12. java.lang package

1. Introduction
2. Object class
3. String class
4. StringBuffer class
5. StringBuilder class
6. Wrapper classes
7. Auto boxing & Auto unboxing.

1. Introduction:—

→ For writing any Java program, whether it is small or complex the most commonly required classes & interfaces are defined in a separate package which is nothing but java.lang package.

→ we are not required to import java.lang package explicitly bcoz by default is available to every Java program.

DEMO

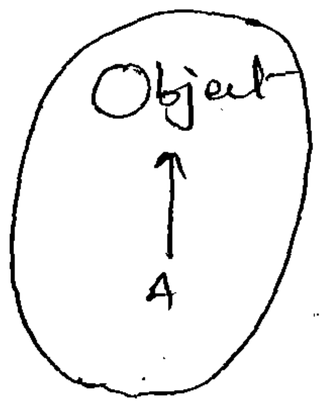
2. Object class:—

→ For all Java classes whether it is predefined or customized the most commonly required methods are defined in Object class.

→ SUN people define Object class as root for all Java classes. So that it's methods by default available to every Java class through inheritance.

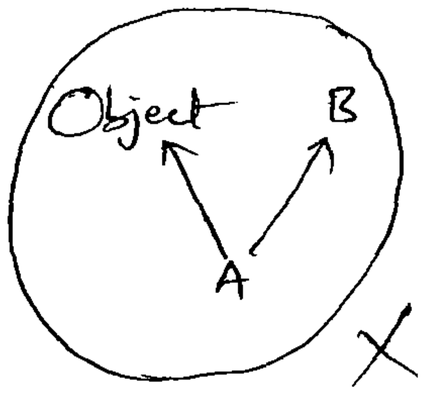
Note:— If our class doesn't extend any other class then only it is the direct child class of Object

Ex:
```
class A
{
}
```

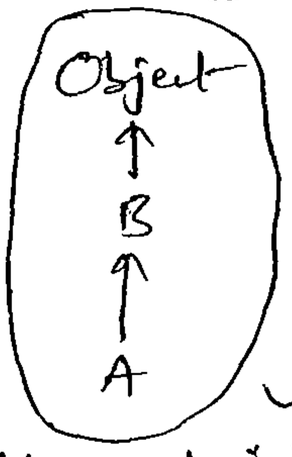


→ If one class extending any other class then it is the indirect child class of Object.

Ex: class A extends B
{
}



Multiple inheritance ✗



Multi-level inheritance. ✓

Q: which of the following statements is valid?

1. Every class in Java is the direct child class of Object. ✗
2. Every class in Java is the child class of Object either directly or indirectly. ✓

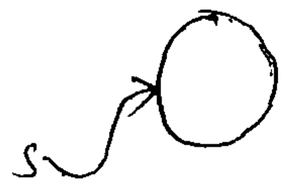
→ Object class defines the following 11 methods.

1. public String toString()
2. public native int hashCode()
3. public boolean equals(Object o)
4. protected native Object clone() throws CloneNotSupportedException
5. protected void finalize() throws Throwable
6. public final Class getClass()
7. public final void wait() throws InterruptedException
8. public final native void wait(long ms) throws InterruptedException
9. public final void wait(long ms, int ns) throws InterruptedException
10. public native final void notify()
11. public native final void notifyAll()

1. toString():—

→ We can use toString() method to get String representation of an object.

→ Whenever we are trying to print any object reference internally toString() method will be called.

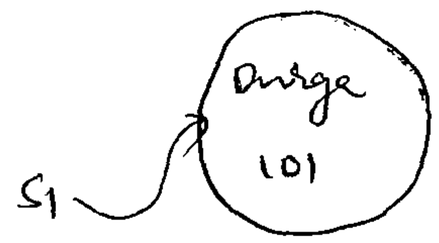


Ex: Student s = new Student();

S.o.p(s); ⇒ S.o.p(s.toString());

→ If our class doesn't contain toString() method then Object class toString() method will be called.

Ex:
```
class Student
{
    String name;
    int rollno;
    Student(String name, int rollno)
    {
      this.name=name;
      this.rollno = rollno;
    }
    p s v m(-)
    {
      Student s1 = new Student("Durga", 101);
      Student s2 = new Student("Ravi", 102);
      S.o.p(s1); ⇒ o/p : Student@3e25a5
      S.o.p(s1.toString()); ⇒ o/p : Student@3e25a5
      S.o.p(s2); ⇒ o/p : Student@19821f
    }
}
```



DEMO

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

```
public String toString()
{
    return getClass().getName+"@"+Integer.toHexString(hashCode());
}
```

i.e., className@hexadecimal_string_of_hashCode.

→ To return more meaningful String representation we can override toString() method in our class.

→ Whenever we are trying to print Student reference to return his name and rollno we have to override toString() method as follows.

Ex: public String toString()
{
// return name;
// return name+"..."+rollno;
return "This is Student with name :"+name+" and rollno:"
+rollno;
}

→ In String class, StringBuffer class, all Wrapper classes, all Collection classes toString() method is overridden for meaningful String representation.

→ Hence it is highly recommended to override toString() method in our class also.

DEMO

Ex: class Test
{
public String toString()
{
return "test";
}
p s v m(-)
{
String s=new String("durga");
Integer i=new Integer(10);
Test t=new Test();
S.o.p(s); ⇒ o/p : durga
S.o.p(i); ⇒ o/p : 10
S.o.p(t); ⇒ o/p : test
}
}

2. hashCode() :—

→ For every object JVM will generate a unique number which is nothing but Hashcode.

→ JVM will use hashcode while saving objects into hashing related data structures like HashSet, Hashtable and HashMap.

→ If the objects are saved according to hashcode then the advantage is Search operation will become easy.

→ If we are not overriding hashCode() method Then Object class hashCode() method will be executed which will generate hashcode based on address of object.

→ If we override hashCode() method then its no longer related to address.

→ It is highly recommended to override hashCode() method. So that we can customize order of elements in hashing related data structures.

DEMO

→ Overriding hashCode() method is said to be proper iff for every object we have to generate a unique no. as hashcode.

Ex:

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

This is Improper way of overriding hashCode() method becoz for all objects we are generating same hashcode.

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

It is proper way of overriding hashCode() method becoz we are generating a unique no. as hashcode for every object.

*** toString() Vs hashCode() :-

→ If we are giving the chance to internally calls hashCode() method.

→ But if we are overriding toString() method it may not call hashCode() method.

```
class Test
{
  int i;
  Test(int i)
  {
    this.i=i;
  }
  public int hashCode()
  {
    return i;
  }
  public String toString()
  {
    return i+" ";
  }
  p s v m(-)
  {
    Test t1=new Test(10);
    Test t2=new Test(100);
    S.o.p(t1);
    S.o.p(t2);
  }
}
```

O/p : 10
100

```
class Test
{
  int i;
  Test(int i)
  {
    this.i=i;
  }
  public int hashCode()
  {
    return i;
  }
  p s v m(-)
  {
    Test t1=new Test(10);
    Test t2=new Test(100);
    S.o.p(t1);
    S.o.p(t2);
  }
}
```

O/p : Test@a
Test@64

16|100
6-4
⇓
(64)

Object ⇒ toString()
⇓
Test ⇒ hashCode()

```
class Test
{
  int i;
  Test(int i)
  {
    this.i=i;
  }
  p s v m(-)
  {
    Test t1=new Test(10);
    Test t2=new Test(100);
    S.o.p(t1);
    S.o.p(t2);
  }
}
```

O/p : Test@3e25a5
Test@19821f

Object ⇒ toString()
⇓
Object ⇒ hashCode()

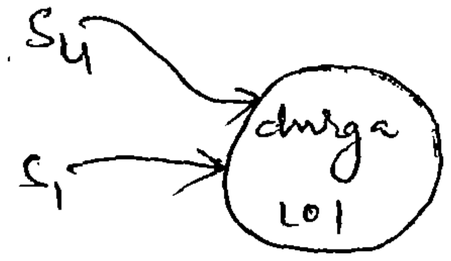
Object class toString() method it internally calls hashCode() method.

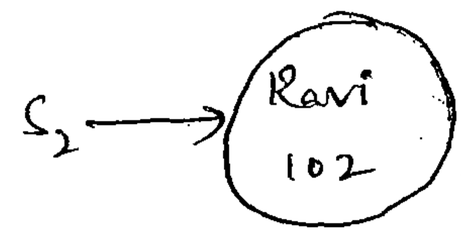
PLOT NO.56, HARSHA CHAMBERS, SAI NAGAR,NEAR BIG C MOBILES, MADHAPUR-500034, MOBILE:9505905786

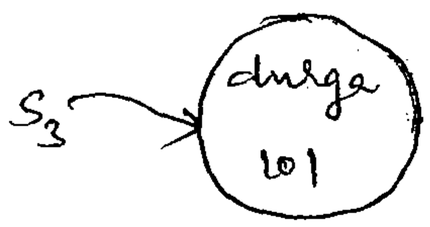
3. equals(Object o):—

→ We can use equals(—) method to check equality of 2 objects.

→ If our class doesn't contain equals(—) method then Object class equals(—) method will be executed.







Ex:
```
class Student
{
    String name;
    int rollno;
    Student(String name, int rollno)
    {
        this.name=name;
        this.rollno=rollno;
    }
    p s v m(—)
    {
        Student s1=new Student("durga", 101);
        Student s2=new Student("Ravi", 102);
        Student s3=new Student("durga", 101);
        Student s4=s1;
        S.o.p(s1.equals(s2)); ⇒ o/p: false
        S.o.p (s1.equals(s3)); ⇒ o/p: false
        S.o.p (s1.equals(s4)); ⇒ o/p: true
    }
}
```

DEMO

→ In the above example, Object class equals(—) method got executed which is meant for Reference comparison (Address comparison).
i.e., if two references pointing to the same object then only equals(—) method returns true.

→ Instead of reference comparison if we want content comparison then we can override equals(—) method in our class.

→ Whenever we are overriding equals(-) method in our class we have to consider the following things

1. What is the meaning of content comparison?
2. If we pass different type of objects then our equals(-) method should return false, but not ClassCastException i.e., we have to handle CCE to return false.
3. If we pass null argument our equals(-) method should return false, but not NullPointerException i.e., we have to handle NPE to return false.

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

```
public boolean equals(Object o)
{
   try
    {
        String name1 = this.name;
        int rollno1 = this.rollno;
        Student s = (Student)o;
        String name2 = s.name;
        int rollno2 = s.rollno;
        if (name1.equals(name2) && rollno1 == rollno2)
            return true;
        else
            return false;
    }
    catch (ClassCastException e)
    { return false;
    }
    catch (NullPointerException e)
    {return false;
    }
}
```

Ex: Student s1 = new Student ("durga", 101);

Student s2 = new Student ("Ravi", 102);

Student s3 = new Student ("durga", 101);

Student s4 = s1;

S.o.p (s1.equals(s2)); ⇒ o/p : false

S.o.p (s1.equals(s3)); ⇒ o/p : true

S.o.p (s1.equals(s4)); ⇒ o/p : true

S.o.p (s1.equals("durga")); ⇒ o/p : false

S.o.p (s1.equals(null)); ⇒ o/p : false

Simplified version of equals(-) method:—

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

DEMO

More simplified version of equals(-) method:-

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

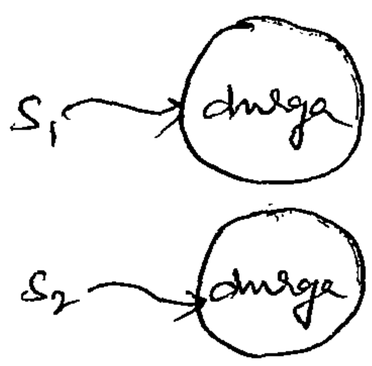
Note:- To make above equals(-) method more efficient we have to place the following code at beginning of equals(-) method.

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

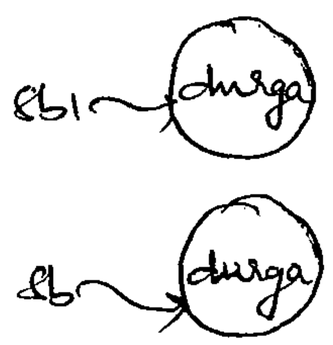
DEMO

→ In String class, all Wrapper classes and all Collection classes equals(-) method is overridden for content comparison, but in StringBuffer equals(-) method is not overridden for content comparison.

Ex:
```
String s1=new String("durga");
String s2=new String("durga");
S.o.p(s1==s2); ⇒ o/p: false
S.o.p(s1.equals(s2)); ⇒ o/p: true
```




```
StringBuffer sb1=new SB("durga");
SB sb2=new SB("durga");
S.o.p(sb1==sb2); ⇒ o/p: false
S.o.p(sb1.equals(sb2)); ⇒ o/p: false
```



In String, .equals(_) method is overridden for content comparison. Hence eventhough objects are different .equals(_) method return true if the content is same.

In StringBuffer class, .equals(_) method is not overridden for content comparison. Hence Object class equals(_) method will be executed which is meant for reference comparison.

Due to this if objects are different .equals(_) method returns false eventhough content is same.

*** Comparison b/w == operator and .equals(_) method:-

1. If two objects are equal by == operator then these objects are always equal by .equals(_) method also i.e.,

DEMO

if $r_1 == r_2$ returns true then r_1.equals(r_2) is always true.

2. If $r_1 == r_2$ returns false then we can't conclude anything about .equals(_) method, it may returns true or false.

3. If r_1.equals(r_2) returns true then we can't conclude anything about == operator, it may returns true or false.

4. If r_1.equals(r_2) returns false then $r_1 == r_2$ is always false.

*** Differences b/w == operator and .equals(_) method:-

== operator	.equals(_)
1. It is an operator applicable for both primitives and Object types.	1. It is a method applicable only for Object types, but not for primitives.

== operator	.equals(-)
2. In case of Object references ==operator meant for reference comparison.	2. By default equals(-) method present in Object class also meant for reference comparison.
3. It is not possible to override == operator for content comparison.	3. It is possible to override equals(-) method for content comparison.
4. If there is no relation btw argument types then we will get compile time error saying incomparable types.	4. If there is no relation btw argument types we won't get any CE or RE equals(-) method simply returns false.

Ex: String s1 = new String ("durga");

SB s2 = new SB ("durga");

S.o.p (s1 == s2); ⟶ CE: incomparable types: String & StringBuffer

S.o.p (s1.equals(s2)); ⇒ o/p: false.

Note:- For any object reference r, the following expressions returns false.

r == null

r.equals(null); ⟹ false.

***

Q: What is the difference btw == operator & .equals(-) method?

Ans: In general we can use == operator for reference comparison whereas .equals(-) method for content comparison.

*** Contract b/w .equals(-) method and hashCode() method :-

→ Two equivant objects should be placed in the same Bucket, but all objects present in the same Bucket need not be equal.

***
1. Two equivalent objects must have same hashcode. i.e.,

if r_1.equals(r_2) is true then r_1.hashCode()==r_2.hashCode()

should returns true.

2. If two objects are not equal by .equals(-) method then there is no restrictions on their hashcodes, may be same or may not be same.

3. If hashcodes of two objects are equal then these objects may or may not equal by .equals(-) method.

4. If hashcodes of two objects are not equal then these objects are always not equal by .equals(-) method.

DEMO

→ To satisfy above contract b/w equals(-) method & hashCode() method whenever we are overriding equals(-) method compulsorey we have to override hashCode() method, o.w. we won't get any CE or RE but it is not a good programming practice.

⇒ Consider the following Person class.

```
class Person
{
  public boolean equals(Object o)
  {
    if(o instanceof Person)
    {
      Person p=(Person)o;
      if(name.equals(p.name) && age==p.age)
        return true;
```

```
        else false;
      } return false;
    return false;
  }
}
```

Q: which of the following is appropriate hashCode() method for Person class?

① ✗
```
public int hashCode()
{ return 100;
}
```

② ✗
```
public int hashCode()
{
  return age+height;
}
```

③ ✓
```
public int hashCode()
{
  return name.hashCode()+age;
}
```

④ ✗ No restrictions

***Note:— Based on which parameters we are overriding equals() method use same parameters while overriding hashCode() method also.

Ex: String s1=new String("durga");
String s2=new String("durga");
S.o.p(s1.equals(s2)); ⇒ o/p: true
S.o.p(s1.hashCode()); ⇒ o/p: 9595
S.o.p(s2.hashCode()); ⇒ o/p:

4. clone() :—

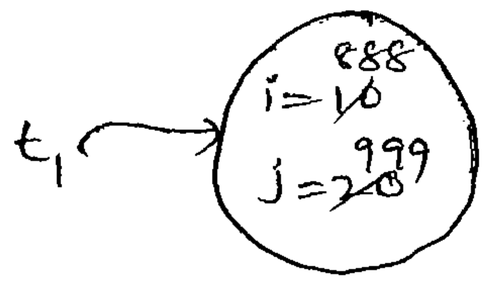
→ The process of creating exact duplicate object is called Cloning.
→ The main purpose of cloning is to maintain back up purpose.
→ we can create cloned object by using clone() method of Object class.

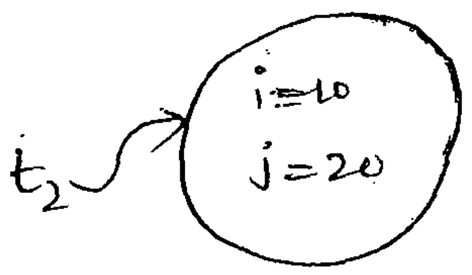
protected native Object clone() throws CloneNotSupportedException

Ex: class Test implements Cloneable

```
{
  int i=10;
  int j=20;
  p s v m(-) throws CloneNotSupportedException
  {
    Test t1=new Test();
    Test t2=(Test)t1.clone();
    t1.i=888;
    t1.j=999;
    S.o.p(t2.i+"..."+t2.j);
  }
}
```

O/P : 10...20





→ We can perform cloning only for Cloneable objects.

→ An object is said to be Cloneable iff the corresponding class implements Cloneable interface.

→ Cloneable interface present in java.lang package and it doesn't contain any methods. It is a Marker interface.

→ If we are trying to perform cloning for non-Cloneable objects then we will get RE saying CloneNotSupportedException.

DEMO

*** Shallow Cloning Vs Deep Cloning :-

Shallow Cloning :-

→ The process of creating bitwise-copy of an object is called Shallow cloning.

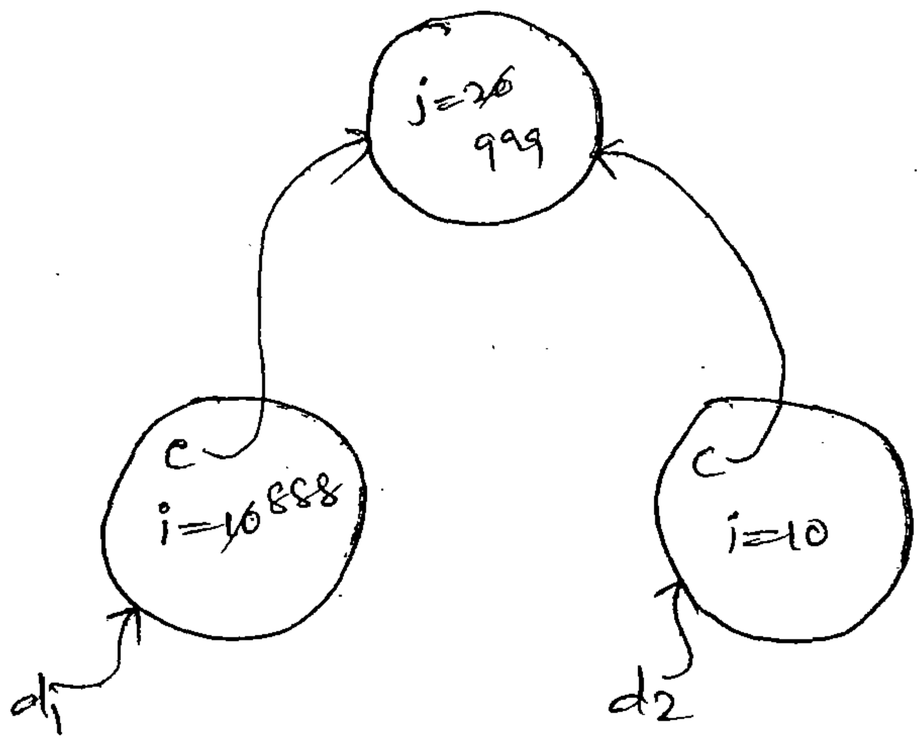
→ If the main object contains any primitive variables exact duplicate copy will be created in cloned object.

→ 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 to cloned object.

→ By default Object class clone() method meant for Shallow Cloning.



Ex:
```
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
  {
    return super.clone();
  }
}
class ShallowCloning
{
  p s v m() throws CNSE
  {
    Cat c=new cat(20);
    Dog d1=new Dog(c,10);
    S.o.p(d1.i+"..."+d1.c.j); ⇒ o/p : 10...20
```

DEMO

```
Dog d2 = (Dog) d1.clone();
d1.i = 888;
d1.c.j = 999;
S.o.p(d2.i + "..." + d2.c.j); ⇒ o/p : 10... 999
}
}
```

→ Shallow cloning is the best choice if 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 to the cloned object also.

→ To overcome this problem we should go for Deep Cloning.

Deep Cloning:-

→ The process of creating exactly duplicate independent object (including contained objects) is called Deep Cloning.

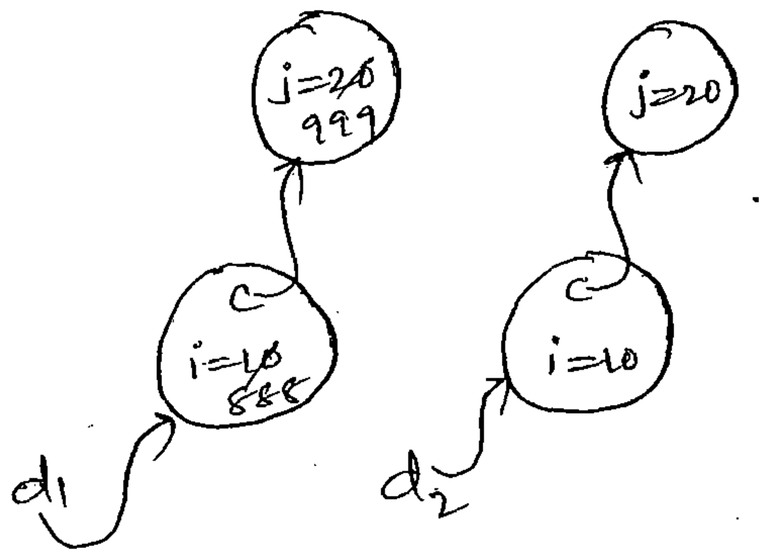
→ In Deep cloning, if main object contains any reference variable then the corresponding object copy will be created in cloned object.

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

Ex:

```
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 CNSE
{
   Cat c1=new Cat(c.j);
   Dog d = new Dog(c1,i);
   return d;
}
}
```




```
class DeepCloning
{
   p s v m(-) throws CNSE
   {
      Cat c=new Cat(20);
      Dog d1=new Dog(c,10);
      S.o.p(d1.i+"..."+d1.c.j); =) olp : 10...20
      Dog d2=(Dog)d1.clone();
      d1.i=888;
      d1.c.j=999;
      S.o.p(d2.i+"..."+d2.c.j); => olp : 10...20
   }
}
```

DEMO

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

*** Which cloning is best?

→ If the object contains only primitive variables then Shallow cloning is the best choice.

→ If the object contains reference variable then Deep cloning is the best choice.

6. getClass():—

→ This method returns runtime class definition of an object.

Ex: To print Connection interface vendor specific implementation class name we have to write code as follows.

```
Connection con = DriverManager.getConnection(url, uname, pwd);
S.o.p (con.getClass().getName());
```

5. finalize():—

→ Just before destroying an object Garbage Collector always call finalize() method to perform clean up activities.

→ Once finalize() method completes automatically GC destroys that object.

wait(), notify() and notifyAll() methods:—

→ Two threads can communicate with each other by using wait(), notify() and notifyAll() methods. i.e., these methods meant for Interthread Communication.

DEMO

*** 3. java.lang.String:—

*** Case(i):

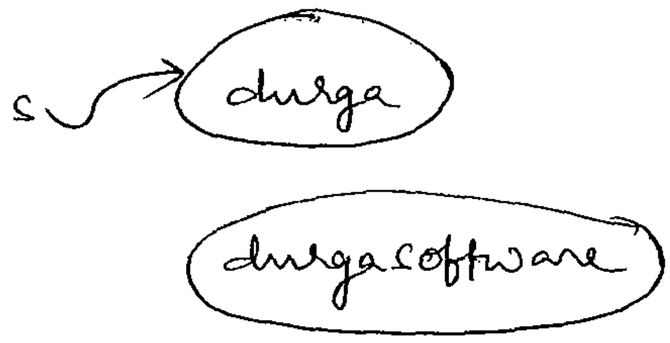
```
String s=new String("durga");
s.concat("software");
S.o.p(s);
```

Once we created 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 object will be created. This behaviour is called Immutability

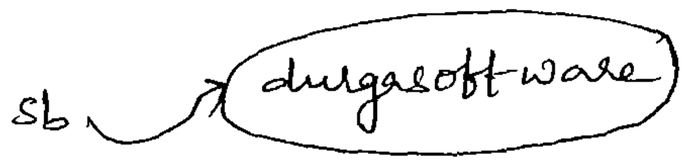
```
StringBuffer sb=new SB("durga");
sb.append("software");
S.o.p(sb); ⇒ o/p: durgasoftware
```

Once we created a StringBuffer object we can perform any type of changes in the existing object this changeable behaviour is called Mutability of the

of String object.



StringBuffer object.



***

Case (ii):

```
String s1=new String("durga");
String s2=new String("durga");
S.o.p(s1==s2); ⇒ o/p: false
S.o.p(s1.equals(s2)); ⇒ o/p: true
```

In String class, .equals(-) method is overridden for content-comparison. Hence if the content is same then .equals(-) method returns true even though objects are different.

```
SB sb1=new SB("durga");
SB sb2=new SB("durga");
S.o.p(sb1==sb2); ⇒ o/p: false
S.o.p(sb1.equals(sb2)); ⇒ o/p: false.
```

In SB class, .equals(-) method is not overridden for content comparison. Hence Object class .equals(-) method will be executed which is meant for reference comparison. Due to this, if objects are different .equals(-) method returns false eventhough content is same.

DEMO

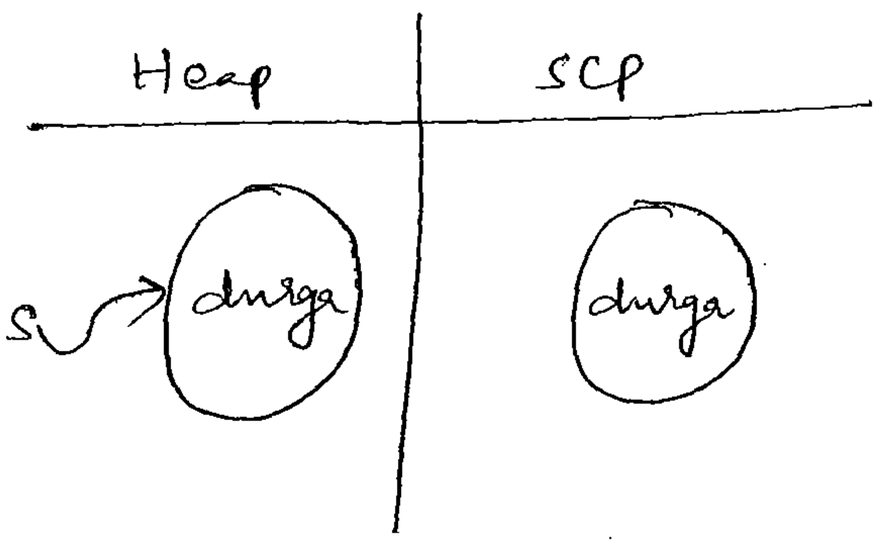
Case (iii):

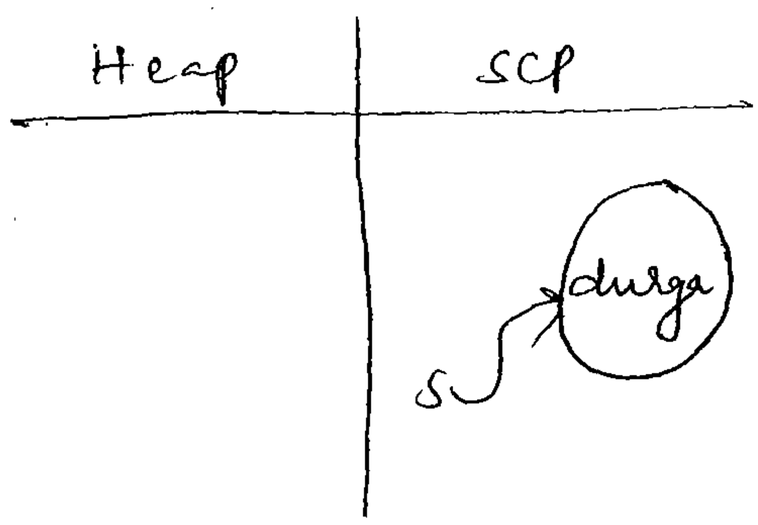
Ex①: String s=new String("durga");

In this case, 2 objects will be created. One is in the Heap & other is in SCP (String Constant Pool) and s is always pointing to Heap object.

String s="durga";

In this case, only one object will be created in SCP and s is always pointing to that object.





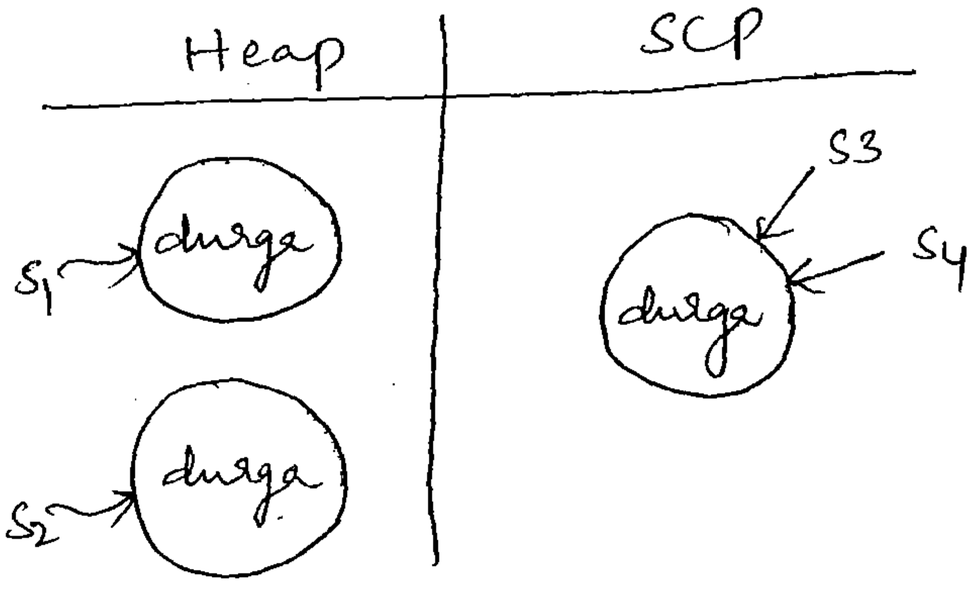
Note:- ① Garbage Collector is not allowed to access SCP area. Hence eventhough object doesn't contain any reference which is not eligible for GC if it is present in SCP area.

② All SCP objects will be destroyed automatically at the time of JVM shut down.

③ Object creation in SCP is always optional. First JVM will check is any object already present in SCP with required content or not. If it is already available then JVM will reuse that object. If it is not already available then only a new object will be created

Ex②: String s1=new String("durga");
String s2=new String("durga");
String s3="durga";
String s4="durga";

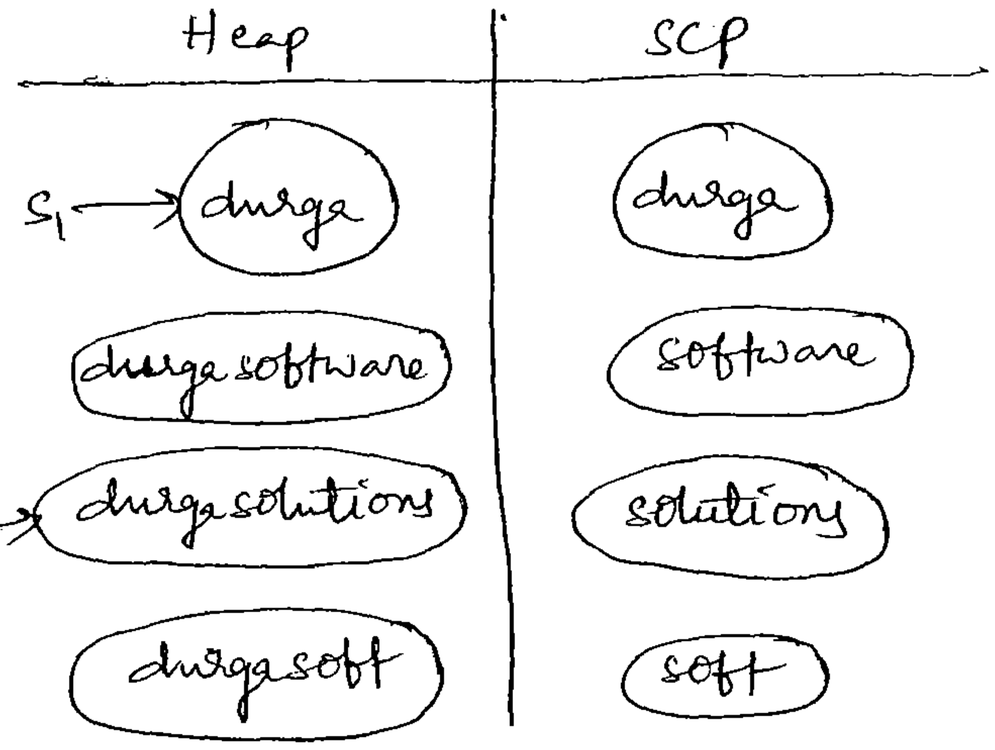
Total : 3 objects



Note:- ① Whenever we are using new operator compulsory a new object will be created on the Heap.

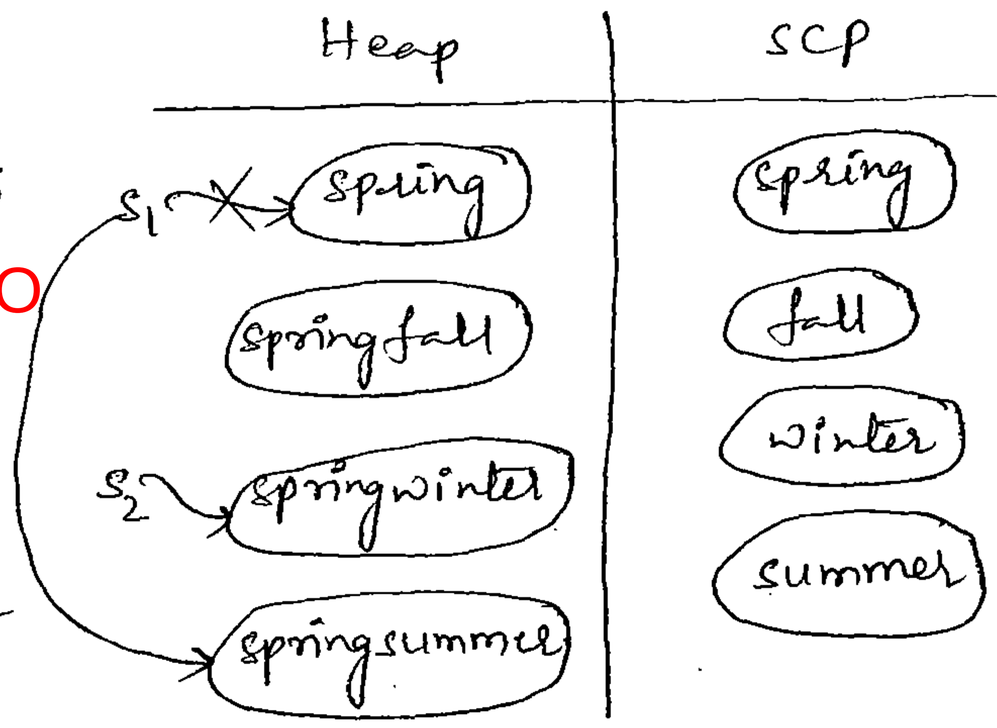
② There may be a chance of 2 objects with same content on Heap, but there is no chance of existing 2 objects with the same content on SCP i.e., duplicate objects are possible on the Heap, but not on SCP.

Ex③: String s1 = new String ("durga");
s1.concat("software");
String s2 = s1.concat("solutions");
s1.concat("soft");
S.o.p(s1); ⇒ o/p : durga
S.o.p(s2); ⇒ o/p : durgasolutions
Total : 8 objects



→ For every String constant, one object will be placed in SCP area
→ Becoz of some runtime operation like a method call if an object is required to create, it will be created only on the Heap, but not in SCP.

Ex④: String s1=new String("spring");
s1.concat("fall");
String s2 = s1+"winter";
s1=s1.concat("summer");
S.o.p(s1); ⇒ o/p : springsummer
S.o.p(s2); ⇒ o/p : springwinter
Total : 8 objects



Ex⑤: String s1=new String("you can't change me!");
String s2=new String("you can't change me!");
S.o.p(s1==s2); ⇒ o/p: false
String s3 = "you can't change me!";
S.o.p(s1==s3); ⇒ o/p : false
String s4 ="you can't change me!";

S.o.p (s3==s4); ⇒ o/p : true

String s5 = "you can't" + "change me!";

S.o.p (s3==s5); ⇒ o/p : true

String s6 = "you can't";

String s7 = s6 + "change me!";

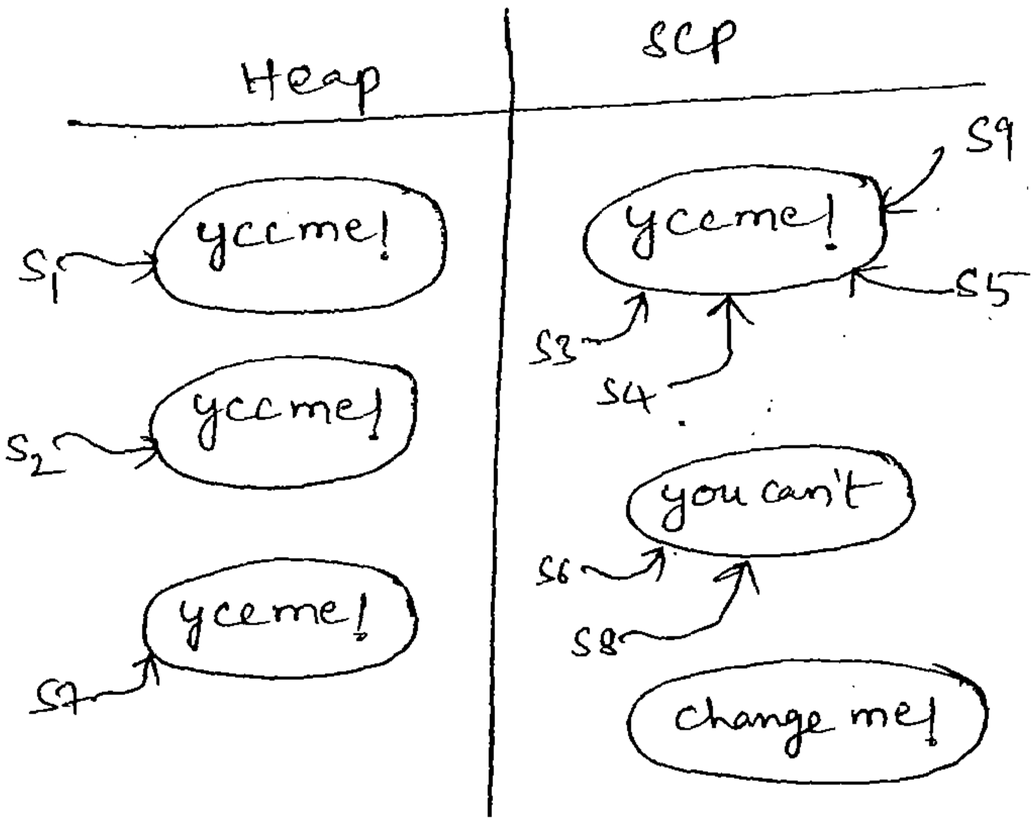
S.o.p (s3==s7); ⇒ o/p : false

final String s8 = "you can't";

String s9 = s8 + "change me!";

S.o.p(s3==s9) ⇒ o/p : true

S.o.p (s6==s8) ⇒ o/p : true



***

Note:- ① If all arguments are constants then that operation should be performed at compile time only.

② If atleast one argument is normal variable then that operation should be performed at runtime only.

***

Interning of String objects:-

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

Ex: ① String s1 = new String("durga");

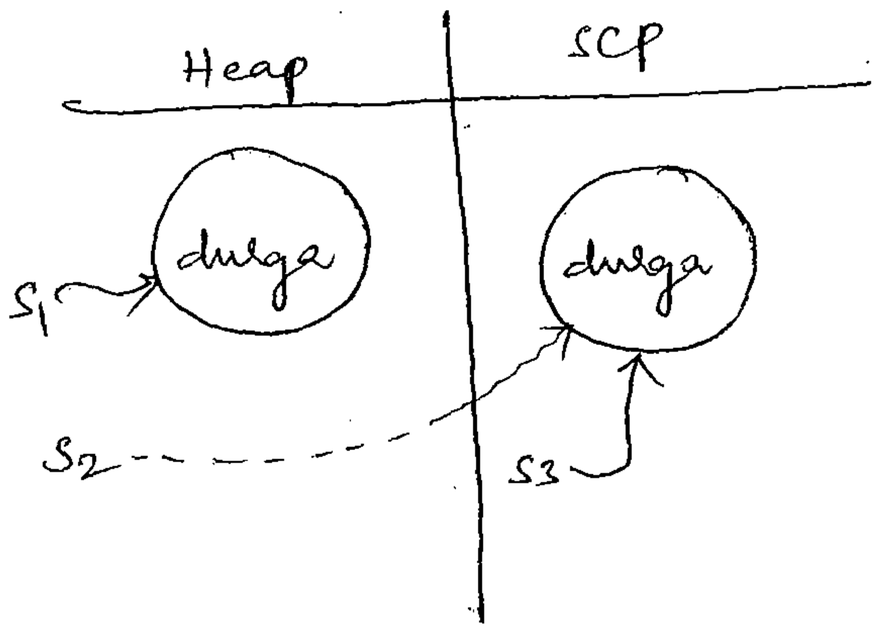
String s2 = s1.intern();

S.o.p (s1==s2); ⇒ o/p : false

S.o.p(s1);

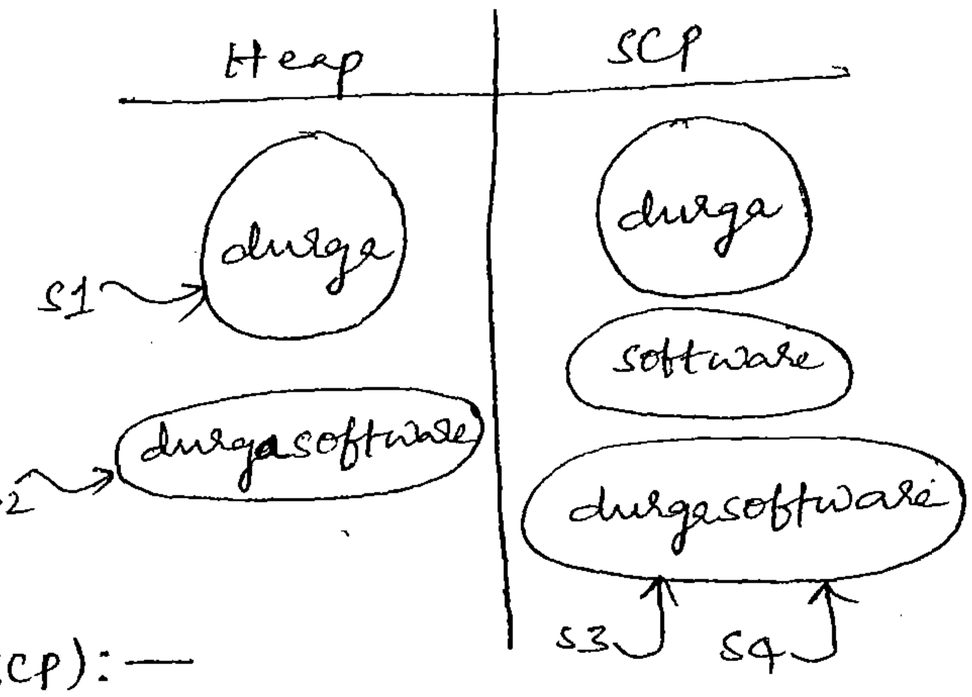
String s3 = "durga";

S.o.p(s2==s3); ⇒ o/p : true



→ If the corresponding SCP object is not available then intern() method will create that object & returns it.

Ex②: String s1 = new String("durga");
String s2 = s1.concat("software");
String s3 = s2.intern();
String s4 = "durgasoftware";
S.o.p(s3 == s4); ⇒ O/P : true

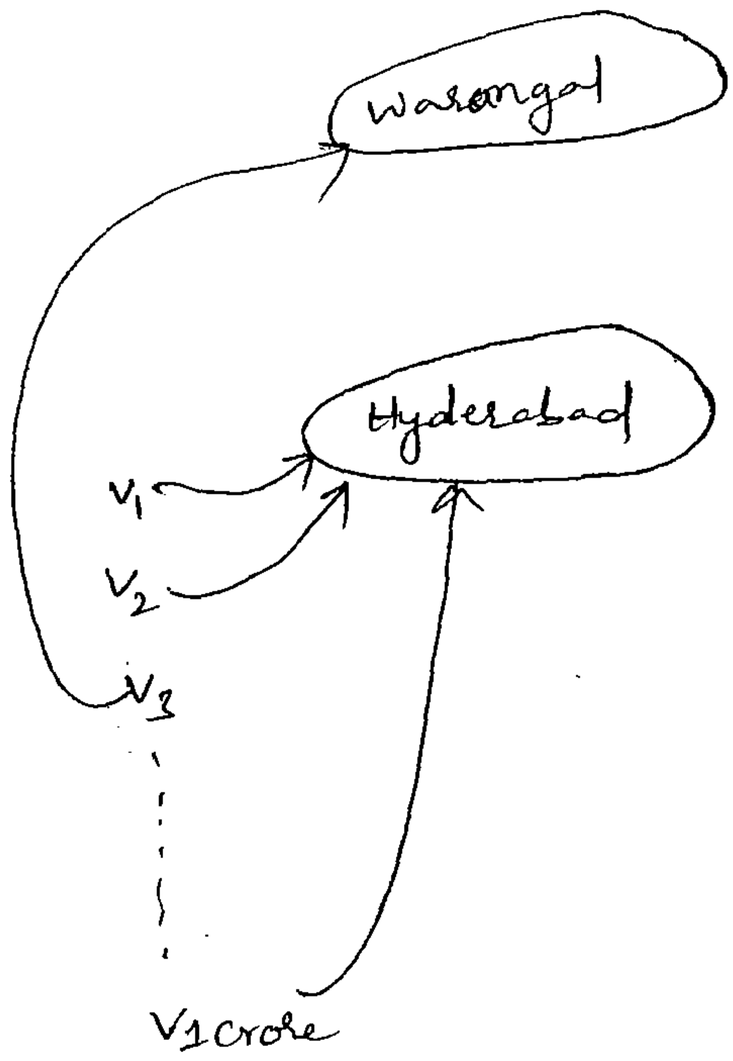


Importance of String Constant Pool (SCP):—

Voter Registration Form

Name: chiranjeevi
Age : 60
DOB : 22-08-1950
photo
Father Name: Venkat Rao
Mother Name: xxx
Address:
HNO: 22-3/425
Street: Banjara Hills
City : Hyd
Mandal: Hyd
Dist : Ranga Reddy
State: AP (TG)
Pin : 500026
land mark: —
Identification mark1: —
Identification mark2: —

DEMO



→ In our program, any String object is required to use repeatedly then it is not recommended to create a separate object for every requirement beeoz memory utilization & performance will be reduced.

→ Instead of creating a separate object every time we can create only one object & we can reuse the same object every time. So that performance and memory utilization will be improved.

→ We can achieve this by using SCP. Hence the main advantages of SCP are memory utilization & performance will be improved.

→ The main disadvantage of SCP is as several references pointing to same object in SCP by using one reference if we are trying to perform any change the remaining references will be impacted.

→ To overcome this problem SUN people defined String objects as Immutable.

DEMO

→ According to this Once we creating String object we can't perform any changes in the existing object. If we are trying to perform any changes then with those changes a new object will be created.

→ Hence SCP is the only reason why String objects are Immutable.

FAQ:

Q1: What is the difference blw String and StringBuffer?

Q2: Explain Immutability & Mutability with an example?

Q3: What is the difference blw `String s = new String("durga");` and `String s = "durga";` ?

Q4: Other than immutability & mutability is any other difference blw String & StringBuffer?

Ans: In String class, equals(-) method meant for content comparison

where as in StringBuffer, equals(-) method meant for reference comparison.

Q5: What is SCP?

Ans: A specially designed memory area for String constants.

Q6: What is the advantage of SCP?

Ans: Instead of creating a separate object for every requirement we can create only one object and we can reuse the same object for every similar requirement. So that performance & memory utilization will be improved.

Q7: What is the disadvantage of SCP?

Ans: As several references pointing to same object in SCP by using one reference if we are allowed to change in the content then remaining references will be impacted.

To overcome this problem we are forced to make String objects are immutable i.e., SCP is the only reason why String objects are immutable.

Q8: Why SCP like concept available only for String but not for StringBuffer?

Ans: String objects are most commonly used objects in Java. Hence SUN people defined specially designed memory area (SCP) for String objects.

StringBuffer objects are not commonly used objects in Java. Hence SUN people won't define any specially designed memory area for StringBuffer objects.

Q9: Why String objects are immutable? where as StringBuffer objects are mutable?

Ans: In case of String objects, just beccoz of SCP a single object is referred by multiple references.

By using one reference if we are allowed to change the content then remaining references will be impacted.

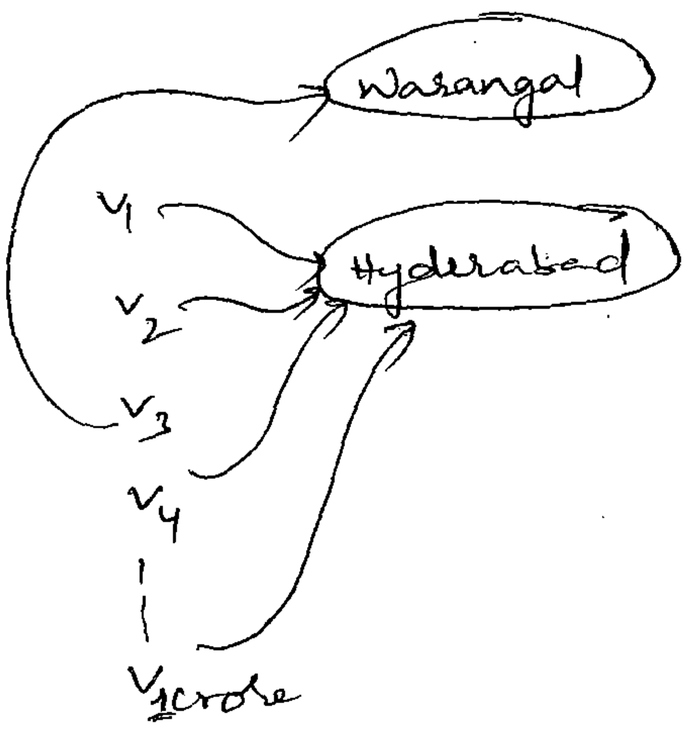
To overcome this problem SUN people made String objects as Immutable.

But in case of StringBuffer, for every requirement a separate object will be Created.

By using one reference if we are allowed to change the content then remaining references won't be impacted.

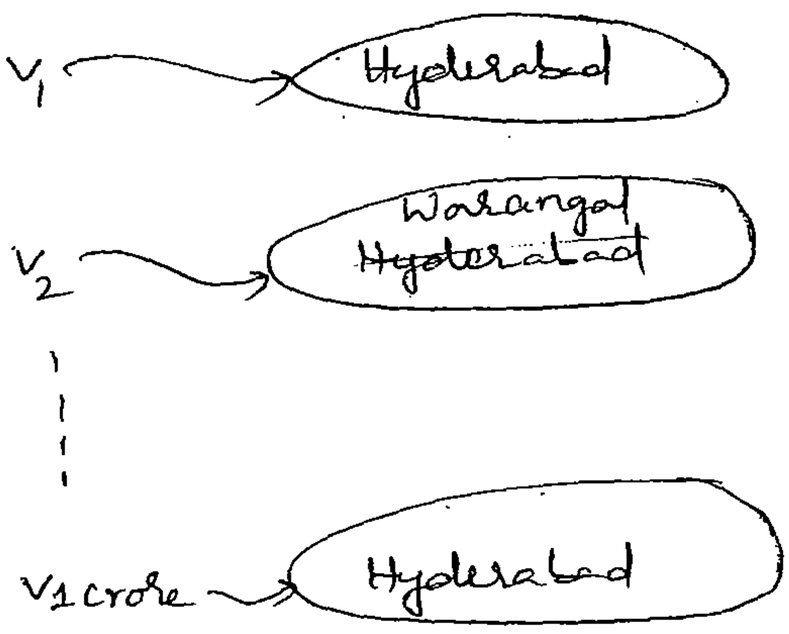
Hence immutability concept is not required for StringBuffer objects.

String



DEMO

StringBuffer



Q10: Similar to String objects is any other objects are immutable?

Ans: In addition to String objects, all wrapper class objects also immutable in Java.

Q11: Is it possible to create our own immutable class?

Ans: Yes.

Q12: Explain with an example how to create our own immutable class?

***Q13: final means non-changeable where as immutable means non-changeable then what is the difference b/w final & immutable?

String class Constructors:—

① String s=new String();

Creates an empty String object.

② String s=new String(String literal);

To create an equivalent String object on the Heap for the given String literal.

③ String s=new String(StringBuffer sb);

DEMO

To create an equivalent String object for the given StringBuffer.

④ String s=new String(char[] ch);

Create an equivalent String object for the given char[] array.

Ex: char[] ch={'a','b','c','d'};

String s=new String(ch);

S.o.p(s);=) O/P: abcd.

⑤ String s=new String(byte[] b);

Creates an equivalent String object for the given byte[] array.

Ex: byte[] b = {100, 101, 102, 103};

String s = new String(b);

S.o.p(s); ⇒ o/p : defg.

Important methods of String class :—

1. public char charAt(int index):—

returns the character locating at specified index.

Ex:— String s = "durga";

S.o.p(s.charAt(3)); ⇒ o/p : g

S.o.p(s.charAt(10)); → RE : ArrayIndexOutOfBoundsException.

2. public String concat(String s):—

→ The overloaded + and += operators also meant for concatenation only.

Ex: String s = "durga";

DEMO

s = s.concat("software");

// s = s + "software";

// s += "software";

S.o.p(s); ⇒ o/p : durgasoftware.

3. public boolean equals(Object o):—

→ To perform content comparison where case is important.

→ This is overriding version of Object class equals(-) method.

4. public boolean equalsIgnoreCase(String s):—

→ To perform content comparison where case is not important.

Ex: String s = "java";

S.o.p(s.equals("JAVA")); ⇒ o/p : false

S.o.p(s.equalsIgnoreCase("JAVA")); ⇒ o/p : true.

Note:- Usually we can use equalsIgnoreCase(-) method to compare User id where case is not important where as equals(-) method to compare passwords where case is important.

5. public String substring(int begin):-

→ Return substring from begin index to end of the String.

6. public String substring(int begin, int end):-

→ Return substring from begin index to end-1 of the String.

Ex:- String s="abcdefg";

S.o.p(s.substring(3)); ⇒ o/p: defg

S.o.p(s.substring(2,5)); ⇒ o/p: cde.

7. public int length():-

→ Return no. of characters present in the String.

Ex: String s="java";

DEMO

S.o.p(s.length); → CE: cannot find symbol
symbol: variable length
location: java.lang.String

S.o.p(s.length()); ⇒ o/p: 4.

***
Note:- length() method applicable for String objects where as length variable applicable for Array objects.

8. public String replace(char old, char new):-

Ex: String s="ababa";

S.o.p(s.repace('a','b')); ⇒ o/p: bbbbb

9. public String toLowerCase();

10. public String toUpperCase();

11. public String trim():-

→ It removes all blank spaces present at beginning & end of the String, but not middle blank spaces.

12. public int indexOf(char ch) :—

→ It returns index of first occurrence of specified character.

13. public int lastIndexOf(char ch):—

→ It returns index of last occurrence of specified character.

Ex: String s="ababa";

S.o.p(s.indexOf('a')); ⇒ o/p : 0

S.o.p(s.lastIndexOf('a')); ⇒ o/p : 4.

*** → Becoz of runtime operation if there is change in the content then with those changes a new object will be created on the Heap

If there is no change in the content new object won't be created existing object will be reused.

DEMO

This rule is same whether the current object present on Heap or SCP.

Ex: ① String s1=new ("durga");

String s2=s1.toUpperCase();

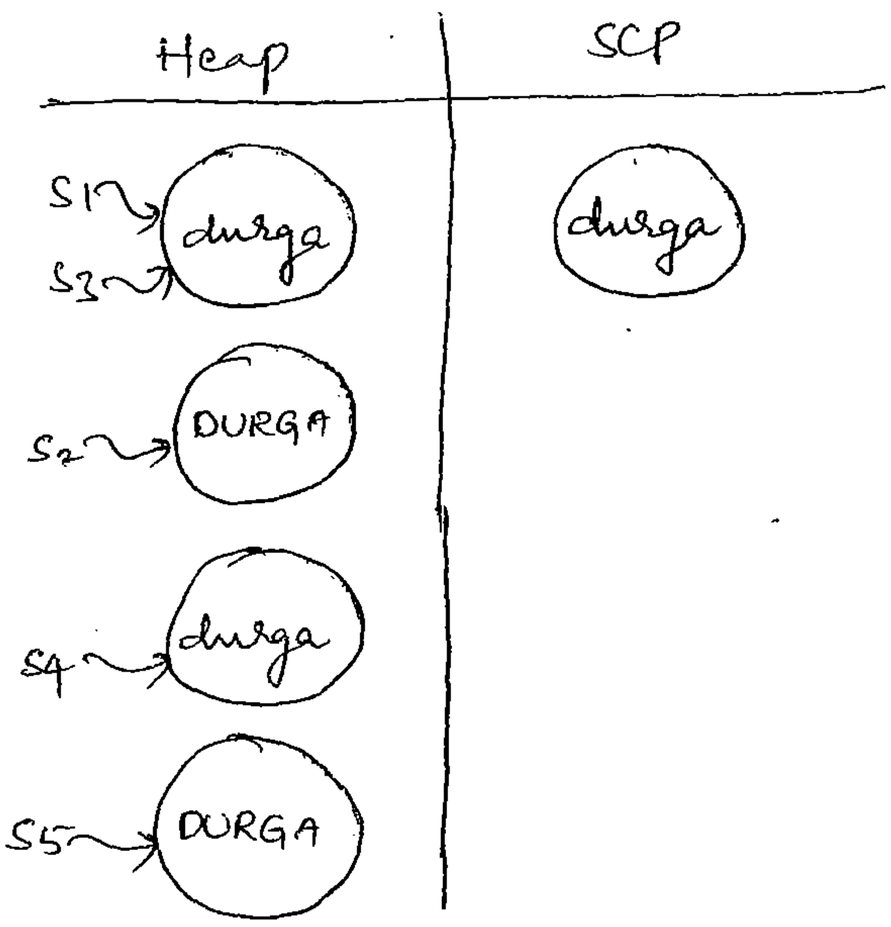
String s3=s1.toLowerCase();

S.o.p(s1==s2);//false

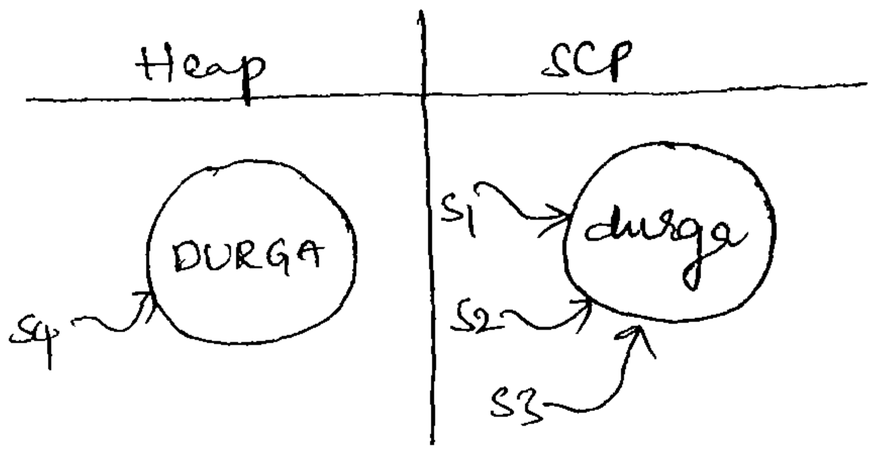
S.o.p(s1==s3); ⇒ o/p : true

String s4=s2.toLowerCase();

String s5=s4.toUpperCase();



Ex②: String s1 = "durga";
String s2 = s1.toString();
String s3 = s1.toLowerCase();
String s4 = s1.toUpperCase();



***
Creation of our own Immutable class :—

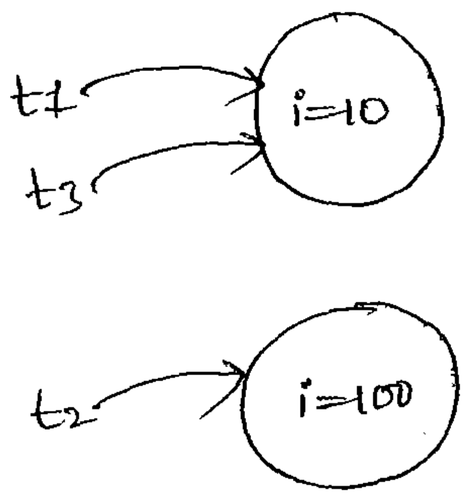
→ Once we created an object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created.

→ If there is no change in the content with our operation then existing object will be reused.

Ex:
```
final public class Test
{
    private int i;
    Test(int i)
    {
     this.i=i;
    }
    public Test modify(int i)
    {
        if (this.i==i)
            return this;
        else
            return new Test(i);
    }
}
```

DEMO

Test t1 = new Test(10);
Test t2 = t1.modify(100);
Test t3 = t1.modify(10);
S.o.p(t1==t2); ⇒ O/P : false
S.o.p(t1==t3); ⇒ O/P : true



→ Once we created a Test 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.

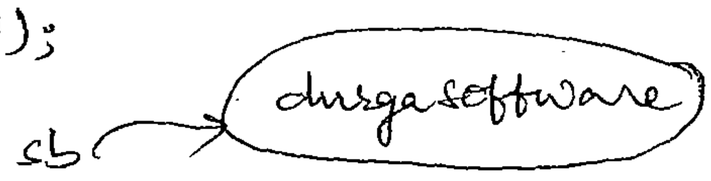
*** final Vs Immutability :—

→ final applicable for variables, but not for objects where as Immutability applicable for objects, but not for variables.

→ By declaring a reference variable as final we won't get any immutability nature in the corresponding object we can perform any type of changes.

→ But we can't perform reassignment for that referenee variable.

Ex: final StringBuffer sb = new SB("durga");
sb.append("software");
S.o.p(sb); ⇒ O/P : durgasoftware
SB sb = new SB("solutions"); → CE: cannot assign a value to final variable sb.



Q: Which of the following is meaningful?

1. final variable ✓
2. final object X
3. Immutable variable X
4. Immutable object ✓

4. StringBuffer :-

→ If the content is fixed and won't change frequently then it is the recommended to use String.

→ If the content is not fixed and keep on changing then it is not recommended to use String objects bcoz for every change a new object will be created internally.

→ To handle this requirement we should go for StringBuffer.

→ The main advantage of StringBuffer over String is all changes will be performed in the existing object only instead of creating new object.

Constructors :-

① `StringBuffer sb = new StringBuffer();`

Creates an empty StringBuffer object with default initial capacity 16.

DEMO

→ Once StringBuffer reaches its max. capacity then a new StringBuffer object will be created with

`new capacity = (current capacity + 1) * 2`

Ex: `StringBuffer sb = new StringBuffer();`

`S.o.p (sb.capacity());` ⇒ O/P : 16

`sb.append("abcdefghijklmnop");`

`S.o.p(sb.capacity());` ⇒ O/P : 16

`sb.append("q");`

`S.o.p (sb.capacity());` ⇒ O/P : 34

② `StringBuffer sb = new StringBuffer(int initialcapacity);`

Creates an empty StringBuffer object with the specified initial

*** capacity.

③ `StringBuffer sb=new StringBuffer(String s);`

Creates an equivalent StringBuffer object for the given String with

`capacity = s.length()+16`

Ex: StringBuffer sb=new StringBuffer("durga");

S.o.p(sb.capacity()); ⇒ o/p: 21

Methods :—

1. public int length();
2. public int capacity();
3. public char charAt(int index);

DEMO

Ex: StringBuffer sb=new StringBuffer("durga");

S.o.p(sb.charAt(3)); ⇒ o/p: g

S.o.p(sb.charAt(20)); → RE: StringIndexOutOfBoundsException.

4. public void setCharAt(int index, char ch);

To replace the character locating at specified index with provided character.

5. public StringBuffer append(String s)
   (int i)
   (float f)
   (double d)
   (boolean b)
   (Object o)

   } overloaded methods.

Ex: StringBuffer sb = new StringBuffer ();
sb.append("PI value is :");
sb.append (3.14);
sb.append ("It is exactly :");
sb.append (true);
S.o.p(sb); ⇒ o/p : PI value is : 3.14 It is exactly : true.

⑥ public StringBuffer insert(int index, String s);
(int index, int i);
(int index, float f);
(int index, double d);
(int index, boolean b);
(int index, Object o);

} Overloaded methods

Ex: StringBuffer sb=new StringBuffer("abcdefgh");
sb.insert (2, "xyz");
S.o.p (sb); ⇒ o/p : abxyzcdefgh

⑦ public StringBuffer delete(int begin, int end);

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

⑧. public StringBuffer deleteCharAt(int index);

⑨. public StringBuffer .reverse();

⑩ public void setLength(int length);

Ex: StringBuffer sb=new StringBuffer("aishwaryalakshmi");
sb.setLength(9);
S.o.p(sb); ⇒ o/p : aishwarya

***11. public void ensureCapacity(int capacity);

To increase capacity on fly based on our requirement.

Ex: StringBuffer sb = new StringBuffer();
S.o.p(sb.capacity()); ⇒ o/p : 16
sb.ensureCapacity(1000);
S.o.p(sb.capacity()); ⇒ o/p : 1000.

***12. public void trimToSize();

To deallocate extra allocated free memory.

Ex: StringBuffer sb = new StringBuffer(1000);
sb.append("abc");
sb.trimToSize();
S.o.p(sb.capacity()); ⇒ o/p : 3

*** Note:- Every method present in the StringBuffer is synchronized. Hence at a time only one thread is allowed to operate on StringBuffer. It increases waiting time of threads and creates performance problems.

To overcome this problem SUN people introduced StringBuilder in 1.5 version.

DEMO

5. StringBuilder:-

→ It is exactly same as StringBuffer (including constructors and ***methods) except the following differences.

StringBuffer	StringBuilder
1. Every method present in StringBuffer is synchronized.	1. Every method present in StringBuilder is non-synchronized.

StringBuffer	StringBuilder
2. At a time one thread is allowed to operate on StringBuffer object and hence StringBuffer object is Thread Safe.	2. At a time multiple threads are allowed to operate on StringBuilder object and hence it is not Thread Safe.
3. It increases waiting time of threads & hence relatively performance is low.	3. Threads are not required to wait & hence relatively performance is high
4. Introduced in 1.0 version.	4. Introduced in 1.5 version.

***

String Vs StringBuffer Vs StringBuilder :-

1. If the content is fixed and won't change frequently then we should go for String.
2. If the content is not fixed and keep on changing, but Thread Safety is required then we should go for StringBuffer.
3. If the content is not fixed and keep on changing, Thread safety is not required then we should go for StringBuilder.

Method Chaining:-

→ For most of the methods in String, StringBuffer & StringBuilder returntypes are same type only.

→ Hence after applying a method call on the result we can call another method which forms Method Chaining.

```
sb.m1().m2().m3().m4().m5().-------
```

→ In method chaining, all method calls will be performed from left to right.

Ex: StringBuffer sb = new StringBuffer();

sb.append("durga").append("software").append("solutions").

insert(2, "xeyz").delete(7,15).reverse().append("hyd");

S.o.p(sb);

6. Wrapper classes :—

→ The main objectives of Wrapper classes are

1. To wrap primitives into object form. so that we can handle primitives also just like objects.
2. To define several utility methods which are required for primitives.

Constructors :—

→ Almost all wrapper classes define 2 constructors. One can take corresponding primitive and the other can take String argument.

DEMO

Ex: ① Integer I = new Integer(10); → primitive

Integer I = new Integer("10"); → String

Ex ②: Double D = new Double(10.5); → primitive

Double D = new Double("10.5"); → String

→ If the String argument is not representing number then we will get RE saying NumberFormatException.

Ex: Integer I = new Integer("ten"); → RE: NumberFormatException.

→ Float class contains 3 constructors with float, double & String arguments.

Ex: Float f = new Float(10.5f); ✓

Float f = new Float("10.5f"); ✓

Float f = new Float(10.5); ✓

Float f = new Float("10.5"); ✓

→ Character class contains only one constructor with char primitive as argument.

Ex: Character ch = new Character('a'); ✓

Character ch = new Character("a"); X

→ Boolean class contains 2 constructors with boolean primitive & String arguments.

→ If we pass boolean primitive as argument then allowed values are true or false (where case should be in lowercase).

Ex: Boolean b = new Boolean(true); ✓

Boolean b = new Boolean(false); ✓

Boolean b = new Boolean(True); X

Boolean b = new Boolean(Durga); X

DEMO

→ If we pass String argument then content and case both are 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.

Ex: Boolean b = new Boolean("true"); ⇒ o/p: true

Boolean b = new Boolean("false"); → false

Boolean b = new Boolean("True"); → true

Boolean b = new Boolean("Durga"); → false

Ex: Boolean x = new Boolean("Yes"); → false

Boolean y = new Boolean("No"); → false

S.o.p(x); ⇒ o/p: false

S.o.p(y); ⇒ o/p: false

S.o.p(x==y); ⇒ o/p : false

S.o.p(x.equals(y)); ⇒ o/p : true

Wrapper class	Constructor arguments
Byte	byte or String
Short	short or String
Integer	int or String
Long	long or String
Float	float or String or double
Double	double or String
Character	char or ~~String~~
Boolean	boolean or String.

Note ①:- In all wrapper classes, toString() method is overridden to return its content.

② In all wrapper classes, equals(-) method is overridden for content comparison.

Utility Methods:-

1. valueOf()
2. xxxValue()
3. parseXxx()
4. toString()

1. valueOf():-

→ We can use valueOf() methods to create wrapper object for the given String or primitive, as alternative to constructor.

Form ①:

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

```
public static wrapper valueOf(String s);
```

Ex: Integer I = Integer.valueOf("10");

Double D = Double.valueOf("10.5");

Boolean B = Boolean.valueOf("durga");

Form ②:

→ Every Integral type wrapper class (Byte, Short, Integer, Long) contains the following valueOf() method to create Wrapper object for the given specified radix String.

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

DEMO

*** The allowed range of radix is 2 to 36

Ex: Integer I = Integer.valueOf("111", 2);

S.o.p(I); ⇒ o/p : 7

base-2 → 0 to 1
base-3 → 0 to 2
base-4 → 0 to 3
base-10 → 0 to 9
base-16 → 0 to 9, a to f
base-36 → 0 to 9, a to z

Form ③:

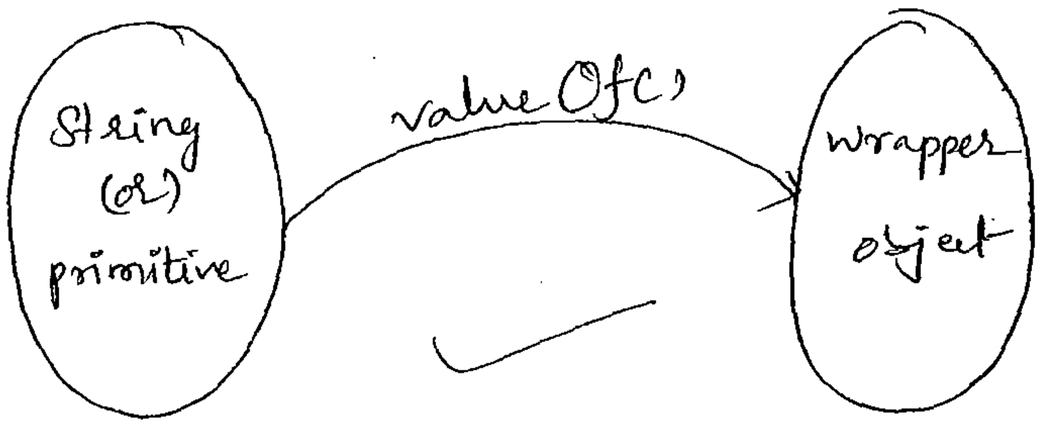
→ Every wrapper class including Character class defines the following valueOf() method to create wrapper object for the given primitive.

```
public static wrapper valueOf(primitive p);
```

Ex: Integer I = Integer.valueOf(10);

Character Ch = Character.valueOf('a');

Boolean B = Boolean.valueOf(true);



2. xxxValue():—

→ We can use xxxValue() method to find primitive values for the pri given wrapper object.

→ Every number type wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following xxxValue() method to find primitive for the given wrapper object.

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

DEMO

Ex: Integer I = new Integer(130);

S.o.p(I.byteValue()); ⇒ o/p : -126

S.o.p(I.shortValue()); ⇒ o/p : 130

S.o.p(I.intValue()); ⇒ o/p : 130

S.o.p(I.longValue()); ⇒ o/p : 130

S.o.p(I.floatValue()); ⇒ o/p : 130.0

S.o.p(I.doubleValue()); ⇒ o/p : 130.0

charValue():—

→ Character class contains charValue() method to find char primitive for the given Character object.

```
public char charValue()
```

Ex: Character ch = new Character('a');

char c = ch.charValue();

S.o.p(c); ⇒ o/p : 'a'.

booleanValue() :-

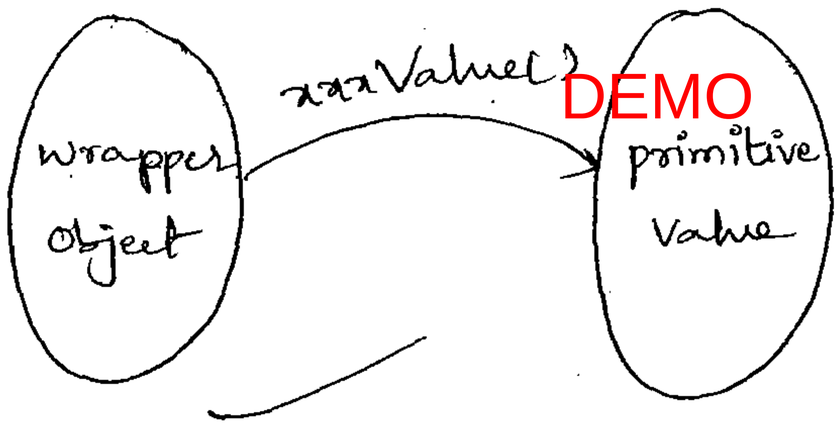
→ Boolean class contains booleanValue() method to find boolean primitive for the given Boolean object.

```
public boolean booleanValue()
```

Ex: Boolean B = Boolean.valueOf("durga");

boolean b = B.booleanValue();

S.o.p(b); ⇒ o/p : false.



Note:- In total, there are 38 (= 6x6+1+1) xxxValue() methods.

3. parseXxx() :-

→ We can use parseXxx() method to convert String to primitive.

Form ① :

→ Every wrapper class except Character class contains the following parseXxx() method to convert String to primitive.

```
public static primitive parseXxx(String s);
```

Ex: int i = Integer.parseInt("10");

Double d = Double.parseDouble("10.5");

```
boolean b = Boolean.parseBoolean("true");
```

Form②:

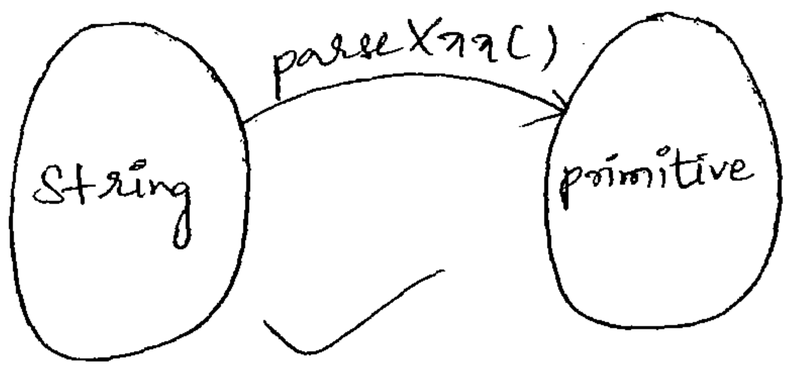
→ Every Integral type wrapper class contains the following parseXxx() method to convert specified radix String to primitive.

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

*** The allowed range of radix is 2 to 36.

Ex: `int i = Integer.parseInt("100",2);`

`S.o.p(i);` ⇒ o/p : 4.



DEMO

4. toString :—

→ We can use toString() method to convert wrapper object & primitives to String.

Form①:

→ Every wrapper class contains the following toString() method to convert wrapper object to String.

```
public String toString()
```

→ It is the overriding version Object class toString() method.

→ Whenever we are trying to print any wrapper object reference internally this method will be called.

Ex: `Integer I = new Integer(10);`

`S.o.p(I);` ⇒ `(I.toString());` → o/p : 10.

String s = I.toString();

S.o.p(s); ⇒ o/p : 10

Form ②:

→ Every wrapper class including Character class contains the following static toString() method to convert primitive to String.

public static String toString(primitive p)

Ex: String s = Integer.toString(10);

String s = Boolean.toString(true);

String s = Character.toString('a');

Form ③:

→ Integer & Long classes contain the following toString() method to convert primitive to specified radix String form.

public static String toString(primitive p, int radix)

DEMO

Ex: String s = Integer.toString(7,2);

S.o.p(s); ⇒ o/p : 111.

Form ④: toXxxString() :-

→ Integer & Long classes contain the following toXxxString() methods.

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

Ex①: String s = Integer.toBinaryString(10);

S.o.p(s); ⇒ o/p : 1010.

2 | 10
2 | 5 — 0
2 | 2 — 1
2 | 1 — 0

1010

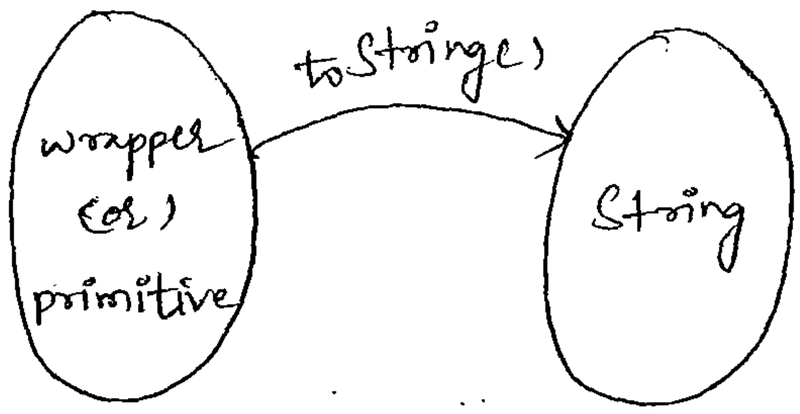
Ex②: String s = Integer.toOctalString(10);

S.o.p(s); ⇒ o/p : 12

8 | 10
1 -2 ↑ ⇒ 12

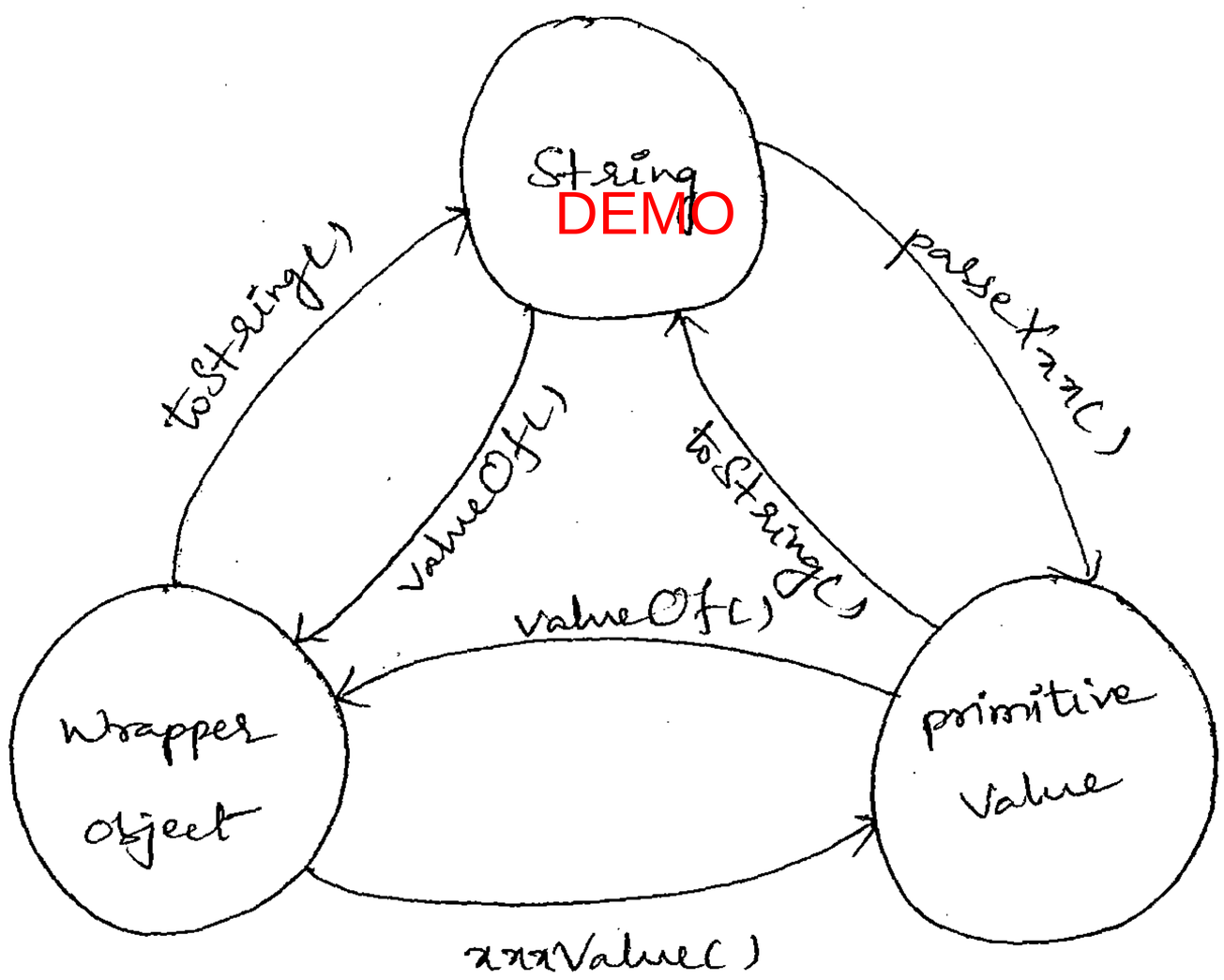
Ex③: String s = Integer.toHexString(10);

S.o.p(s); ⇒ o/p : a

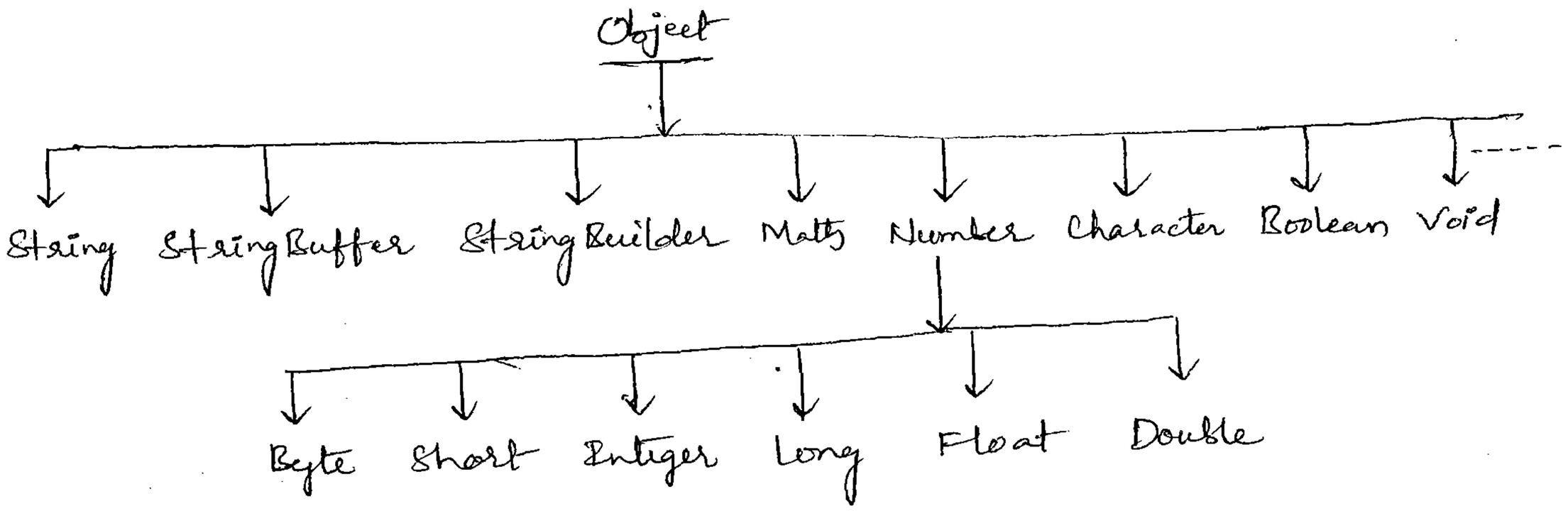


***

Dancing blw wrapper object, primitive and String:-



Partial hierarchy of java.lang package :—



1. The wrapper classes which are not child class of Number are Character and Boolean.
2. Wrapper classes which are not direct child class of Object are Byte, Short, Integer, Long, Float and Double.
3. In addition to String, all wrapper objects are immutable.
4. All wrapper classes, String, StringBuffer and StringBuilder are final classes.

DEMO

Void class :—

→ Sometimes we may consider Void class also wrapper class.

→ It is the direct child class of Object and it is the final class.

→ It doesn't contain any methods and it contains only one variable TYPE.

```
public static final java.lang.Class TYPE;
```

→ We can use Void class in Reflections to check whether return type of method is void or not.

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

7. Auto boxing & Auto unboxing :—

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

Ex ①:
```
ArrayList l=new AL();
l.add(10); → CE:

Integer I=new Integer(10);
l.add(I); ✓
```

Ex ②:
```
Boolean B=new Boolean("true");
if(B)
{
=
}
```
→ CE: incompatible types
found: j.l.Boolean
required: boolean

DEMO

```
boolean b=B.booleanValue();
if(b)
{
=
}
```
✓

→ But from 1.5 version onwards, we can provide primitive value in the place of wrapper object & wrapper object in the place of primitive. All required conversions will be performed automatically by the compiler.

→ These automatic conversions are called Auto boxing and Auto unboxing.

Auto boxing :—

→ Automatic conversion of primitive to wrapper object by compiler is called Autoboxing.

Ex: Integer I=10; [Compiler converts into to Integer automatically by Autoboxing]

→ After compilation the above line will become

Integer I = Integer.valueOf(10); i.e.,

→ Internally Autoboxing is implemented by using valueOf() method.

Autounboxing :—

→ Automatic conversion of wrapper object into primitive by compiler is called Auto unboxing.

Ex: Integer I = new Integer(10);

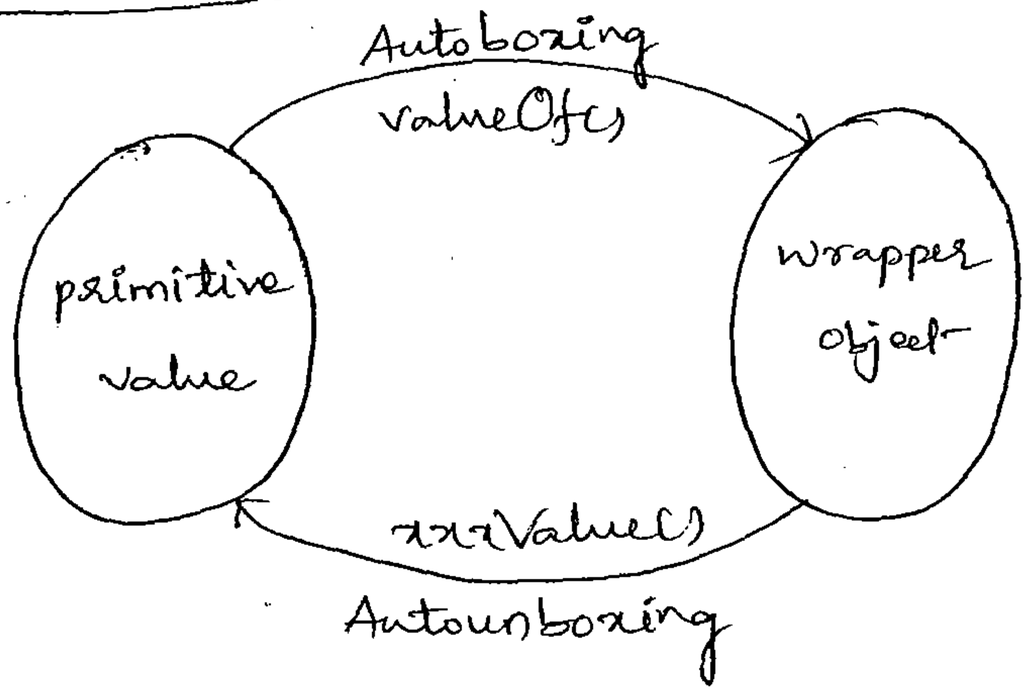
int i = I; [Compiler converts Integer to int automatically by Autounboxing]

DEMO

→ After compilation the above line will become

int i = I.intValue(); i.e.,

→ Internally Autounboxing concept implemented by using xxxValue() method.



Ex①:
```
class Test
{
    static Integer I=10;  ---> ① A.B
    p s v m(-)
    {
        int i=I;  ---> ② A.U.B
        m1(i);  ---> ③ A.B.
    }
    public static void m1(Integer I)
    {
        int k=I;  ---> ④ A.U.B.
        S.o.p(k); => o/p : 10
    }
}
```

→ It is valid in 1.5 version, but invalid in 1.4 version.

Note:- From 1.5 version onwards we can use primitives and wrapper objects interchangeably.

DEMO

Ex②:
```
class Test
{
    static Integer I=0;
    p s v m(-)
    {
        int k=I;
        S.o.p(k); => o/p : 0
    }
}
```

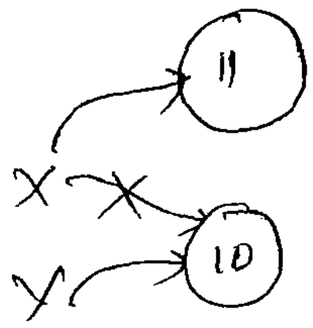
```
class Test
{
    static Integer I;
    p s v m(-)
    {
        int k=I;  ---> int k=I.intValue();
                                ↓
                               null
        S.o.p(k); ---> RE : NPE
    }
}
```

Note: ① If we are trying to perform Autounboxing for null references we will get NullPointerException.

② The default value for int type is 0 (zero) where as for Integer type is null.

Ex③:
```
Integer X=10;
Integer Y=X;
```

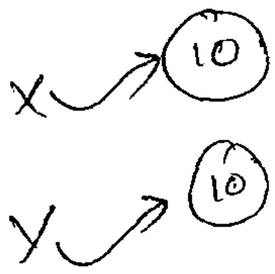
```
X++;
S.o.p(x); ⇒ o/p : 11
S.o.p (Y); ⇒ o/p : 10
S.o.p (X==Y); ⇒ o/p : false
```



Note:- All wrapper objects are immutable i.e., once we created a wrapper object we can't perform any change in the existing object. If we are trying to perform any change with those changes a new object will be created.

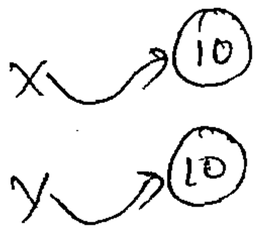
Ex④:

ⓐ
```
Integer X = new Integer(10);
Integer Y = new Integer(10);
S.o.p(X==Y); ⇒ o/p : false
```

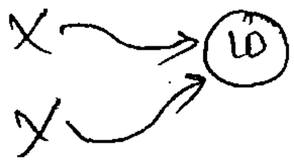


ⓑ
```
Integer X=new Integer(10);
Integer Y=10;
S.o.p(X==Y); ⇒ o/p : false
```

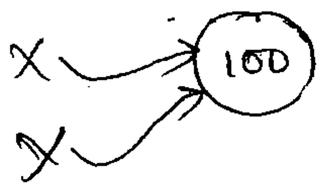
DEMO



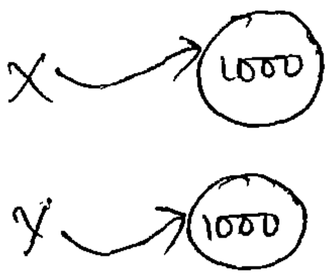
ⓒ
```
Integer X=10;
Integer Y=10;
S.o.p(X==X); ⇒ o/p : true
```



ⓓ
```
Integer X=100;
Integer Y=100;
S.o.p(X==Y); ⇒ o/p : true
```



ⓔ
```
Integer X=1000;
Integer Y=1000;
S.o.p(X==Y); ⇒ o/p : false
```



Conclusion:-

→ Internally to provide support for autoboxing a Buffer of wrapper objects will be created at the time of wrapper class loading.

→ By Autoboxing, if an object is required to create first it will check is it already there in the buffer or not.

→ If it is already available then existing buffer object will be reused.

→ If it is not present in buffer then only a new object will be created.

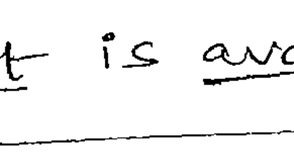
→ But buffer concept is available only in the following cases.

1.	Byte	→ Always
2.	Short	→ -128 to 127
3.	Integer	→ -128 to 127
4.	Long	→ -128 to 127
5.	Character	→ 0 to 127
6.	Boolean	→ Always

DEMO

→ Except this range in all remaining cases a new object will be created.

Ex: Integer x=127;
Integer y=127;



S.o.p(x==y); ⇒ o/p : true

Integer x=128;
Integer y=128;
S.o.p (x==y); ⇒ o/p : false

Boolean x=true;
Boolean y=true;
S.o.p (x==y); ⇒ o/p : true

Double x=10.0;
Double y=10.0;
S.o.p (x==y) ⇒ o/p : false

→ Internally Autoboxing concept is implemented by using valueOf() method.

→ Hence Buffering concept is applicable for valueOf() methods also.

Ex: Integer X = new Integer(10);
Integer Y = new Integer(10);
S.o.p(X==Y) ⇒ o/p : false

Integer X = 10;
Integer Y = 10;
S.o.p(X==Y) ⇒ o/p : true

Integer X = Integer.valueOf(10);
Integer Y = Integer.valueOf(10);
S.o.p(X==Y); o/p : true.

Integer X = Integer.valueOf(10);
Integer Y = 10;
S.o.p(X==Y); o/p : true

***

Overloading w.r.t widening, var---arg & Autoboxing :-

Case (i): Widening Vs Autoboxing :-

Ex: class Test DEMO
{
public static void m1(long l)
{
S.o.p("widening");
}
p s v m1(Integer I)
{
S.o.p("Autoboxing");
}
p s v m(-)
{
int x=10;
m1(x); ⇒ o/p : widening.
}
}

→ Widening dominates Autoboxing

Case(ii): Widening Vs var-arg method :-

```
Ex: class Test
    {
      p s v m1(long l)
      {
        S.o.p("widening");
      }
      p s v m1(int... i)
      {
        S.o.p("var-arg");
      }
      p s v m(-).
      {
        int x=10;
        m1(x); => O/P : widening
      }
    }
```

→ widening dominates var-arg method.

Case (iii): Autoboxing Vs var-arg method :- DEMO

```
Ex: class Test
    {
      p s v m1(Integer I)
      {
        S.o.p("Autoboxing");
      }
      p s v m1(int... i)
      {
        S.o.p("var-arg");
      }
      p s v m(-)
      {
        int x=10;
        m1(x); => O/P : Autoboxing.
      }
    }
```

→ Autoboxing dominates var-arg method.

→ In general var-arg method will get least priority i.e., if no other method matched then only var-arg method will get chance.

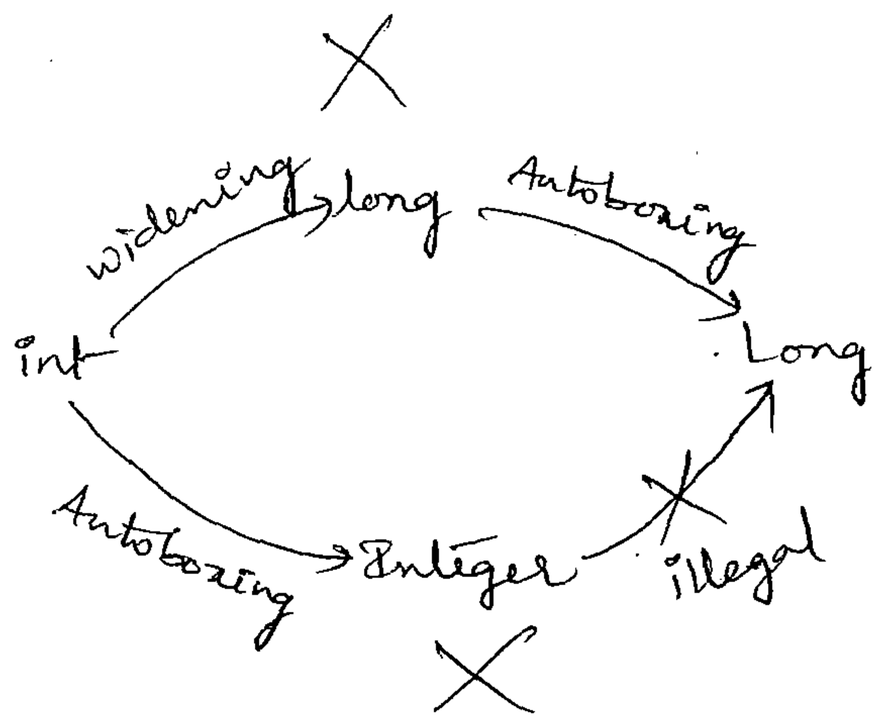
→ It is exactly same as default case inside switch.

*** Note:- While resolving overloaded methods compiler will always gives the precedence in the following order.

1. Widening
2. Autoboxing
3. var-arg method.

Case (iv):

Ex:
```
class Test
{
    p s v m1(Long l)
    {
        S.o.p("Long");
    }
    p s v m(-)
    {
        int x=10;
        m1(x); → CE: m1(j.l.Long) in Test can't be applied to (int)
    }
}
```



DEMO

*** → Widening followed by Autoboxing is not allowed in Java, but Autoboxing followed by widening is allowed.

Case (v):

Ex:
```
class Test
{
    p s v m1(Object o)
    {
        S.o.p("Object");
    }
```

```
p s v m(—)
{
  int x=10;
  m1(x); ⇒ O/P : Object
}
```

int ⟶ Integer ⟶ Object
Autoboxing ✓ widening

Case(vi):

Q: which of the following assignments are valid?

int i=10; ✓

Integer I=10; ✓ (Autoboxing)

int i=10L; ⟶ CE: PLP found: Long required: int

Long l=10L; ✓

Long l=10; X → (Autoboxing)

long l=10; ✓

Object o=10; ✓

double d=10; ✓

Double D=10; X

Number n=10; ✓

CE: incompatible types found: int required: j.l.Long

DEMO

widening

Autoboxing followed by widening

widening

CE: incompatible types found: int required: j.l.Double

Autoboxing followed by widening

DEMO