Avishek

7

* Java.lang.Package (105 - 119)
* File I/O (120 - 124)

#105 Java.lang.package

05 May 23

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

* Introduction

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

→ For writing any java program wether it is simple or complese the most commanly required classes & interfaces are grouped into a seperate package which is java.lang package.

→ We are not required to import java.lang package explicitly bcz. all classess & interfaces present in lang package by default available to every java program.

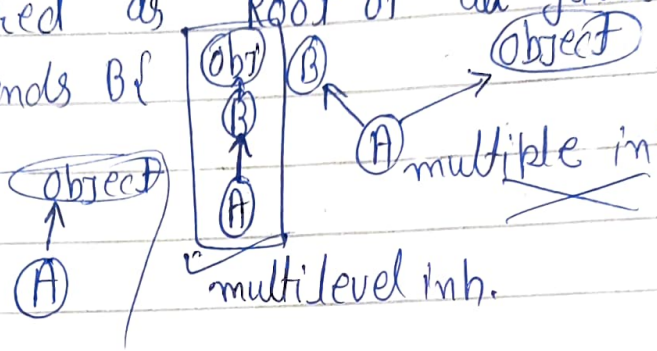
* Java.lang.Object / Object class.

→ The most commanly required method for every Java class (wether it is pre-defined class or customized class) are defined in seperate class which is Object class.

→ Every class in Java is a child class of object either directly (or) in-directly so that object class methods by-default available to every java cl
→ If is considered as Root of all java class.

> class A extends B{
}

> class A {
}



→ If our class doesn't extend any other class then only our class is the direct class of object
→ If our class extends any other class then our class is in-direct child class of object.
⇒ Either directly or indirectly Java won't provide support for multiple inheritance.

* Object class defines 11 methods.
> p String toString()
> p native int hashCode()
> p bool equals(Object o)
> protected native Object clone() throws CNSE
> protected void finalize() throws Throwable.
> p final class getClass()
> p final void wait() throws InterruptedException.
> p final native void wait(long ms) throws IE
> p final void wait(long ms, int ns) throws IE
> p final native void notify()
> p native final void notifyAll()

→ Strictly speaking object class contain 12 method extra method is registerNatives()

> private static native void registerNatives();

→ This method internally require for object class.

(46:00)

① toString()

06 May 23

→ We can use toString() method to get string represention of an Object.

> String s = obj.toString();

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

> Sopln(s); ⇒ Sopln(s.toString());

```
> class Student {
        String name;
        int rollno;
        Student(Str name, int rollno){
            this.name=name;
            this.rollno = rollno;        }
    p s v m (S[] args) {
        Student S1 = new Student("durga", 101);
        Student S2 = new Student("Ravi", 102);
        Sopln(S1);
        Sopln(S1.toString());
        Sopln(S2);
    }
```

Internal implementation

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

→ We can override toString based on our requirement

> public String toString() {
    return name + "---" + rollno;
  }

> class Test {
    p s v m(S[] a) {
      String s = new String("durga");
      Sopln(s); durga ✓
      Integer I = new Integer(10);
      Sopln(I); -10 ✓
      AL l = new AL();
      l.add("A"); l.add("B");
      Sopln(l); [A, B]
      Test t = new Test();
      Sopln(t); Test@375927 } }

→ In all wrapper classes, in all collection classes String class, StringBuffer & StringBuilder classes. toString() method is overrid for meaningful string representation

→ It is recommended to override toString() in our class also.

#106 Object Class . Hashcode

06/May/23

* 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 object into hashing related D.S. like Hashtable, HasMap, HashSet etc.
→ The main adv. of saving objects based on hashCode is search operation will become easy (O(n))
> public native int hashCode();
→ hashCode() method generate hashcode based on address of object, it doesn't mean hashCode represent address of object.
→ Based on our requirement we can override.

```
> Class Student {

    public int hashCode() { return 100; }   } Improper

    public int hashCode() { return rollno; } } proper way

}
```

→ Overriding hashCode method is said to be proper iff for every object we are generating a unique number.

* toString Vs hashCode()

→ If we are executing toString(), it internally call hashCode() method.

→ If we are overriding toString() method then our toString() may not call hashCode() method.

```
> Class Test {
      int i;
      Test(int i) {
         this.i = i; }
      public static v main(S) {
         Test t1 = new test(10);
         Test t2 = new test(100);
         Sopln(t1); Test@ ~~
         Sopln(t2); Test@ ~~
```

object ⇒ toString()
⇓
object ⇒ hashCode()

```
> class Test {
      int i;
      Test(int i) {
         this.i = i; }
      p s v m (s[]a) {
         Test t1 = new Test(10);
         Test t2 = new Test(100);
         Sopln(t1); Test@ a
         Sopln(t2); Test@ 64
   } }
}
```

```
public int hashCode() {
   return i;  }
```

object ⇒ toString()
⇓
Test ⇒ hashCode()

16 | 100
6-4

#107 equals()

08/May/23

```
> class Test {
      int i ;
      Test (int i) {
         this.i = i }
      public string tostring ( ) {
         return i + " " ;      }
      public int hashCode() {
         return i ;    }                    Test => tostring()
      p s v m (S[]a){
         Test t1 = new Test(10);
         Test t2 = new Test(100);
         Sopln(t1) ; //10
         Sopln(t2); //100
   } }
```

107

* equals()

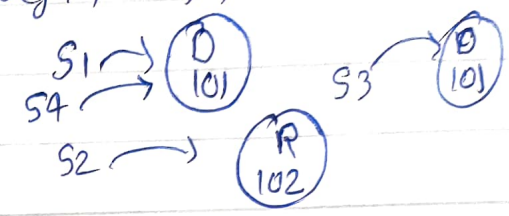
→ We use equals() method to check equality of two objects.

> Obj1. equals(Obj2).

→ If our class doesn't contain equals method then object class equals method will be executed.

```
> class Student {
      string name ;
      int rollno ;
      Student (str name, int rollno) {
         this.name = name;
         this.rollno = rollno;      }
```

```
p s v m (S [] a){
    Student s1 = new student("Durga", 101);
     S  s2      = new S("Ravi", 102);
          S s3 = new S("Durga", 101);
          S s4 = s1;
    Sop1n (s1.equals(s2)); f
    Sop1n (s1.equals (s3)); f.
    Sop1n (s1. equals (s4)); t
}
```



→ Object class .equals() method got executed which is ment for reference comparision.

→ We can override equals() method for content comparision.

→> public boolean equals (Object obj) {
for{
String name1 = this.name;
int rollno1 = this.rollno;
S s = (Student) obj ~~name~~);
S name2 = s.name;
int rollno2 = s.rollno;
if(name1.equals(name2) && rollno1 == rollno2){
return true;
}else {
return false;
}} catch (CCE e) { return false;}
}, s1.equals(s2); false }Catch(NPE e){return false;}
> s1.equals (s3); true
> s1. equals (s4); true.

our override.

> Sopln (s1.equals ("durga")); RE: CCE (classCastExcep)
↳ false (after modification)

> Sopln (s1.equals (null)); → RE: NPE
↳ false (after modification)

→ While overriding .equals() for content comparision we have to take care about the following.

1) What is the meaning of equality i.e. (name only or roll no only or both)
2) If but we are passing diff. type of object our equals method should not raise CCE i.e we have to handle CCE to return false.
3) If we are passing null arg. then our equals() method should not raise NPE i.e. we have to handle NPE to return false.

* Symplified version of equals()

```
> public boolean equals(Obj obj){
    try{ Student s = (Student) obj;
      if (name.equals(s.name) && rollno == s.rollno){
        return true;
      } else {
        return false
      }
    }Catch(CCE e){ return false;}
     Catch (NPE e){ return false;}
  }
```

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

Note :- To make above equals() more efficient we have to write following code in the begining

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

```
String s1 = new String("durga");
String s2 = new String("durga");
Sop(s1 == s2);                    false
Sop(s1.equals(s2));               true
```

```
StringBuffer sb1 = new SB("durga");
SB sb2 = new SB("durga");
Sop(sb1 == sb2);                  false.
Sop(sb1.equals(sb2));             false
```

→ In string .equals() are overriden for content comparision but in SB .equals() is not overriden then object method got executed.

#108 finalize() || getClass() || notify()

07/May/23

* getClass()

→ We can use getClass() method to get runtime class defination of an object.

> public final class getClass()

→ by using Class class object we can access class level properties like, name of class, methods info., constructors info. etc.

```
> p s v m(S[] a){
    Object o = new String("durga");
    Class c = o.getClass();
    Sopln(c.getName());
    Method[] m = c.getDeclaredMethods();
    for(Method m1 : m){
        Sopln(m1.getName());
        Count++;
    }
}
```

→ To display DB vendor specific Connection (I) implemented class name.

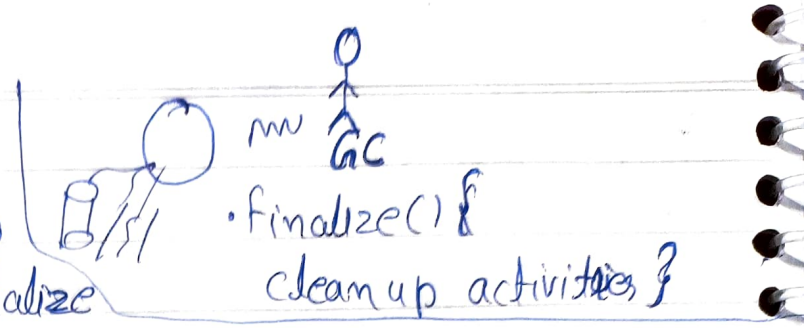
→ Connection con = DriverManager.getConnection(--);
Sopln(con.getClass().getName());

Note :- After loading every .class file JVM will create an object of type java.lang.Class in the heap area, we can use this class obj. to get class level information.

→ We use this method in reflection.

* finalize()

→ Just before distroying an object GC calls finalize method to perform clean-up activities.



→ Once finalize() completes automically GC distroyes the object.

* wait(), notify(), notifyAll() #2(P) #1(C)

→ We can use these method for inter-thread communication.

Que .wait();
.notify()

→ The thread which is expecting updation, it is responsible to call wait() method then immediatly 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 the notification & continue its execution with those update.

#109 String Class

07/May/23

{Kathy Sierra}

* String Class :~

C-I

> String s = new String("durga");
s.concat("Software");
Sop(s); durga ✓

s → durga → durgaSoftware

> StringBuffer sb = new SB("durga");
sb.append("Software");
Sop(sb); durgaSoftware

sb → durgaSoftware

→ Once we create a String object we can't perform any changes in the existing object. If we try to perform any change with those changes a new object will be created. This non-changable bevior is immutability of String.

→ Once we create a StringBuffer object we can perform any chang in the existing object this changable bevibour is mutability of StringBuffer

C-II

> String s1 = new S("durga");
String s2 = new S("durga");
Sop(s1==s2); false
Sop(s1.equals(s2)); true

> SB s1 = new SB("durga");
SB s2 = new SB("durga");
Sop(s1==s2); false
Sop(s1.equals(s2)); false

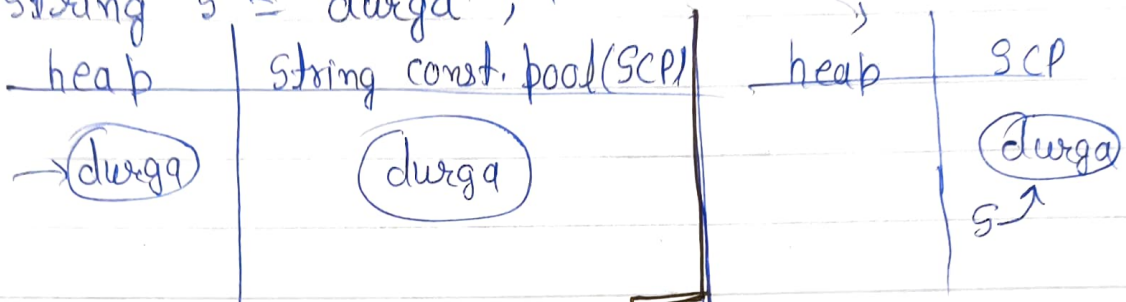
→ In String class .equals() method is overriden for content comparision.

→ In StringBuffer class .equals() method is not overriden for content comparision. hence object class .equals() got executed.

C-III

String s = new String("durga");

String s = "durga";



→ In this case 2 object will be created one in the heap area & another one in SCP. & s is always pointing to heap obj.

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

⇒ Object creation in SCP is optional first it will check is there any object already present in SCP with required content.

→ If obj. already present then existing obj. will be re-used.

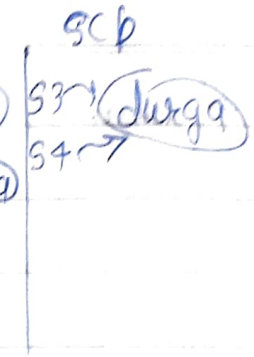
→ If obj. not already available then only a new obj will be created.

→ This rule is applicable only for SCP but not for heap.

⇒ GC is not allow to access SCP area. hence even though object doesn't contain ref. var it is not elegible to GC.

→ All SCP objects will be destroyed automatically at the time of JVM shutdown.

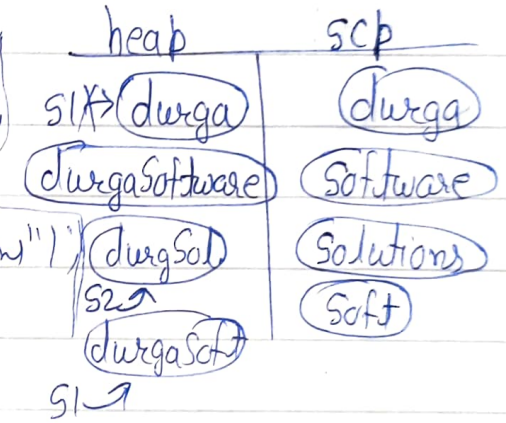
> String s1 = new String("durga"); s1→durga
String s2 = new String("durga"); s2→durga
String s3 = "durga";
String s4 = "durga";



→ Total object created → 3

Note:- Whenever we are using new operator compulsory a new object will be created in the heap area. hence there may be chance of existing 2 objects with same content in heap area but not in SCP.

> String s1 = new String("durga");
s1.concat("software");
String s2 = ~~new~~ s1.concat("solutions");
s1 = s1.concat("soft");
Sopln(s1); durgasoft
Sopln(s2); durgasolutions.

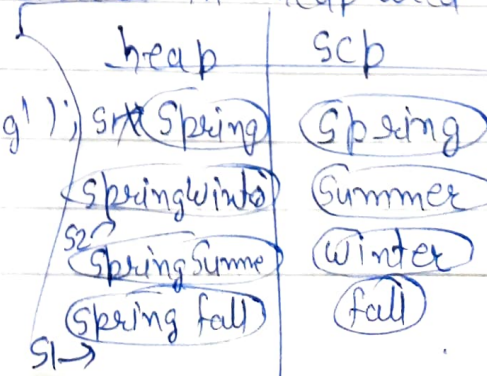


⇒ Total object created = 8

Note:→ For every string constant one object will be placed in SCP area.

→ bcz of runtime operation if an obj. is required to create that object will be placed in heap area but not in SCP area.

> String s1 = new String("Spring");
s1.concat("Summer");
String s2 = s1.concat("winter");
s1 = s1.concat("fall");
Sopln(s1); Sopln(s2);



#110 String class Constructor

07/May/23

* Constructor

① String s = new String();
② String s = new String(String literal);
③ String s = new String(StringBuffer sb);
④ String s = new String(char[] ch);
⑤ String s = new String(byte[] b);

eg byte[] b = {100, 101, 102, 103};
String s = new String(b);
Sop ln(s); defg

* Methods.

> public char charAt(int index);
> public String concat(String s);
we can use '+' also.

-> String s = "durga"
s = s.concat("Soft");
Sop ln(s); durgaSoft.

> public bool equals(Object o)
To perform content comp. where is case is imp.

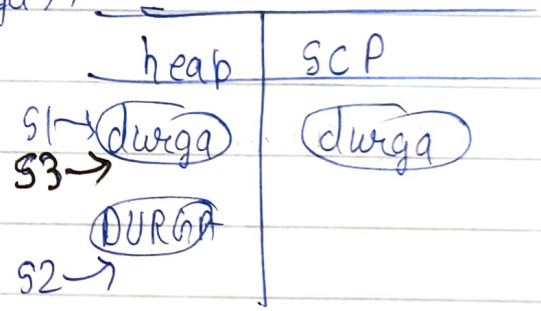
> public bool equalsIgnoreCase(String s)
Case is not important.

> public String subString(int begin);
String s = "abcdefg";
Sop ln(s.subString(3)); defg
Sop ln(s.substring(2,6)); cdef

> public String subString(int begin, int end)
Till end-1

> public int length()
> public String replace(char oldch, char newch)
> public String toLowerCase();
> public String toUpperCase();
> public String trim(); => To remove blank space.
> public int indexOf(char ch)
> public int lastIndexOf(char ch)

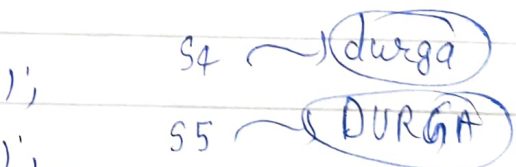
```
String s1 = new String("durga");
String s2 = s1.toUpperCase();
String s3 = s1.tolowercase();
Sopln(s1==s2); false
Sopln(s1==s3); true
```



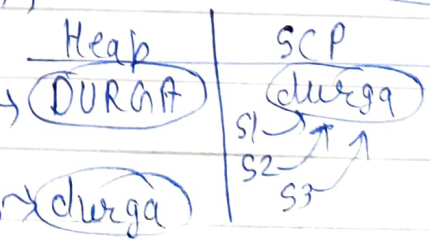
*** Note:- Because of Runtime operation if there is chang in content then with those changes a new object will be created on the heap.

→ If there is no change in content then existing object will be re-used & new object won't be created.

```
String s4 = s2.toLowerCase();
String s5 = s4.toUpperCase();
```



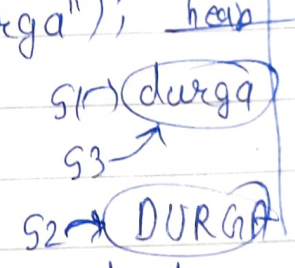
→ String s1 = "durga";
String s2 = s1.toString(); s1==s2; true
String s3 = s1.toLowerCase();
String s4 = s1.toUpperCase();
String s5 = s4.toLowerCase();



* How to create our own immutable class?

→ Once we create an object we can't perform any changes in that object, if we are trying to perform any change & if there is a change in content then with those changes a new object will be created. If there is no change in content then existing object will be re-used. This beaviour is immutability.

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

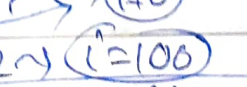


→ We can create our own immutable class.

final

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


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



Sopln (t1==t2) F, (t1==t3) true.

→ Once we create Test object we can't perform any chang in the existing object. If we trying to perform any change & if there is any change in the content then with those changes a new object will be created. If no change in content then existing object will be used.

* final Vs immutability

→ final applicable for variables but not for objects. whereas immutability applicable for objects but not for variables.

→ By declaring a ref. var. as final we won't get any immutability nature eventhough ref. var. is final we can perform any type of changes in the corresponding object. but we can't perform re-assignment for the variable.

hence final & immutable both are different concepts.

> final StringBuffer sb = new SB("durga");
sb.append("Software");
sopln(sb); durgaSoftware
sb = new SB("Solutions");
↳ CE;

sb → durgaSoftware

#111 StringBuffer

07/May/23

* StringBuffer :-

→ If content is fixed & won't chang frequently then String is recommended.

→ If content is not ~~not~~ fixed and keep on changing then ~~Str~~ It is not recommended to use string bcz every change a new object will be created which effects performance.

To handle this requirement we should go to string buffer.

→ The main adv. of SB is the all required changes will be performed on string only.

* Constructors

① StringBuffer sb = new SB();

→ 16

new capacity = (curr. cap. +1) * 2

② StringBuffer sb = new SB(int init.cap);

③ StringBuffer sb = new SB(String s);

eg: StringB sb = new StringB("durga");

Sopln (sb.capacity()); 21

capacity = s.length() + 16

* Methods

> length()

> Capacity()

> charAt(int Index)

> void setCharAt(int index, char ch);

for replace char at specific index.

> StringBuffer **append(String s)** → overloaded method
↳ int, long, char, bool, ---

> StringBuffer sb = new SB();
sb.append("Pi value :");
sb.append(3.14);
sb.append("It is exactly :");
sb.append(true);
Sop(sb); // Pi value : 3.14 It is exactly : true

⇒ StringBuffer **insert(int index, String s)** → overloaded.
↳ int, double, char, bool
↳ To insert at specific index.

> SB sb = new SB("abcde");
sb.insert(2, "xyz");
Sop(sb); // abxyzcde

> StringBuffer **delete(int begin, int end)**;
from begin to end-1

> StringBuffer **deleteCharAt(int index)**
↳ delete char at specific index.

> StringBuffer **reverse()**

> SB sb = new SB("durga");
Sop(sb.reverse()); // agrud

> void **setLength(int length)**;

SB sb = new SB("aishwarya abhi");
sb.setLength(8);
Sop(sb); // aishwarya.

> void ensureCapacity(int capacity) → To inc. capacity on fly.

```
SB sb = new SB();
Sopln(sb.capacity()); 16
sb.ensureCapacity(1000);
Sopln(sb.capacity); 1000
```

> void trimToSize();

```
SB sb = new SB(1000);
sb.append("abc");
sb.trimToSize();
Sopln(sb.Capacity()); 3
```

* StringBuilder

→ Every method present in StringBuffer is synchronized & hence it may create perf. problem to handle this requirement SUN introduced **StringBuilder** concept in 1.5 V.

→ StringBuilder is exactly same as StringBuffer except foll. diff.

SBuffer	SBuilder
→ methods are Sync.	→ methods are non-Sync
→ Thread safe	→ Not thread safe.
→ Waiting time ↑ perf. ↓	→ waiting time ↓ performance ↑
→ Came in 1.0 V	→ Came in 1.5 V

Buffer → Builder
sync → Remove

StringBuffer.Java
StringBuilder

* String Vs StringBuffer Vs StringBuilder.

→ If content is fixed & won't chang frequently then we should go to String.

→ If content is not fixed & keep on changing but thread safety required then go for StringBuffer

→ If content is not fixed & keep on changing but thread safety is not required then go for StringBuilder.

* Method chaining

```
> SB sb = new SB();
> sb.append("durga").append("Soft").append("Sal").insert(2,"xyz")
    .reverse().delete(2,10);
> Sop(sb);
```

→ For most of the method in String, SBuffer, & SBuilder return types are same type hence after applying a method on result we can call another method which forms method chaining.

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

→ In method chaining method call will be executed from left to write.

#112 String, SBuffer & SBuilder. 7/May/23

#113 Wrapper class

08|May|23

* Wrapper class

→ The main objective of wrapper classes are
→ To wrap primitive into object form so that we can handle it like object.
→ To define several utility method which are required for the primitives.

* Constructors :-

① Integer I = new Integer(10); → int primitive
② Integer I = new Integer("10"); → String
→ If str not a number then RE: NFE

→ Float class contains 3 const. with float, double & Str.

```
Float f = new Float(10.5f);   ✓
Float f = new Float("10.5f"); ✓
Float f = new Float(10.5);    ✓
Float f = new Float("10.5");  ✓
```

Wrapper	corresponding const. arguments.
Byte	byte or String
Short	short or String
Integer	int or String
Long	long or String
* Float	float or String @ double
Double	double @ String
* Character	char or ~~String~~

→ char class contain only one char const.

Boolean	boolean or String.

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

> Boolean x = new Boolean("yes"); → false left
Boolean y = new Boolean("no"); → false right
Sop(x.equals(y)); true

→ If we are passing string type of arg. then case & content both are not important. If content is case insensitive string of "true" then it is treated as true. otherwise it is treated as false.

> B b = new B("true") → true
> B b = new B("True") → true
> B b = new B("TRUE") → true
> B b = new B("malaika") → false
> B b = new B("mallika") → false

→ In all wrapper classes ~~to~~ toString() method is overriden to return content directly.
→ .equals() method is overriden for content comparision

* methods
① valueOf()
② xxxValue()
③ parseXxx()
④ toString()

* valueOf()

→ We can use valueOf method to create wrapper object for the given premitive or string.

Form 1:

→ Every wrapper class except char. class contains a static value of method to create wrapper obj. for the given string.

b	static	wrapper	valueOf(String s)

eg:

```
Integer I = Integer.valueOf("10");
D d = D.valueOf("10.5");
B b = B.valueOf("durga");
```

Form 2:

Every integral type wrapper class(B,S,I,L) contains the following values of method to create wrapper object for given specified radix string.

b	s	wrapper	valueOf(Str s, int radix);

(radix : 2 to 36)

```
I i = I.valueOf("100", 2);
Sop(I) // 4
I i = I.valueOf("101", 4);
Sop(I) // 7
```

Form 3:

Every wrapper class including char.(C) contains a stdic valueOf() method to create wrapper object for the given premitive.

p	s	wrapper	valueOf(primitive p);

valueOf()

Primitive / String → wrapper object

* xxxValue()

wrapper obj → primitive

→ we can use it to get primitive for the given wrapper object.

> I i = new I(130);
> Sopln(i.byteValue()); -126
> Sopln(i.shortValue()); 130
> Sopln(i.intValue()); 130
> Sopln(i.longValue()); 130
> Sopln(i.floatValue()); 130.0f
> Sopln(i.doubleValue()); 130.0

→ Every number type wrapper class (I, B, S, L, F, D) contains following 6 methods. to get primitive m/c.
b, s, i, l, f, d.

* charValue()

> character class contains charValue() to get char primitive for the given char. wrapper object.
> p char charValue()
> booleanValue()

→ boolean (C) contain booleanValue() method to get boolean primitive for given bool. object.

> p bool booleanValue()

→ 38 (6x6+1+1) xxxValue() possible.

#114 Wrapper class 11 Utility methods

08/May/23

* parseXxx()

→ We use parseXxx() method to convert string to primitive.

Form - 1

> p static primitive parseXxx(String s);

→ Every wrapper class except char. class contains the following parseXxx() to find primitive for the given String object:

```
int i = Integer.parseInt("10");
Double d = Double.parseDouble("10.5");
```

Form - 2

→ Every integral type wrapper class (B,S,I,L) contains the following parseXxx() method. to convert specified radix string to primitive.

> p s primitive parseXxx(Str s, int radix);

radix : 2 to 36.

eg. `int i = Integer.parseInt("1111", 2);`
`Sop(i);` 15

* toString()

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

Form - I

public String toString();

→ Every wrapper class contains toString method to convert wrapper object to string type.
→ It is overriding version of object class toString() method

> Integer I = new I(10);
  String S = I.toString();
  Sop ln(S); 10 ✓

Form-2

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

> p. static String toString(primitive p)

eg String S = Integer.toString(10);

Form-3

> p static static toString(pri. p, int radix) → 2 to 36

→ Integer & Long classes contains this method to convert primitive to specified radix string.

eg. String S = Integer.toString(15,2);
  Sop ln(S); 1111

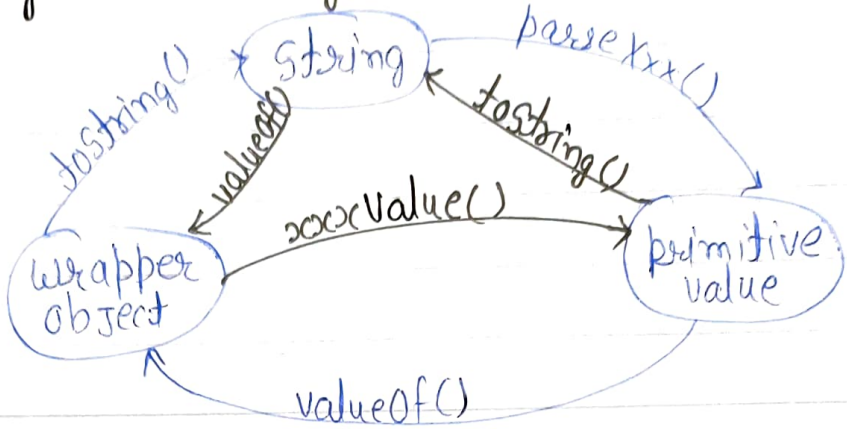
Form-4 toXxx String()

→ Integer & long classes only.

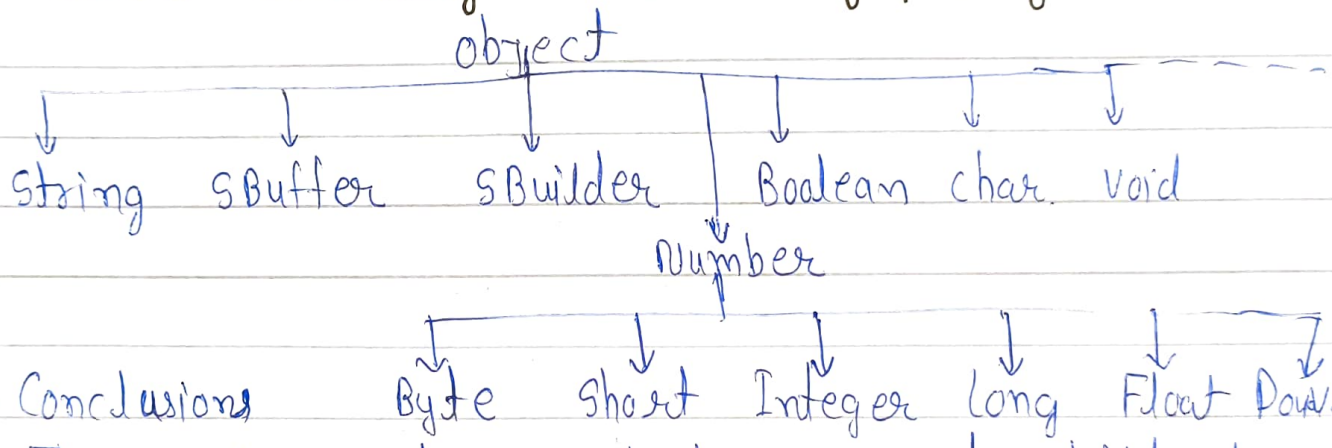
> p S String toBinaryString(pri p)
> p S String toOctalString(pri p)
> p S String toHexString(pri p)

Wrapper Prd. → toString() → String

* Dancing b/w String, wrapper obj. & primitive.



* Partial hierarchy of java.lang.package.



* Conclusions
→ The wrapper classes which are not child class of ~~number~~ Number are (Bool, char) ~~are~~
→ The wrapper class not direct child of obj. {B,S,I,L,F,D}
→ All wrapper classes are final classes.

* Void class.
→ It is final & direct child of object.
→ It doesn't contain any method & contain only one var {Void TYPE}
> if (getMethod("m1").getReturnType() == Void.TYPE) { }

#115 Autoboxing & Autounboxing

08/May/23

* Autoboxing (1.5V)

→ Integer I = 10; [compiler converts int to Integer automatically by Autoboxing]

→ Automatic conversion of primitive to wrapper object by compailer is called autoboxing.

[Integer I = Integer.valueOf(10);] after compilation

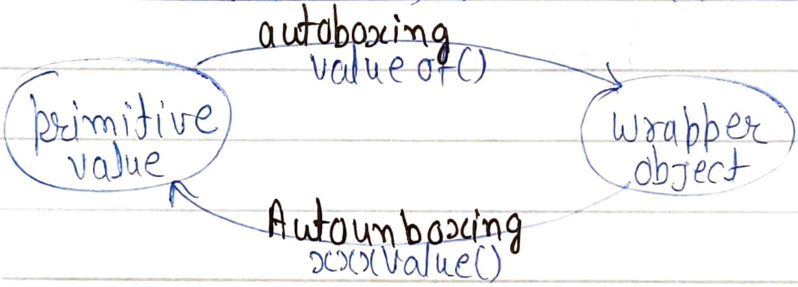
* Autounboxing

→ Automatic conversion of wrapper object to primitive by compiler is called auto-unboxing.

> Integer I = new Integer(10);
> int i = I; [compiler converts Integer to int automatically by autoUnboxing.

⇓ After compilation

[int i = I.intValue();] [xxxValue()]methods.




```
> class Test{
      static Integer I=10;          --- ① AB
      p s v m(S[] a){
          int i=I;                  -- ② A.U.B
          m1(i);     }  → ③ A.B
      p s v m1(Integer k){
          int m=k; →              ④ A.U.B
          Sopln(m);
      }
  }
```

eg-2

```
class Test {
    static Integer I = 10;
    p s v m(S[] a){
        int m = I;
        Sop ln(m);
    }
}
```

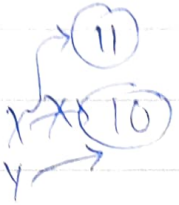
O

```
class Test {
    static Integer I;
    p s v m(S[] a){
        int m = I;
        Sop ln(m);
    }
}
```

RE: NPE

eg-3

```
Integer x = 10;
Integer y = x;
x++;
Sop ln(x);
Sop ln(y);
Sop ln(x==y);
```



11
10
false

→ All wrapper class is immutable i.e. once we create wrapper class object we can't perform any changes in that object. If we try to perform any changes with those changes a new object will be created.

eg-4

① `Integer x = new Integer(10);`
`I y = new I(10);`
`Sop ln(x==y);` false

② `I x = new I(10);` x → 10
`I y = 10;` y → 10
`Sop ln(x==y);` false

③ `I x = 10;`
`I y = 10;`
`Sop ln(x==y);` true

④ `I x = 100;`
`I y = 100;`
`Sop ln(x==y);` true

⑤ `I x = 1000;`
`I y = 1000;`
`Sop ln(x==y);`
false.
x → 1000
y → 1000

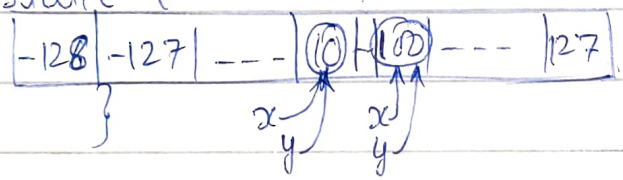
* Conclusion

→ To internally to provide support for ~~abo~~ autoboxing a buffer of wrapper object will be created at the time of wrapper class loading. ~~bu~~

→ By autoboxing if an object is required to create first JVM will check wether this object already present in buffer or not. If it is already present in buffer then exesting buffer object will be used. If it is not already available in the buffer then JVM will create a new Object

Class Integer {

Static {



}

}

> I x = 127;	> I x = 128
I y = 127;	I y = 128
Sopln(x==y); true	Sopln(x==y) false
> B x = false;	> D x = 10.0;
B y = false;	> D y = 10.0;
Sopln(x==y); true	Sopln(x==y); false.

Byte → Always.
Short → -128 to 128
Integer → -128 to 127
long → -128 to 127
Char → 0 to 127
Bool → Always.

#116 Autoboxing

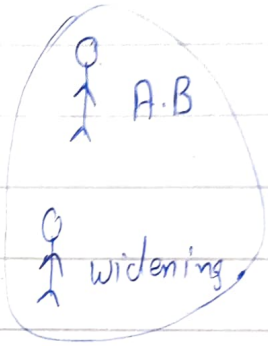
08/May/23

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

Case-I Autoboxing Vs Widening

```
class Test {
    p s v m1(I i) {
        Sop.println("Auto Boxing"); }
    p s v m1(L i) {
        Sop.println("widening"); }
    p s v m(S[]a) {
        int x=10;
        m1(x);
    }
}
```

overloaded method



o/p: Widening

→ Widening domenates autoBoxing.

Case-II Widening Vs var-arg method.

```
class Test {
    p s v m1(int... x) {
        Sop.println("var-arg"); }
    p s v m1(long l) {
        Sop.println("widening"); }
    p s v m(S[]a){
        int x=10;
        m1(x); }
}
```

o/p: widening

→ Widening domenates var-arg methods.

Case-III Autoboxing Vs var-arg method

```
class Test {
    p s v m1(int... x) {
        Soln("var-arg"); }
    p s v m1(long x) {
        Soln("Auto Boxing"); }
    p s v m(S[] a) {
        int x=10;
        m1(x)
    }
}
```

o/p: Auto Boxing

→ Autoboxing dominates var-arg method.

→ In General var-arg method will get least priority.

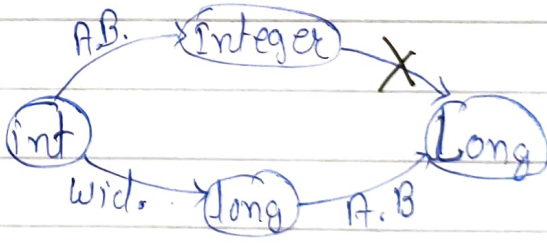
Widening > Autoboxing > var-arg.

while resolving overloaded method.

Case-IV

```
class Test {
    p s v m1(L l) {
        Soln("Long"); }
    p s v m1(S[] ar) {
        int x=10;
        m1(x);
    }
}
```

CE:



W → A ⇒ ✗

A → W ⇒ ✓

→ Widening followed by AutoBoxing is not allowed.
but Autoboxing followed by widening is allowed.

Long l = 10; CE:

Long l = 10; ✓

Case-IV

```
class Test {
    psv m1(Object o) {
        Sop("obj.ver"); }
    psv m(S[]a){
        int x=10;
        m1(x)      } o/p. obj.ver
}
```

A → W ✓

Object o = 10; ✓

Number n = 10; ✓

Q. which are legal.

int i = 10; ✓

Integer I = 10; ✓ (A.B)

int i = 10L; ✗ CE: PLP

Long l = 10L; ✓

Long l = 10; ✗ CE:

long l = 10; ✓ (Widening)

Object o = 10; ✓ (A.B. → W)

double d = 10; ✓ (Widening)

Double D = 10; CE:

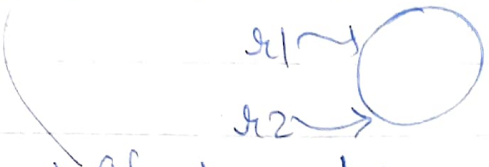
Number n = 10; ✓ (A.B → W)

#117. equals() || Hashcode()

08|May|23

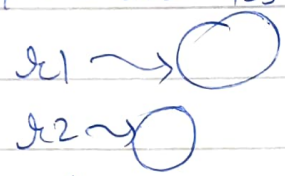
* Relation b|w '==' operator & equals() method.

① If r1 == r2 is true then r1.equals(r2) is always tr



→ If two obj are equal by double "==" operator then these objs. are always equal by .equal().

② If r1 == r2 is false then r1.equals(r2) ?



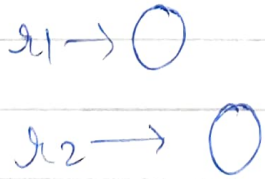
→ If two obj. are not equal by "==" operator then we can't conclude anything about .equals() method

③ If r1.equals(r2) is true then r1 == r2 ?

→ If two obj. are equal by .equal() method then we can't conclude anything about "==" operator.

④ If r1.equals(r2) is false then r1 == r2 is always

→ If two obj. are not equal by .equal() method fals e. then these obj. are always not equal "==" oper



* Diff. blw '==' operator & equals() method.

→ To use '==' operator, compulsary there should be some relation blw argument types (C to P, PtoC, or same type) otherwise we will get CE:

→ If there is no relation blw argument type .equal() method don't raise any CE or RE. Simply it returns false.

```
> String s1 = new S("durga");
  S s2 = new S("durga");
  SB sb1 = new SB("durga");
  SB sb2 = new SB("durga");
  Sop(s1==s2); false
  Sop(s1.equals(s2)); true
  Sop(sb1==sb2); false
  Sop(sb1.equals(sb2)); false
  Sop(s1==sb1); CE
  Sop(s1.equals(sb1)); false
```

== operator	equals() method
① It is operator applicable for both primitive & object.	① It is method applicable only for obj. types
② It uses reference comparision	② default reference comp.
③ Can't override	③ can override for content comparision.
④ There should be relation blw argument type. (P→C, C→P, or same type) CE:	④ no relation blw arg. type then return false.

⇒ Ans. in one line :- In general we use '==' operator for reference comp. & .equals() for content comparision.

r == null
r.equals(null) ⇒ always false

**Note:- Hashing related DS follow the following fundamental rule.

→ Two equivalent objects should be placed in same bucket but all object present in same bucked need not be equal.

* Contract b/w .equals() & hashCode()

→ If two objects are equal by .equals() method then there hashcode must be equal. i.e. two equivalent objects should ~~be placed~~ have same hashCode.

① If r1.equals(r2) is true then r1.hashCode() == r2.hashCode() is always true.

→ Object class .equals() & hashCode() follows above contract hence whenever we are overriding .equals() method compulsary we should override hashCode() method to satisfy above contract.

→ 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.

→ If hashCodes of two objects are not equal then equals these objects are always not equal by .equals() method.

** Note :- To satisfy contract blw .equals() & hashCode() methods whenever we are overriding .equals() method compulsary we have to override hashCode method.

→ In string class .equals() method is overriden for content comparision & hence hashcode method is also overriden. to generate hashCode based on content

> S s1 = new S("durga");
> S s2 = new S("durga");
> Sop(ln(s1.equals(s2)); true
> Sop(ln(s1.hashCode()); 95950491
> Sop(ln(s2.hashCode()); 95950491

> SB sb1 = new SB("durga");
> SB sb2 = new SB("durga");
Sop(ln(sb1.equals(sb2)); false
Sop(ln(sb1.hashCode()); 19621457
Sop(ln(sb2.hashCode()); 4872882

→ In SB .equals method is not overriden for content comparision hence hascode() is also not overriden.

> class Person {

```
    p bool equals(obj. obj){
      if (obj instance Person){
        Person p = (Person) obj;
        if (name.equals(p.name) && age == p.ag){
          return true; }
        } else {
          return false;
        }
      } }
    return false;
  }
```

✗ ① p int hashCode () { return 100; }

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

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

✗ ④ no Restrictions.

→ Based on which paremeters we override .equals() method, it is highly recommende to use same same paremater while overriding hashCode() method also.

→ In all C.C & Wrapper C in Str. equals() is overriden for content comp. So, it is rec. to equals() method also. for content comp.

#118 clone()

9|May|23

* clone()

> protected native Object clone() throws cloneNotSupportedEx.

→ The process of creating exactly duplicate obj is called cloning.

→ The main objective of cloning is to maintain backup copy & to preserve state of an object.

> class Test implements Cloneable ③ RE

```
int i=10;
int J=20;
p s v m(S[] a) throws CNSE ② CE
  Test t1 = new Test();
  Test t2=(Test)t1.clone(); ① CE:
  t2.i=888;
  t2.J=999;
  Sop(t1.i+---+t1.J); #10---20
}
```

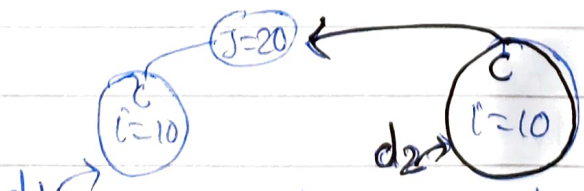
t1 → (i=10, J=20)

t2 → (i=10 888, J=20 999)

→ We can perform cloning only for clonable objects.

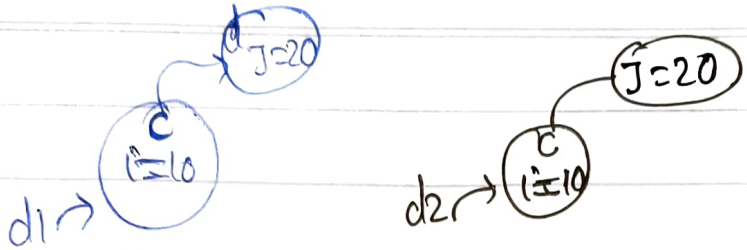
→ It present in java.lang package & it is a marker interface. (doesn't contain any method)

* Shallow cloning Vs Deep cloning



Shallow cloning

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



Deep cloning

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

* Shallow cloning :-

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

→ If the main object contain primitive variables then exactly duplicate copy will be created in the cloned object.

→ If the main object contain any reference variable the corresponding object won't be created just duplicate ref. var. will be created pointing to old content object.

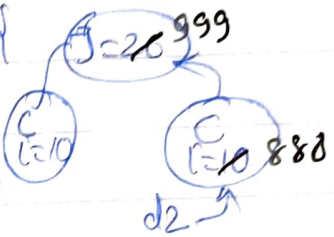
→ Object class clone() method ment for shallow clon

> class Cat {
int J;
Cat (int J) {
this.J = J }
}

> class Dog imp. Clonable {
Cat C ;
int i ;
Dog (Cat c, int i) {
this.c = c ;
this.i = i ;
}

protected Object clone() thr. CNSE
{
return super.clone();
}

```
class ShallowCloneDemo {
    p s v m (S[]'a) throws CNSE {
        Cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
        Dog d2 = (Dog) d1.clone();
        d2.i = 888;
        d2.c.j = 999;
        Sop(d1.i + --- + d1.c.j); // 10 -- 999
    }
}
```



→ In shallow cloning by using cloned obj. ref if we perform any change to contained obj. then those changes will be reflected to the main obj.

To overcome this we should go for deep cl.

* Deep cloning

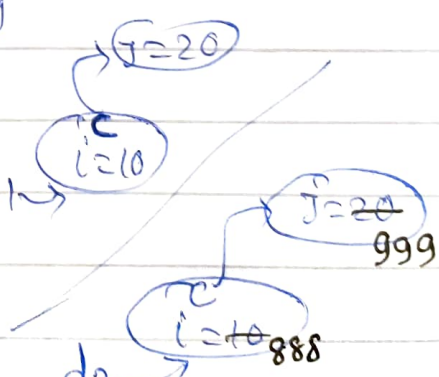
→ The process of creating exactly duplicate independent copy including contained object is called deep cloning.

→ In deep cloning if the main obj. contain any primitive variables then in the cloned obj. duplicate copies will be created.

→ If the main object contain any reference var. then the corresponding contained object also will be created in the cloned copy.

→ By default object class clone method ment for shallow cloning but we can implement deep cloning explicitly by overriding clone() method in our class.

```
> class Cat { }
      int J;
      Cat (int J) {
          this.J = J; }
  }
  class Dog imp. cloneable {
      cat c;
      int i;
      Dog (cat c, int i) {
        this.c = c;
        this.i = i;     }
      p. object clone() throws CNSE{
        Cat C1 = new Cat(c.J);
        Dog d2 = new Dog (C1, i);
        return d2;                    }
  }
> class DeepClonDemo {
      p s v m (S[]a) th. CNSE{
        cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
        Sopln(d1.i+--+d1.J); 10--20
      Dog d2 = (Dog) d1.clone();
        d2.i = 888;
        d2.c.J = 999;
      Sopln( d1.i+---+d1.c.J);   10 --- 20
```



#119 String class

09/May/23

→ By using cloned object reference, if we perform any change to the contained object then those changes wont be reflected to the main object.

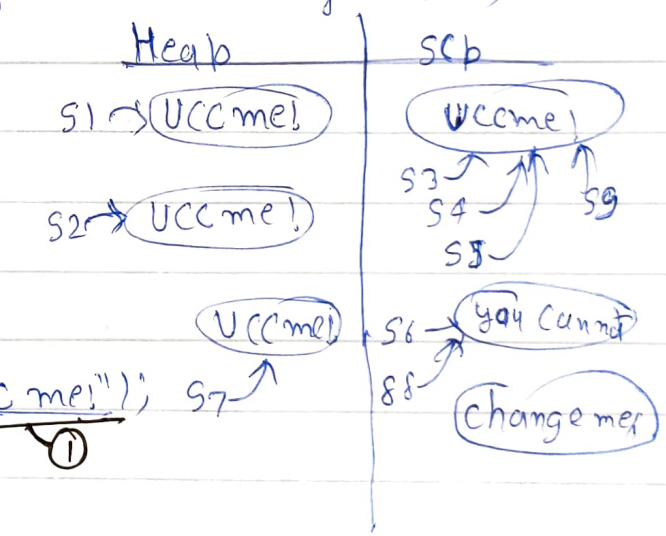
* which cloning is best

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

→ If object contains reference variables then deep cloning is best choice.

* String class

```
> String s1 = new S("you cannot chang me!");
> String s2 = new S("you cannot chang me!");
  Sop(s1==s2); false
  S s3 = "you cannot chang me!"
  Sop(s1==s3); false
  S s4 = "you cannot chang me!";
  Sop(s3==s4); true
  String s5 = "you cannot"+"change me!"; ①
  Sop(s3==s4);
  S s6 = "you cannot";
  S s7 = s6+"change me!"; ②
  Sop(s3==s7); false
  final String s8 = "you cannot";
  St. s9 = s8+"change me!"; → ③
  Sop(s3==s9); true.
  Sop(s6==s8); true.
```



→ at line ① This operation will be performed at compile-time only bcz both arguments are compile-time constants.

→ at line ② This operation will be performed at Runtime only bcz atleast one argument is normal variable.

→ at line ③ This operation will be performed at compile-time only bcz both arg. are compile-time constant.

* Interning of String object.

```
> S S1 = new S("durga");
> S S2 = S1.intern();
  Sopln(S1==S2); false.
  S S3 = "durga";
  Sopln(S2==S3); true
```

heap: S1 → (durga)

scp: (durga) ← S2, S3

→ We can use intern() method to get corresponding scp object reference by using heap object reference.

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

```
> S S1 = new S("durga");
> S S2 = S1.concat("Soft");
  S S3 = S2.intern();
  Sopln(S2==S3);
  S S4 = "durgaSoft";
```

S1 → (durga)

(durgaSoft) ← S2

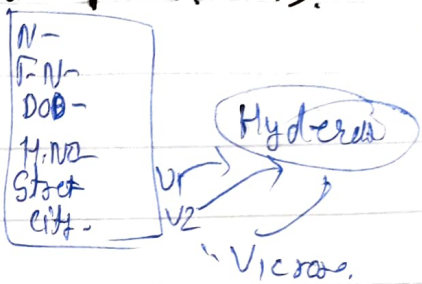
(S3==S4)

(durga) (Soft) (durgaSoft) ← S3, S4

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

* Importance of String constant pool (SCP).

→ In our program if a string is repeadly required then it is not recommended to create seperate object for ever requirement bcz it creates performance & memory problems.



→ Instead of creating seperate object for every requirement we have to create only one object & we can re-use the same object for every requirement so that performance & memory utilization will be improved.

→ This this is possible bcz of SCP. (1:08)

→ The main problem with SCP is, As several reference pointing to the same object, by using one reference if we are trying to change the content then remaining references will be affected to overcome this problem SUN implemented String objects as immutable i.e once we create string object we can't perform any changes in existing object. If we try to perform any changes with those changes a new object will be created.

* FAQ

1. What is diff. blw String & StringBuffer?
2. Explain about Immutability & mutability with an example?
3. What is diff. blw
   String s = new String("durga"); &
   String s = "durga";
4. Other than immutability & mutability is any other diff. blw String & StringBuffer?
5. What is SCP?
6. What is advantag of SCP?
7. What is disadvantage of SCP?
8. Why SCP like concept is available only for String but not for StringBuffer?
9. Why String objects are immutable where as StringBuffer objects are mutable?
10. In addition to String objects any other objects are immutable in java? => All wrapper (@) object
11. It is possible to create our own immutable class?
12. How to create our own immutable class explain with example.?
13. Immutable means non-changable where as final means also non-changable. Then what is the difference blw final & immutable?

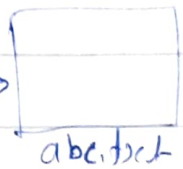
#120 File I/O

10-May-23.

1. File
2. FileWriter
3. FileReader
4. BufferedWriter
5. BufferedReader
6. PrintWriter

* File

① `File f = new File("abc.txt");`



`sop(f.exists());` false

② `f.createNewFile();`

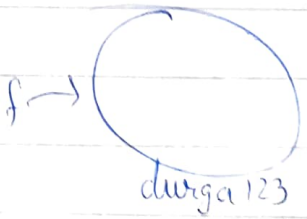
`sop(f.exists());` true.

	1st run	2nd run
	false	true
	true	true

→ Line ① won't create any physical file first it will check is there any physical file named with abc.txt is available or not is it is available then f simply refers that file. If it is not available then we are just creating java file obj. to represent abc.txt.

→ We can use java file object to represent directory also.

```
File f = new File("durga123");
sop(f.exists()); false.
f.mkdir();
sop(f.exists()); true
```

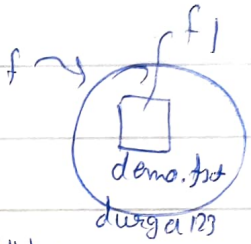


Note In unix every thing is treated as file java file I.O concept implemented based on unix operating system.

* File class Constructors :

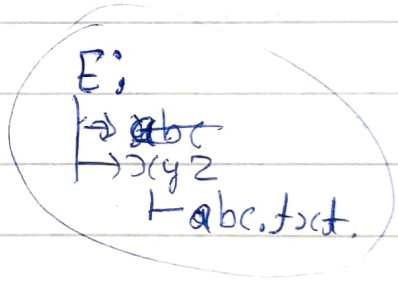
① File f = new File (String name);
② File f = new File (Str subdirname, Str name);
③ File f = new File (File subdir, Str name);

eg. File f = new File("abc.txt");
f.createNewFile();



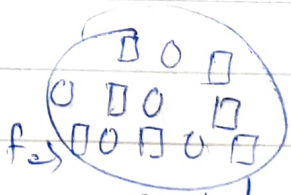
> F f = new F ("durga123")
f.mkdir();
F f1 = new F ("durga123", "demo.txt");
F f1 = new F(f, "demo.txt");
f1.createNewFile();

> F f = new F("E:\\xyz", "abc.txt");



* Methods

> bool exists();
> bool createNewFile(); → first check then create.
> bool mkdir();
> bool isFile();
> bool isDirectory();
> String [] list()
> long length()
> bool delete();




```
int count=0;
F f = new F("C:\\durga_dar");
String [] s = f.list();
for(String s: s1) {
    count = count + 1;
    sout(s1);
}
sout(count);
}
}
```

```
F f1 = new F(f,s1);
if(f1.isFile()) {
}
```

#121 FileWriter

10/May/23

* FileWriter

→ We can use FileWriter to write, char data to file.

* Constructors

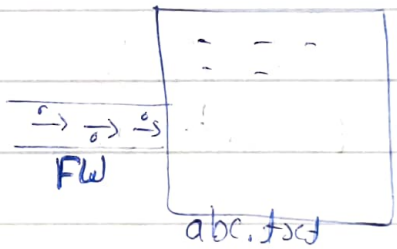
overwrite
1. FileWriter fw = new FileWriter(Str fname);
2. Filewriter fw = new FileWriter(File f);

append
3. Filewriter fw = new FileWriter(Str. fname, bool append);
4. Filewriter fw = new FileWriter(File f, boal append);

→ If specified file is not already available then all the above cons. will create that file.

* methods.

① write(int ch)
② write(char[] ch)
③ write(String s)
④ flush()
⑤ close()




```
> class FileWRDemo {
    p s v m(S[] a) throws IOE {
    FW fw = new FW("abc.txt", true);   ↳ To append data.
    fw.write(100);
    fw.write("Durga \n Soft Sol");
    fw.write('\n');
    char[] ch1 = {'a','b','c'};
    fw.write(ch);
    fw.write('\n');
    fw.flush();
    fw.close();
  }
}
```

#122 FileReader || Important method BufferedWriter

10/May/23

→ The main problem with file writer is we have to insert lines operator(\n) manually which is varied from system to system.
→ We can solve it by using BufferedWriter & printWriter classes.

* FileReader

→ We use FileReader to read char

* Constructors :-

1. FileReader fr = new FileReader(String fname);
2. FileReader fr = new FileReader(File f);

* Methods :~

1. int read()
   ↳ Unicode value. {if not then -1}

```
FR fr = new FR("abc.txt")
int i = fr.read();
while (i != -1) {
    sopln((char) i);
    i = fr.read();
}
```

2. int read(char[] ch)
   ↳ ~~attempts~~ to read enough char from file into char arr & returns no. of chars copied from the file.

```
char[] ch = new char[10];
FR fr = new FR("abc.txt");
fr.read(ch);
```

```
File f = new F("abc.txt")
char[] ch = new char[(int) f.length()];
FR fr = new FR(f);
fr.read(ch);
for (char chi:ch){
    sopln(chi);
}
```

③ void close()

Note:- By using FileReader we can read data character by char. which is not convinent to program.

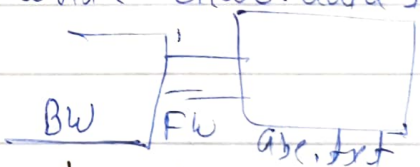
✱ Uses of FileReader & FileWriter is not recommended.

To overcome problems we should go for

④ BufferedWriter & BufferedWriter.

✱ BufferedWriter

→ We can use BufferedWriter to write char. data to the file.



✱ Constructor

① BufferedWriter bw = new BW(Writer w);

② BufferedWriter bw = new BW(Writer w, int bufferSize);

Note:- BW can't communicate directly with the file it can communicate via some writer object.

✱ Methods

1. write(int ch)
2. write(char[]ch)
3. write(String s)
4. flush()
5. close()
6. newLine()
   ↳ To insert a line separator.

* BufferedReader

→ We can use BR to read char data from the file.

→ The main adv. of BR when compared with FR is we can read data line by line in addition to char by char.

* Constructors

① BufferedReader br = new BR(Reader r);

② BufferedReader br = new BR(Reader r, int buffersize);

Note:- BR can't communicate directly with the file & it can communicate via some Reader object.

* Methods

1. int read()
2. int read(char[] ch)
3. void close()
4. String readLine()
   ↳ If attempts to read next line from the file & returns it. If next line not available then this method returns null.

```
> FR fr = new FR("abc.txt");
  BR br = new BR(fr);
  String line = br.readLine();
  while (line != null) {
      sop(line);
      line = br.readLine()
  }
  br.close();
```

After closing BR underlying reader will also be closed.

#123 PrintWriter

10|May|23

* PrintWriter

→ It is the most inhanced writer to write char. data to the file.

→ The main adv. of printWriter over BW & FW is we can write any type of primitive data directly to the file.

* Constructer

① PrintWriter pw = new PrintWriter(String fname);

② PrintWriter pw = new PrintWriter(File f);

③ PrintWriter pw = new PrintWriter(Writer w);

Note:- PW can communicate directly with the file & can communicate vaia some writer object also.

* Methods

① write(inh ch);

② write(char[] ch);

③ write(String s);

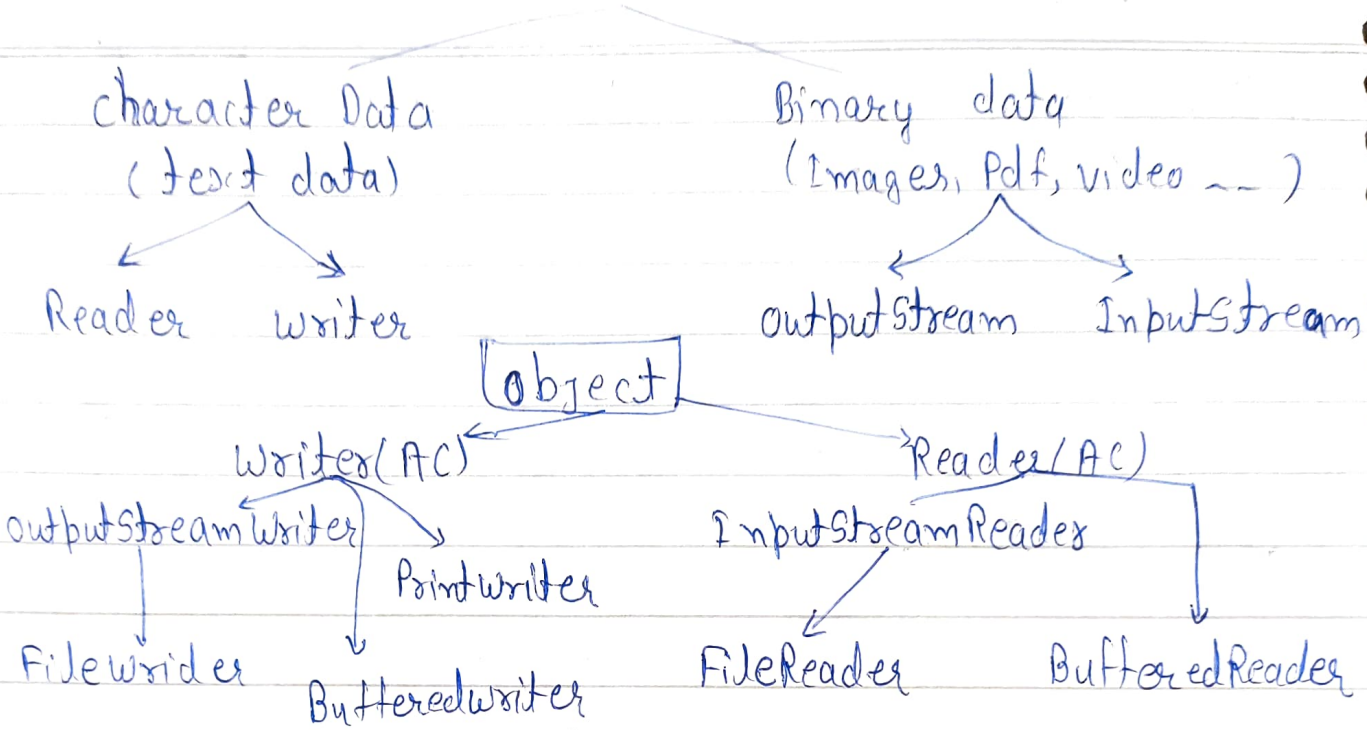
④ flush()

⑤ close()

⑥ print(char ch)
↳ ch int, double, bool, String---

⑦ println(char ch)
↳ int, double, boolean, String---

what is diff. writ(100) & print(100)
'd' ↙ ↳ 100.



Q WAP to merge data from 2 file to a 3rd file.

```
> PW pw = new PW("file3.txt");
  BR br = new BR(new FR("file1.txt"));
  String line = br.readLine();
  while (line != null) {
      pw.println(line);
      line = br.readLine();
  }

  br = new BR(new FR("file2.txt"));
  line = br.readLine();
  while (line != null) {
      pw.println(line);
      line = br.readLine();
  }

  pw.flush();
  br.close()
  pr.close()
```

#124 PrintWriter - Excamples

10-May-23

Q. WAP to merg two fides where merging should be done line by line alternatively.

A:-

Q:- WAP to perform file extraction programme

Output = Input - delete

Soln:-

```
PW pw = new PW("Output.txt")
BR br = new BR(new FR("input.txt"))
String line = br.readline();
while (line != null) {
    BR br2 = new BR(new FR("delete.txt"));
    str target = br1.readline();
    while(target != null) {
        if (line.equals(target)) {
            available = true;
            break;
        }
        target = br1.readline();
    }
    if (available == false) {
        pw.println(line); }
    line = br.readline();
}
```

input.txt

222
333
444
555
666
777
888

delete.txt

555
888
222

output.txt

333
444
[illegible]
777
999

* WAP. to remove duplicates from given file.*

Ans:-