Java.lang package

→ The most Commonly required classes & Interfaces which are required for writing any java program whether it is simple or Complex, are encapsulated into a Seperate package which is nothing but lang package

→ It is not required to import lang package explicitly because bydefault it is available to every java program.

→ The following are some of the commonly used classes in lang package

1. Object
2. String
3. StringBuilder
4. StringBuffer
5. Wrapper classes (Auto boxing & Auto unboxing)

① Object :-

→ The most Common methods which are required for any java object are encapsulated into a Seperate class which is nothing but Object class.

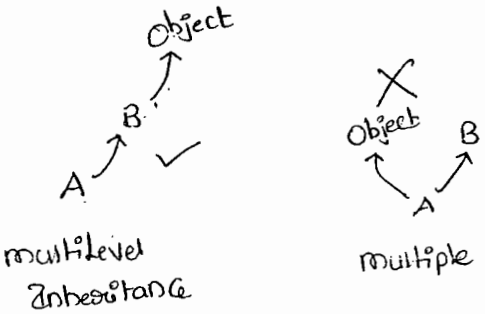
→ SUN people made this class as parent for all Java classes so that its methods are bydefault available to every Java class Automatically

→ Every class in java is the child class of object either directly or indirectly, if our class doesn't extend any other class then only our class is direct child class of Object.

Ex!- Class A

→ if our class extends any other class then our class is not direct child class of Object. it extends object class indirectly.

Ex!- Class A extends B



→ 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) public final Class getClass();

(6) protected void finalize() throws Throwable

(7) public final void wait() throws InterruptedException

(8) public final native void wait(long ms) throws IE

(9) public final native void wait(long ms, int ns) throws IE

(10) public final native void notify();

(11) public final native void notifyAll();

① toString() method :-

→ We can use this method to find String representation of an Object

→ whenever we are trying to print any object reference internally toString() 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(String[] args)
    {
        Student s1 = new Student("durga", 101);
        Student s2 = new Student("Sathu", 102);
        S.o.pln(s1); ⇒ S.o.pln(s1.toString()); Student@3e25a5
        S.o.pln(s2);                          Student@19821f.
    }
}
```

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

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

Student @ 3e25a5

→ Top.

→→ classname@hexadecimal String representation of hash code.

→ To provide our own String representation we have to override toString() in our class which is highly recommended.

→ when ever we are trying to print Student Object reference to return his name & roll number we have to override toString() as follows

```
public String toString()
{
// return name;
//return name + "-----" + rollno;
// return "this is Student with name:" + name + ", with rollno:"
                                        + rollno;
}
```

* In String, StringBuffer & in all wrapper classes toString() method is Overridden to return proper String form. Hence, it is highly recommended to override toString() method in our class also.

Ex:-

```
Class Test
{
 p.s.v.m
 Public String toString()
 {
  return "test";
 }
 public . s. v. m( ——— )
 {
  Test t = new Test();
  String s = new String("durga");
  Integer i = new Integer(10);
  S.o.pln(t);      test
  S.o.pln(s);      durga
  S.o.pln(i);      10
 }
}
```

test@a23504

(ii) hashCode() :-

→ For every object Jvm ~~will always~~ will assign one unique id. which is nothing but hashCode.

→ Jvm uses hashCode, will saving objects into hashtable or hashSet or hashmap

→ Based on our requirement we can generate hashCode by over-riding hashCode() method in our class.

→ If we are not overriding hashCode() method then Object class

hashCode() method will be executed which generates hashCode based on Address of the object But whenever we are overriding hashCode() method then hashCode is no longer related to Address of the object.

→ Overriding hashCode() method is said to be proper iff for every object we have to generate a unique number.

Ex:- ① Case ①:-

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

Case ②:-

```
Class Student
{
  ==
  ==
  ==
  public int hashCode()
  {
    return *rollno;
  }
  :
}
```

Case ①:- It is improper way of overriding hashCode() because we are generating same hashCode for every object

Case ②:- It is proper way of overriding hashCode() because we are generating a different hashCode for every object.

toString() Vs hashCode() :-

Ex:

```
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.pln(t1);   Test@1a3b2b
     S.o.pln(t2);   Test@2a4b2a
   }
}
```

Object → toString()
⇓
Object → hashCode()

0-15

0
1
2
1
1
9
a(10)
b(11)
c(12)
d(13)
e(14)
f(15)

16 | 100
6 - 4
⟶
64

10

Ex (2):

```
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.pln(t1);   Test@a
    S.o.pln(t2);   Test@64
  }
}
```

Object → toString()
⇓
Test → hashCode()

16 | 100
~~64~~ 4↑
64

10 ⇒ a
in hashCode

ex3:-

```
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.ph(t1);      10
   S.o.ph(t2);      100

  }
}

  Test → toString()
```

Note:-

→ if we are giving opportunity to Object class toString() method than it will call internally hashCode() method.

→ if we are giving opportunity to our class toString() method than it may not call hashCode() method.

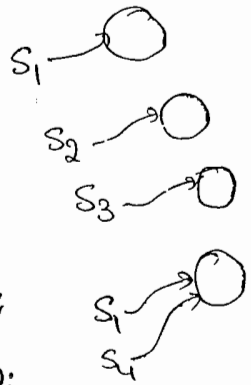
③ equals() method :-

→ We can use equals() method to check equality of two objects

```
public boolean equals(Object o)
```

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 ("pavan", 102);
        Student S3 = new Student ("durga", 101);
        Student S4 = S1;
        S.O.pln (S1.equals(S2));   false
        S.O.pln (S1.equals(S3));   false
        S.O.pln (S1.equals(S4));   true
    }
```



↳ In the above Case Object class .equals() method will be executed which is always ment for Reference Comparision (address Comparision).

↳ i.e, if two References pointing to the Same object Then only .equals() method returns true. This behaviour is Exactly Same as == operator

↳ If we coant to perform Content Comparision instead of Reference Comparision we have to Override .equals() method in our class.

→ When ever we are overriding .equals() method we have to Consider the following things,

(1) What is the meaning of equality

(2) In the Case of diff. Type of Objects (Hetrogeneous) equals method should return false but not ClassCastException.

(3) If we are passing Null arrguement our .equals method should returns false but not a NullPointerException.

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

Ex:

```
public bookean equals(Object o)
{
  try
  {
    String name1 = this.name;
    int rollno1 = this.rollno;
    Student s2 = (Student) o;
    Student name2 = s2.name;
    int rollno2 = s2.rollno;
```

```
if(name1.equals(name2) && rollno1==rollno2)
 {
   return true;
 }
 else
 {
   return false;
 }
}
Catch(CCE e)
 {
   return false;
 }
 catch(NPE e)
   {
     return false;
   }

Student S1 = new Student("durga",101);
Student S2 = new Student("pavan",102);
Student S3 = new Student("durga",101);
Student S4 = S1;
```

S.o.pln(S1.equals(S2)); false

S.o.pln(S1.equals(S3)); True

S.o.pln(S1.equals(S4)); True

S.o.pln(S1.equals("durga")); false

S.o.pln(S1.equals(null)); false

Short way of writting equals() method :-

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

Relationship b/w == operator & .equals() method :-

→ if $r_1 == r_2$ is True, then r_1.equals(r_2) is always True.

→ if $r_1 == r_2$ is false, then we can't expect about r_1.equals(r_2) Exactly It may returns True or false.

→ if r_1.equals(r_2) returns True, we can't conclude anything about $r_1 == r_2$. it may returns either True or false.

→ if r_1.equals(r_2) is false, then $r_1 == r_2$ is always false.

differences b/w == operator & .equals() method :-

== operator	.equals()
① It is an operator applicable for both primitives & object references	① It is a method applicable only for object references but not for primitives.
② In the case of object references == operator is always meant for reference comparision. i.e, if two references pointing to the same object then only == operator returns T	② By default .equals() method present in object class is also meant for reference comparision only.
③ we can't override == operator for content comparision	③ we can override .equals() method for content comparision.
④ In the case of Heterogeneous type objects ~~equal~~ == operator gives causes compiletime Error saying incomparable types	④ In the case of Heterogeneous objects .equals() method simply return false & we won't get any compiletime or runtime Error
⑤ for any object reference r, r == null is always false.	⑤ for any object reference r, r.equals(null) is always false.

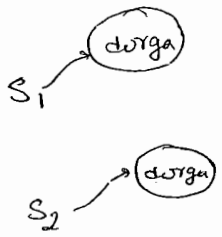
Note:-

(8) What is the difference b/w Double equal operator (==) & .equals()

→ "==" operator is always meant for reference comparision, where as .equals() method meant for content comparision.

Ex:-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln(s1 == s2); false
S.o.pln(s1.equals(s2)); true
```



→ In String, ~~all wrapper~~ classes .equals() is overriden for content comparision.

→ In StringBuffer class .equals() is not overriden for content comparision hence object class .equals() got executed which is meant for reference comparison.

→ In wrapper class .equals() is overriden for content comparision

Contract b/w .equals() & hashCode():-

1. If two objects are equal by .equals() compulsory their hashCodes must be same.
2. If two objects are not equal by .equals() then there are no restrictions on hashCode(), they can be same or different.
3. If hashCodes of 2 objects are equal, then we can't conclude above .equals(), it may returns True or false.

4. If hashCodes of 2 objects are not equals then we can always conclude .equals() returns false.

Conclusion :-

→ To satisfy the above contract b/w .equals() and hashCode(), whenever we are overriding .equals() compulsory we should override hashCode().

→ If we are not overriding we won't get any compile time & run-time errors.

→ But it is not a good program practice.

Q) Consider the following .equals()

```
public boolean equals(Object obj)
{
  if(!(obj instanceof person))
  {
    return false;
  }
  person p = (person) obj;
  if (name.equals(p.name) && (age == p.age))
      return true;
  else   return false;
}
```

1) Which of the following hashCode() are said to be properly implemented.

✗ ① public int hashCode()
{
return 100;

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

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

✗ ④ public int hashCode()
{
return (int)height;
}

⑤ public int hashCode()
{
return age + name.length();
}

Note:-

To maintain a Contract b/w .equals() and hashCode(), what ever the parameters we are using while over riding .equals() we have to use the same parameters while overriding hashCode() also.

Clone():-

→ The process of Creating exactly duplicate objects is called Cloning

→ The main objective of cloning is to maintain backup.

① we can get cloned object by using clone() of Objects class.

protected native object clone() throws CloneNotSupportException

```
Class Test implements Cloneable
{
    int i=10;
    int j=20;
  P.S.V.m(——) throws CloneNotSupportException
  {
    Test t1 = new Test();
    Test t2 = (Test) t1.clone();
      t2.i = 888;
      t2.j = 999;
    S.o.pln(t1.i + "----" + t1.j);
  }
}
S.o.pln(t1.hashCode() == t2.hashCode()); // false
S.o.pln(t1 == t2); // false.
```

→ We can call clone() only on cloneable objects.

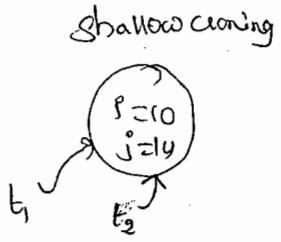
→ An object is said to clonable iff the corresponding class implements Clonable interface. Cloneable interface() presently java.lang package & doesn't contain any methods. It is a masiker interface.

Deep cloning & shallow cloning:-

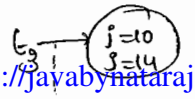
→ The process of creating just duplicate reference variable but not duplicate object is called shallow cloning.

→ The process of creating exactly duplicate independent objects is by bydefault considered as deep cloning.

ex:-
```
Test t1 = new Test();
Test t2 = t1; // shallow cloning
Test t3 = (Test)t1.clone(); // Deep cloning
```



By default cloning means deep cloning.



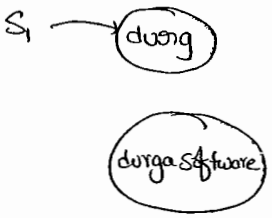
String class

24/05/11

Case(1) :-

Immutable

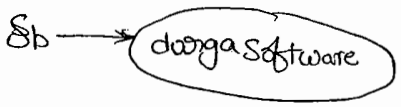
```
String s = new String("durga");
s.concate("software");
s.o.p(s);   durga
```



→ 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 nothing but, "immutability of String objects"

mutable

```
SB s = new SB("durga");
s.append("software");
s.o.pln(s); // durgasoftware
```



→ ONCE we created a StringBuffer Object we can perform any changes in the existing object. this behaviour is nothing but "mutability of StringBuffer Object".

---

getClass() :-

this method returns run-time class definition of an object

Eg:-

```
Test ob = new Test();
S.o.pln("class name: " + ob.getClass().getName());
```

Case(2) :-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.println(s1==s2) ; false
S.o.println(s1.equals(s2)); true
```

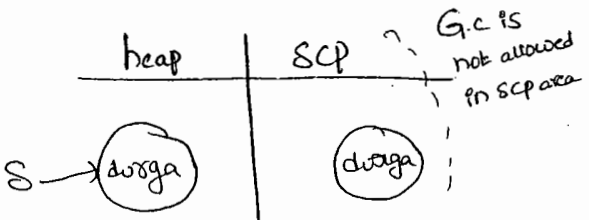
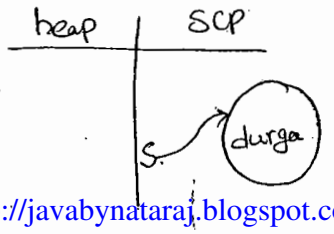
→ In String class .equals() method is overridden for Content Comparision. Hence .equals() method returns True if Content is same eventhough objects are different.

```
StringBuffer sb1 = new StringBuffer("durga");
SB sb2 = new SB("durga");
S.o.println(sb1==sb2) ; false
S.o.println(sb1.equals(sb2)); false
```

→ In StringBuffer class .equals() method is not overridden for Content Compasision. Hence object class .equals() method will be executed which is ment for reference comparision due to this .equals() method returns false eventhough content is same if objects are different

Case (3) :-

* What is the difference b/w following?

String s = new String("durga");	String s = "durga";
→ In this case two objects will be Created one is in heap, & the other is in SCP. and 's' is always pointing to heap object	→ In this case only one object will be Created in SCP and 's' is always pointing to that Object
	

Note:-

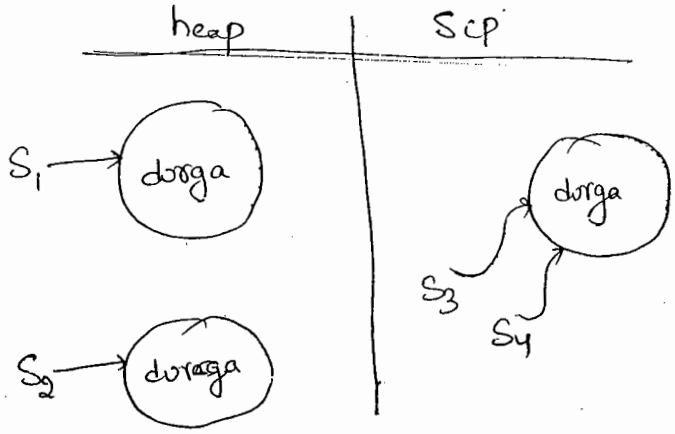
① G.C is not allowed to access in scp area hence eventhough object doesn't have any reference variable still it is not eligible for G.C, if it is present in scp area.

② All objects present on scp will be destroyed automatically at the time of JVM shutdown.

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 it will reuse existing object instead of creating new object. if it is not already available then only a new object will be created. hence, there is no chance of two objects with the same content in scp. i.e, Duplicate objects are not allowed in scp.

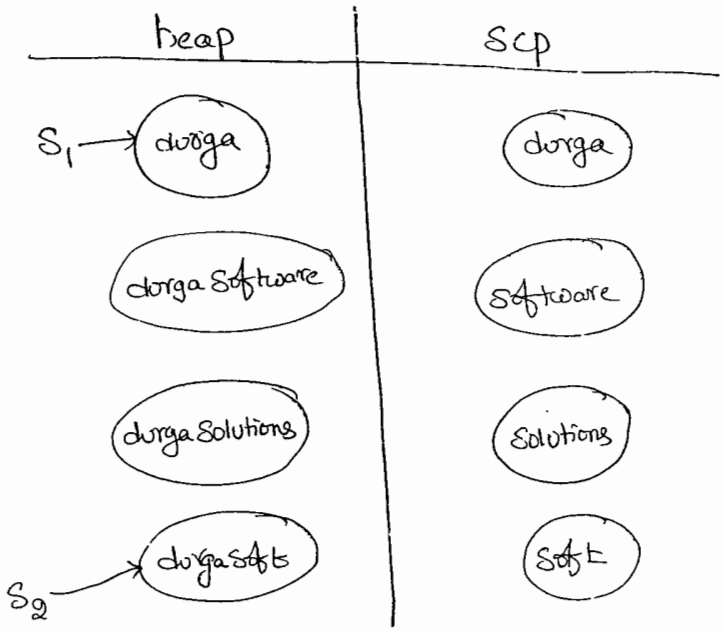
Ex③:-

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



Ex③ :-

```
String s1 = new String("durga");
s1.concate("software");
s1.concate("solutions");
String s2 = new s1.concate("soft");
```



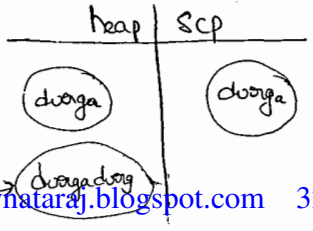
Note :-

→ for every String Constant Compulsory one object will be Created in SCP area.

→ Because of some runtime operation if an object is required to Created that object should be Created only on heap but not in SCP

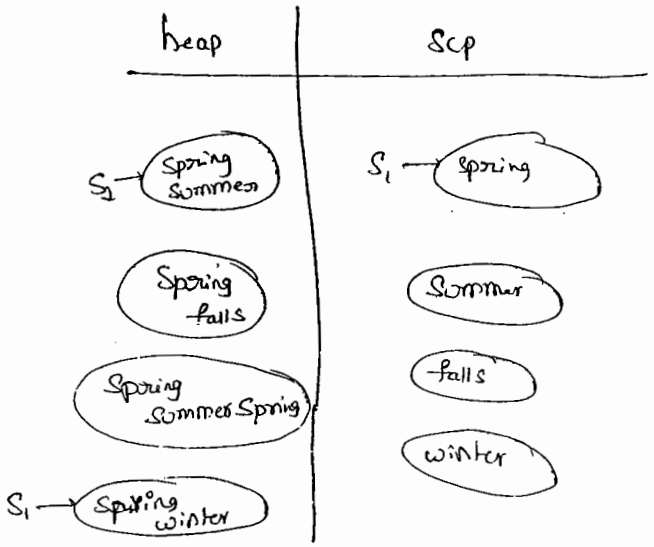
Ex④ :-

```
String s = "durga" + new String("durga");
```



Eg3:-

```
String S1 = "Spring";
String S2 = S1 + "Summer";
S1.concat("falls");
S2.concat(S1);
S1 += "winter";
S.o.pln(S1);
S.o.pln(S2);
```



Eg:- Note:-

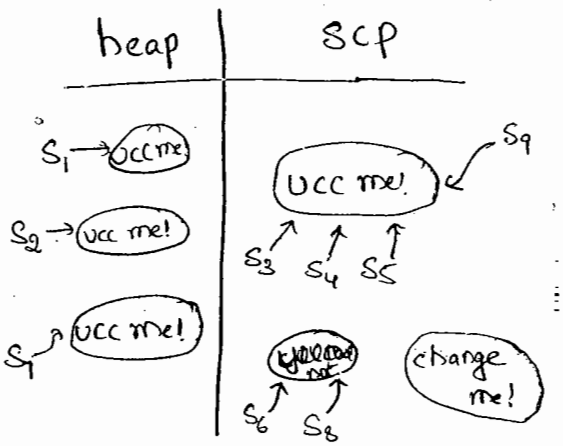
final String S = "raghu"; S is a Constant

String S = "raghu"; S is a normal variable.

Eg:-

String S1 = new String("you can't

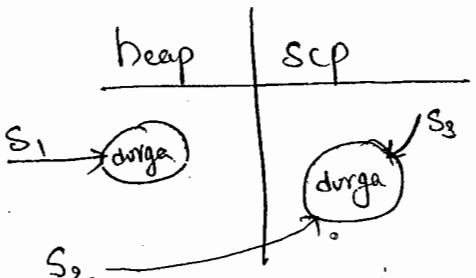
```
String s1 = new String("you cannot change me!");
String s2 = new String("you Cannot change me!");
S.o.pln(s1==s2); false
String s3 = "you cannot change me!";
String s4 = "you cannot change me!";
S.o.pln(s1==s4); true
S.o.pln(s1==s3); false
String s5 = "you Cannot "+"change me!";
S.o.pln(s3==s5); true
String s6 = "you cannot";
String s7 = s6+"change me!";
S.o.pln(s3==s7); false
final String s8 = "you cannot";
String s9 = s8+"change me!";
S.o.pln(s3==s9); true
S.o.pln(s6==s8); true
```



Interning of String :-

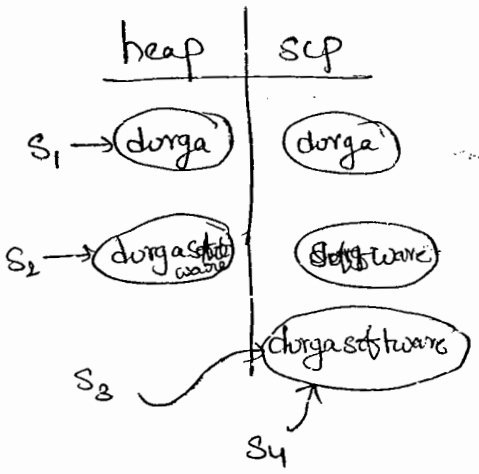
→ By using heap object reference if you want to get corresponding scp object reference then we should go for intern().

Ex:-
```
String s1 = new String("durga");
String s2 = s1.intern();
S.o.pln(s1==s2); false
String s3 = "durga";
S.o.pln(s2==s3); true
```



→ If the corresponding object not available in scp, then intern() creates that object & returns it.

Eg:- String s1 = new String("durga");
String s2 = s1.concat("software");
String s3 = s2.intern();
String s4 = "durgasoftware";
S.o.println(s3 == s4) ; true



Constructors of the String class :-

① String s = new String();

② String s = new String(String constant);

③ String s = new String(StringBuffer sb);

④ String s = new String(char[] ch);

Eg:- char[] ch = {'a', 'b', 'c', 'd'};
String s = new String(ch);
S.o.println(s); abcd

⑤ String s = new String(byte[] b)

Eg:- byte[] b = {100, 101, 102, 103};
String s = new String(b);
S.o.println(s); defg.

Important Methodes of String class :-

① public char charAt(int index);

Eg:- String s = "durga";
S.o.pln(s.charAt[3]); g
S.o.pln(s.charAt[30]); R.E :- StringIndexOutOfBoundsException

② public String concat(String s);

Eg:- String s = "durga";
s = s.concat("Software");
// s = s + "Software";
// s += "Software";
S.o.pln(s); durgasoftware

→ The overloaded +, += operators also ment for Concatination only

③ public boolean equals(Object obj) ment for Content Comparision where the case is also important.

④ public boolean equalsIgnoreCase(String s) ment for Content Comparision where the case is not important.

Ex:- String s = "JAVA";
S.o.pln(s.equals("Java")); false
S.o.pln(s.equalsIgnoreCase("java")); true

Note :- In General to perform Validation of User name we have to go for equalsIgnoreCase method where the case is not important. where as to perform password validation we have to go for equals() where the case is important.

⑤ public String substring(int begin); returns the substring from begin index to End of the String.

⑥ public String substring(int begin, int end); returns the substring from begin index to End-1 index.

Eg:- String s = "abcdefg";

s.o.println(s.substring(3)); defg

s.o.println(s.substring(2,6)); cdef

⑦ public int length();

Eg:- String s = "aabbbb";

s.o.println(s.length); ⟶ C.E: Can't find Symbol
Symbol : variable length
location : class java.lang.String

s.o.println(s.length()); 5

Note:-

length variable applicable for arrays where as length() is applicable for string objects.

⑧ public String replace(char old, char new);

Eg:- String s = "aabbb";

s.o.println(s.replace('a','b')); bbbbb

⑨ public String toLowerCase();

⑩ public String toUpperCase();

⑪ Public String trim();-

→ To remove the blank spaces present at begining & End of the String But not blankspaces present at middle of the String.

⑫ public int indexOf(char ch) :-

→ It returns indexof first occurance of the Specified character

⑬ public int lastIndexOf(char ch);-

Importance of String Constant pool (scp):-

Voter Registration Form

Name of Consistency : chpet

Name : Srinivas

fathername: SitaRamaiah

Age : 22

DOB :

H.NO : 9-133

Street : Ramnagar

Substreet: Ramnagar

City : Ganapavaram

District : Guntur

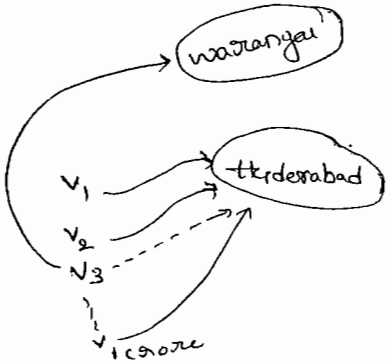
State : A.P

Country : India

PIN : 522619

Identification Name : xxxx xxxx

Submit



→ In our program if any String object required to use repetadely, it is not recommended to create a seperate object for every requirement. this approach reduces performance & memory utilization.

→ we can resolve this problem by creating only one object & share the same object with all required references.

→ This approach improves memory utilization & performance. we can acheive this by using String Constant pool.

→ In scp, a single object will be shared for all required References. Hence the main advantages of scp are memory - utilization & performance will be improved.

→ But the problem in this approach is, As several references pointing to the same object by using one reference, if we are perfoam any change all remaining references will be impacted.

→ To resolve these SUN people declare String objects as Immutable

→ Accoading to that once we created a String object we can't perfoam any change in the existing object. if we are toying to perform any change with

So, that there is no effect on remaining references

→ Hence, "The main disadvantage of scp is we should compulsary maintain String objects as Immutable".

Q) Why SCP like Concept is defined only for String object But not for StringBuffer?

A) → In any Java program, the most Commonly used object is String. Hence with respect to memory & performance Special arrangement is required. for this SCP Concept required.

→ But StringBuffer is not Commonly used object. Hence Special Concepts like SCP is not required.

Q) What are the Advantages of SCP?

A) → Instead of Creating a Seperate object for every requirement we Can Create only one object in SCP & we Can reuse the Same object for Every requirement. So that performance & memory utilization will be increased.

Q) What is the disAdvantage of SCP?

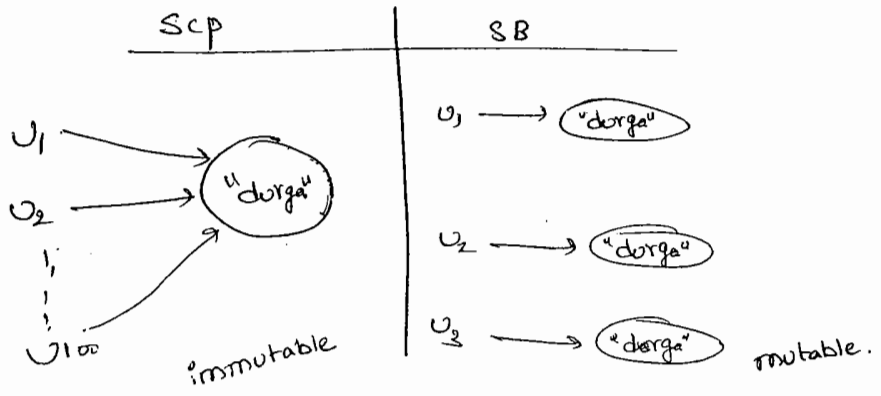
A) → Compulsory we should make String objects as immutable.

Q) Why String objects are immutable where as StringBuffer objects are mutable?

A) → In the Case of String Several references Can Pointing to the Same object. By using one reference, if we are performaining any change in the Existing object the remaining references will be impacted. to resolve this problem SUN people declared as String objects are immutable. According to this ONCE we Created a String object we can't perform any changes in the existing object.

If we are taying to perform any changes, with those changes a new object is created. i.e Scp is the only reason why the String objects are immutable.

→ But in case of StringBuffer for every requirement Commpulsary a Seperate object will be created. Revising the same StringBuffer object, there is no chance. In one StringBuffer object if we are performing any change there is no impact of remaining references. Hence we can perform any changes in the StringBuffer object & StringBuffers objects are mutable.



10/03/11

Q) Is it possible to create our own immutable class?

A) Yes,

Note:

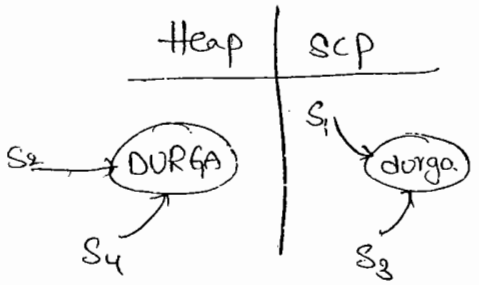
→ Once we created a String object we can't perform any changes in the existing object. If we are taying to perform any change with those changes a new object will be created on the Heap.

→ Because of our runtime method call if there is a change in content then only new object will be created.

→ If there is no change in Content existing object only will be reused.

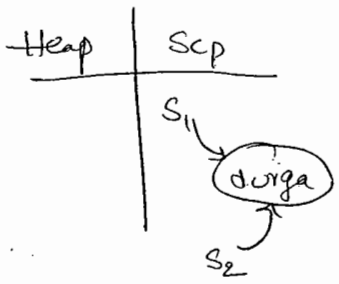
Ex:①

```
String s1 = "durga";
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
String s4 = s2.toUpperCase();
S.o.pln(s1 == s2); false
S.o.pln(s1 == s3); True
S.o.pln(s2 == s4) True
```



Ex:②

```
String s1 = "durga";
String s2 = s1.toString();
S.o.pln(s1 == s2); True
```



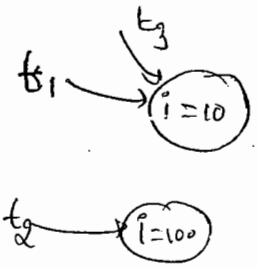
Creation of Our Own Immutable class :-

Sol:- We can create our own immutable classes also.

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

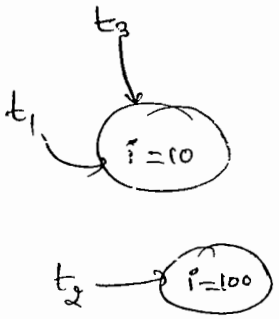
→ Because of our runtime method call if there is no change in the content then existing object only will be returned.

Ex:-



Ex:-

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




```
Test t1 = new Test (10);
Test t2 = new Test (100);
Test t3 = new Test (10);
S.o.pln (t1 == t2);   false.
S.o.pln (t1 == t3);   true
```

Q) In Java which objects are Immutable ?

A) (1) String objects ?

(2) All wrapper objects are immutable

StringBuffer :-

→ If the Content will change frequently then it is never recommended to go for String. Because for every change Compulsary a new object will be Created.

→ To handle this requirement Compulsary we should go for StringBuffer where all changes will be performed in existing object only instead of Creating new object.

Constructors :-

→① StringBuffer sb = new StringBuffer();

→ Creates an Empty StringBuffer object with default initial Capacity 16

→ Once StringBuffer reaches its max. Capacity a new SB object will be Created with.

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

Ex:-

```
StringBuffer sb = new StringBuffer();
S.o.pln(sb.capacity()); //16
sb.append("abcdefghijklmnop");
S.o.pln(sb.capacity()); 16
sb.append("q");
S.o.pln(sb.capacity()); 34.
```

② StringBuffer sb = new StringBuffer(int initialCapacity);

→ Creates an Empty SB object with Specified initialCapacity

③ StringBuffer sb = new StringBuffer(String s);

→ Creates an equivalent SB object for the given String with,

Capacity = 16 + s.length();

Important Methods of StringBuffer class:

(1) public int length()

(2) public int capacity()

(3) public char charAt(int index);

ex:- StringBuffer sb = new StringBuffer("durga");

S.o.pln(sb.charAt(2)); g

S.o.pln(sb.charAt(30)); } R.E:- StringIndexOutOfBoundException

S.o.pln(sb.charAt(5)); }

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

→ To replace the character Locating at Specified index with the provided Character.

(5) public StringBuffer append(String s)
append(int i)
append(boolean b)
(double d)
(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.pln(sb);

⑥ public StringBuffer insert(int index, String s);

(int index, int i);

(" boolean b);

(" double d);

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

sb.insert(3, "sainu");

S.o.pln(sb); dursainuga.

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

→ To delete the characters Present at begin index to End-1 index

⑧ public StringBuffer deleteCharAt(int index);

→ To delete the character Locating at Specified index.

⑨ public StringBuffer reverse();

eg:- SB sb = new SB("durga");

S.o.pln(sb.reverse()); agrud.

⑩ public void setLength(int Length);

⑩ public void setLength(int Length);

Eg:- StringBuffer sb = new StringBuffer("durga123456");
sb.setLength(8);
S.o.pln(sb); durga123

⑪ public void ensureCapacity(int Capacity);

→ To set the Capacity based on our requirement.

Eg:- StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); 16
sb.ensureCapacity(2000);
System.out.println(sb.capacity()); 2000

⑫ public void trimToSize()

→ To release extra allocated free memory. after calling this method, Length & Capacity will be equal.

Eg:- StringBuffer sb = new StringBuffer();
sb.ensureCapacity(2000);
sb.append("durga");
sb.trimToSize();
S.o.pln(sb.capacity()); 5

StringBuilder :-

→ Every method present in StringBuffer is Synchronized, Hence at a time only one Thread is allowed to access StringBuffer object. It Increases waiting time of the Threads & effects performance of the System.

→ To resolve this problem SUN people introduced StringBuilder in 1.5 version.

→ StringBuilder is exactly Same as StringBuffer (including methods & Constructors) except the following differences :**

(#)

StringBuffer	StringBuilder
① Every method is Synchronized	① No method is Synchronized.
② SB object is Thread Safe. Because SB object can be accessed by only one thread at time.	② StringBuilder is not Thread Safe Because it can be accessed by multiple Threads Simultaneously.
③ Relatively performance is Low	③ Relatively performance is High.
④ Introduced in 1.0 Version	⑤ Introduced in 1.5 Version

* String Vs StringBuffer Vs StringBuilder :-

→ If the Content will not ~~only~~ change frequently then we should go for String

→ If Content will change frequently & ThreadSafety is required. then we should go for StringBuffer.

→ If Content will change frequently & ThreadSafety is not required. then we should go for StringBuilder.

Method chaining :-

→ For most of the methods in String, StringBuffer & StringBuilder the return-type is same type only. Hence after applying a method on the result we can call another method with forms method chaining

```
Sb.m1().m2().m3().m4().m5().----;
```

→ In method chaining all methods will be executed from Left to Right.

Ex :-
```
StringBuffer sb = new StringBuffer();
sb.append("durga").insert(2,"xyz").reverse().delete
                                    delete(2,7).append("solutions");
S.o.println(sb); //agdsolutions
```

final vs immutable :-

300

→ If a reference variable declareded as the final then we can't reasign that reference variable to some other object.

Ex:-
```
final StringBuffer sb = new StringBuffer("durga");
sb = new StringBuffer("Software");
```
C.E:- Can't assign a value to final variable sb.

→ Declaring a reference variable as final we wont get any immutability natuae, in the corresponding object we can perform any type of change Eventhough reference variable declared as final.

Ex:-
```
final StringBuffer sb = new StringBuffer("durga");
sb.append("software");
S.o.pln(sb); durgasoftware
```

→ Hence final variable & Immutability both Concepts are different.

* Wrapper classes :-

→ The main objectives of wrapper classes are

(i) To wrap primitives into object form, so that we can handle primitives just like objects.

(ii) To define several utility methods for the primitives.

Constructors of wrapper classes (or)

Creation of wrapper objects :-

→ Almost all wrapper classes contains two constructors, one can take corresponding premitive as argument & the other can take String as argument.

Ex:-

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

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

→ If the String is not properly formatted then we will get R.E saying NumberFormatException.

Ex:- Integer I = new Integer("ten"); R.E:- NFE

→ Float class contains 3 constructors one can take float premitive, and the other can take String & 3rd one can take double argument

Ex:- 1) Float F = new Float (10.5f); ✓

2) Float F = new Float ("10.5f"); ✓

3) Float F = new Float (10.5); ✓ ⟶ double

* Character class Contains only one Constructor which Can take Char premitive as aargument.

Ex:- 1) Character ch = new Character ('a'); ✓

2) Character ch = new Character ("a"); ✗

* Boolean class Contains two Constructors one Can take boolean premitive as the aargument & other Can take String as aargument.

→ If we are passing boolean premitive as aargument the only allowed values are true, false. by mistake if we are providing any other we will get CompiletimeError.

Ex:- ✓ Boolean B = new Boolean (true);

✗ Boolean B = new Boolean (True);

→ If we are passing String aargument to the Boolean Constructor then the Case is not important & Content also not important.

↔ If the Content Case insensitive String ~~(true)~~, otherwise it is treted as false.

Ex:- (1) Boolean b = new Boolean ("true"); ✓ true

(2) Boolean b = new Boolean ("True"); ✓ true

(3) Boolean b = new Boolean ("TRUE"); ✓ true

(4) Boolean b = new Boolean ("durga"); ✓ false

(5) Boolean b = new Boolean ("") ✓ false

Wrapper classes	Corresponding Constructor arrgeuement
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
* Boolean	boolean or String

Q):- Which one is True & false

(1) Boolean b1 = new Boolean("yes");

(2) Boolean b2 = new Boolean("no");

S.o.pln(b1.equals(b2)); ⟶ true

S.o.pln(b1==b2); ⟶ false

S.o.pln(b1); false

S.o.pln(b2); false.

Note:-

→ In Every wrapper class toString() is overridden to return it's Content.

→ In Every wrapper class .equals() is overridden for Content Comparision.

Utility Methods :-

There are 4 methods

(i) valueOf()
(ii) xxxValue()
(iii) parseXxx()
(iv) toString()

(i) valueOf() :-

→ We can use valueOf() methods for Creating wrapper object as alternative to Constructor.

form 1:-

→ Every wrapper class Except Character class Contains a Static valueOf() method for Converting for Converting String to the wrapper Object.

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

Eg:- Integer I1 = Integer.valueOf("10"); ✓

Boolean b1 = Boolean.valueOf("true"); ✓

Double D = Double.valueOf("p.s"); ✓

form(2) :-

→ Every Integral type wrapper class (Byte, Short, Integer, Long) Contains the following valueOf() method to Convert Specified Radix String form to Corresponding Wrapper object.

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

Ex:-

```
Integer I1 = Integer.valueOf("1010", 2);
S.o.pln(I1); 10
Integer I2 = Integer.valueOf("1111", 2);
S.o.pln(I2); 15
```

2 to 36

base-10 : 0-9
base-11 : 0-9, a
!
base-16 : 0-9, a-f
base-17 : 0-9, a-g
!
base-36 : 0-9, a-z.
10 + 26
= 36

form(3) :-

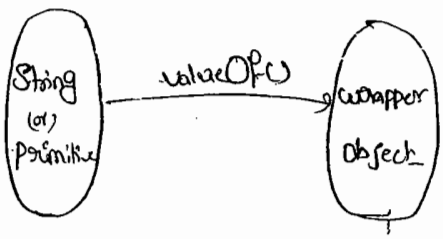
→ Every wrapper class including Character class Contains the following valueOf() to Convert primitive to Corresponding wrapper Object

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

Eg:-

1) Integer I = Integer.valueOf(10); ✓
2) Character ch = Character.valueOf('a'); ✓
3) Boolean B = Boolean.valueOf(true); ✓

Note:



15/03/11

(ii) xxxValue():-

→ We can use xxxValue() methods to convert wrapper object to primitives.

→ Every number type wrapper class contains the following six xxxValue() methods.

→ The methods are

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

eg:-

(1)
```
Double D = new Double(130.456);
S.o.pln(D.byteValue());   -126
S.o.pln(D.shortValue());  130
S.o.pln(D.intValue());    130
S.o.pln(D.longValue());   130
S.o.pln(D.floatValue());  130.0
S.o.pln(D.doubleValue()); 130.0
```

charValue():-

→ Character class contains charValue method to convert Character Object to the char primitive.

* public char charValue();

eg:- Character ch = new Character('@');

char ch1 = ch.charValue();

S.o.pln(ch1); '@'

booleanValue():

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

public boolean booleanValue();

eg:- Boolean B = Boolean.valueOf("durga");

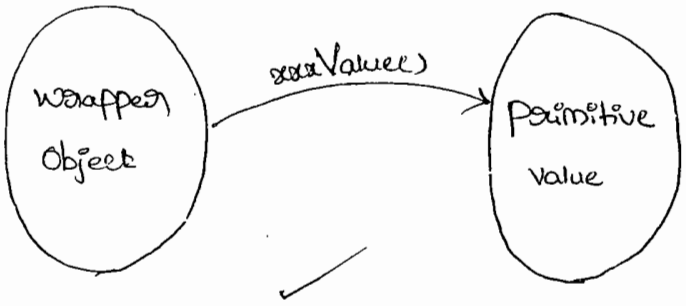
boolean b = B.booleanValue();

S.o.pln(b); false.

6×6=36
+1
+1
= 38

Note:-

→ In total 38 (=6×6+1+1) xxxValue() are vailable.



(iii) parseXxx() :-

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

form1 :-

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

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

Eg:-

```
int i = Integer.parseInt("10");
double d = Double.parseDouble("10.5");
long l = Long.parseLong("10L");
Boolean b = Boolean.parseBoolean("durga"); // false
```

form2 :-

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

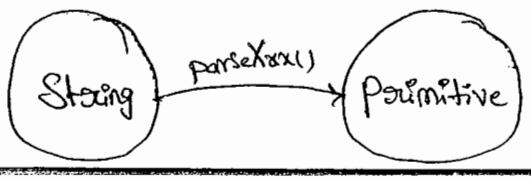
Eg:- public static primitive parseXxx(String s, int radix);

Eg:- int i = Integer.parseInt("1111", 2);

So.pln(i); 15

2 to 36.

Note:-



(iv) toString() :-

→ We can use toString() to Convert Wrapper Object or primitive to String.

form1 :-

→ Every wrapper class contains the following toString(), to Convert Wrapper Object to String type.

```
public String toString();
```

→ It is the Overriding version of object class toString()

Eg:- ① Integer I = new Integer(10);

S.o.pln(I.toString()); 10

form2 :-

→ Every wrapper class Contains a Static toString(), to Convert primitive to String form.

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

✓ String s = Integer.toString(10);

✓ String s = Boolean.toString(true);

form3 :-

→ Integer & Long classes contains toString() to Convert primitive to Specified radix String form.

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

2 to 36

Eg: String s = Integer.toString(15, 2);

s.o.p(s); 1111

Form 4:-

→ Integer & Long classes contains the following toXxxString().

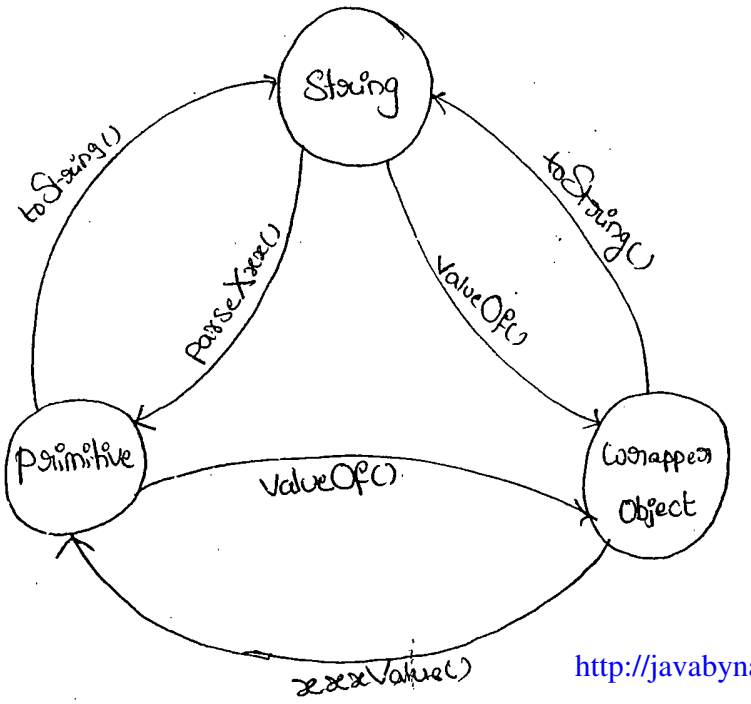
1. public static String toBinaryString(primitive p);
2. public static String toOctalString(primitive p);
3. public static String toHexString(primitive p);

Ex:- String s = Integer.toHexString(123)

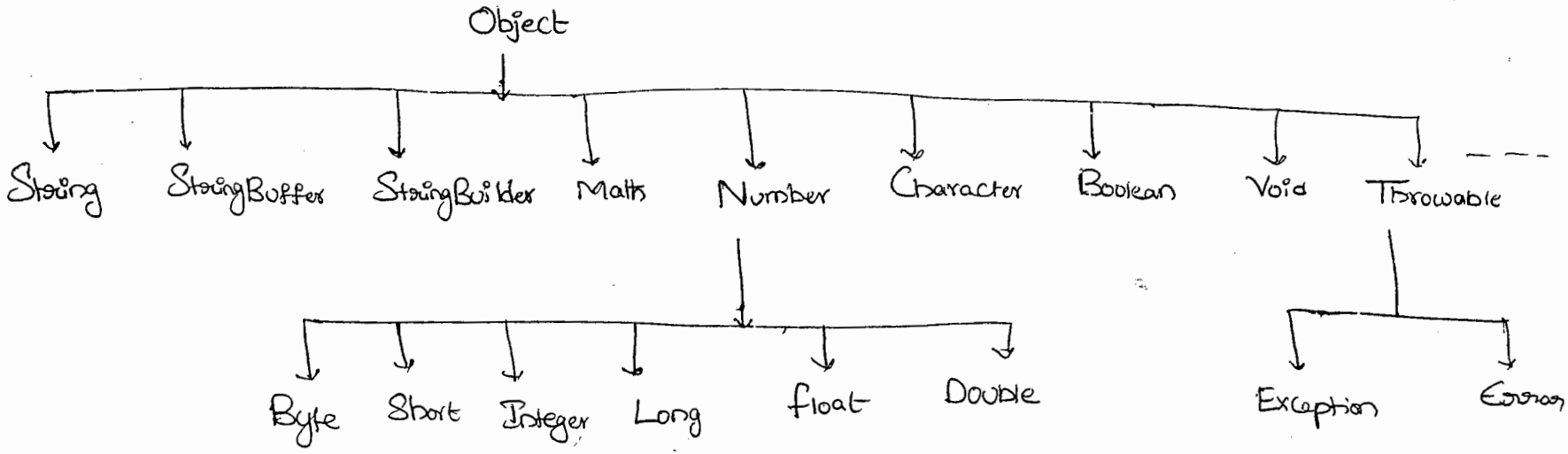
s.o.p(s); "7b"

16 | 123
7 - b

Dancing b/w String, Wrapper Object & primitive Value:-



Partial hierarchy of java.lang package:-



* String, StringBuffer, StringBuilder, All Wrapper Classes are final.
* The Wrapper Classes which are not child classes of Number, are Character & Boolean.
* the wrapper classes which are not direct child classes of Object are Byte, Short, Integer, Long, Float, Double
* Some-times we Can Consider Void also as Wrapper Classes
* In addition to String object all Wrapper objects are Immutable.

16-3-11

305

Autoboxing & Autounboxing :- (1.5v)

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

Ex:-

① ArrayList l = new ArrayList();

l.add(10); ✗ C.E!.

Integer I = new Integer(10);

l.add(I); ✓

② Boolean B = new Boolean(true);

```
if(B)
{
  S.o.pln("Hello");
}
```

C.E!- Incompatible types
found : Boolean
required : boolean

```
boolean b = B.booleanValue();
if(b) ✓
{
  S.o.pln("Hello"); ✓
}
```

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

Conversions are called Autoboxing & Auto unboxing.

Autoboxing :-

→ Automatic Conversion of primitive value to the wrapper Object by Compiler is called "Autoboxing".

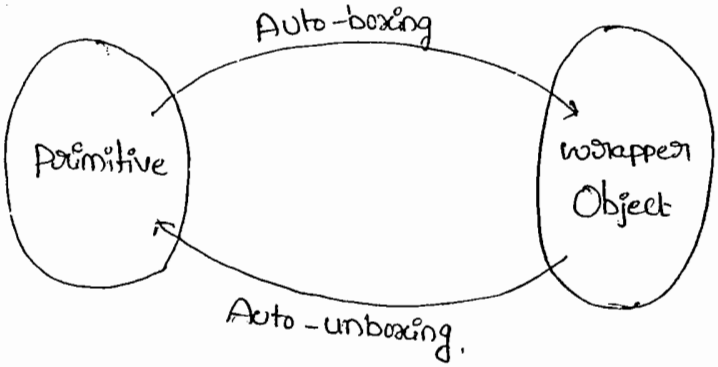
Ex :- ✓D Integer I = 10; [Compiler Converts int to Integer automatically by Autoboxing]

Auto-unboxing :-

→ Automatic Conversion of wrapper Object to the primitive type by Compiler is called "Auto-unboxing".

Ex :- ✓D int i = new Integer(10); [Compiler Converts Integer to int automatically by Auto-unboxing]

Note :-



Ex :-

① Integer I = 10;

↳ after Compilation this line will become

Integer I = Integer.valueOf(10);

i.e, Autoboxing Concept internally implimented by using valueOf()

Ex②:-

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

→ After Compilation this Line will become

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

i.e, Autounboxing Concept internally implemented by using xxxValue().

Exam purpose:-

ex①:-

```
class Test
{
  static Integer I=10; ——→ ① A.B
  p.s.v.m(String[] args)
  {
    int i = I; ——→ ② A.U.B
    m1(i);
  }                  →③ A.B
  p.s.v.m1(Integer I)
  {
    int k = I; ——→ ④ A.U.B
    S.o.println(k); 10
  }
}
```

Note:-

→ Because of Autoboxing & Auto-unboxing, from 1.5Version onwards there is no diff. blw primitive Value & Wrapper Object. we can use interchangeably.

Ex2 :-

```
class Test
{
Static Integer I=0;
P.S.V.m(String[] args)
{
  int i = I;
  S.o.pln(i); //0
}
}
```

int i = I.intValue();

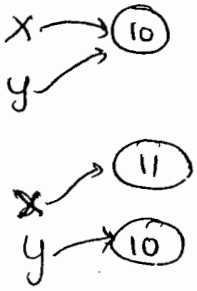
```
class Test
{
  Static Integer I;
  P.S.V.m(String[] args)
  {
    int i = I;
    S.o.pln(i);
  }
}
```

→ R.E:- NPE

int i = I.intValue()
↓
null

Ex3 :-

```
Integer x = 10;
Integer y = x;
 x++;
✓ S.o.pln(x); 11
✓ S.o.pln(y); 10
✓ S.o.pln(x==y); false
```



Note :-
because if we want to changes after creating an object, then that new changed object is created with the same reference name.

Ex4 :-

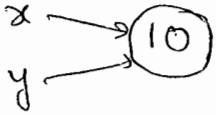
① Integer X = new Integer(10);
Integer y = new Integer(10);
S.o.pln(x==y); false ✓

② Integer x = new Integer(10);
Integer y =10;
S.o.pln(x==y); false ✓

③ Integer x = 10;
Integer y = 10;

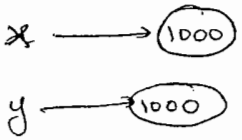
S.o.pln(x == y); true ✓

④ Integer x = 100;
Integer y = 100;



S.o.pln(x == y); true ✓

⑤ Integer x = 1000;
Integer y = 1000;



S.o.pln(x == y); false ✓

Conclusion :-

→ By AutoBoxing if an Object is required to Create Compiler won't Create that object immidiately. first check is any object already Created

→ If it is already Created then it will reuse existing object. instead of Creating new one.

→ If it is not already there, then only a new object will be Created.

→ But this rule is applicable 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

→ Except the above range in all other Cases Compulsory a new object - will be Created.

Ex:-

① Integer I_1 = 127;
Integer I_2 = 127;
S.o.pln (I_1 == I_2); true

② Integer I_1 = 128;
Integer I_2 = 128;
S.o.pln (I_1 == I_2); false

③ Float f_1 = 10.0f;
Float f_2 = 10.0f;
S.o.pln (f_1 == f_2); false

④ Boolean b_1 = true;
Boolean b_2 = true;
S.o.pln (b_1 == b_2); true.

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

→ Overloading w.r.t Auto-boxing, widening & Var-Arg methods:-

case(1):-

Widening Vs Auto-boxing:-

Ex:-
```
Class Test
{
 p.s.v.m1(long l)
 {
  S.o.pln("widening");
 }
 p.s.v.m1(Integer I)
 {
  S.o.pln("Autoboxing");
 }
}
```

```
P.S.V.m(String[] args)
{
  int x=10;
    m1(x);   o/p:- widening
}
}
```

→ Widening dominates Auto-boxing

Case(2):-

→ Widening Vs Var-arg():-

Eg:- Class Test
{
P.S.V.m1(long l)
{
S.o.pln("widening");
}
P.S.V.m1(int... i)
{
S.o.pln("Var-arg");
}
P.S.V.main(String[] args)
{
int x=10;
m1(x);   o/p:- widening
}
}

→ Widening dominates Var-arg().

Case 3:-

→ Auto-boxing Vs Var-arg:-

Ex:- Class Test

```
{
 p.s.v.m1(Integer I)
 {
  S.o.pln("Autoboxing");
 }
 p.s.v.m1(int... i)
 {
  S.o.pln("Var-arg");
 }
 P.S.v.m(String[] args)
 {
   int x=10;
    m1(x);      o/p:- Autoboxing.
 }
}
```

→ In General Var-arg() will get least priority, if no other method matched then only Var-arg() will be Executed.

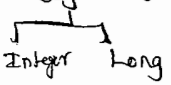
→ While resolveing overloaded methods Compiler will always keeps the percidence in the following oorder.

(i) Widdening

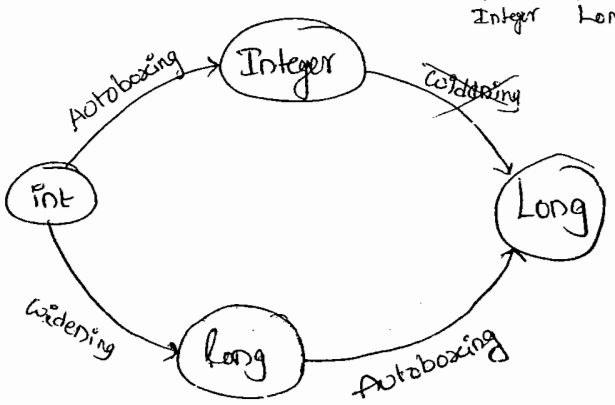
(ii) Auto-boxing

(iii) Var-arg().

Case 4 :-



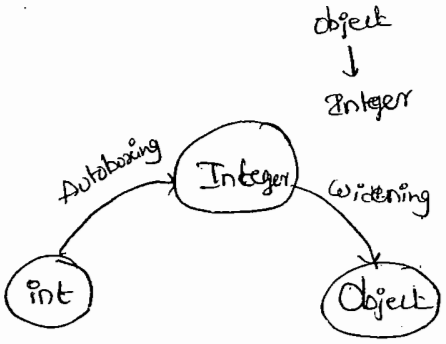

```
Class Test
{
  p.s.v.m1(Long l)
  {
   S.o.pln("Long");
  }
  p.s.v.main(String[] args)
  {
   int x=10;
   m1(x);
  }
}
```



C.E:-

m1(java.lang.Long) in Test cannot be applied to (int)

→ Widening followed by Auto-boxing is not allowed in java. where as Autoboxing followed by widening is allowed.



Ex!-

```
Class Test
{
  p.s.void m1(Object o)
  {
   S.pln("Object");
  }
  p.s.void.main(String[] args)
  {
   int x=10;
   m1(x); // Object
  }
}
```

Q) Which of the following declarations are valid.

✓ ① long l = 10;

✗ ② Long l = 10;

✓ ③ Object o = 10;

✓ ④ double d = 10;

✗ ⑤ Double d = 10;

✓ ⑥ Number n = 10;