java.lang Package

1. Introduction
2. Object class
3. String class
4. StringBuffer class
5. StringBuilder class
6. Wrapper classes
7. Autoboxing & Autounboxing

✳ Introduction

⇝ For writing any java program, whether it is simple or complex, the most commonly required classes & Interfaces are grouped into a separate package, which is nothing but java.lang package

⇝ we are not required to import java.lang package explicitly, bcz all classes & interfaces present in lang package by default available to every java program

Ex.

```
class Test (1)
{
    public static void main(String[] arg) (2)
    {
        System (3).out.println("Hello" (4));
    }
}
```

java.lang.Object class

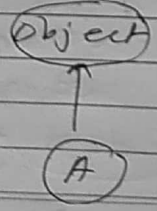
→ The most commonly required methods for every java class (whether it is pre-defined class or customized class) are defined in a separate class which is nothing but Object class.

→ Every class in java is the child class of Object either directly or indirectly so that Object class methods by default available to every java class. Hence, Object class is considered as Root of all java classes.

Note :

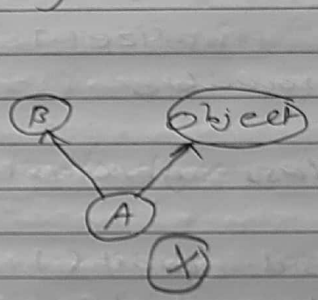
① If our class doesn't extend any other class then only our class is direct child class of Object

```
class A
{

}
```

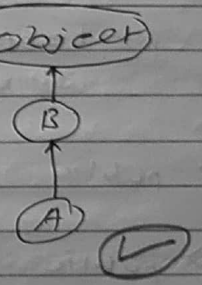


② If our class extends any other class then our class is indirect child class of Object

```
class A extends B
{

}
```



Multiple Inheritance



Multilevel Inheritance

→ Conclusion : Either directly (or) Indirectly Java don't provide support for multiple Inheritance w.r.t classes.

→ Object class defines the following 11 methods :-

Remember

1) public String toString()

Scanned with CamScanner

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();

Note:- Strictly speaking Object class contains 12 methods. The extra method is :-

registerNatives()

private static native void
registerNatives();

This method internally required for Object class & Not available to child classes (As it is private) Hence we are not required to consider this method

① toString()

→ we can use toString() method to get string representation of an object.

String s = Obj.toString();

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

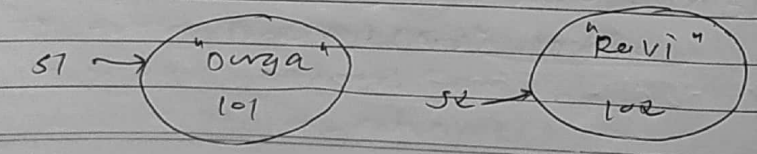
```
Student st = new Student();
S.O.P(S); ⇒ S.O.P(s.toString());
```

Scanned with CamScanner

~ If our class doesn't containing toString() method, then Object class toString() method will be executed.

Ex:

```
class Student
{
    public static void main(String[] args)
    {
        Student s1 = new Student
                        ("Durga", 101);
        Student s2 = new Student
                        ("Ravi", 102);

        S.O.P(s1);
        S.O.P(s1.toString());
        S.O.P(s2);
    }
    String name;
    int rollNo;

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



O/P:
Student@1888759
Student@1888759
Student@6e1408

~ In the above example Object class toString() method got executed, which is implemented as follows:-

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

(classname @ hashcode_in_hex_form)

~ Based on our requirement we can override toString() method to provide our own string representation.

~ For example, whenever we are trying to print Student reference to print his name & rollNo, we have to override toString() method as follows:-

Scanned with CamScanner

```
public String toString()
{
    return name + "....." + rollNo;
}
```

O/P :- Durga ... 101
Ravi ... 102

⇒ In all Wrapper classes, In all Collection classes, String class, StringBuffer & StringBuilder class toString() method is overridden for meaning string representation, Hence it is highly recommended to override toString() method in our class also.

```
class Test
{
    public String toString()
    {
        return "test";
    }
    p s v m(Str)
    {
        String s = new String("Abc");
        S.O.P(s); // Abc
```

```
        Integer I = new Integer(10);
        S.O.P(I); // 10

        AL al = new AL();
        l.add("A"); l.add("B");
        S.O.P(al); [A,B]

        Test t = new Test();
        S.O.P(t); // test
    }
}
```

★ How to find NO. of Methods in any class?

```
import java.lang.reflect.*;
class Test
{
    public static void main(String[] arg)
                                throws Exception
    {
        int count=0;
        Class c = Class.forName
                    ("java.lang.Object");

        Method[] m =
            c.getDeclaredMethods();

        for (Method m1 : m)
        {
```

Scanned with CamScanner

```
        count++;
        S.O.P(m1.getName());
    }
    S.O.P("No. of Methods : " + count);
  }
}
```

O/P :- registerNatives
getClass
hashCode
equals
clone
toString
notify
notifyAll
wait
wait
wait
finalize
The No. of methods : 12

[Note :- Method, class is belongs to java.lang.reflect package]

(2) hashCode()

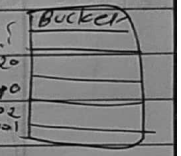
→ For every object, A [unique] number generated by JVM, which is nothing but hashCode.

→ HashCode won't represent address of Object.

→ JVM will use hashcode while saving objects into Hashing related Data structure like Hashtable, HashMap, HashSet etc.

→ The main Advantage of saving Objects based on Hashcode is search operation will become easy.

→ The most powerful search Algorithm upto today is ["Hashing"]



which has time complexity = O(1)

[Other searching Algorithm
(1) Linear search = O(n)
(2) Binary search = O(log n) + time require to sort the elements]

→ Using Java, we can never know the exact memory address of any object, means it is impossible (& size of object as well)

Bcz, Java is programmer friendly language unlike Machine friendly language like C & C++.

~ So, Hashcode is just a number assigned by JVM to object.

~ If you are giving the chance to object class hashcode() method, it will generate Hashcode based on Address of the object (Though it doesn't mean Hashcode represents address of object)

(like, guess our object's address is 1024
then hashcode() might use some algorithm randomly like

(int) (((1024 * 2 * 1.5) / 8.2) / 3.2)
Address

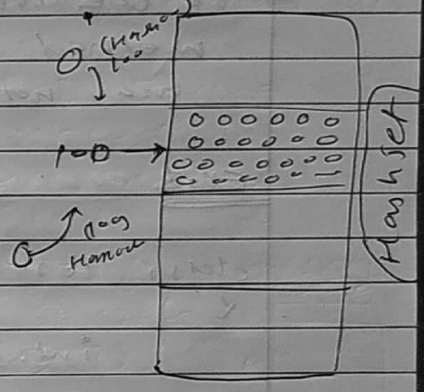
= 132
↳ Hashcode)

~ Based on our requirement we can override hashcode() in our class to generate Hashcode. (our own)

~ overriding hashcode method is said to be proper if & only if it generates unique number for each objects.

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

Improper Way

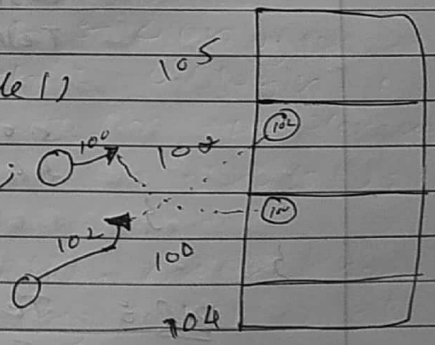


Bcz, For all student object we are generating same No. as hashcode.

So, problems remains.

- class Student
{
public int hashCode()
{
return rollNo;
}
}

Properway



Scanned with CamScanner

toString() vs hashCode()

- If we are giving chance to Object class toString() method, it will internally calls hashcode method.

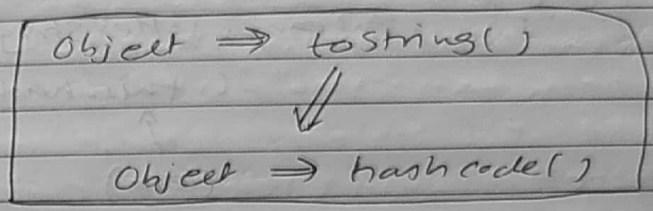
- If we are overriding toString() method then our toString() method may not call hashcode() method.

Ex-1

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

        S.O.p(t1);
        S.O.p(t2);
    }
}
```

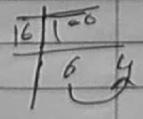
O/p : Test@abce
Test@bced

We can't predict o/p here.



Ex-2

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

        S.O.p(t1);
        S.O.p(t2);
    }
}
```



O/p: Test@a
Test@64

Scanned with CamScanner

Date / /
Page

∴ In toString() method of Object

→ Integer.toHexString
(this.hashCode());
↑
(Test)

Integer.toHexString(10); &
Integer.toHexString(100);

we can predict o/p here.

Object ⇒ toString()
↓
Test ⇒ hashCode()

Ex.3

```
class Test
{
    int i;
    Test(int i)
    {
        this.i = i;
    }
    public String toString()
    {
        return i+" ";
    }
```

Date / /
Page

```
    public int hashCode()
    {
        return i;
    }
    p s v m(sm)
    {
        Test t1 = new Test(10);
        Test t2 = new Test(100);

        S.O.P(t1);
        S.O.P(t2);
    }
}
```

O/P ⇒ 10
100

Test ⇒ toString()

(3) equals()

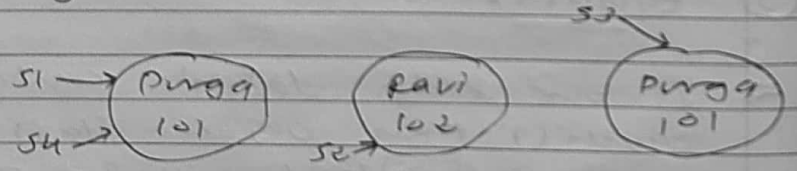
→ If our class doesn't contains equals() then Object class equals() method will be executed.

Ex.

```
class Student
{
    String name;
    int no;
```

Scanned with CamScanner

```
Student (String name, int rno)
{
    this.name = name;
    this.rno = rno;
}
psvm(SW)
{
    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)); // false
    S.O.p(s1.equals(s3)); // false
    S.O.p(s1.equals(s4)); // true

}
}
```



⇝ In above example, Object class equals() method got executed which also meant for reference/address comparison, which means if two references pointing to the same object then only .equals() method returns true.

⇝ Based on our requirement, we can override equals() method for content comparison.

⇝ By overriding equals() method for content comparison, we have to take care about the following:

1) What is the meaning of equality (whether we have to check only names or only rno or both)

2) If we are passing different type of object, our equals() method should not rise CCE. (i.e we have to handle CCE to return false)

3) If we are passing null argument then our equals() method should not rise NPE (i.e we have to handle NPE to return false)

Scanned with CamScanner

Date / / Page

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

```
public boolean equals(Object obj)
{
    try
    {
        String name1 = this.name;
        int rno1 = this.rno;

        Student s = (Student)obj;

        String name2 = s.name;
        int rno2 = s.rno;

        if(name1.equals(name2) &&
                rno1 == rno2)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    catch(ClassCastException e)
    {
        return false;
    }
    catch(NullPointerException e)
    {
        return false;
    }
}
```

```
s.o.p(s1.equals(s2)); false
s.o.p(s1.equals(s3)); true
s.o.p(s1.equals(s4)); true
s.o.p(s1.equals("durga")); false
s.o.p(s1.equals(null)); false
```

⇒ simplified return of equals() method:

```
public boolean equals(Object obj)
{   try{
    Student s = (Student) obj;

    if(name.equals(s.name) &&
            rno==s.rno)
    {
        return true;
    }
    else
    {
        return false;
    }
}
catch(CCE e){ return false; }
catch(NPE e){ return false; }
}
```

Scanned with CamScanner

Bcz,

```
public void m1()          // instance method
{
    s.o.p(x);  ⇒  s.o.p(this.x);
}
```

⇝ Move simplified version of equals method :-

```
public boolean equals(Object obj)
{
    if (obj instanceof Student)
    {
        Student s = (Student) obj;

        if (name.equals(s.name) &&
                rno == s.rno)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    return false;
}
```

⇝ To make above equals() method more efficient we have to write following at the begining inside equals() method.

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

According to this if both references pointing to same object then without performing any comparison .equals() method returns true directly.

⇝

```
String s1 = new String("durga");
String s2 = new String("durga");

s.o.p(s1 == s2);                // false
s.o.p(s1.equals(s2));           // true
```

```
StringBuffer sb1 = new StringBuffer
                        ("durga");
StringBuffer sb2 = new StringBuffer
                        ("durga");
s.o.p(sb1 == sb2);  // false
s.o.p(sb1.equals(sb2));  // false
```

Scanned with CamScanner

In String class .equals() method it overridden for content comparison Hence, even though Object's are different, based on content of object .equals() method returns true.

In StringBuffer .equals() method it Not overridden for content comparison, Hence if object are different .equals() method returns false, even though content is same.

④ getClass()

→ we can use getClass() method to get Runtime class definition of an object.

```
public final Class getClass()
```

→ By using this Class class object we can access, class level properties like fully qualified name of class, method info, constructors info etc.

```
import java.lang.reflect.*;

class Test
{
    p s v m (Str[] a)
    {
        int count = 0;
        Object O = new String("durga");
        Class c = O.getClass();
        S.O.P("Fully Qualified Class Name :" + c.getName());

        Method[] m = c.getDeclaredMethods();

        S.O.P("Methods information:");

        for(Method mt : m)
        {
            count++;
            S.O.P(mt.getName());
        }
        S.O.P("No. of method :" + count);
    }
}
```

O/P :

```
Fully Qualified Class Name :
    java.lang.String
Methods Info :
    equals
    hashCode
```

Scanned with CamScanner

toString()

No. of Methods : 73

Ex.

To Display Database vendor specific connection interface implemented class Name.

```
Connection cn = DriverManager.
                getConnection(.....);
S.o.p(cn.getClass().getName());
```

★ Note :-

(1) After loading every .class file JVM will create an object of the type java.lang.Class in the Heap area., programmer can use this class object to get class level info".

(2) We can use getClass() method very frequently in reflection API.

(5) finalize()

→ Just before destroying an object Garbage collector calls finalize() method, to perform cleanup activities.

↝ once finalize() method completes, Automatically Garbage Collector destroys that object.

(6) wait(), notify(), notifyAll()

↝ we can use this methods for inter thread communication.

↝ The Thread which is expecting updation, it is responsible to call wait() method & then immediately the thread will enter into waiting state.

↝ The thread which is responsible to perform updation, After performing updation, the thread can call notify method, the waiting thread will get that notification & continue it execution with those updates.

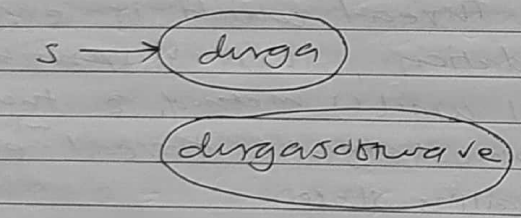
Scanned with CamScanner

String class - (Part-1)

case-1 :-

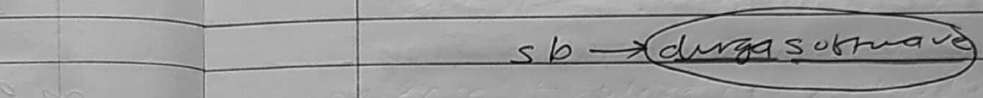
```
String s = new String("durga");
s.concat("software");
s.o.p(s);  // durga
```

Once we creates a string object we can't perform any changes in existing object, if we are trying to perform any change with those changes a new object will be created, this Non-changeable behavior is called Im-mutability of String.




```
StringBuffer sb = new StringBuffer("durga");
sb.append("software");
s.o.p(sb);  // durgasoftware
```

Once we creater a StringBuffer object we can perform any change in the existing object this changeable behavior is Nothing but mutability of StringBuffer



case-2 :-

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

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

```
StringBuffer sb1 = new StringBuffer("durga");
StringBuffer sb2 = new StringBuffer("durga");
s.o.p(sb1==sb2);  // false
s.o.p(sb1.equals(sb2));  // false
```

In StringBuffer class .equals() method is not overridden for content comparison, hence objects (not

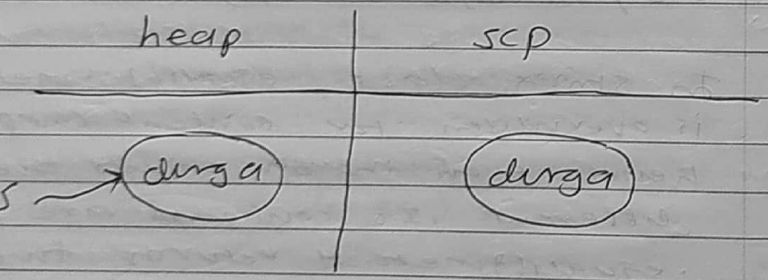
Scanned with CamScanner

.equals() method got executed which is meant for reference comparison (Address comparison), due to this if Object are different .equals() method returns false even though Contents are same.

Case-3 :-

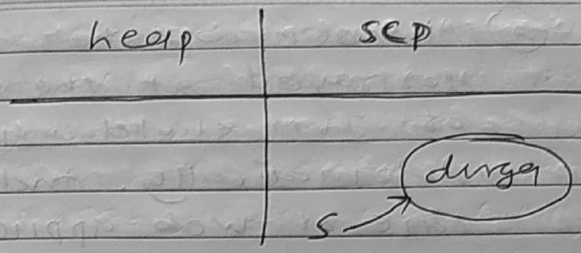
⇒ String s = new String("durga");

In this case, two objects will be created, one in the heap Area & the other in SCP & s is always pointing to heap object



⇒ String s = "durga";

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



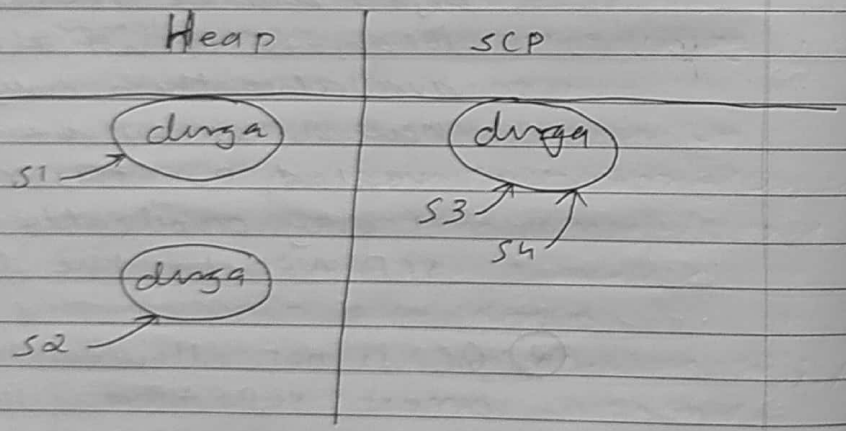
Note :- ① Object creation in SCP is optional, first it will check, is there any object already present in SCP with required content, if object already present then existing object will be reused & if object is not already available then only the new object will be created, But this rule is only applicable for SCP Not for the Heap

② GC is not allowed to access SCP Area, Hence even though object doesn't contain reference variable, it is not eligible for GC, if it is present in SCP Area.

Scanned with CamScanner

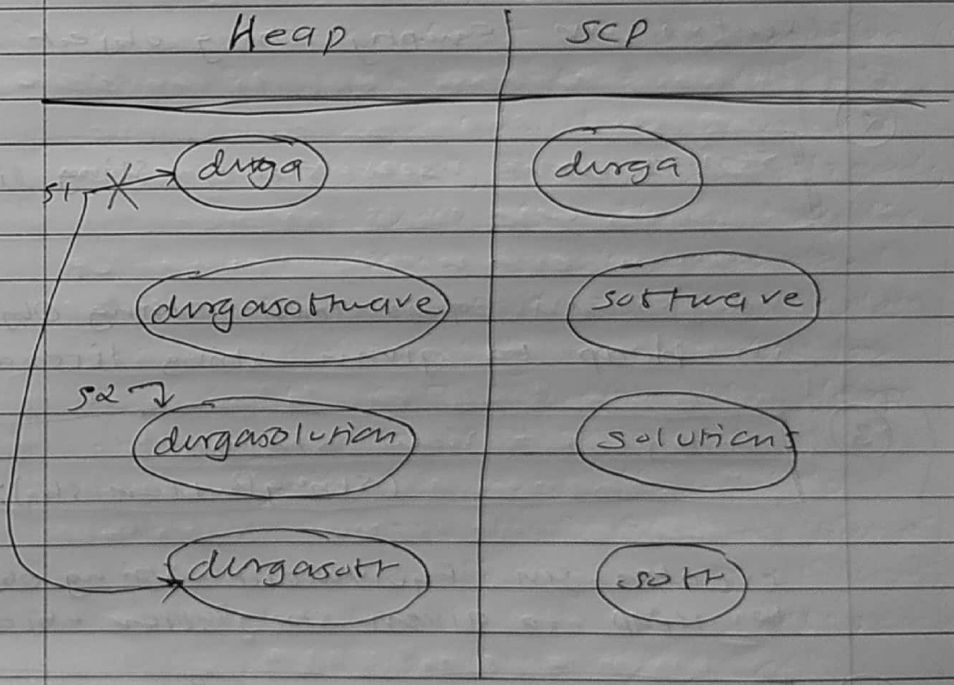
(3) All SCP will be destroyed at the time of JVM shutdown Automatically. And in case of web-application at the time of server shutdown

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



- Whenever we are using new operator compulsory a new object is created in the Heap Area & hence there may be chance of existing two objects with same content in Heap Area, but Not in SCP, i.e duplicate objects are possible in the Heap Area but Not in SCP.

⇒ String s1 = new String ("durga");
s1.concat ("software");
String s2 = s1.concat ("solution");
s1 = s1.concat ("soft");
s.o.p (s1); //durgasoft
s.o.p (s2); //durgasolution



- For every unique string constant one object will be placed in SCP Area

Scanned with CamScanner

- Bcz of some Runtime operation if an object is required to create that object will be placed only in the Heap Area, but Not in SCP Area.

(*) Constructors in String class

① `String s = new String();`

creates an Empty string object

② `String s1 = new String(String literal);`

creates an Equivalent string Object in Heap to given string literal

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

creates an Equivalent string object in Heap to given StringBuffer object

④ `String s = new String(char[] ch);`

creates an equivalent string object for given char[].

ex. `char[] ch = {'a','b','c','d'};`
`String s = new String(ch);`
`s.o.p(s); //abcd`

⑤ `String s = new String(byte[] b);`

Creates an Equivalent string Object for given byte[]

ex. `byte[] b = {100,101,102,103};`
`String s = new String(b);`
`s.o.p(b); //defg`

(*) Methods in String class

① `public char charAt(int index);`

Returns character locating at specified index.

ex. `String s = "durga";`
`s.o.p(s.charAt(3)); //g`
`s.o.p(s.charAt(30));`

R.E. StringIndexOutOfBoundsException

Scanned with CamScanner

Date / /
Page

② public String concat (String s)

overloaded + and += operators are meant for concatenation purpose only.

ex. String s = "durga";
s = s.concat ("software");
// s = s + "soft"; // durgasoft
// s += "soft"; // durgasoft
s.o.p (s); // durgasoftwave

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")); // false
s.o.p (s.equalsIgnoreCase ("java"));
// true

Date / /
Page

⑤ public String substring (int begin);

Returns substring from begin index to end of the string.

⑥ public String substring (int begin, int end);

Returns substring from begin index to end-1 index.

ex. String s = "abcdefg";
s.o.p (s.substring (3)); // defg
s.o.p (s.substring (2, 5)); // cde

⑦ public int length ()

Return Number of characters present in string.

ex. String s = "durga";
s.o.p (s.length); ↓
// CE: cannot find symbol
symbol: variable length
location: j.l.String

s.o.p (s.length()); // 5

Note: length variable applicable for Arrays but not for String objects

Scanned with CamScanner

Date __/__/____
Page ____

(8) `public String replace(char oldch, char newch)`

Replace old character with new character

ex. `String s = "ababa";`
`s.o.p(s.replace('a', 'b'));` //bbbbb

(9) `public String toLowerCase()`

converts string to lowercase

(10) `public String toUpperCase()`

converts string to uppercase

(11) `public String trim()`

To remove blank spaces present at beginning & End of string but not in-between

(12) `public int indexOf(char ch)`

Return index of first occurence of given character

Date __/__/____
Page ____

(13) `public int lastIndexOf(char ch)`

ex. `String s = "ababa";`
`s.o.p(s.indexOf('a'));` // 0
`s.o.p(s.lastIndexOf('a'));` //4

★★ Note :- ① Bcz of Runtime operation if there is a change in content then with those changes a new object will be created on the heap, If there is no change in content then existing object will be reused & new object won't be created unlike new operator which ~~creates~~ is Runtime operation only, though creates new object everytime irrespective of same-content-object exist or Not.

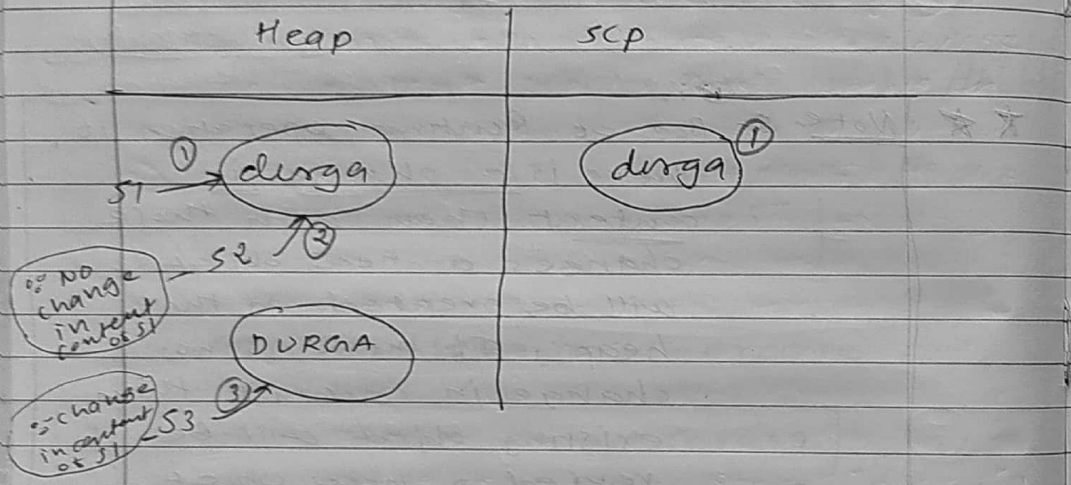
② Whether the object present in Heap (or) scp the rule is same (above rule)

Scanned with CamScanner

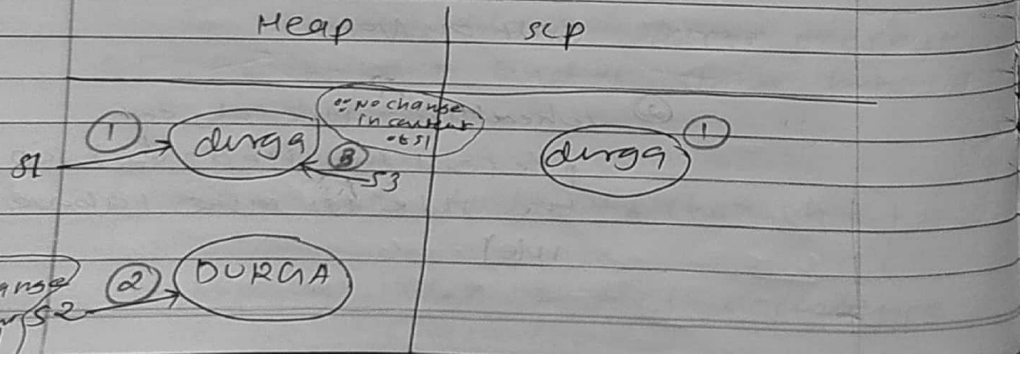
Date ___/___/_____
Page _____

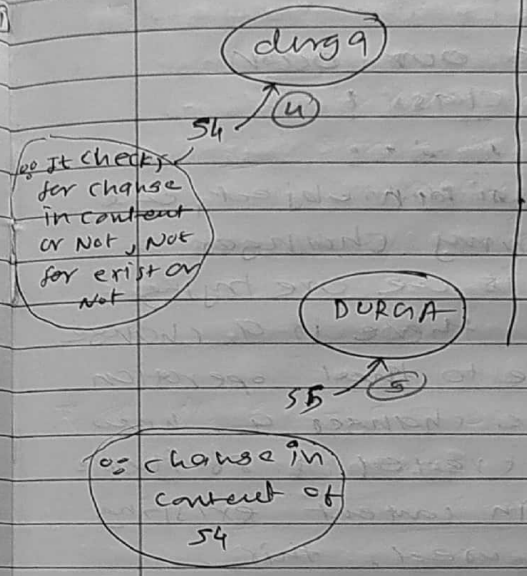
★★

Ex. String s1 = new String("durga"); —①
String s2 = s1.toLowerCase(); —②
String s3 = s1.toUpperCase(); —③
⇒ s.o.p(s1==s2); // true
s.o.p(s1==s3); // false

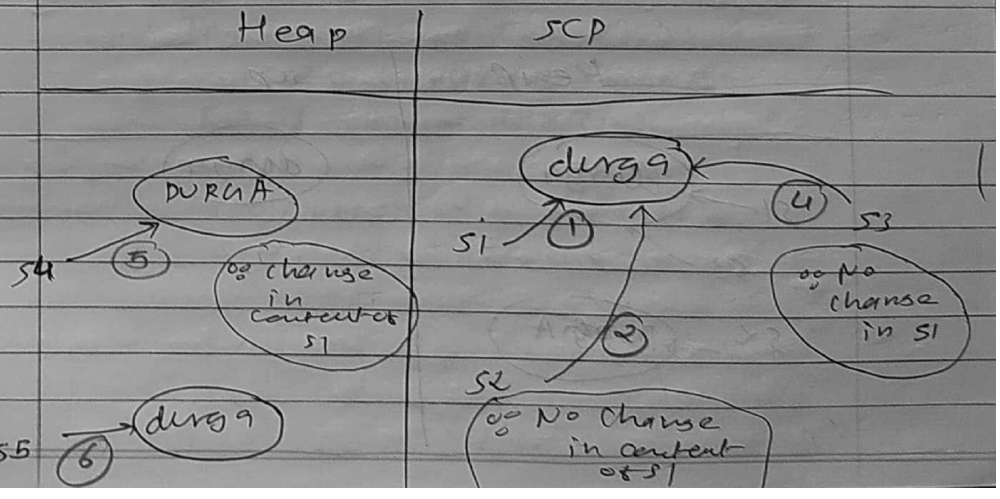


★★ Ex. String s1 = new String("durga"); —①
String s2 = s1.toUpperCase(); —②
String s3 = s1.toLowerCase(); —③
String s4 = s2.toLowerCase(); —④
String s5 = s4.toUpperCase(); —⑤





★★ Ex. String s1 = "durga"; —①
String s2 = s1.toString(); —②
s.o.p(s1==s2); —③
String s3 = s1.toLowerCase(); —④
String s4 = s1.toUpperCase(); —⑤
String s5 = s4.toLowerCase(); —⑥



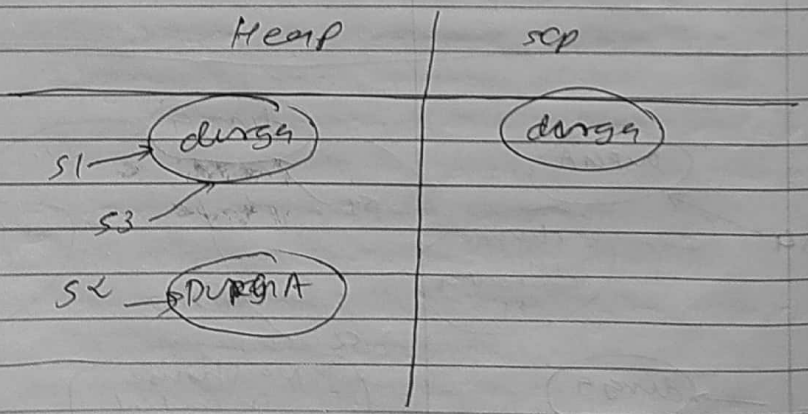
Scanned with CamScanner

(*) How to create our own Immutable class?

⇒ Once we created an object, we can't perform any changes in that object if we are trying to perform operation & if there is a change in content due to that operation then with those changes a new object will be created & if there is no change in content existing content is re-used, this behavior is nothing but "Immutability".

ex.

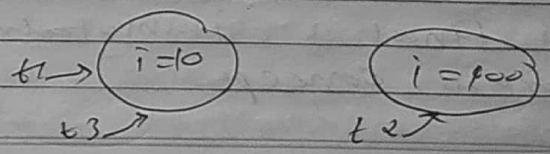
```
String s1 = new String("durga");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
```



ex.

```
public final 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));
        }
    }
}
```

```
Test t1 = new Test (10);
Test t2 = t1.modify (100);
Test t3 = t1.modify (10);

s.o.p (t1 == t2); // false
s.o.p (t1 == t3); // true
```



Scanned with CamScanner

Once we created a Text object we can't perform any change in existing object, if we do so & if there is a change in content then with there change a new object will be created & if there is no change in content then same object will be reused.

final (vs) im-mutability

→ final applicable for variable, but Not for objects whereas Im-mutability applicable for Object, but not for variables.

→ By declaring a reference variable as final we won't get any immutability nature even though reference variable is final we can perform any kind of change in that object, but we can't perform re-assignment for that variable.

→ Hence final & immutable are different concept.

Ex.

```
final SB sb = new SB("durga");
sb.append("soft");
s.o.p(sb); // durgasoft
sb = new SB("sol");
```

CE: cannot assign a value to final variable sb

durgasoft
sb

→ Which of the following is meaningful?

1) final variable ✓
2) Immutable variable ✗
3) final Object ✗
4) Immutable Object ✓

StringBuffer

→ If the content is fixed & won't change frequently then it is recommanded to go for string.

→ But If content is Not fixed & keep on changing then it is not recommanded to go for string bcz for every change a new object will be created, which affects performance of system.

Scanned with CamScanner

Date / /
Page

↝ To handle this requirements, we should go for StringBuffer.

↝ The Main Advantage of StringBuffer over String is all required changes will be performed in the existing objects only.

⇛ Constructors :

① `StringBuffer sb = new StringBuffer();`

Creates an Empty StringBuffer object with Default Initial Capacity = 16

Once a StringBuffer reaches its max capacity, a new StringBuffer object will be created with the

$$\text{New Capacity} = \left(\text{Current Capacity} + 1\right) * 2$$

Ex.
```
SB sb = new SB();
s.o.p(sb.capacity()); //16
sb.append("abcdefghijklmnop");
s.o.p(sb.capacity()); //16
sb.append("q");
s.o.p(sb.capacity()); //34
```

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

Creates an Empty StringBuffer object with specified initial-capacity.

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

Creates an Equivalent StringBuffer for given String with

(Capacity = s.length() + 16)

Ex.
```
StringBuffer sb = new SB("durga");
s.o.p(sb.capacity()); //21
```

⇛ Methods :

① `public int length();`

② `public int capacity();`

③ `public char charAt(int index);`

Scanned with CamScanner

ex.

```
SB sb = new SB("durga");
s.o.p (sb.charAt (3)); // g
s.o.p (sb.charAt (10)); // R.E SIOBE
```

(4) (instead replace())

```
public void setCharAt
            (int index, char ch)
```

To replace character located at specified index with new character

(5) (instead of concat())

```
public StringBuffer append (String s)
                           (int i)
                           (long l)
                           (float f)
                           (double d)
                           (char ch)
                           (boolean b)
                              :
                              :
```

overloaded

ex.

```
SB sb = new SB();
sb.append("PI value is : ");
sb.append(3.14);
sb.append("It is exactly: ");
sb.append(true);
s.o.p(sb);
```

// PI value is : 3.14 It is exactly : true

(6)

```
public StringBuffer insert (int index,
                            String s)
                            (int index, int i)
                            (int index, boolean b)
                               :
```

ex.

```
SB sb = new SB("abcdefgh");
sb.insert(2, "xyz");
s.o.p(sb); // abxyzcdefgh
```

(7.)

```
public StringBuffer delete (int begin,
                            int end)
```

To delete character located from begin index to end-1 index

(8.)

```
public StringBuffer deleteCharAt (int index)
```

To delete the character located at specified index

(9.)

```
public StringBuffer reverse()
```

ex.

```
StringBuffer sb = new SB("durga");
s.o.p(sb.reverse()); // agrud
```

Scanned with CamScanner

(10.) public void setLength(int length)

ex.

```
SB sb = new SB("AishAbhi");
sb.setLength(4);
S.O.P(sb); // Aish
```

(11.) public void ensureCapacity(int capacity)

To increase capacity on fly based on your requirement

ex.

```
SB sb = new SB();
S.O.P(sb.capacity()); //16
sb.ensureCapacity(1000);
S.O.P(sb.capacity()); //1000
```

(12.) public void trimToSize();

To deallocate extra allocated free memory.

ex.

```
SB sb = new SB(1000);
sb.append("abc");
sb.trimToSize();
S.O.P(sb.capacity()); //3
```

(*) StringBuilder

⇒ Every method present in StringBuffer is synchronized, so only one thread is allowed to operate on SB object at a time, which may creates performance problem, To handle this requirement SUN people introduce StringBuilder concept in 1.5V.

⇒ StringBuilder is exactly same as StringBuffer except the following differences:-

StringBuffer	StringBuilder
1) Every method present in StringBuffer is synchronized.	Every method present in StringBuilder is Non-synchronized
2) At a time only one thread allowed to operate on S.Buffer Object ∴ Thread-safe	At a time multiple threads are allowed to operate on S.Builder object ∴ Not Thread safe

t1 → S.Buffer

t1, t2 → S.Builder

Scanned with CamScanner

3)	Thread are required to wait to operate on S.Buffer object ∴ Relatively performance low	Threads are not required to wait to operate on S.Builder object ∴ Relatively performance is high
4)	1.0 V	1.5 V

Except above differences everything is same (like constructors & methods) in S.Buffer & S.Builder

⇒ String (VS) StringBuffer (VS) StringBuilder

(1) If content is fixed & won't change frequently then we should go for String.

(2) If content is Not fixed & keep on changing, but thread-safety required, then we should go for StringBuffer.

(3) If content is Not fixed & keep on changing, but Thread-safety Not required, then we should go for StringBuilder.

✳ Wrapper Classes

⇒ The main objectives of wrapper classes are :-

(1) To wrap primitive into object form so that we can primitives also just like objects.

(2) To define several utility methods which are required for primitives.

⇒ Constructors :-

⇒ Almost all wrapper classes contains two constructors one can take corresponding primitive as argument & other can take string as argument

Ex.1

```
Integer I = new Integer(10);      // int primitive
Integer I = new Integer("10");    // string
```

Ex.2

```
Double D = new Double(10.5);
Double D = new Double("10.5");
```

Scanned with CamScanner

Date / / Page

⇒ If string argument not representing a number then we will get R.E

Ex.

Integer I = new Integer("ten"); ✗

R.E. NumberFormatException

⇒ Float class contains 3 constructors with the float, double, string argument

Ex.

Float f = new Float (10.5f); ✓
Float f = new Float ("10.5f"); ✓
Float f = new Float (10.5); ✓
Float f = new Float ("10-9"); ✓

⇒ Character class contains only 1 constructor which can take char Argument

Ex

Character ch = new Character('a'); ✓
Character ch = new Character("a"); ✗

⇒ Boolean class contains 2 constructors one can take primitive as argument & other can take string as argument If we pass boolean primitive as argument the only allowed values are true / false i.e. case is important

Ex.

Boolean B = new Boolean(true); ✓
Boolean B = new Boolean(false); ✓
Boolean B = new Boolean(True); ✗
Boolean B = new Boolean(durga); ✗

- If we are passing String type as argument, then case & content both are Not important, If the content is case-insensitive string of true then it is treated as true otherwise it is treated as false.

Ex.

Boolean B1 = new Boolean("true"); true
Boolean B2 = new Boolean("True"); true
Boolean B3 = new Boolean("TRUE"); true
Boolean B4 = new Boolean("Abc"); false
Boolean B5 = new Boolean("xyz"); false
Boolean B6 = new Boolean("yes"); false
s.o.p(B1); true s.o.p(B6) false
s.o.p(B1.equals(B2)); true
s.o.p(B1.equals(B4)); false
s.o.p(B5.equals(B2)); false

Scanned with CamScanner

Date / / Page

→ Wrapper class | Corresponding Constructor Argument

	Wrapper class	Corresponding Constructor Argument
	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

→ In Wrapper classes , .toString() method is overridden to return content directly, 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 given primitive or string.

→ Form1 :-

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

```
public static Wrapper valueOf(String str)
```

eg:

```
Integer I = Integer.valueOf("10");
Double D = Double.valueOf("10.5");
Boolean B = Boolean.valueOf("durga");
```

★ → Form2 :-

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

Scanned with CamScanner

public static wrapper valueOf (String str, int radix)

The Allowed range of radix is :
2 to 36

Bcz,
Base-2 :- 0, 1
Base-3 :- 0 to 2
Base-4 :- 0 to 3
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

eg.

```
Integer I = Integer.valueOf("100", 2);
S.O.P(I); // 4

Integer I = Integer.valueOf("10", 4);
S.O.P(I); // 17
```

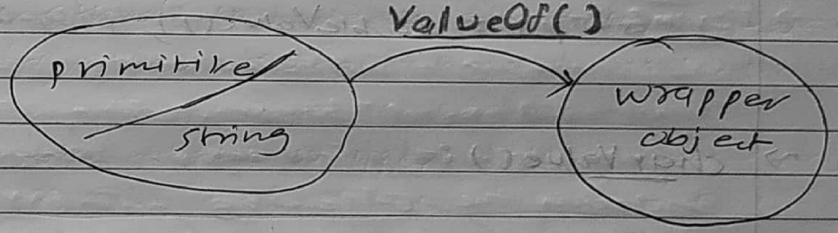
→ Form3 :-

Every wrapper class including Character class contains a static valueOf() method to create a wrapper object for given primitive.

public static wrapper valueOf (primitive p);

eg.

```
Integer I = Integer.valueOf(10);
Character ch = Character.valueOf('a');
Boolean B = Boolean.valueOf(true);
```



(2) xxxValue()

→ we can use xxxValue() methods to get primitive for the given Wrapper Object.

→ Every Number type Wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following 6 methods to get primitive for given Wrapper Object

1) public byte byteValue()
2) public short shortValue()
3) public int intValue()
4) public long longValue()

Scanned with CamScanner

5) public float floatValue();
6) public double doubleValue()

ex:

```
Integer I = new Integer(130);
S.O.p(I.byteValue()); // -126
S.O.p(I.shortValue()); // 130
S.O.p(I.intValue()); // 130
S.O.p(I.longValue()); // 130
S.O.p(I.floatValue()); // 130.0
S.O.p(I.doubleValue()); // 130.0
```

→ charValue() :-

Character class contains charValue() method to get char primitive for given character object

```
public char charValue()
```

ex:

```
Character ch = new Character('a');
char c = ch.charValue();
S.O.p(c); // a
```

→ booleanValue())

Boolean class contains booleanValue() method to get boolean primitive for given Boolean Object.

```
public Boolean booleanValue()
```

ex:

```
Boolean B = Boolean.valueOf("durga");
boolean b = B.booleanValue();
S.O.p(b); // false
```

Note :- In total 38 (= 6*6 + 1 + 1) xxxValue() methods are possible

xxxValue()

(3) parseXXX()

→ we can use parseXXX() method, to convert string to primitive.

→ Form1 :-

Every wrapper class except Character class contains the following parseXXX() methods to find primitive for the given string object.

Scanned with CamScanner

Date / / Page

`public static primitive parseXXX(String s)`

eg.

```
int i = Integer.parseInt("10");
double d = Double.parseDouble("10.5");
boolean b = Boolean.parseBoolean("true");
```

→ Form 2 :-

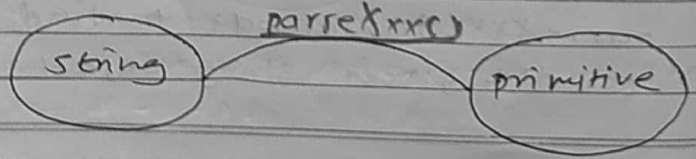
Every Integral type wrapper class (Byte, short, Integer, Long) contains the following parseXXX() method to convert specified radix string to primitive.

`public static primitive parseXXx(String s, int radix)`

The allowed range of radix is (2 to 36)

eg.

```
int i = Integer.parseInt("1111", 2);
S.O.P(i); //15
```



Date / / Page

(4) toString()

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

→ Form I :-

Every wrapper class including Character contains the following toString() method to convert wrapper object to string type.

`public String toString()`

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

Whenever we are trying to print wrapper object reference, internally this toString() method will be called.

Ex.

```
Integer I = new Integer(10);
String s = I.toString();
S.O.P(s); //10

S.O.P(I); ⇒ S.O.P(I.toString()); //10
```

Scanned with CamScanner

Date / /
Page

→ Form2 :

Every wrapper class contains the following static toString() method to convert primitive to String

```
public static String toString
                (primitive p)
```

eg:

```
String s = Integer.toString(10);
String s = Boolean.toString(true);
String s = Character.toString('a');
```

★ → Form3 :

~~Every~~ ~~not~~ Integer & Long classes contains the following toString() method to convert primitive to specified radix string.

```
public static String toString
        (primitive p, int radix)
```

Allowed range for radix (2 to 36)

ex.

```
String s = Integer.toString(15,2);
s.o.p(s); // 1111
```

Date / /
Page

→ Form4 : toXxxString()

Integer & Long classes contains the following toXxxString() methods:

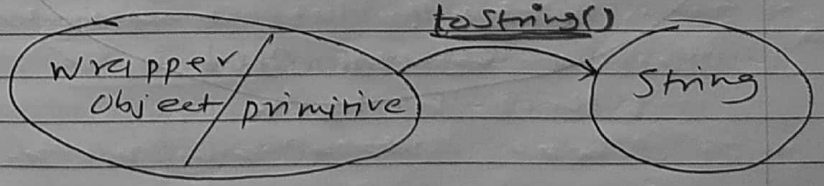
① public static String toBinaryString(primitive p)

② public static String toOctalString(primitive p)

③ public static String toHexString(primitive p)

ex.

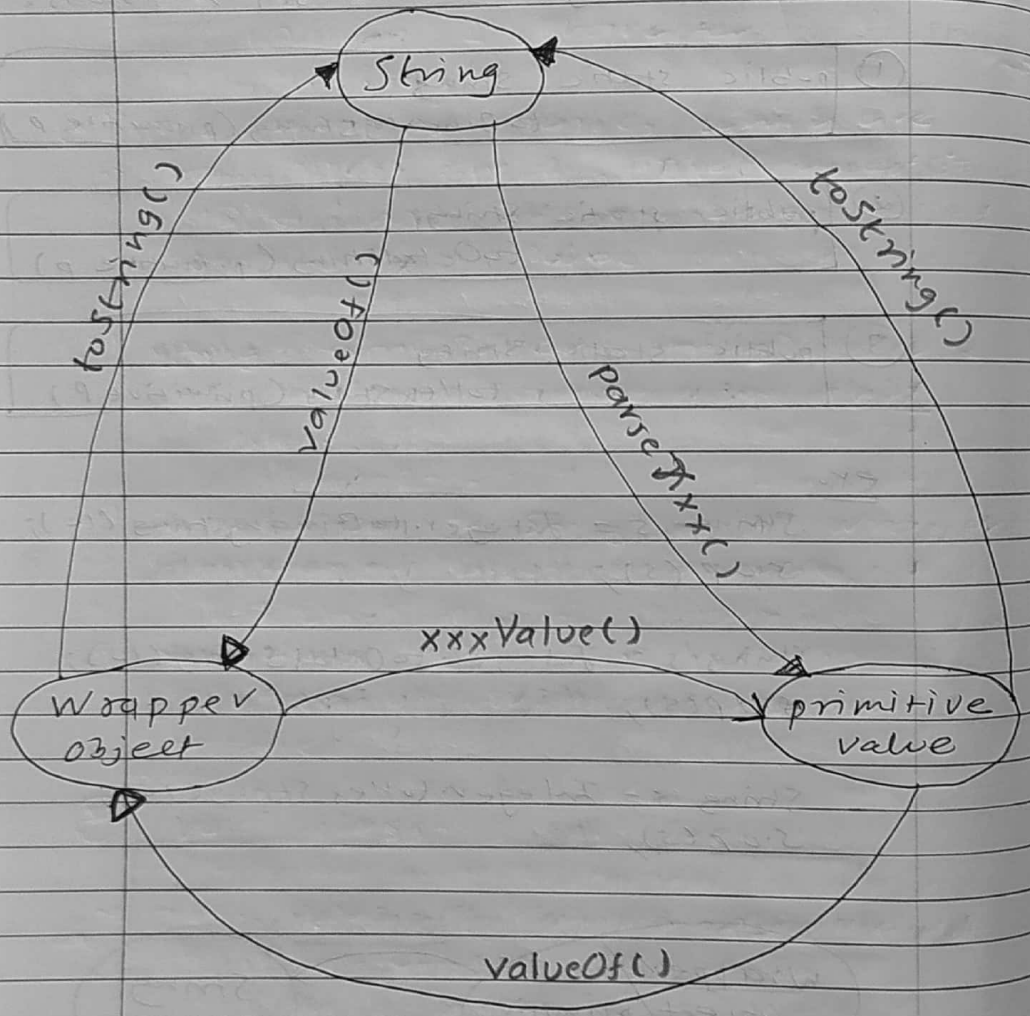
```
String s = Integer.toBinaryString(10);
s.o.p(s); // 1010

String s = Integer.toOctalString(10);
s.o.p(s); // 12

String s = Integer.toHexString(10);
s.o.p(s); // a
```



Scanned with CamScanner

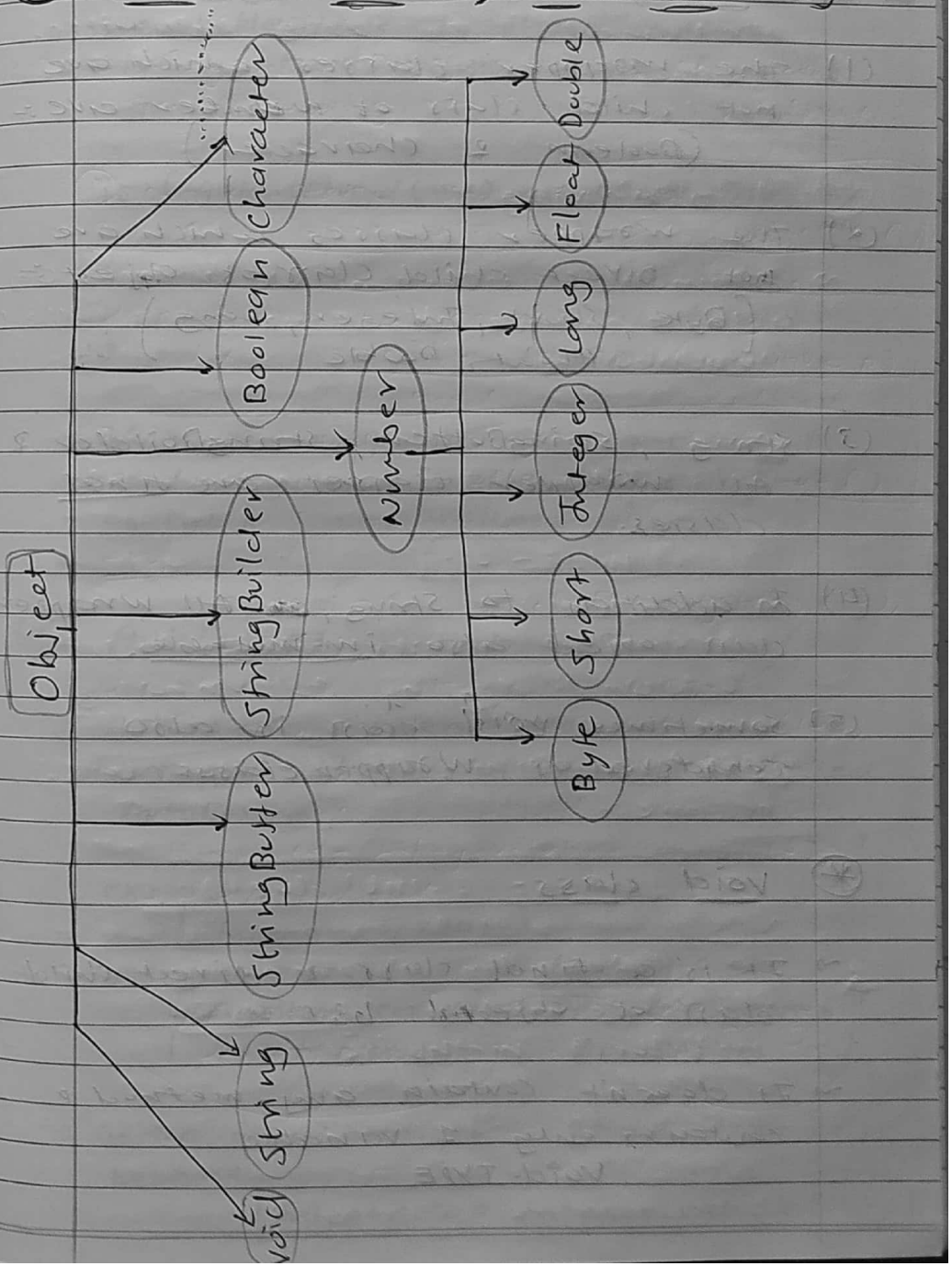
Date / /
Page

⇒ Dancing b/w String, Wrapper Object & primitive



Date / /
Page

⊛ Partial Hierarchy of java.lang



Scanned with CamScanner

Date / /
Page

⇒ Conclusions :-

(1) The wrapper classes which are not child class of Number are :- (Boolean & Character)

(2) The wrapper classes which are not direct child class of Object :- (Byte, Short, Integer, Long, Float, Double)

(3) String, StringBuffer, StringBuilder & All wrapper classes are final classes.

(4) In addition to String, All Wrapper class object also im-mutable.

(5) Sometimes Void class is also considered as Wrapper class.

⊛ Void class

~ It is a final class & direct child class of Object.

~ It doesn't contain any method & contain only 1 variable
Void.TYPE

Date / /
Page

~ In general we can use Void class in reflections to check whether the method return type is void or not.

```
if (getmethod(m1).getReturnType()
        == Void.TYPE)
{
   ;
}
```

~ Void is the class representation of void keyword in java (like Integer is of int).

⊛ AutoBoxing

~ Automatic conversion of primitive to wrapper object by compiler is called AutoBoxing

ex.

```
Integer I = 10;
```

[compiler convers int to Integer automatically by AutoBoxing]

~ After Compilation, the above line will became

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

Scanned with CamScanner

Date / / Page

★ i.e internally AutoBoxing concept is implemented by using valueOf() methods.

(*) AutoUnboxing

→ Automatic conversion of wrapper Object into primitive by compiler is called AutoUnboxing

ex.

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

int i = I;
```

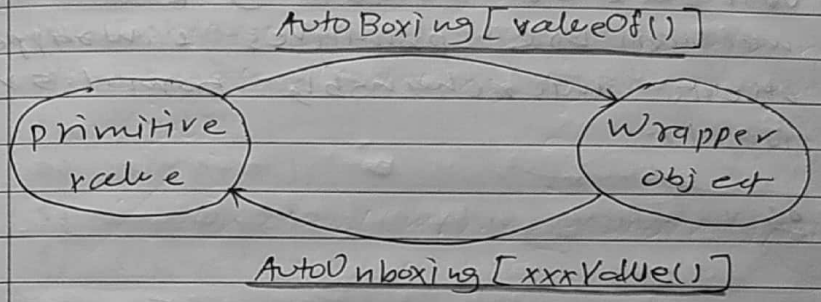
[compiler converts Integer to int Automatically by AutoUnboxing]

After compilation, above line will become:-

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

★ i.e internally AutoUnboxing concept is implemented by using xxxValue() methods.

Date / / Page



Ex.1

```
class Test
{
    static Integer I = 10;  ---> (1) AB

    p s v main(Str)
    {
        int i = I;  ---> (2) AUB
        m1(i);
    }                (3) AB

    p s v m1(Integer K)
    {
        int m = K;  ---> (4) AUB
        S.O.P(m);
    }
}
```

O/P :- 10

★ [It is valid from 1.5V onwards but invalid till 1.4V]

Scanned with CamScanner

★ ⇝ Just bcz of AutoBoxing & AutoUnboxing we can use primitives & Wrapper Objects interchangably from 1.5 version.

Ex.2

```
class Test
{
    static Integer
        I = 0;

    p s v main(str)
    {
        int m = I;
        s.o.p(m);
    }
}
```

O/P : 0

```
class Test
{
    static Integer I;

    p s v m(str)
    {
        int m = I;
        s.o.p(m);
    }
}
```

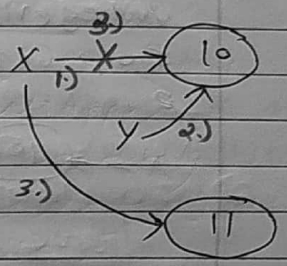
O/P (R.E : NPE)

[∵ int m = I.intValue(); ← null]

Note : On null reference if we are trying to perform AUB then we will get R.E NPE.

Ex.3

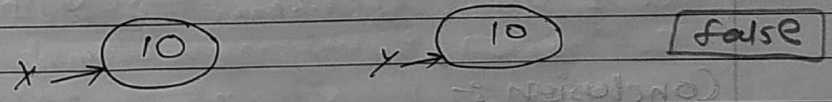
1.) Integer x = 10;
2.) Integer y = x;
3.) x++;
4.) s.o.p(x);
5.) s.o.p(y);
6.) s.o.p(x==y);



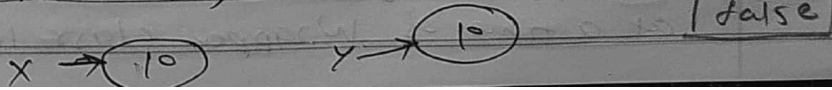
Note : All Wrapper class objects are Im-mutable i.e once we create Wrapper class object, we can't perform any changes in that object, if we are trying to perform any changes, with more changes a new object will be created.

Ex.4

1) Integer x = new Integer(10);
Integer y = new Integer(10);
s.o.p(x==y);

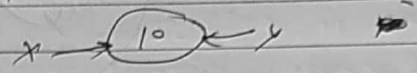


2) Integer x = new Integer(10);
Integer y = 10;
s.o.p(x==y);



Scanned with CamScanner

3.) Integer x = 10;
Integer y = 10;
S.O.P (x==y);

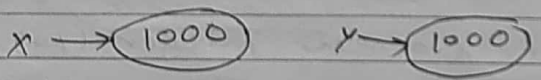


true

4.) Integer x = 100;
Integer y = 100;
S.O.P(x==y);

true

5.) Integer x = 1000;
Integer y = 1000;
S.O.P (x==y);

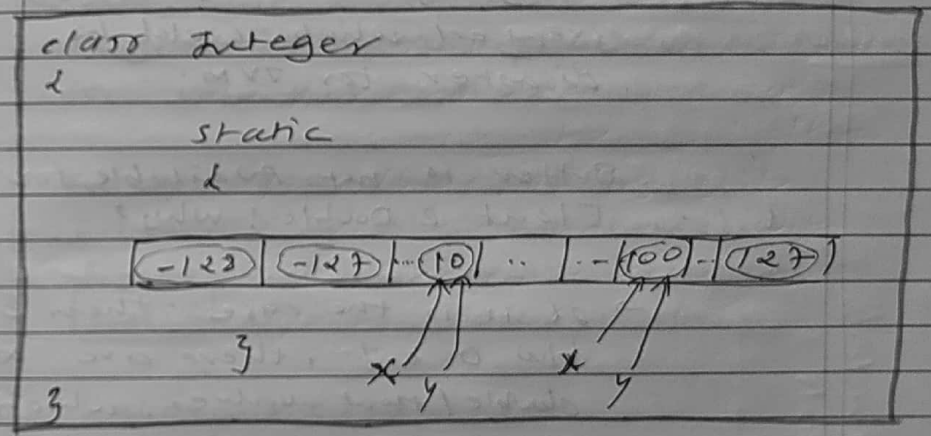


false

Conclusion :-

~ Internally to provide support for AutoBoxing, A buffer of wrapper object will be created at a time of Wrapper class loading

~ By AutoBoxing, if an object is required to create, first JVM will check whether this object already present in buffer or not, if it is already present in buffer then existing buffer object will be used, if it is not already available in buffer then JVM will create a new object.



~ But buffer concept is available only in the following ranges.

Byte →	Always
Short →	-128 to 127
Integer →	-128 to 127
Long →	-128 to 127
Character →	0 to 127
Boolean →	Always

Scanned with CamScanner

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

Note :- Buffer is Not Available for Big ranges! Why?

→ If this is the case, then at class load time only, there many object needs to be created which could be disaster for JVM.

Buffer is not Available for Float & Double! Why?

→ If it is the case, then only b/w 0 & 1, there are ∞ double/float values, which can not be described in buffer.

⇝ Due to above reasons SUN provides Buffer for very commonly used range.

Ex. Integer x = 127;
(1.) Integer y = 127;
S.O.P(x==y);
true

(2.) Integer x = 128;
Integer y = 128;
S.O.P(x==y);
false

(3.) Boolean x = false;
Boolean y = false;
S.O.P(x==y);
true

(4.) Double x = 10.0;
Double y = 10.0;
S.O.P(x==y);
false

★ ⇝ Internally, AutoBoxing concept is implemented by using valueOf() methods.

Hence, Buffer concept is applicable for valueOf() method also.

Ex. Integer x = new Integer(10);
(1.) Integer y = new Integer(10);
S.O.P(x==y);
false

(2.) Integer x = 10;
Integer y = 10;
S.O.P(x==y);
true

(3.) Integer x = Integer.valueOf(10);
Integer y = Integer.valueOf(10);
S.O.P(x==y);
true

(4.) Integer x = 10;
Integer y = Integer.valueOf(10);
S.O.P(x==y);
true

Scanned with CamScanner

Date / / Page

⇒ So, we can say that for commonly used range, valueOf() method is much better than constructor.

⇒ Overloading w.r.t Autoboxing, Widening & var-arg method

case-1 :- Autoboxing (vs) Widening

```
class Test
{
    public static void m(Integer I)
    {
        s.o.p("AutoBoxing");
    }
    public static void m(long l)
    {
        s.o.p("widening");
    }
    p s v m(s[])
    {
        int x=10;
        m(x);
    }
}
```

(overloaded methods)

O/P :- widening

⇒ widening dominates AutoBoxing, bcz Widening came in 1.0 V & AutoBoxing came in 1.5V, so to maintain legacy codes it is happening.

case-2 :- Widening (vs) var-args

```
class Test
{
    public static void m(int... x)
    {
        s.o.p("var-arg method");
    }
    public static void m(long l)
    {
        s.o.p("widening");
    }
    p s v main(s[])
    {
        int x=10;
        m(x);
    }
}
```

(overloaded methods)

O/P :- widening

⇒ (same reason)

Scanned with CamScanner

Case-3 :- Var-args (VS) AutoBoxing

```
class Test
{
    public static void m(int... x)
    {
        S.O.P("var-args method");
    }
    public static void m(Integer I)
    {
        S.O.P("AutoBoxing");
    }
    p s v main(St-)
    {
        int x=10;
        m(x);
    }
}
```

O/P :- AutoBoxing

⇝ AutoBoxing dominates var-args methods In general var-arg method will get least priority i.e if no other method matched then only var-arg method get change. It is exactly same as default case in switch case

⇝ order of priority while resolving overloaded methods

1.) Widening (Highest)
2.) AutoBoxing
3.) Var-arg method (Lowest)

Case-4 Confusion in AB & Widening

```
class Test
{
    p s v m1(Long l)
    {
        S.O.P("Long wrapper class");
    }
    p s v main(St-)
    {
        int x=10;
        m1(x);
    }
}
```

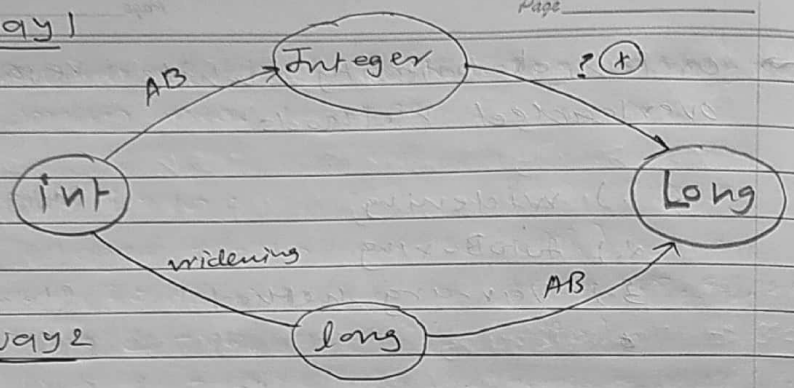
C.E. m(j.l.Long) in Test cannot be applied to(int)

For method resolving in above case something like this should happen.

Scanned with CamScanner

Date / / Page

way1



way2

~ In way1, AB can be done but there is no relation blw Integer & Long so casting not possible.

~ In way2, everything goes fine but widening following by AB not possible in java, though AB followed by widening can be possible.

W → AB ⟹ ✗

AB → W ⟹ ✓

~ So, In this case, no solution possible for method resolving & we will get C.E.

~ So, Long l = 10; ✗ C.E: Incompatible types F: int R: v. l. Long

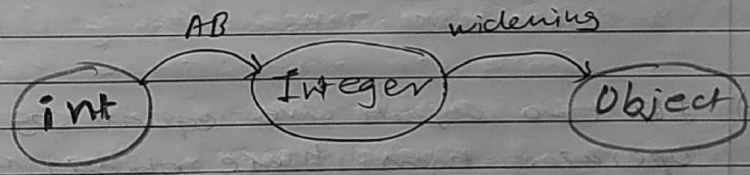
long l = 10; ✓ (widening)

Date / / Page

~

```
class Test
{
    p s v m(Object O)
    {
        S.O.P("Object version");
    }
    p s v main(Sm)
    {
        int x = 10;
        m(x);
    }
}
```

O/P : Object version.



AB → W ⟹ ✓

~ Object O = 10; ✓

Number N = 10; ✓

Scanned with CamScanner

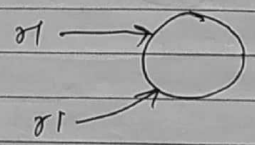
Date / / Page

⇒ Which of the following assignment options are legal?

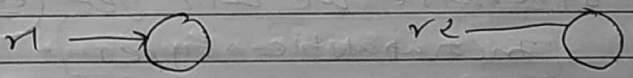
1) int i = 10;	✓	(Normal assignment)
2) Integer I = 10;	✓	(AB)
3) int i = 10L;	✗	(C.E: PLP Long R.Int)
4) Long L = 10L;	✓	(AB)
5) Long L = 10;	✗	(C.E: Incompatible types)
6) long l = 10;	✓	(W)
7) Object O = 10;	✓	(AB → W)
8) double d = 10;	✓	(W)
9) Double D = 10;	✗	(C.E: Incompatible type)
10) Number N = 10;	✓	(AB → W)

(*) Relation b/w (== operator) and [.equals()] method :-

(1) If two objects are equal by == operator then these objects are always equal by .equals() method.
i.e
If (r1 == r2) is true then
[r1.equals(r2)] is Always true.

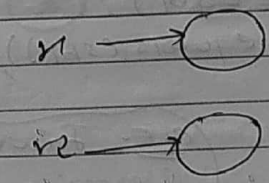


(2) If two objects are Not equal by == operator then we can't conclude anything about .equals() method, It may return true/false.
i.e
If (r1 == r2) is false then
[r1.equals(r2)] may return true/false we can't expect anything



(3) If two objects are equal by .equals() method then we can't conclude anything about == operator It may return true/false
i.e
If [r1.equals(r2)] is true then
(r1 == r2) may return true/false

(4) If two objects are Not equal by .equals() method then these objects are always not equal by == operator.
i.e
If [r1.equals(r2)] is false then
(r1 == r2) is Always false



Scanned with CamScanner

Date / / Page

(*) Differences blw (== operator) & [.equals()] method

⇒ To use == operator compulsory there should be some relation blw Argument type either (child to parent / parent to child / same) otherwise we will get (C.E) Incompatible types

⇒ If there is no relation blw Argument types, then .equals() method won't rise any C.E / R.E simply it returns false.

Ex.

```
String s1 = new String("durga");
String s2 = new String("durga");
StringBuffer sb1 = new SB("durga");
StringBuffer sb2 = new SB("durga");

S.O.P(s1==s2); // false
S.O.P(s1.equals(s2)); // true
S.O.P(sb1==sb2); // false
S.O.P(sb1.equals(sb2)); // false
S.O.P(s1==sb1); (X)
S.O.P(s1.equals(sb1)); // false
```

C.E. Incompatible types
J.L.String and J.L.StringBuffer

== Operator	.equals() method
1.) It is an Operator in java applicable for both primitive & object types	It is a method applicable only for object types, not for primitives
2.) In the case of object references == meant for Reference/Address comparison	By default .equals() present in Object class meant for Reference comparison
3.) We can't overrride == Operator for content comparison	We can overrride .equals() method for content comp.
4.) To use == Operator compulsory there should be some relation blw Argument type. otherwise we will get C.E. Incomparable types.	If there is no relation blw argument types then .equals() method won't rise any C.E/R.E & simply returns false.

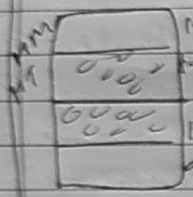
⇒ For Any Object reference r,

r==null
r.equals(null) ⇒ Always returns (false)

Scanned with CamScanner

Thread t = new Thread();
s.o.pl(t==null); // false
s.o.pl(t.equals(null)); // false.

★ **Note:** Hashing related Data structure follow the following fundamental Rule:-



- Two equivalent objects should be placed in same bucket but All objects present in same bucket need not be equal.

★ ⊛ Contract b/w [.equals()] and [hashCode()]

→ If two objects are equal by .equals() method then their Hashcode must be equal i.e two equivalent objects should have same Hashcode i.e
If [r1.equals(r2)] is true then (r1.hashCode()==r2.hashCode()) it Always true.

→ Object class .equals() method & hashCode() method follows above concept, Hence whenever we are overriding .equals() method compulsory we should override hashCode() method to satisfy the above contract.

(i.e two equivalent object should have same Hashcode)

→ If two objects are not equal by .equals() method then there is no restriction on hashcodes may be equal or may not be equal.

→ If hashcodes of two objects are equal then we can't conclude anything about .equals() method it may return true/false.

→ If hashcodes of two objects are Not equal then these objects are Not equal by .equals() method.

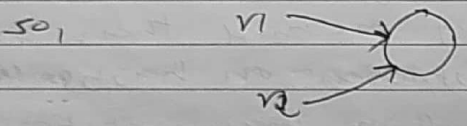
Note: To satisfy contract b/w equals() & hashCode() methods, whenever we are overriding .equals() method compulsory we should override

.hashCode() method.

Though we won't get any C.E/R.E but it is not a good programming practice.

Q How Object class following this contract?

Ans:→ Object class .equals() method meant for reference comparison.



so, r1.equals(r2) returns true only when referencing to same objects so hashcodes always be same.

↝ In String class .equals() method is overridden for content comparison & Hence hashCode() method is also overridden to generate Hashcode based on content.

String s1 = new String("durga");
String s2 = new String("durga");
S.O.P(s1.equals(s2)); // true
S.O.P(s1.hashCode()); // 95950491
S.O.P(s2.hashCode()); // 95950491

↝ In StringBuffer .equals() method is not overridden for content comparison & hence hashCode() Method is also not overridden.

SB sb1 = new SB("durga");
SB sb2 = new SB("durga");
S.O.P(sb1.equals(sb2)); // false
S.O.P(sb1.hashCode()); // 19621457
S.O.P(sb2.hashCode()); // 4872882

↝ Consider the following Person class.

```
class Person
{
    :
    public boolean equals(Object obj)
    {
        if(obj instanceof Person)
        {
            Person p = (Person)obj;

            if(name.equals(p.name) &&
                age==p.age)
            {
```

Scanned with CamScanner

```
        return true;
    }
    else
    {
        return false;
    }
  }
  return false;
 }
}
```

Which of the following hashCode() methods are appropriate for Person class ?

	name	age	ssno
P1 :	shiva	30	60
P2 :	shiva	30	40

(a.)
```
public int hashCode()
{
    return 100;
}
```

(b.)
```
public int hashCode()
{
    return age + ssno;
}
```

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

(d.) No Restrictions

Note :- Based on which parameters we override .equals() method, It is highly recommended to use same parameters while overriding hashCode() methods also.

→ In all Collection class, In all Wrapper classes & In String class .equals() method is overridden for content comp. Hence it is highly recommended to override .equals() method In our class also for content comparison.

✱ clone()

↝ The process of creating exactly duplicate object is called cloning.

↝ The main purpose of cloning is to maintain backup copy & to preserve a state of an object.

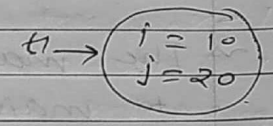
↝ we can perform cloning by using clone() method of Object class.

Scanned with CamScanner

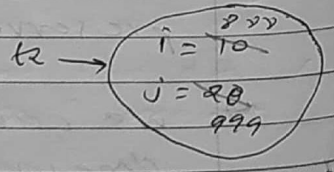
protected native Object clone()
throws CloneNotSupportedException

Ex.

```
class Test implements Cloneable          (3)
{
    int i = 10;
    int j = 20;

    p s v main (Str )
            throws CloneNotSupportedEx    (2)
    {
        Test t1 = new Test();
        Test t2 = (Test) t1.clone();      (1)

        t2.i = 888;
        t2.j = 999;

        S.O.p(t1.i + "---" + t1.j);

    }
}
```



O/P : 10 --- 20



⇝ If we don't write Part (1) we will get C.E. Incompatible types
Found: j.l.Object Required: j.l.Test

⇝ If we don't write part (2) means if we don't throws CNSE which is thrown by clone() method then we will get
C.E. Unreported Exception
CNSE: Test

⇝ If we don't write part (3) then code compiles fine but we will get
R.E. CNSE

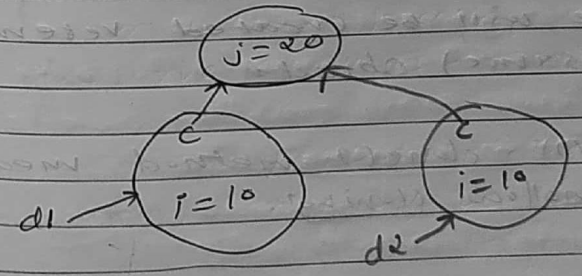
⇝ So, we can perform cloning only on Cloneable classes.

⇝ An object is said to be Cloneable if & only if it implements Cloneable interface.

⇝ Cloneable interface present in java.lang package & as it doesn't contain any methods, so it is "Marker Interface".

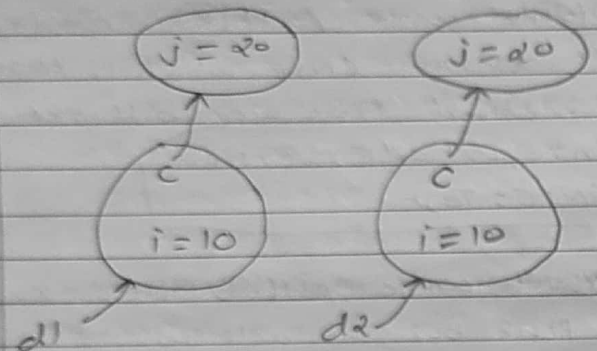
⇝ Shallow Cloning (VS) Deep Cloning

⇝



Dog d2 = (Dog) d1.clone();

Scanned with CamScanner



Deep cloning

Dog d2 = (Dog) d1.clone();

⇒ The process of creating bitwise copy of an object is called "shallow cloning".

- If the main object contains primitive variable then exactly duplicate copies will be created in the cloned object.

- If the main object contains Any Reference variable then corresponding object won't be created for cloned object, just duplicate reference variable will be created referring to old (same) object.

★ - object class clone() method meant for "shallow cloning".

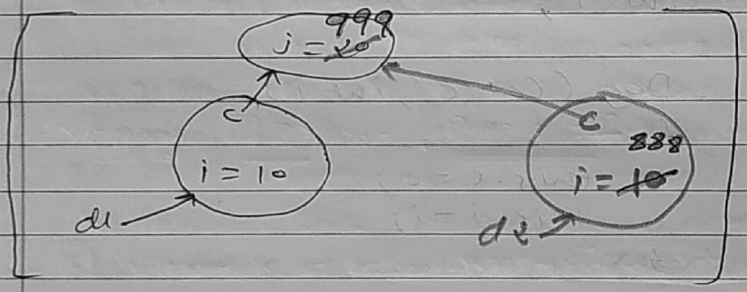
Ex:

```
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 CNSE
    {
        return super.clone();
    }
}
class ShallowCloningDemo
{
    p s v main (Str) throws CNSE
    {
        Cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
```

Scanned with CamScanner

```
s.o.p(d1.i + "...." + d1.c.j);

Dog d2 = (Dog) d1.clone();
d2.i = 888;
d2.c.j = 999;

s.o.p(d1.i + "...." + d2.c.j);
}
}
```

O/P :- 10....20
10....999



→ In shallow cloning, by using cloned object reference, if we perform any change to contained object then those changes will be reflected to the original object.

⇒ To overcome this problem, we should go for "Deep cloning".

⇒ The process of creating exactly duplicate inelependate copy including contained object is called "Deep cloning".

⇒ In Deep cloning If the main object contain any primitive variables then exactly duplicate copies will be created in the cloned object.

- If the main object contains any reference variable then corresponding contained object also will be created in the cloned copy.

- By default, Object class clone() method meant for shallow cloning But we can implement Deep cloning by overriding clone() method in our class.

- All code remains same, only we have to change the version of clone() method means we have to just override clone() method as follows:-

```
public Object clone() throws CNSE
{
    Cat c1 = new Cat(c.j);
    Dog d = new Dog(c1, i);
    return d;
```

Scanned with CamScanner

Date / / Page

O/P :- 10 20
[0 . . . . 20]

Q. ⇒ Which cloning is best?

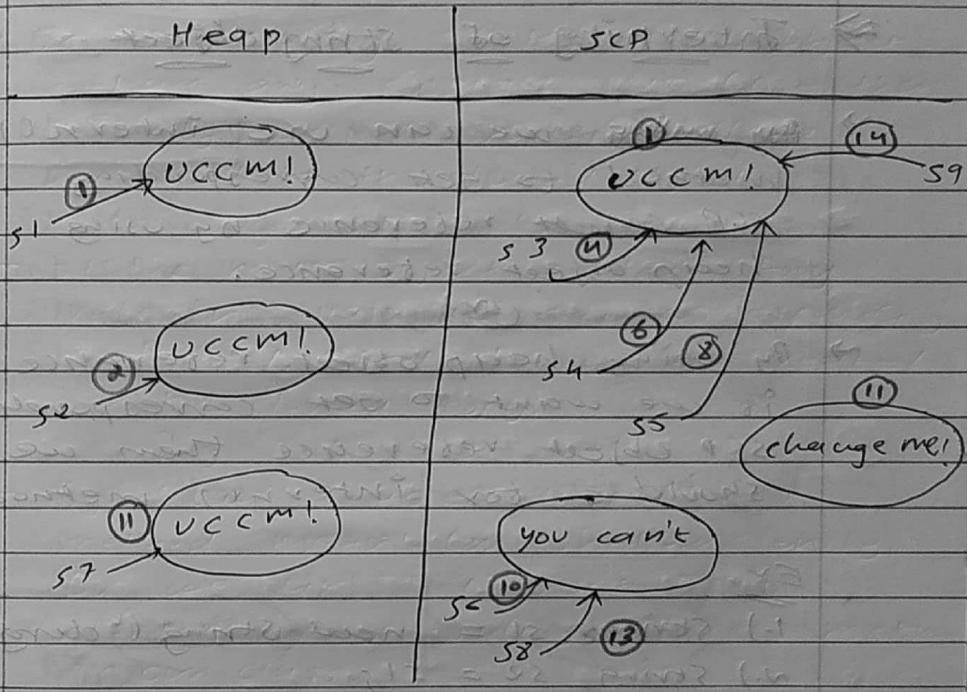
Ans: It Object contains only primitive variable then shallow cloning is the best choice.

It Object contain Reference variables then Deep cloning is the best choice.

✱ String Class - (Part-2)

Ex.

1.) String s1 = new String ("you can't change me!");
2.) String s2 = new String ("you can't change me!");
3.) s.o.p (s1 == s2); // false
4.) String s3 = "you can't change me!";
5.) s.o.p (s1 == s3); // false
6.) String s4 = "you can't change me!";
7.) s.o.p (s3 == s4); // true
8.) String s5 = "you can't " + "change me!";
9.) s.o.p (s3 == s5); // true
10.) String s6 = "you can't";
11.) String s7 = s6 + "change me!";
12.) s.o.p (s3 == s7); // false
13.) final String s8 = "you can't";
14.) String s9 = s8 + "change me!";
15.) s.o.p (s3 == s9); // true
16.) s.o.p (s6 == s8); // true



⇝ Line-(8) This Operation will be performed at compile-time, only bcz both arguments are compile time constant.

⇝ Line-(11) This Operation will be performed at Runtime only bcz atleast one-argument is Normal variable which can vary at compile-time.

Scanned with CamScanner

→ Line-(14) This operation will be performed at compile-time only bcz both arguments are C.T. constants.

Interning of String Object

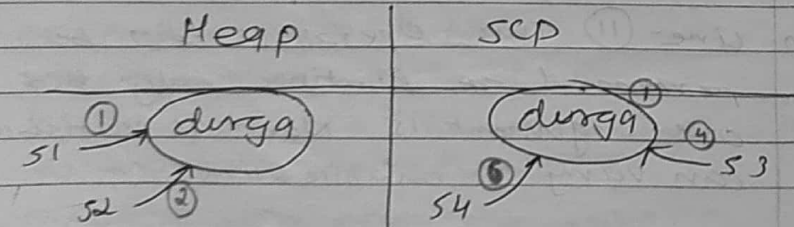
→ By ~~using~~ we can use intern() method to get corresponding SCP object reference by using heap object reference.

(or)

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

Ex.

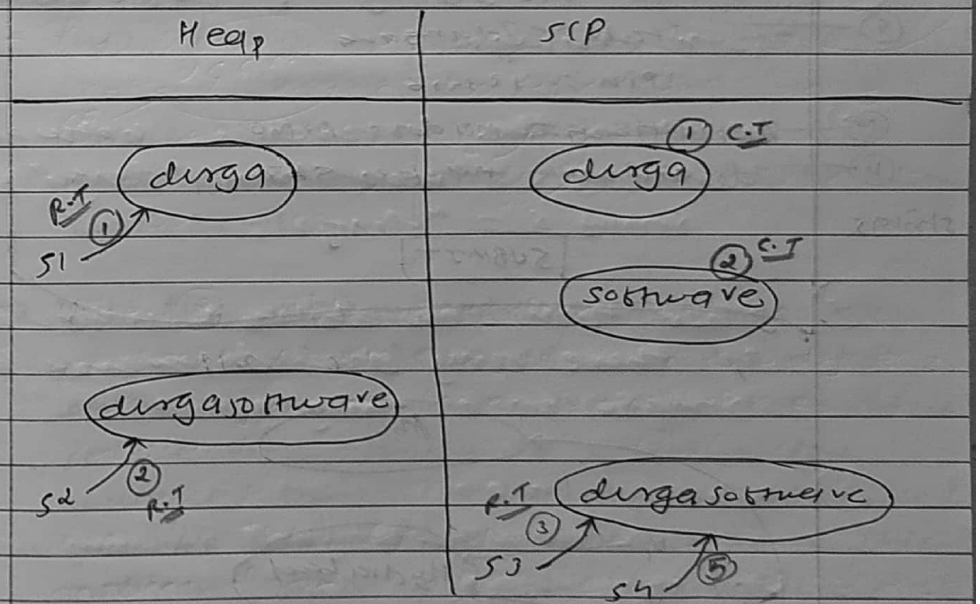
```
1.) String s1 = new String("durga");
2.) String s2 = s1;
3.) S.O.p(s1==s2); // true
4.) String s3 = s1.intern();
5.) S.O.p(s1==s3); // false
6.) String s4 = "durga";
7.) S.O.p(s3==s4); // true
```



→ If the corresponding SCP object is not available then intern() method itself will create the corresponding SCP object.

Ex.

```
1.) String s1 = new String("durga");
2.) String s2 = s1.concat("software");
3.) String s3 = s2.intern();
4.) S.O.p(s2==s3); //
5.) String s4 = "durgasoftware";
6.) S.O.p(s3==s4); //
```



Scanned with CamScanner

⇒ Importance of String Constant Pool

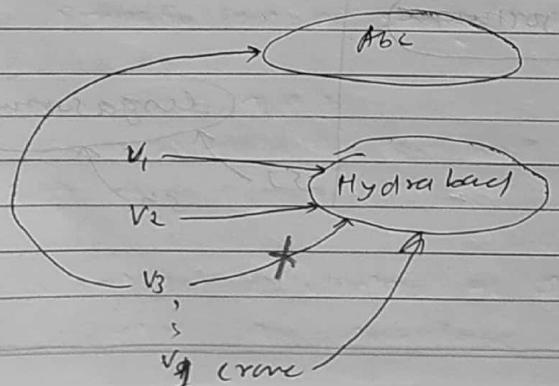
Voter Registration Form

1. Name :- Abc
2. Father's Name :- pqr
3. DOB :- 02-09-'80
   Age :- 39
   Address :-
4. H.NO :- 1-203
5. Street :- Street-1e
6. Village/city :- XYZ
7. Mandal :- Hydrabad
8. District :- ~~~
9. State :- Telangana
   Pin :- 123456
10. Identification Mark 1 :- MNO
11. Identification Mark 2 :- SRT

Photo

SUBMIT

strings



⇒ In our program, If a string object is repeatdly required then its not recommanded to create a separate object for every requirement bcz it creates performance & memory problems.

⇒ Instead of creating a separate object for every requirement we have to create only one object & we can reuse the same object for every requirement, so that performance & memory utilization is improved.

- This thing is possible bcz of SCP.

- Hence, The main Advantages of SCP are Memory utilization & Performance will be improved.

⇒ But the main problem with SCP is, as several references pointing to same object, by using one reference if we are trying to change the content then remaining reference will be affected, To overcome this problem, SUN people implemented String Objects as Immutable i.e once we created a String object, we can't perform any changes in

existing object, If we trying to do so with those changes a new object will be created.

- Hence, SCP is the only reason why immutability introduction for string objects.

⇒ Important Questions

(1) What is the difference b/w String and StringBuffer?

(2) Explain Mutability & Immutability with example.

(3) What is the difference between.
```
String s = new String("durga");
String s = "durga";
```

(4) Other than mutability & immutability is there any other difference b/w StringBuffer & String? (Hint: .equals() method)

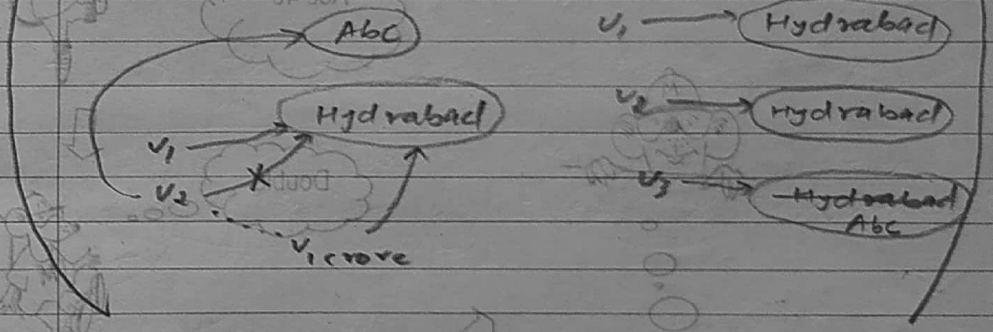
(5) What is SCP?
(It is a specially designed memory area for string object)

(6) What is Advantage of SCP?

(7) What is Disadvantage of SCP?

(8) Why SCP like concept is available only for String but not for StringBuffer?
(String is the most commonly used object & hence SUN people provided special memory mgmt for String object, But StringBuffer is not commonly used object)

(9) Why String objects are immutable where as StringBuffer objects are mutable?
In case of String, Due to SCP, several reference can refer to same object, so immutability required, but In case of StringBuffer Everytime new object get created



(10) In addition to String Objects any other objects are immutable in java?
(All Wrapper classes & we can create our own also)

(11) How to create our own immutable class? Explain with example? (on previous pages)

(12) Difference b/w Immutable and final?
(on previous pages)

Scanned with CamScanner