every java program.

2.. Object Class

For any java object whether it is predefined or customized the most commonly required methods are encapsulated into a separate class which is nothing but **Object Class**. As Object Class acts as a root (or) parent (or) super class for all java classes, by default its methods are available to every java class.

Note: If our class doesn't extends any other class then it is the direct child class of object. If our class extends any other class then it is the indirect child class of Object.

2.1 Methods Present in Object Class

The following is the list of all methods present in Object class:

2.1.1 public String toString();

We can use this method to get string representation of an object. Whenever we are try to print any object reference internally toString() method will be executed. If our class doesn't contain toString() method then Object class toString() method will be executed.

```
class Student {
    String name;
    int rollNumber;
    Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }

    public static void main(String[] args) {
        Student s1 = new Student("Vikash", 101);
        Student s2 = new Student("Akash", 102);
        System.out.println(s1);
        System.out.println(s1.toString());
        System.out.println(s2);
        System.out.println(s2.toString());
    }
}
```

```
java -cp /tmp/LbMhOxeeb8 Student
Student@379619aa
Student@379619aa
Student@123a439b
Student@123a439b
```

In the above program Object class toString() method got executed which is implemented as follows.

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
//Here getClass().getName() => classname@hexa_decimal_String_representation_of_hashCode
```

To provide our own String representation we have to override toString() method in our class. Example: For example whenever we are try to print student reference to print his a name and roll no we have to override toString() method as follows.

```
class Student {
    String name;
    int rollNumber;
    Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }
    public String toString() {
        return this.name + " ---> " + this.rollNumber;
    }
    public static void main(String[] args) {
        Student s1 = new Student("Vikash", 101);
        Student s2 = new Student("Akash", 102);
        System.out.println(s1);
        System.out.println(s1.toString());
        System.out.println(s2);
        System.out.println(s2.toString());
    }
}
```

```
java -cp /tmp/LbMhOxeeb8 Student
Vikash ---> 101
Vikash ---> 101
Akash ---> 102
Akash ---> 102
```

In String class, StringBuffer, StringBuilder, wrapper classes and in all collection classes toString() method is overridden for meaningful string representation. Hence in our classes also highly recommended to override toString() method.

2.1.2 public native int hashCode();

For every object JVM will generate a unique number which is nothing but **hashCode**. Jvm will use this hashCode while saving objects into hashing related data structures like HashSet, HashMap, and Hashtable etc. If the objects are stored according to hashCode searching will become very efficient (The most powerful search algorithm is hashing which will work based on hashCode).

If we didn't override hashCode() method then Object class hashCode() method will be executed which generates hashCode based on address of the object but it doesn't mean hashCode represents address of the object. Based on our programming requirement we can override hashCode() method to generate our own hashCode. Overriding hashCode() method is said to be proper if and only if for every object we have to generate a unique number as hashCode for every object.

```
class Student
{
public int hashCode()
{
return 100;
}
}
```

It is *improper way* of overriding hashCode() method because for every object we are generating same hashcode.

```
class Student
{
int rollno;
public int hashCode()
{
return rollno;
}
}
```

It is *proper way* of overriding hashcode() method because for every object we are generating a different hashcode.

```
class Test
{
int i;
Test(int i)
{
this.i=i;
}
public static void main(String[]
args){
Test t1=new Test(10);
Test t2=new Test(100);
System.out.println(t1);
System.out.println(t2);
}
}
Object==>toString() called.
Object==>hashCode() called.
```

In this case *Object class toString( )* method got executed which is internally calls *Object class hashCode( )* method.

```
class Test{
int i;
Test(int i){
this.i=i;
}
public int hashCode(){
return i;
}
public static void main(String[]
args){
Test t1=new Test(10);
Test t2=new Test(100);
System.out.println(t1);
System.out.println(t2);
}}
Object==>toString() called.
Test==>hashCode() called.
```

In this case *Object class toString( )* method got executed which is internally calls *Test class hashCode( )* method.

Note: If we are giving opportunity to Object class toString() method it internally calls hashCode() method. But if we are overriding toString() method it may not call hashCode() method. We can use toString() method while printing object references and we can use hashCode() method while saving objects into HashSet or Hashtable or HashMap.

2.1.3 public boolean equals(Object o);

We can use this method to check equivalence of two objects. If our class doesn't contain equals() method then object class equals() method will be executed which is always meant for reference comparison [address comparison]. i.e., if two references pointing to the same object then only equals() method returns true .

```java
class Student {
    String name;
    int rollNumber;
    Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }
    public String toString() {
        return this.name + " ---> " + this.rollNumber;
    }
    public static void main(String[] args) {
        Student s1 = new Student("Vikash", 101);
        Student s2 = new Student("Akash", 102);
        System.out.println(s1.equals(s2));
    }
}
```

```
java -cp /tmp/LbMhOxeeb8 Student
false
```

In the above program Object class equals() method got executed which is always meant for reference comparison that is if two references pointing to the same object then only equals(() method returns true. In object class equals() method is implemented as follows which is meant for reference comparison.

```java
public boolean equals(Object obj) {
  return (this == obj);
    }
```

Based on our programming requirement we can override equals() method for content comparison purpose. Whenever we are overriding equals() method we have to consider the following things:

- Meaning of content comparison i.e., whether we have to check the names are equal (or) roll numbers (or) both are equal.
- If we are passing different type of objects (heterogeneous object) our equals() method should return false but not **ClassCastException** i.e., we have to handle **ClassCastException** to return false.
- If we are passing null argument our equals() method should return false but not **NullPointerException** i.e., we have to handle **NullPointerException** to return false.

The following is the proper way of overriding equals() method for content comparison in Student class.

```java
class Student {
    String name;
    int rollNumber;
    Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }
    public boolean equals(Object O) {
        try {
            String name1 = this.name;
            int rollNumber1 = this.rollNumber;
            Student s2 = (Student)O;
            String name2 = s2.name;
            int rollNumber2 = s2.rollNumber;
            if(name1.equals(name2) && rollNumber1 == rollNumber2)
                return true;
            else
                return false;
        }catch(ClassCastException ex1) {
            return false;
        } catch(NullPointerException ex2) {
            return false;
        }
    }
    public static void main(String[] args) {
        Student s1 = new Student("Vikash", 101);
        Student s2 = new Student("Akash", 102);
        Student s3 = new Student("Vikash", 101);
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s1.equals("Vikash"));
        System.out.println(s1.equals(null));
    }
}
```

```
java -cp /tmp/ZOULloqepW Student
false
true
false
false
```

Simplified version of .equals() method:

```
public boolean equals(Object o){
try{
  Student s2=(Student)o;
  if(name.equals(s2.name) && rollno==s2.rollno){
    return true;
  }
   else return false;
}
catch(ClassCastException e) {
   return false;
 }
catch(NullPointerException e) {
  return false;
 }
}
```

More simplified version of .equals() method :

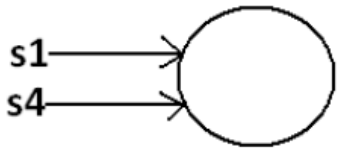
```
public boolean equals(Object o)  {
  if(this==o)
  return true;
 if(o instanceof Student)  {
     Student s2=(Student)o;
  if(name.equals(s2.name) && rollno==s2.rollno)
     return true;
  else
     return false;
   }
  return false;
}
```

To make equals() method more efficient we have to place the following code at the top inside equals() method.

```
if(this==o)
return true;
```

If two references pointing to the same object then equals() method return true directly without performing any content comparison this approach improves performance of the system.

Student s1=new Student("vijayabhaskar",101);
Student s4=s1;



`String s1 = new String("ashok");`
`String s2 = new String("ashok");`
`System.out.println(s1==s2);`
`//false`
`System.out.println(s1.equals(s2) );`
`//true`
	`StringBuffer s1 = new`
	`StringBuffer("ashok");`
	`StringBuffer s2 = new`
	`StringBuffer("ashok");`
	`System.out.println(s1==s2);  //false`
	`System.out.println(s1.equals(s2) );`
	`//false`

In String class .equals( ) is overridden for content comparision hence if content is same .equals( ) method returns true , even though ths objects are different.	**In StringBuffer class .equals( ) is not overriden for content comparision hence Object class .equals( ) will be executed which is meant for reference comparision , hence if objects are different .equals( ) method returns false , even though content is same.**

Note: In String class, Wrapper classes and all collection classes equals( ) method is overridden for content Comparision.

Relationship between == Operator and equals() method

- If r1==r2 is true then r1.equals(r2) is always true i.e., if two objects are equal by == operator then these objects are always equal by .equals( ) method also.
- If r1==r2 is false then we can't conclude anything about r1.equals(r2) it may return true (or) false.
- If r1.equals(r2) is true then we can't conclude anything about r1==r2 it may returns true (or) false.
- If r1.equals(r2) is false then r1==r2 is always false.

Differences between == Operator and equals() method

== (double equal operator)	.equals() method
It is an operator applicable for both primitives and object references.	**It is a method applicable only for object references but not for primitives.**
In the case of primitives == (double equal operator) meant for content comparison, but in the case of object references == operator meant for reference comparison.	**By default .equals() method present in object class is also meant for reference comparison.**
We can't override== operator for content comparison in object references.	**We can override .equals() method for content comparison.**
If there is no relationship between argument types then we will get compile time error saying incompatible types.(relation means child to parent or parent to child or same type)	**If there is no relationship between argument types then .equals() method simply returns false and we won't get any compile time error and runtime error.**
For any object reference r, r==null is always false.	**For any object reference r, r.equals(null) is also returns false.**

```
String s = new String("ashok");
StringBuffer sb = new StringBuffer("ashok");
System.out.println(s == sb); // CE : incomparable types : String and
StringBuffer
System.out.println(s.equals(sb));    //false
```

Contract between equals() method and hashCode() method

- If two objects are equal by equals() method compulsory their hashCodes must be equal (or) same. That is If r1.equals(r2) is true then r1.hascode()==r2.hashcode( ) must be true.
- If two objects are not equal by equals() method then there are no restrictions on hashCode() methods. They may be same (or) may be different. That is If r1.equals(r2) is false then r1.hashCode()==r2.hashCode() may be same (or) may be different.
- If hashCodes of 2 objects are equal we can't conclude anything about .equals() method it may returns true (or) false. That is If r1.hashCode()==r2.hashCode() is true then r1.equals(r2) method may returns true (or) false.
- If hashCodes of 2 objects are not equal then these objects are always not equal by equals() method also. That is If r1.hashCode()==r2.hashCode() is false then r1.equals(r2) is always false.

To maintain the above contract between equals() and hashCode() methods whenever we are overriding .equals() method compulsory we should override hashCode() method. Violation leads to no compile time error and runtime error but it is not good programming practice.

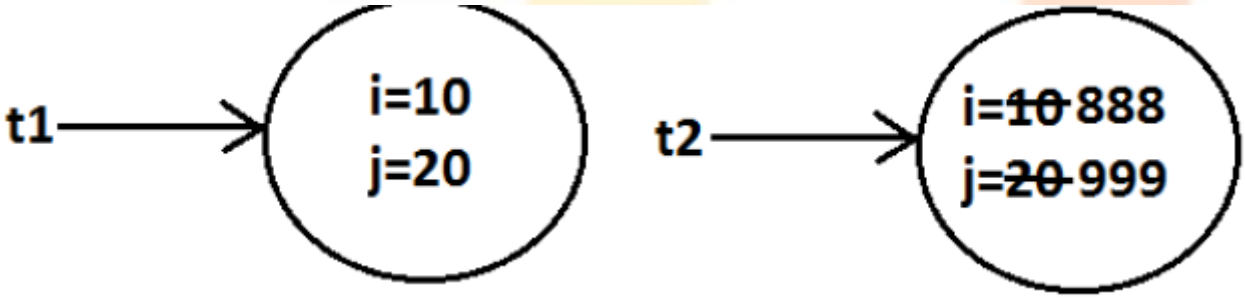
Based on whatever the parameters we override "equals() method" we should use same parameters while overriding hashCode() method also.

2.1.4 protected native Object clone()throws CloneNotSupportedException;

The process of creating exactly duplicate object is called **cloning**. The main objective of cloning is to maintain backup purposes. (i.e., if something goes wrong we can recover the situation by using backup copy.) We can perform cloning by using clone() method of Object class.

```
1  class Test implements Cloneable {
2      int i = 10;
3      int j = 20;
4
5      public static void main(String[] args) throws CloneNotSupportedException {
6          Test t1 = new Test();
7          Test t2 = (Test)t1.clone();
8          t2.i = 888;
9          t2.j = 999;
10         System.out.println(t1.i + " ----> " + t1.j);
11         System.out.println(t2.i + " ----> " + t2.j);
12     }
13 }
```

```
java -cp /tmp/ZOULloqepW Test
10 ----> 20
888 ----> 999
```



We can perform cloning only for **Cloneable** objects. An object is said to be **Cloneable** if and only if the corresponding class implements **Cloneable** interface. **Cloneable** interface present in **java.lang** package and does not contain any methods. It is a marker interface where the required ability will be provided automatically by the JVM. If we are trying to perform cloning or non-cloneable objects then we will get RuntimeException saying **CloneNotSupportedException**.

Cloning or Copy are of two Types:

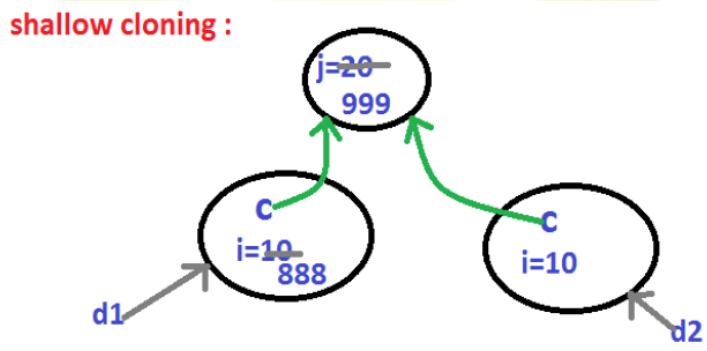
- Shallow Cloning or Shallow Copy
- Deep Cloning or Deep Copy

Shallow Cloning

The process of creating bitwise copy of an object is called **Shallow Cloning**. If the main object contains any primitive variable, then exactly duplicate copies will be created in cloned object. If the main object contains any reference variable, then the corresponding object won't be created just reference variable will be created by pointing to old contained object. By using main object reference if we perform any change to the contained object then those changes will be reflected automatically to the cloned object, by default Object class clone() meant for Shallow Cloning.

```
1  class Cat {
2      int j;
3      Cat(int j) {
4          this.j = j;
5      }
6  }
7  class Dog implements Cloneable {
8      Cat c;
9      int i;
10     Dog(Cat c, int i) {
11         this.c = c;
12         this.i = i;
13     }
14     public Object clone() throws CloneNotSupportedException {
15         return super.clone();
16     }
17 }
18 class ShallowClone {
19     public static void main(String[] args) throws CloneNotSupportedException {
20        Cat c = new Cat(20);
21        Dog d1 = new Dog(c,10);
22        System.out.println(d1.i + " ----> " + d1.c.j);
23        Dog d2 = (Dog)d1.clone();
24        d2.i = 888;
25        d2.c.j = 999;
26        System.out.println(d1.i + " ----> " + d1.c.j);
27        System.out.println(d2.i + " ----> " + d2.c.j);
28     }
29 }
```

```
java -cp /tmp/ZOULloqepW ShallowClone
10 ----> 20
10 ----> 999
888 ----> 999
```



Shallow cloning is the best choice, if the Object contains only primitive values. In Shallow cloning by using main object reference, if we perform any change to the contained object then those changes will be reflected automatically in cloned copy. To overcome this problem we should go for **Deep cloning**.

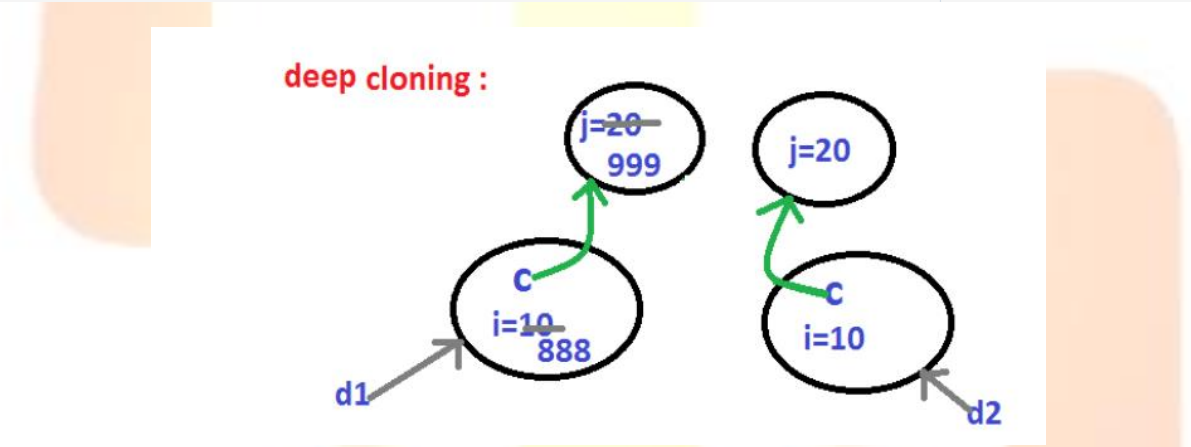
Deep Cloning

The process of creating exactly independent duplicate object (including contained objects also) is called deep cloning. In Deep cloning, if main object contain any reference variable then the corresponding Object copy will also be created in cloned object. Object class clone() method

meant for Shallow Cloning , if we want Deep cloning then the programmer is responsible to implement by overriding clone() method.

```
class Cat {
    int j;
    Cat(int j) {
        this.j = j;
    }
}
class Dog implements Cloneable {
    Cat c;
    int i;
    Dog(Cat c, int i) {
        this.c = c;
        this.i = i;
    }
    public Object clone() throws CloneNotSupportedException {
        Cat c1 = new Cat(c.j);
        Dog d1 = new Dog(c1,i);
        return d1;
    }
}
class DeepClone {
    public static void main(String[] args) throws CloneNotSupportedException {
       Cat c = new Cat(20);
       Dog d1 = new Dog(c,10);
       System.out.println(d1.i + " ----> " + d1.c.j);
       Dog d2 = (Dog)d1.clone();
       d2.i = 888;
       d2.c.j = 999;
       System.out.println(d1.i + " ----> " + d1.c.j);
       System.out.println(d2.i + " ----> " + d2.c.j);
    }
}
```

```
java -cp /tmp/ZOULloqepW DeepClone
10 ----> 20
10 ----> 20
888 ----> 999
```



In Deep cloning by using main Object reference if we perform any change to the contained Object those changes won't be reflected to the cloned object.

2.1.5 public final Class getClass();

This method returns runtime class definition of an object.

```
class GetClassDemo {
    public static void main(String[] args) throws CloneNotSupportedException {
       Object o = new String("Vikash");
       System.out.println(o.getClass().getName());
    }
}
```

```
java -cp /tmp/ZOULloqepW GetClassDemo
java.lang.String
```

2.1.6 protected void finalize()throws Throwable;

Just before destroying an object GC calls finalize( ) method to perform Clean Up activities. More about this method is discussed in Garbage Collection Chapter.

2.1.7 public final void wait() throws InterruptedException;

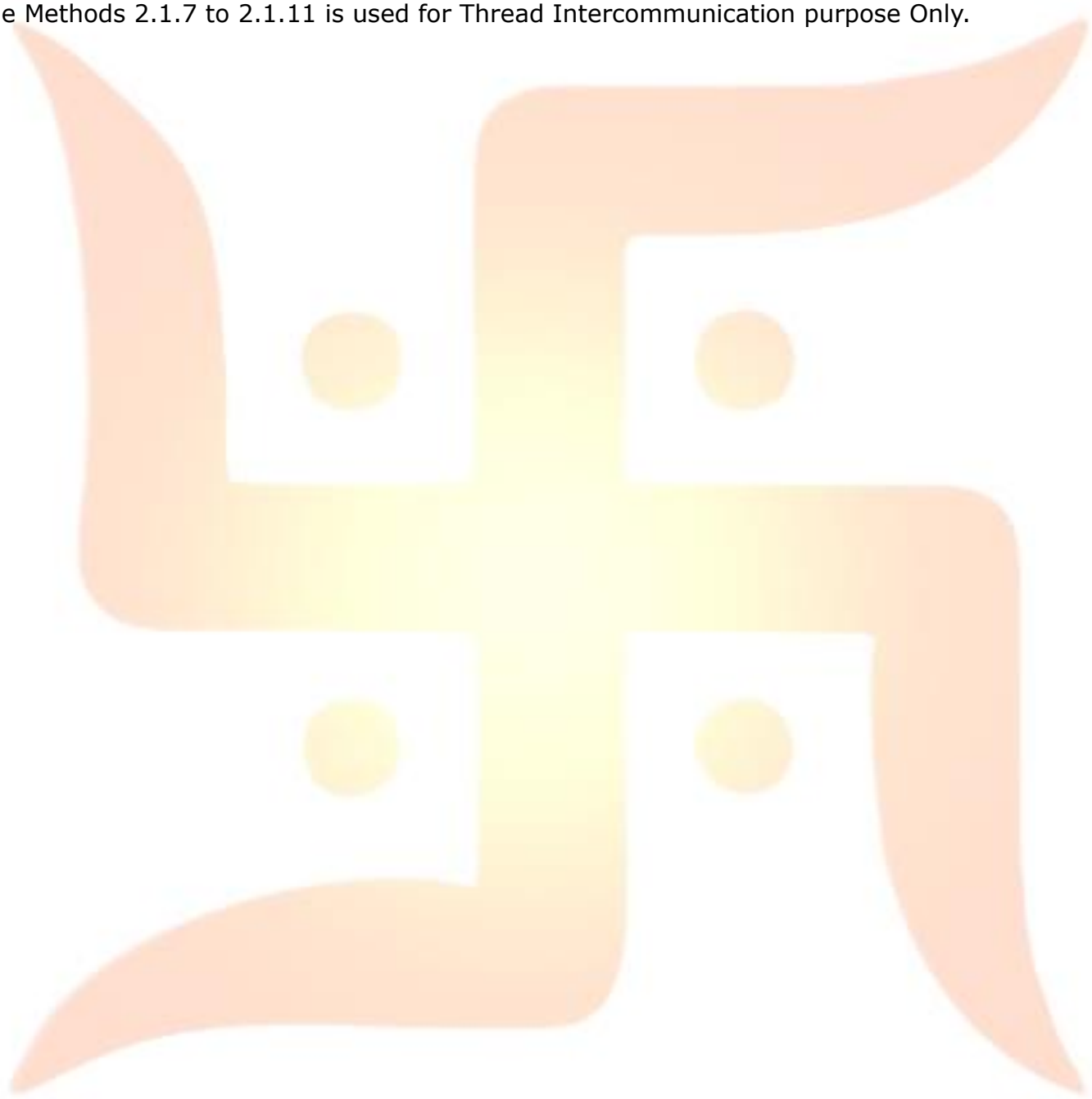
2.1.8 public final native void wait()throws InterruptedException;

2.1.9 public final void wait(long ms,int ns)throws InterruptedException;

2.1.10 public final native void notify();

2.1.11 public final native void notifyAll();

The Methods 2.1.7 to 2.1.11 is used for Thread Intercommunication purpose Only.



3.. String Class

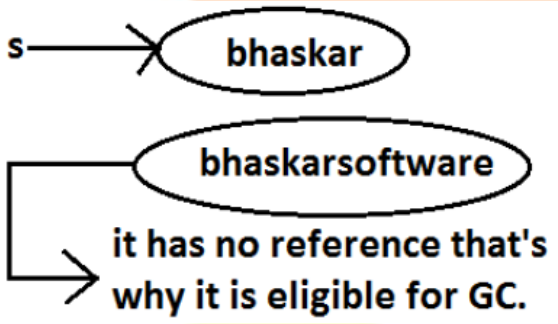
3.1 Different Use Case related to String Class

Case -1 Immutability vs Mutability

Once we create an object of any class say X, we can't perform any changes in the existing object. If we are trying to perform any changes on an existing Object then with those changes a new object will be created. This behavior is called **Immutability**. In Java, **String** Class is considered to be the **Immutable** Class.

```
class StringDemo {
    public static void main(String[] args) {
        String s = new String("baskar");
        s.concat("software");
        System.out.println(s);
    }
}
```

```
java -cp /tmp/NAmTtJCwQP StringDemo
baskar
```



Once we create an object of any class say X, we can't perform any changes in the existing object and this behavior is called **Mutability**. In Java, **StringBuffer** class is a **Mutable** Class.

```
class StringDemo {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer("baskar");
        s.append("software");
        System.out.println(s);
    }
}
```

```
java -cp /tmp/NAmTtJCwQP StringDemo
baskarsoftware
```

Case – 2

In **String** class equals() method is overridden for content comparison hence if the content is same equals() method returns true even though objects are different.

```
class StringDemo {
    public static void main(String[] args) {
        String s1 = new String("Hello");
        String s2 = new String("Hello");
        System.out.println(s1==s2);
        System.out.println(s1.equals(s2));
    }
}
```

```
java -cp /tmp/NAmTtJCwQP StringDemo
false
true
```

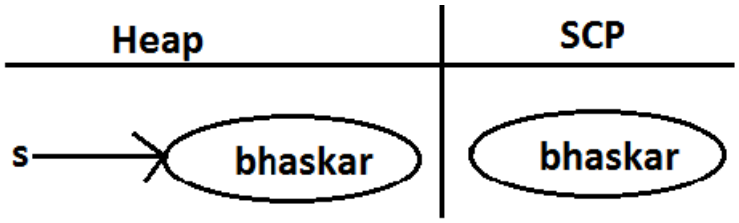
In **StringBuffer** class equals() method is not overridden for content comparison hence Object class equals() method got executed which is always meant for reference comparison. Hence if objects are different equals() method returns false even though content is same.

```java
class StringDemo {
    public static void main(String[] args) {
        StringBuffer s1 = new StringBuffer("Hello");
        StringBuffer s2 = new StringBuffer("Hello");
        System.out.println(s1==s2);
        System.out.println(s1.equals(s2));
    }
}
```

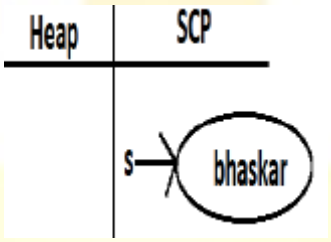
```
java -cp /tmp/NAmTtJCwQP StringDemo
false
false
```

Case – 3

When we write String s=new String("Vikash"); In this case two objects will be created one is on the Heap Memory and the other one is on **SCP (String constant pool)** and "s" is always pointing to Heap Memory object.



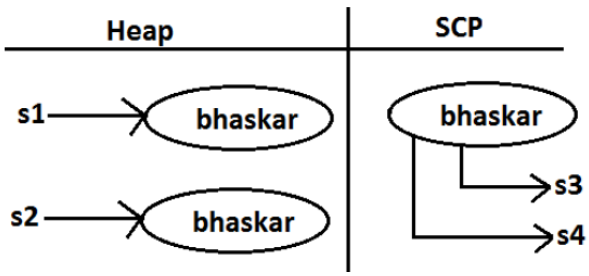
When we write String s="Vikash"; In this case only one object will be created in SCP and "s" is always referring that object.



Note: Object creation in SCP is always optional. First, JVM will check if any object is already created with required content or not. If it is already available, then it will reuse existing object instead of creating new object. If it is not there already, then only a new object will be created. Hence there is no chance of existing two objects with same content on SCP i.e., Duplicate Objects are not allowed in SCP. Garbage collector can't access SCP area hence even though object doesn't have any reference still that object is not eligible for GC if it is present in SCP. All SCP objects will be destroyed at the time of JVM shutdown automatically.

```
Example 1:
String s1=new String("bhaskar");
String s2=new String("bhaskar");
String s3="bhaskar";
String s4="bhaskar";
```

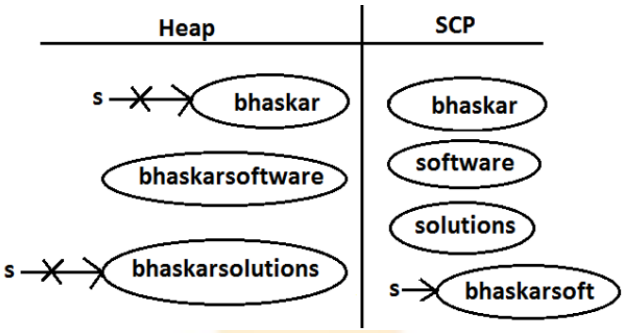
Whenever we are using new operator compulsory a new object will be created on the Heap Memory. There may be a chance of existing two objects with same content on the heap but there is no chance of existing two objects with same content on SCP. i.e., duplicate objects possible in the heap but not in SCP.



Consider the below Program Snippet:

```
class StringDemo {
    public static void main(String[] args) {
        String s=new String("bhaskar");
        s.concat("software");
        s=s.concat("solutions");
        s="bhaskarsoft";
    }
}
```

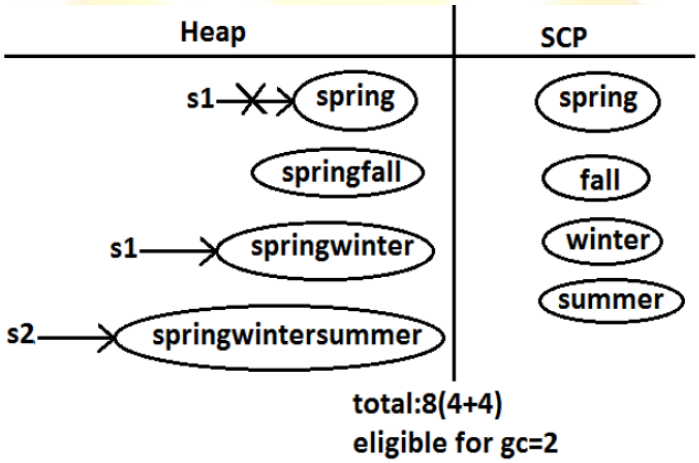
This is how Objects will be created in Heap and in SCP for this Program.



Case – 4

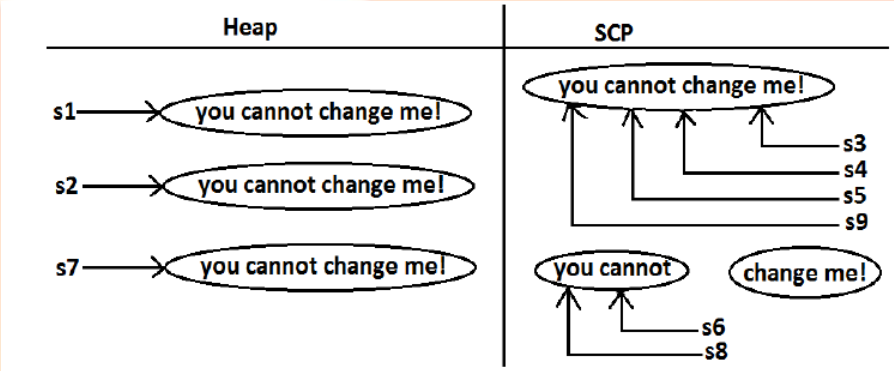
For every String Constant one object will be created in SCP. Because of runtime operation if an object is required to create compulsory that object should be placed on the heap but not SCP.

```
class StringDemo {
    public static void main(String[] args) {
        String s1=new String("spring");
        s1.concat("fall");
        s1=s1+"winter";
        String s2=s1.concat("summer");
        System.out.println(s1);
        System.out.println(s2);
    }
}
```



```java
class StringDemo {
    public static void main(String[] args) {
        String s1 = new String("you cannot change me!");
        String s2 = new String("you cannot change me!");
        System.out.println(s1==s2);
        String s3="you cannot change me!";
        System.out.println(s1==s3);
        String s4="you cannot change me!";
        System.out.println(s3==s4);
        String s5="you cannot "+"change me!";
        System.out.println(s3==s5);//true
        String s6="you cannot ";
        String s7=s6+"change me!";
        System.out.println(s3==s7);
        final String s8="you cannot ";
        String s9=s8+"change me!";
        System.out.println(s3==s9);
        System.out.println(s6==s8);
    }
}
```

```
java -cp /tmp/NAmTtJCwQP StringDemo
false
false
true
true
false
true
true
```



3.2 Importance of String Constant Pool (SCP)

In our program if any String object is required to use repeatedly then it is not recommended to create multiple objects with same content because it reduces performance of the system and effects memory utilization. We can create only one copy and we can reuse the same object for every requirement. This approach improves performance and memory utilization we can achieve this by using **String Constant Pool (SCP).**

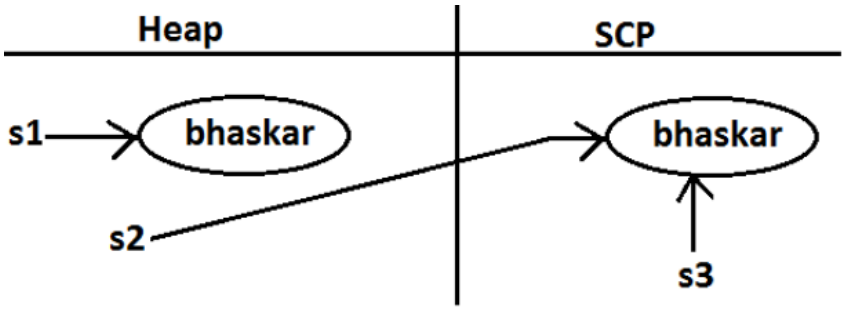
In SCP several references pointing to same object. Hence, the main disadvantage in this approach is by using one reference if we are performing any change the remaining references will be impacted. To overcome this problem sun people implemented **Immutability** concept for String objects. According to this once we create a String object, we can't perform any changes in the existing object if we are trying to perform any changes with those changes a new String object will be created. Hence, **Immutability** is the main disadvantage of SCP.

3.3 Interning of String Objects

By using heap object reference, if we want to get corresponding SCP object, then we should go for intern() method.

```java
class StringDemo {
    public static void main(String[] args) {
        String s1=new String("bhaskar");
        String s2=s1.intern();
        System.out.println(s1==s2);
        String s3="bhaskar";
        System.out.println(s2==s3);
    }
}
```

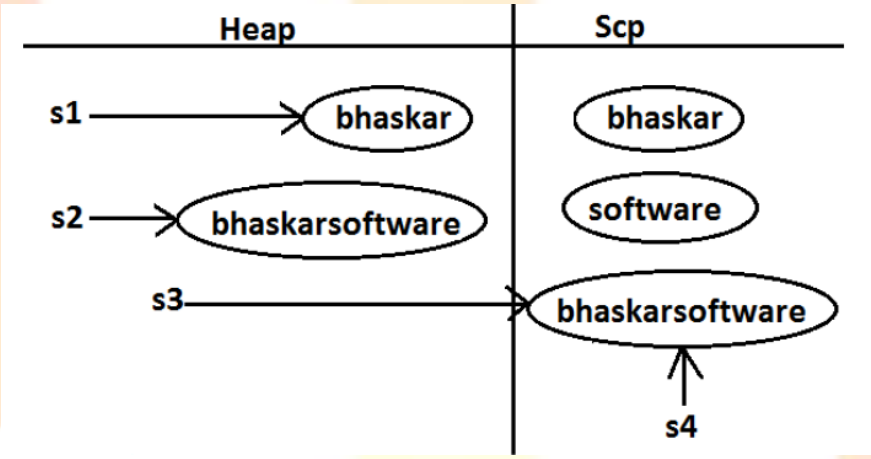
```
java -cp /tmp/anO87ZyvCq StringDemo
false
true
```



If the corresponding object is not there in SCP then intern() method itself will create that object and returns it.

```java
class StringDemo {
    public static void main(String[] args) {
        String s1=new String("bhaskar");
        String s2=s1.concat("software");
        String s3=s2.intern();
        String s4="bhaskarsoftware";
        System.out.println(s3==s4);
    }
}
```

```
java -cp /tmp/anO87ZyvCq StringDemo
true
```



3.4 Constructors of String Class

- **String s = new String();**
  Creates an empty String Object.

- **String s = new String(String literals);**
  To create an equivalent String object for the given String literal on the heap.

- **String s = new String(StringBuffer sb);**
  Creates an equivalent String object for the given StringBuffer.

- **String s=new String(char[] ch);**
  Creates an equivalent String object for the given char[ ] array.

```java
class StringDemo {
    public static void main(String[] args) {
        char[] ch={'a','b','c'} ;
        String s=new String(ch);
        System.out.println(s);
    }
}
```

```
java -cp /tmp/anO87ZyvCq StringDemo
abc
```

- **String s=new String(byte[] by);**
  Create an equivalent String object for the given byte[] array.

```java
class StringDemo {
    public static void main(String[] args) {
        byte[] b={100,101,102};
        String s=new String(b);
        System.out.println(s);
    }
}
```

```
java -cp /tmp/anO87ZyvCq StringDemo
def
```

3.5 Methods of String Class

- **public char charAt(int index);**
  Returns the character locating at specified index.

- **public String concat(String str);**
  Concatenate a String str to a given String S. The overloaded "+" and "+=" operators also meant for concatenation purpose only.

- **public boolean equals(Object o);**
  For content comparison where case is important. It is the overriding version of Object class equals() method.

- **public boolean equalsIgnoreCase(String s);**
  For content comparison where case is not important. For Example: We can validate username by using equalsIgnoreCase() method where case is not important and we can validate password by using equals() method where case is important.

- **public String substring(int begin);**
  Return the substring from begin index to end of the string.

- **public String substring(int begin, int end);**
  Returns the substring from begin index to end-1 index.

- **public int length();**
  Returns the number of characters present in the string.

- **public String replace(char old, char new);**
  To replace every old character with a new character.

- **public String toLowerCase();**
  Converts the all characters of the string to lowercase.

- **public String toUpperCase();**
  Converts the all characters of the string to uppercase.

- **public String trim();**
  We can use this method to remove blank spaces present at beginning and end of the string but not blank spaces present at middle of the String.

- **public int indexOf(char ch);**
  It returns index of 1st occurrence of the specified character if the specified character is not available then return -1.

- **public int lastIndexOf(Char ch);**

It returns index of last occurrence of the specified character if the specified character is not available then return -1.

Example:

```
class StringDemo {
    public static void main(String[] args) {
        String s = new String(" MyNameIsVikashSharma ");
        System.out.println(s.length());
        System.out.println(s.trim());
        System.out.println(s.toLowerCase());
        System.out.println(s.toUpperCase());
        System.out.println(s.substring(3));
        System.out.println(s.substring(3,9));
        System.out.println(s.charAt(4));
        System.out.println(s.indexOf('a'));
        System.out.println(s.lastIndexOf('a'));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringDemo
22
MyNameIsVikashSharma
 mynameisvikashsharma
 MYNAMEISVIKASHSHARMA
NameIsVikashSharma
NameIs
a
4
20
```

```
class StringDemo {
    public static void main(String[] args) {
        String s1 = new String("VikashSharma");
        String s2 = new String("vikashsharMA");
        System.out.println(s1.equals(s2));
        System.out.println(s1.equalsIgnoreCase(s2));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringDemo
false
true
```

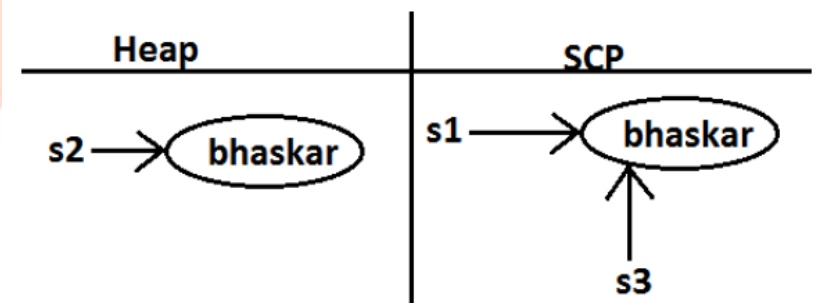
```
class StringDemo {
    public static void main(String[] args) {
        String s1 = new String("Vikash");
        String s2 = s1 + "Sharma";
        String s3 = s2.concat("ComputerEngineer");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s3.replace('e','w'));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringDemo
Vikash
VikashSharma
VikashSharmaComputerEngineer
VikashSharmaComputwrEnginwwr
```

Note: Because runtime operation if there is a change in content with those changes a new object will be created only on the heap but not in SCP. If there is no change in content no new object will be created the same object will be reused. This rule is same whether object present on the Heap or SCP.

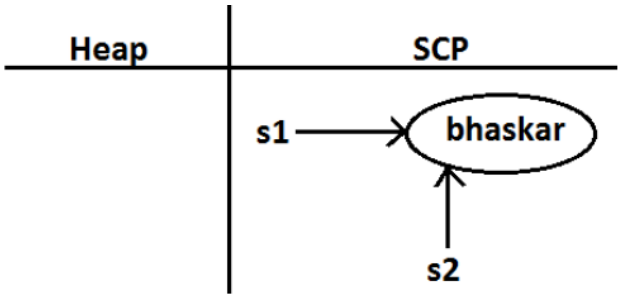
```
class StringDemo {
    public static void main(String[] args) {
        String s1="bhaskar";
        String s2=s1.toUpperCase();
        String s3=s1.toLowerCase();
        System.out.println(s1==s2);
        System.out.println(s1==s3);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringDemo
false
true
```




```
class StringDemo {
    public static void main(String[] args) {
        String s1="bhaskar";
        String s2=s1.toString();
        System.out.println(s1==s2);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringDemo
true
```




```
1  class StringDemo {
2      public static void main(String[] args) {
3          String s1=new String("ashok");
4          String s2=s1.toString();
5          String s3=s1.toUpperCase();
6          String s4=s1.toLowerCase();
7          String s5=s1.toUpperCase();
8          String s6=s3.toLowerCase();
9          System.out.println(s1==s6);
10         System.out.println(s3==s5);
11     }
12 }
```

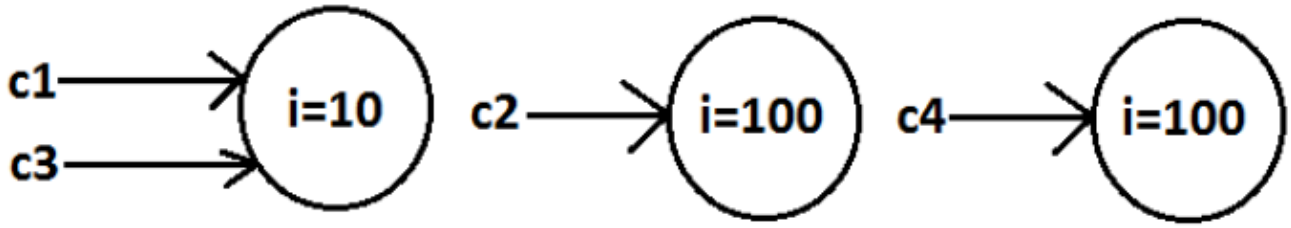
```
java -cp /tmp/2JsjGmJGNP StringDemo
false
false
```

3.6 Creation of Own Immutable Class

Once we created an object, we can't perform any changes in the existing object. If we are trying to perform any changes with those changes a new object will be created. If there is no change in the content then existing object will be reused. This behavior is called **Immutability**.

```
final class CreateImmutable  {
        private int i;
        CreateImmutable(int i)  {
                this.i=i;
        }
        public CreateImmutable modify(int i)  {
                if(this.i==i)
                        return this;
                else
                        return (new CreateImmutable(i));
        }
        public static void main(String[] args)        {
        CreateImmutable c1=new CreateImmutable(10);
        CreateImmutable c2=c1.modify(100);
        CreateImmutable c3=c1.modify(10);
        System.out.println(c1==c2);//false
        System.out.println(c1==c3);//true
        CreateImmutable c4=c1.modify(100);
        System.out.println(c2==c4);//false
        }
}
```

Here, once we create a **CreateImmutable** object we can't perform any changes in the existing object, if we are trying to perform any changes with those changes a new object will be created. If there is no chance in the content then existing object will be reused.

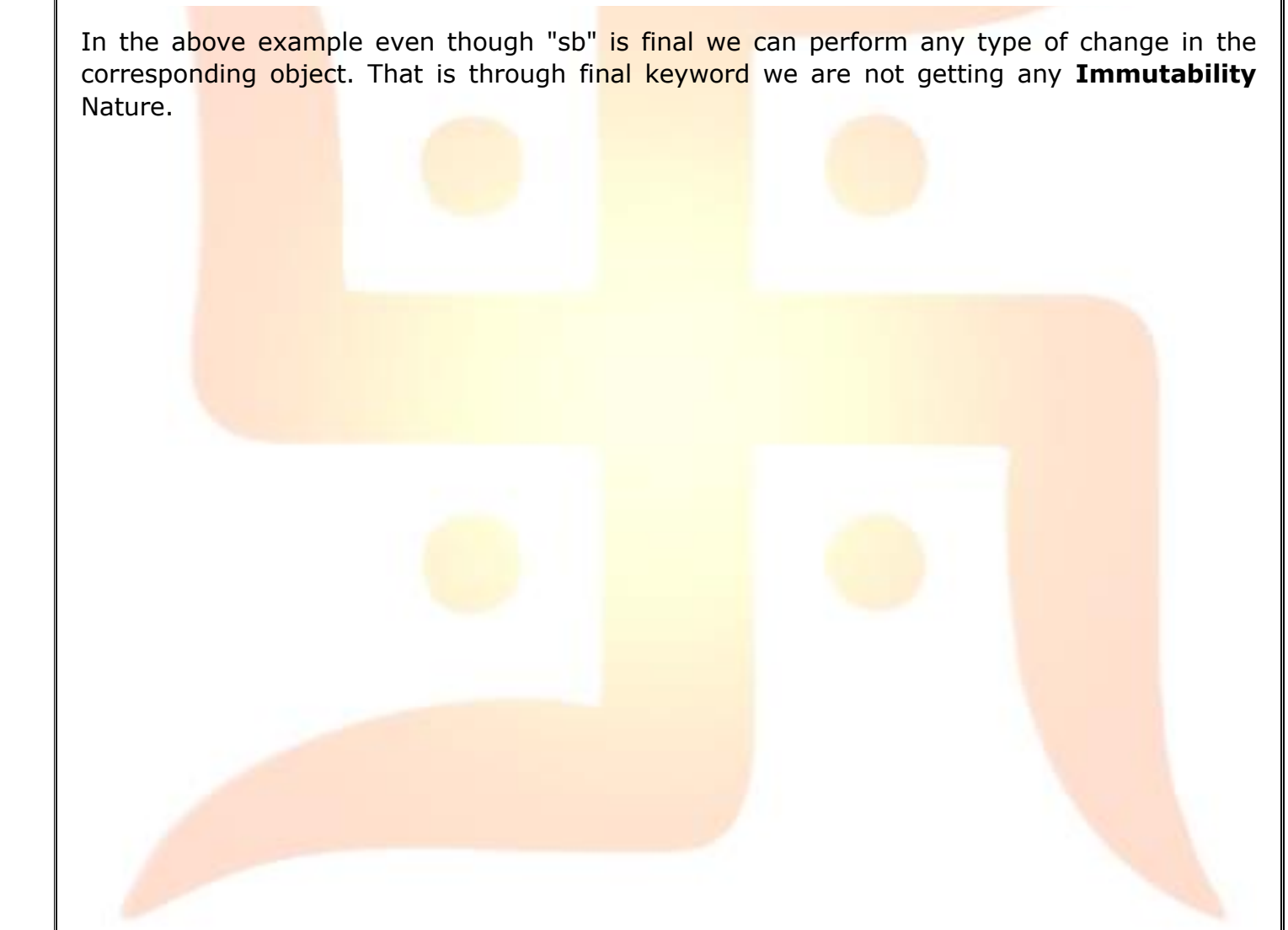


Final versus Immutability

final modifier applicable for variables whereas immutability concept applicable for objects. If reference variable declared as final then we can't perform reassignment for the reference variable it doesn't mean we can't perform any change in that object. That is by declaring a reference variable as final we won't get any immutability nature. Hence, final and immutability both are different concepts.

```
class Test
{
        public static void main(String[] args)
        {
                final StringBuffer sb=new StringBuffer("ashok");
                sb.append("software");
                System.out.println(sb);//ashoksoftware
                sb=new StringBuffer("solutions");//C.E: cannot assign a
value to final variable sb
        }
}
```

In the above example even though "sb" is final we can perform any type of change in the corresponding object. That is through final keyword we are not getting any **Immutability** Nature.



4.. StringBuffer/StringBuilder Class

If the content is going to change frequently then it is never recommended to go for String object because for every change a new object will be created internally. To handle this type of requirement we should go for **StringBuffer** concept. The main advantage of **StringBuffer** over **String** is, all required changes will be performed in the existing object only instead of creating new object.

4.1 Constructors of StringBuffer Class

- **StringBuffer sb=new StringBuffer();**
  Creates an empty StringBuffer object with default initial capacity "16". Once StringBuffer object reaches its maximum capacity a new StringBuffer object will be created with New capacity = (currentcapacity+1)*2.

```
1  class StringBufferDemo {
2      public static void main(String[] args) {
3          StringBuffer sb=new StringBuffer();
4          System.out.println(sb.capacity());
5          sb.append("abcdefghijklmnop");
6          System.out.println(sb.capacity());
7          sb.append("q");
8          System.out.println(sb.capacity());
9      }
10 }
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
16
16
34
```

- **StringBuffer sb=new StringBuffer(int initialCapacity);**
  Creates an empty StringBuffer object with the specified initial capacity.

```
1  class StringBufferDemo {
2      public static void main(String[] args) {
3          StringBuffer sb=new StringBuffer(19);
4          System.out.println(sb.capacity());
5      }
6  }
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
19
```

- **StringBuffer sb=new StringBuffer(String s);**
  Creates an equivalent StringBuffer object for the given String with capacity= s.length()+16;

```
1  class StringBufferDemo {
2      public static void main(String[] args) {
3          StringBuffer sb=new StringBuffer("Vikash");
4          System.out.println(sb.capacity());
5      }
6  }
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
22
```

4.2 Methods of StringBuffer Class

- **public int length();**
  Return the number of Characters present in the StringBuffer.

- **public int capacity();**
  Returns the total no of characters StringBuffer can accommodate(hold).

- **public char charAt(int index);**
  It returns the character located at specified index.

- **public void setCharAt(int index, char ch);**
  To replace the character locating at specified index with the provided character.

- **public StringBuffer delete(int begin, int end);**

To delete characters from begin index to end n-1 index.

- **public StringBuffer deleteCharAt(int index);**
  To delete the character locating at specified index.

- **public StringBuffer reverse();**
  Reverses the StringBuffer Object Content.

- **public void setLength(int length);**
  Consider only specified no of characters and remove all the remaining characters.

- **public void trimToSize();**
  To deallocate the extra allocated free memory such that capacity and size are equal.

- **public void ensureCapacity(int initialCapacity);**
  To increase the capacity dynamically(fly) based on our requirement.

StringBuffer append() methods: Below are overloaded methods.

- public StringBuffer append(String s);
- public StringBuffer append(int i);
- public StringBuffer append(long l);
- public StringBuffer append(boolean b);
- public StringBuffer append(double d);
- public StringBuffer append(float f);
- public StringBuffer append(int index, Object o);

StringBuffer insert() methods: Below are Overloaded methods.

- public StringBuffer insert(int index, String s);
- public StringBuffer insert(int index, int i);
- public StringBuffer insert(int index, long l);
- public StringBuffer insert(int index, double d);
- public StringBuffer insert(int index, boolean b);
- public StringBuffer insert(int index, float f);
- public StringBuffer insert(int index, Object o);

Example:

```
class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb=new StringBuffer("saiashokkumarreddy");
        System.out.println(sb);
        System.out.println(sb.length());
        System.out.println(sb.capacity());
        System.out.println(sb.charAt(14));
        sb.setCharAt(8,'A');
        System.out.println(sb);
        sb.ensureCapacity(1000);
        System.out.println(sb.capacity());
        System.out.println(sb.charAt(30));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
saiashokkumarreddy
18
34
e
saiashokAumarreddy
1000
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: index 30,length 18
at java.base/java.lang.String.checkIndex(String.java:3278)
    at java.base/java.lang.AbstractStringBuilder.charAt(AbstractStringBuilder.java:307)at java.base/java.lang
        .StringBuffer.charAt(StringBuffer.java:247)
    at StringBufferDemo.main(StringBufferDemo.java:12)
```

```
class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb=new StringBuffer(1000);
        System.out.println(sb.capacity());
        sb.append("Vikash");
        System.out.println(sb.capacity());
        sb.trimToSize();
        System.out.println(sb.capacity());
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
1000
1000
6
```

```java
class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb=new StringBuffer();
        sb.append("PI value is :");
        sb.append(3.14);
        sb.append(" this is exactly ");
        sb.append(true);
        System.out.println(sb);
        sb.setLength(7);
        System.out.println(sb);
        System.out.println(sb.reverse());
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
PI value is :3.14 this is exactly true
PI valu
ulav IP
```

```java
class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb=new StringBuffer("abcdefgh");
        System.out.println(sb);
        sb.insert(2, "xyz");
        sb.insert(11,"9");
        System.out.println(sb);
        sb.deleteCharAt(0);
        System.out.println(sb);
        sb.delete(2,6);
        System.out.println(sb);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP StringBufferDemo
abcdefgh
abxyzcdefgh9
bxyzcdefgh9
bxefgh9
```

Note: Every method present in StringBuffer is synchronized hence at a time only one thread is allowed to operate on StringBuffer object, it increases waiting time of the threads and creates performance problems, to overcome this problem we should go for **StringBuilder**.

4.3 StringBuilder Class

Every method present in StringBuffer is declared as synchronized hence at a time only one thread is allowed to operate on the StringBuffer object due to this, waiting time of the threads will be increased and effects performance of the system. To overcome this problem sun people introduced **StringBuilder** concept in 1.5v.

4.4 Difference between StringBuffer and StringBuilder Class

StringBuilder is exactly same as StringBuffer including constructors and methods except the following differences:

StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time only one thread is allow to operate on the StringBuffer object hence StringBuffer object is Thread safe.	At a time Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
It increases waiting time of the Thread and hence relatively performance is low.	Threads are not required to wait and hence relatively performance is high.
Introduced in 1.0 version.	Introduced in 1.5 versions.

4.5 Difference between String, StringBuffer & StringBuilder Class

- If the content is fixed and won't change frequently then we should go for String.
- If the content will change frequently but Thread safety is required then we should go for StringBuffer.
- If the content will change frequently and Thread safety is not required then we should go for StringBuilder.

4.6 Method Chaining

For most of the methods in String, StringBuffer and StringBuilder the return type is same type only. Hence after applying method on the result we can call another method which forms **method chaining**. Example: sb.m1().m2().m3().....In method chaining all methods will be evaluated from left to right.

```
class StringBufferDemo {
  public static void main(String[] args)        {
  sb.append("ashok").insert(5,"arunkumar").delete(11,13)
          .reverse().append("solutions").insert(18,"abcdf").reverse();
          System.out.println(sb); // snofdcbaitulosashokarunkur
          }
  }
```

5.. Wrapper Classes

The main objectives of wrapper classes are to wrap primitives' data types into object form so that we can handle primitives' data types also just like objects and also to define several utility functions which are required for the primitives.

5.1 Constructors of Wrapper Classes

Mostly, all wrapper classes define the following two constructors where one can take corresponding primitive type as argument and the other can take String as argument.

```
1  class WrapperClassDemo {
2      public static void main(String[] args) {
3          Integer i1 = new Integer(10);
4          Integer i2 = new Integer("10");
5          System.out.println(i1);
6          System.out.println(i2);
7      }
8  }
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10
10
```

If the String is not properly formatted i.e., if it is not representing number then we will get runtime exception saying "**NumberFormatException**".

```
1  class WrapperClassDemo {
2      public static void main(String[] args) {
3          Integer i1 = new Integer(10);
4          Integer i2 = new Integer("ten");
5          System.out.println(i1);
6          System.out.println(i2);
7      }
8  }
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
Exception in thread "main" java.lang.NumberFormatException: For input string: "ten"
at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.<init>(Integer.java:1105)
at WrapperClassDemo.main(WrapperClassDemo.java:4)
```

Float class defines three constructors with float, String and double arguments.

```
1  class WrapperClassDemo {
2      public static void main(String[] args) {
3          Float f1 = new Float(10.5f);
4          Float f2 = new Float("10.5f");
5          Float f3 = new Float(10.5);
6          Float f4 = new Float("10.5");
7          System.out.println(f1);
8          System.out.println(f2);
9          System.out.println(f3);
10         System.out.println(f4);
11     }
12 }
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10.5
10.5
10.5
10.5
```

Character class defines only one constructor which can take char primitive as argument there is no String argument constructor.

```
1  class WrapperClassDemo {
2      public static void main(String[] args) {
3          Character c1 = new Character('a');
4          System.out.println(c1);
5      }
6  }
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
a
```

```
1  class WrapperClassDemo {
2      public static void main(String[] args) {
3          Character c1 = new Character("a");
4          System.out.println(c1);
5      }
6  }
7
```

```
ERROR!
javac /tmp/2JsjGmJGNP/WrapperClassDemo.java
/tmp/2JsjGmJGNP/WrapperClassDemo.java:3: error: incompatible types: String cannot be converted to char
        Character c1 = new Character("a");
^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

Boolean class defines two constructors with boolean primitive and String arguments. If we want to pass boolean primitive the only allowed values are true, false where case should be lower case.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Boolean b1=new Boolean(true);
        Boolean b2=new Boolean(false);
        Boolean b3=new Boolean("true");
        Boolean b4=new Boolean("false");
        System.out.println(b1);
        System.out.println(b2);
        System.out.println(b3);
        System.out.println(b4);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
true
false
true
false
```

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Boolean b1=new Boolean(True);
        Boolean b2=new Boolean(False);
    }
}
```

```
ERROR!
javac /tmp/2JsjGmJGNP/WrapperClassDemo.java
/tmp/2JsjGmJGNP/WrapperClassDemo.java:3: error: cannot find symbol
        Boolean b1=new Boolean(True);
                               ^
  symbol:   variable True
  location: class WrapperClassDemo
/tmp/2JsjGmJGNP/WrapperClassDemo.java:4: error: cannot find symbol
        Boolean b2=new Boolean(False);
                               ^
  symbol:   variable False
location: class WrapperClassDemo
2 errors
```

If we are passing String argument then case is not important and content is not important. If the content is case insensitive String of true then it is treated as true in all other cases it is treated as false.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Boolean b1=new Boolean("True");
        Boolean b2=new Boolean("False");
        Boolean b3=new Boolean("true");
        Boolean b4=new Boolean("false");
        Boolean b5=new Boolean("Vikash");
        System.out.println(b1);
        System.out.println(b2);
        System.out.println(b3);
        System.out.println(b4);
        System.out.println(b5);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
true
false
true
false
false
```

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Boolean b1=new Boolean("yes");
        Boolean b2=new Boolean("no");
        System.out.println(b1);
        System.out.println(b2);
        System.out.println(b1 == b2);
        System.out.println(b1.equals(b2));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
false
false
false
true
```

Wrapper class	Constructor summery
Byte	byte, String
Short	short, String
Integer	Int, String
Long	long, String
Float	float, String, double
Double	double, String
Character	char,String
Boolean	boolean, String

Note: In all wrapper classes toString() method is overridden to return its content and equals() method is overridden for content compression.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i1=new Integer(10);
        Integer i2=new Integer(10);
        System.out.println(i1 + " ----> " + i1.toString());
        System.out.println(i2 + " ----> " + i2.toString());
        System.out.println(i1 == i2);
        System.out.println(i1.equals(i2));
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10 ----> 10
10 ----> 10
false
true
```

5.2 Utility Methods of Wrapper Classes

5.2.1 valueOf() method

We can use valueOf() method to create wrapper object for the given primitive or String. This method is alternative to constructor.

Every wrapper class except Character class contains a static valueOf() method to create wrapper object for the given String.

public static wrapper valueOf(String s);

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i=Integer.valueOf("10");
        Double d=Double.valueOf("10.5");
        Boolean b=Boolean.valueOf("ashok");
        System.out.println(i);
        System.out.println(d);
        System.out.println(b);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10
10.5
false
```

Every Integral type wrapper class (Byte, Short, Integer, and Long) contains the following valueOf() method to convert specified radix string to wrapper object.

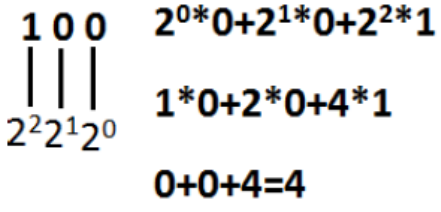
public static wrapper valueOf(String s , int radix);

Note: Here radix means base and the allowed radix range is 2 to 36.

base 2 ——— 0 To 1
base 8 ——— 0 To 7
base 10 ——— 0 To 9
base 11 ——— 0 To 9,a
base 16 ——— 0 To 9,a To f
base 36 ——— 0 To 9,a to z

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i=Integer.valueOf("100", 2);
        System.out.println(i);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
4
```

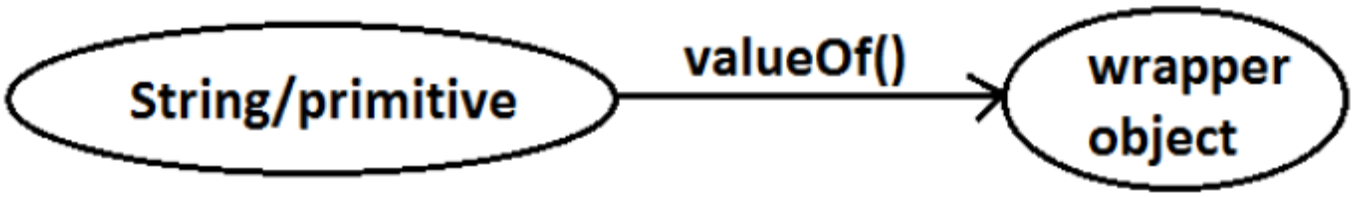


Every wrapper class including Character class defines valueOf() method to convert primitive to wrapper object.

public static wrapper valueOf(primitive p);

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i=Integer.valueOf(10);
        Double d=Double.valueOf(10.5);
        Boolean b=Boolean.valueOf(true);
        Character ch=Character.valueOf('a');
        System.out.println(ch);
        System.out.println(i);
        System.out.println(d);
        System.out.println(b);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
a
10
10.5
true
```



5.2.2 xxxValue() method

We can use xxxValue() methods to convert wrapper object to primitive. Every number type wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following 6 xxxValue() methods to convert wrapper object to primitives.

- public byte byteValue()
- public short shortValue()
- public int intValue()
- public long longValue()
- public float floatValue()
- public double doubleValue();

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i=new Integer(130);
        System.out.println(i.byteValue());
        System.out.println(i.shortValue());
        System.out.println(i.intValue());
        System.out.println(i.longValue());
        System.out.println(i.floatValue());
        System.out.println(i.doubleValue());
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
-126
130
130
130
130.0
130.0
```

Character class contains charValue() method to convert Character object to char primitive.

- public char charValue();

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Character ch=new Character('a');
        char c=ch.charValue();
        System.out.println(c);
    }
}
```

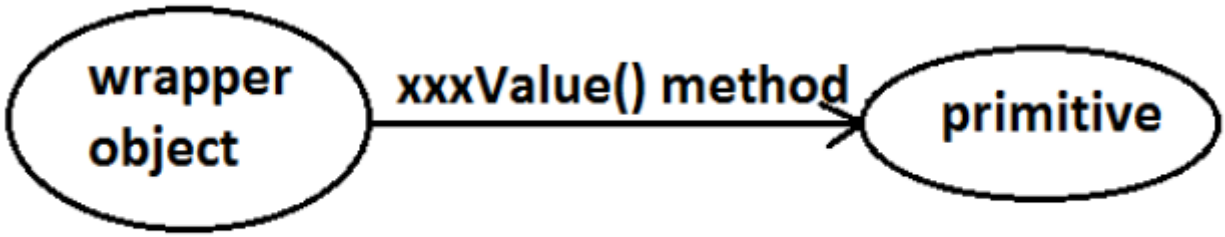
```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
a
```

Boolean class contains booleanValue() method to convert Boolean object to boolean primitive.

- public boolean booleanValue();

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Boolean b1=new Boolean("Vikash");
        boolean b=b1.booleanValue();
        System.out.println(b);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
false
```



In total there are 38(= 6*6+1+1) xxxValue() methods are possible.

5.2.3 parseXxx() method

We can use this method to convert String to corresponding primitive.

Form 1:

Every wrapper class except Character class contains a static parseXxx() method to convert String to corresponding primitive.

public static primitive parseXxx(String s);

```
class WrapperClassDemo {
    public static void main(String[] args) {
        int i=Integer.parseInt("10");
        boolean b=Boolean.parseBoolean("ashok");
        double d=Double.parseDouble("10.5");
        System.out.println(i);
        System.out.println(b);
        System.out.println(d);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10
false
10.5
```

Form 2:

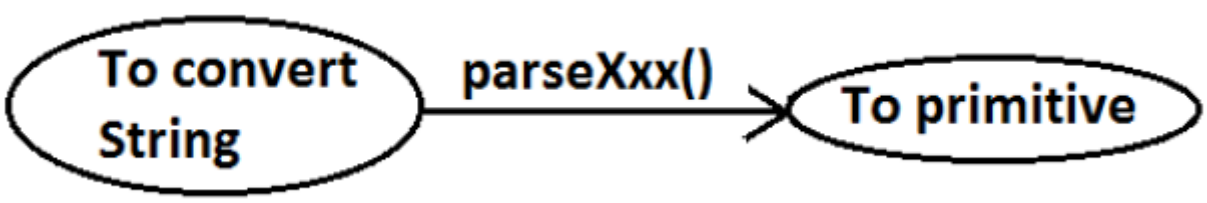
Integral type wrapper classes(Byte, Short, Integer, Long) contains the following parseXxx() method to convert specified radix String form to corresponding primitive.

public static primitive parseXxx(String s, int radix);

The allowed range of radix is: 2 to 36.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        int i=Integer.parseInt("100",2);
        System.out.println(i);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
4
```



5.2.4 toString() method

We can use toString() method to convert wrapper object (or) primitive to String.

Form 1: **public String toString();**

Every wrapper class (including Character class) contains the above toString() method to convert wrapper object to String. It is the overriding version of Object class toString() method. Whenever we are trying to print wrapper object reference internally this toString() method only executed.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        Integer i=Integer.valueOf("10");
        System.out.println(i);
        System.out.println(i.toString());
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10
10
```

Form 2: public static String toString(primitive p);

Every wrapper class contains a static toString() method to convert primitive to String.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        String s1=Integer.toString(10);
        String s2=Boolean.toString(true);
        String s3=Character.toString('a');
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
10
true
a
```

Form 3: public static String toString(primitive p, int radix);

Integer and Long classes contains the following static toString() method to convert the primitive to specified radix String form.

```
class WrapperClassDemo {
    public static void main(String[] args) {
        String s1=Integer.toString(7,2);
        String s2=Integer.toString(17,2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
111
10001
```

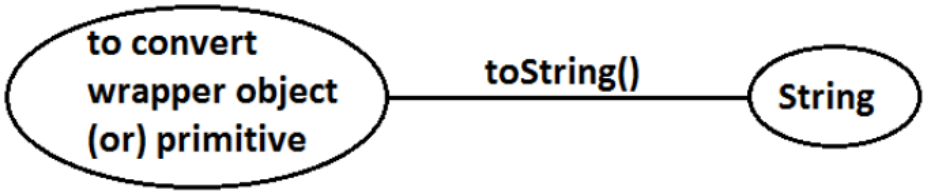
Form 4:

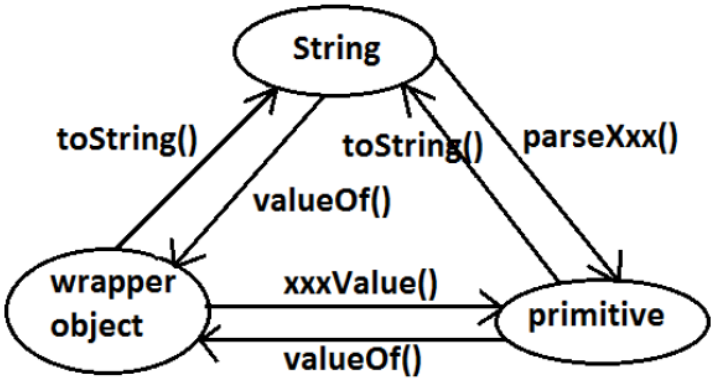
Integer and Long classes contains the following toXxxString() methods.

- public static String toBinaryString(primitive p);
- public static String toOctalString(primitive p);
- public static String toHexString(primitive p);

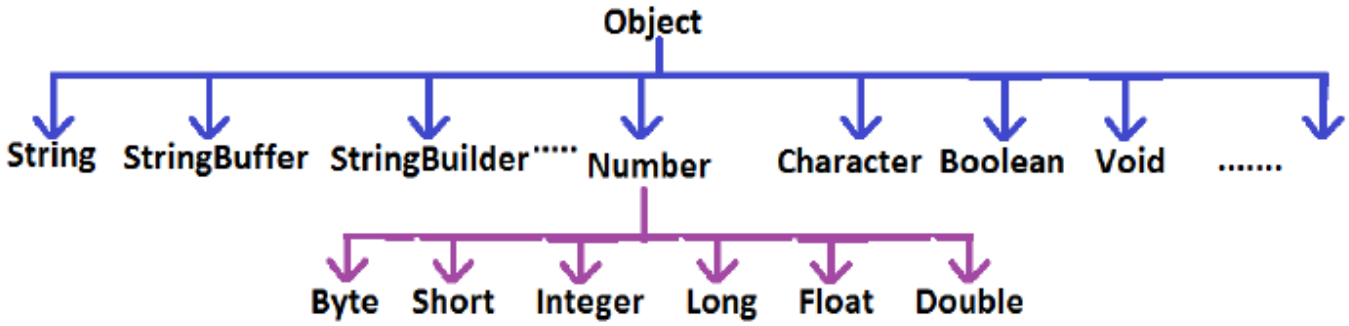
```
class WrapperClassDemo {
    public static void main(String[] args) {
        String s1=Integer.toBinaryString(7);
        String s2=Integer.toOctalString(10);
        String s3=Integer.toHexString(20);
        String s4=Integer.toHexString(10);
        System.out.println(s1);//111
        System.out.println(s2);//12
        System.out.println(s3);//14
        System.out.println(s4);//a
    }
}
```

```
java -cp /tmp/2JsjGmJGNP WrapperClassDemo
111
12
14
a
```





Note:



- String, StringBuffer, StringBuilder and all wrapper classes are final classes.
- The wrapper classes which are not child class of Number are Boolean and Character.
- The wrapper classes which are not direct class of Object are Byte, Short, Integer, Long, Float, Double.
- Sometimes we can consider Void is also as wrapper class.
- In addition to String objects, all wrapper class objects also immutable in java.

Void Class

Sometimes Void class is also considered as wrapper class. Void is class representation of void java keyword. Void class is the direct child class of Object and it doesn't contain any method and it contains only one static variable TYPE. We can use Void class in reflections

Example: To check whether return type of m1( ) is void or not.

```
if(ob.getClass( ).getMethod("m1").getReturnType( )==Void.TYPE) {
 ----------------------------
 ----------------------------
}
```

6.. Autoboxing and Autounboxing (1.5v)

Until 1.4 version we can't provide wrapper object in the place of primitive and primitive in the place of wrapper object all the required conversions should be performed explicitly by the programmer.

```
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        Boolean b=new Boolean(true);
        if(b)
        {
            System.out.println("hello");
        }}}
```

1.4V →

```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:6: incompatible types
found   : java.lang.Boolean
required: boolean
```

```
Program 2:
class AutoBoxingAndUnboxingDemo  {
        public static void main(String[] args)        {
                Boolean b=new Boolean(true);
                if(b)   {
                        System.out.println("hello");
                 }
         }
}
Output:
hello
```

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList();
        l.add(10);
    }
}
```

```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:7: cannot find symbol
symbol  : method add(int)
location: class java.util.ArrayList
```

```
Program 2:
import java.util.*;
class AutoBoxingAndUnboxingDemo  {
        public static void main(String[] args)  {
                ArrayList l=new ArrayList();
                Integer i=new Integer(10);
                l.add(i);
        }
}
```

But from 1.5 version onwards we can provide primitive value in the place of wrapper and wrapper object in the place of primitive all required conversions will be performed automatically by compiler. These automatic conversions are called **Autoboxing** and **Autounboxing.**

Automatic conversion of primitive to wrapper object by compiler is called **Autoboxing**. Internally Autoboxing concept is implemented by using valueOf() method.

Example :

```
Integer i=10;
```

[compiler converts "int" to "Integer" automatically by Autoboxing]

After compilation the above line will become.

```
Integer i=Integer.valueOf(10);
```

Automatic conversion of wrapper object to primitive by compiler is called **Autounboxing**. Autounboxing concept is internally implemented by using xxxValue() method.

Example:

```
Integer I=new Integer(10);
Int i=I;
```

[compiler converts "Integer" to "int" automatically by Autounboxing]

After compilation the above line will become.

```
Int i=I.intValue();
```