S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

JAVA

Abstraction:

* The process of hiding the implementation details of method is called abstraction.

* We can achieve abstraction in two ways.

(i) Using Abstract class
(ii) Using Interface.

(1) Abstract Method:

* A method without implementation is called abstract method

Syntax: abstract Modifiers return_type Methodname(datatype Var-)

ex: abstract public void test();

* Abstract method header must be prefix with abstract keyw
* It should be end with ;

(2) Abstract Class:

* A class prefix with abstract keyword is called Abstract class

Note: A class has atleast one abstract method it must be declared as abstract class otherwise compiler generates CTE.

ex:
```
abstract class A
{
    abstract public void test();//abstract method.
}
```

③ Concrete method:

* A method with implementation is called Concrete method
* Abstract class can have both abstract as well as Concrete method

ex: abstract class B

```
{
    abstract public void test();// abstract method.
    public void test1() // Concrete method
    {
        s.o.pln("from test1()");
    }
}
```

Note: Abstract class can have only Concrete methods

ex: abstract class C

```
{
    public void test()
    {
        s.o.pln("from test()");
    }
}
```

When exactly we should make class abstract?

whenever class has atleast one defined or inherited abstract method it should declared as abstract.

① abstract class D

```
{
abstract public void m1();
}
```

② abstract E extends D

```
{

}
```

* In the above example class D is abstract because there is one defined abstract method.
* class E is abstract because it has inherited abstract method (from class D).

We Cannot Create object of abstract class:

```
abstract class F
{
 abstract public Void test();
}
```

```
class Driver1
{
 p.s.v.m()
 {
  F obj=new F();//CTE
 }
}
```

* We Can provide implementation for the abstract method by Creating Concrete Subclass as shown in the below example.

Concrete class:

* If all the methods of class that is defined as well as inherited methods (or) Concrete methods such classes are Consider as Concrete class.

ex1:
```
abstract class F
{
    abstract public Void test();
}

class G extends F //Concrete subclass for F
{
 public Void test()// providing implementation for super class abstract method.
 {
  S.o.pln("test() implementated");
 }
}

class Driver1
{
 p.s.v.m()
 {
    G obj1 = new G();
    obj1.test();// test() implemented
    F obj2 = obj1;//Upcasting
    obj2.test();// test() implemented
 }
}
```

ex. 2.

```
abstract Class H1
{
 abstract public Void a1();
}
```

→ H1 class abstract because it has defined abstract method.

```
abstract class H2 extends H1
{
  public void a2();
  {
    S.o.pln("from a2()");
  }
}
```

⇒ H2 class is abstract because it inherited abstract with it.

```
class H3 extends H2
{
  public Void a1()
  {
    S.o.pln("a1() implented");
  }
}
```

⇒ H3 provides implementation in abstract method of its Superclass & it is not having any other defined inherited (or) defined.

∴ It is Considered as Concrete Method & We Can't Create object of H3 class in order to use a2 & a1 methods.

```
class Driver2
{
  p.s.v m()
  {
    H3 obj1 = new H3();
    obj1.a1(); || a1() implemented
    obj1.a2(); || from a2();
    H2 obj2 = obj1; ||Upcasting H3 type ref to H2 type
    obj2.a2(); || from a2()
    obj2.a1(); || a1() implemented
    H1 obj3 = obj2; ||Upcasting H2 type ref to H1 type
    obj3.a2(); ||CTE
    obj3.a1(); || a1() implemented.
  }
}
```

Can we make a static method as abstract ?

Since static methods can't be over ~~written~~ riden. We can't declared them as abstract

ex:
```
abstract class I
{
    abstract public static void test();  // CTE
}
```

Interface :-

Interface is a component in java which acts as blue print of a class.

* We can create interface using interface keyword.

Syntax:
```
interface interface_name
{
    // members of interface
}
```

* Interface can have following members
  (i) Abstract Methods
  (ii) Static final Variables (from JDK 8 onwards)
  (iii) Static Concrete Methods (from JDK 8 onwards)

Note:- The abstract methods inside interface by default abstract & public ∴ We don't have to use abstract & public keywords

Before Compiling →
```
interface I1
{
    void test();
}
```

After Compiling
```
interface I1
{
    abstract public void test();
}
```

Note: All the static Concrete methods of an interface will be by default public.

Before Compile

```
interface I2
{
    static void test1()
    {
      s.o.pln("from test1()");
    }
}
```

After Compile

```
interface I2
{
  public static void test1()
  {
    s.o.pln("from test1()");
  }
}
```

Note: A Variable declared inside the interface are by default Considered as public, static & final Variables. (Final Variables should be initialized or gives CTE)

Before Compile

```
interface I3
{
  int a=10;
}
```

After Compile

```
interface I3
{
  public final static int a=10;
}
```

* Interface Can have
  (i) public, abstract, non-static Methods
  (ii) public, static, Concrete methods.
  (iii) public, final, static Variables

Can we Create object for Interface?

For Interface we Can't Create objects, becaus interface is by default abstract.

*

Inheritance b/w interfaces

We can achieve inheritances b/w interfaces by using extends keyword.

* Interface supports all the type of inheritance.

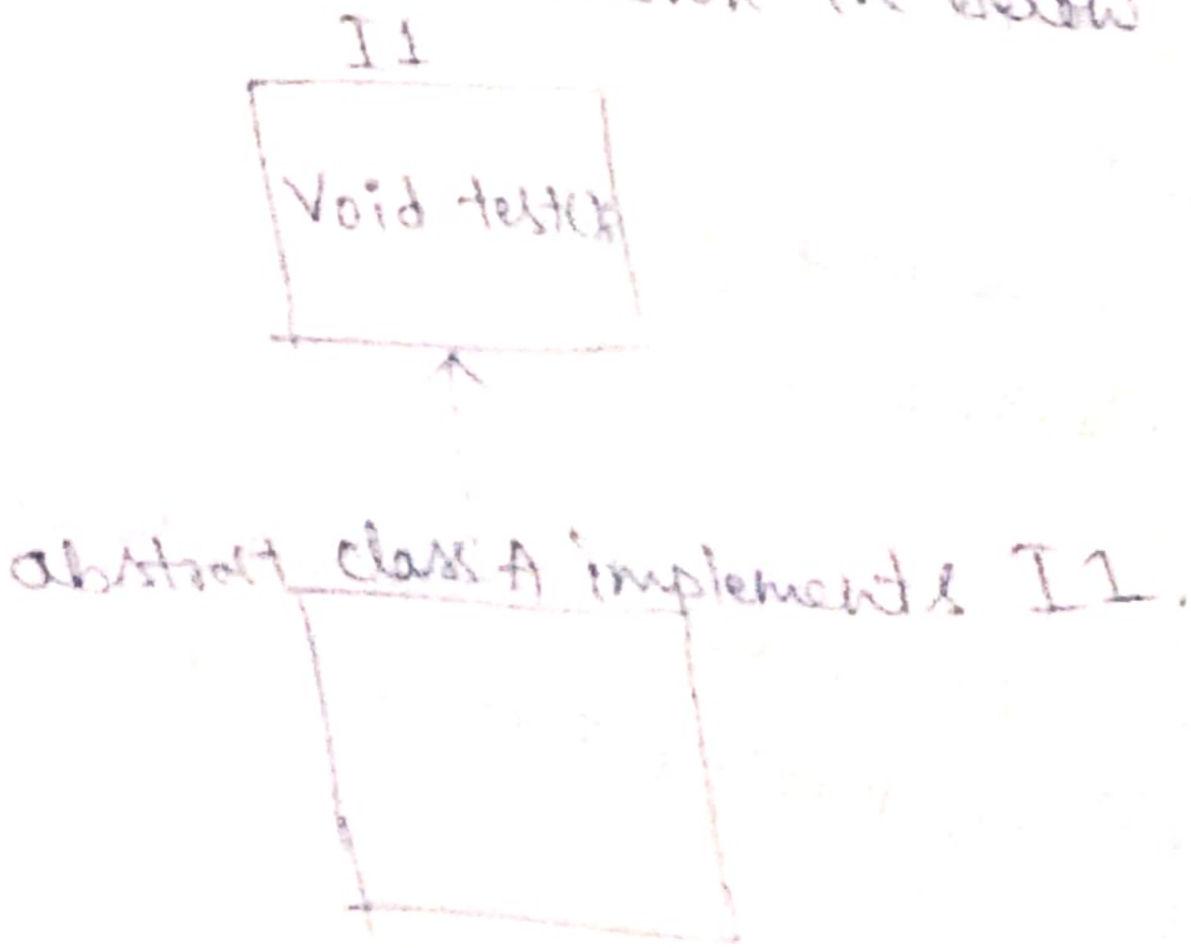
inheritance b/w interface & class:-

* Interface can never inherit a class, but a class can inherit interface.
* To achieve inheritance b/w interface & class we should use "implements" keyword.

ex:
```
interface I1
{
    void test();
}

abstract class A implements I1
{

}
```

In above example class A is implementing interface I1 but it is not providing implementation to the abstract methods of interface I1 ∴ it is declared as abstract class.

* We can represent inheritance b/w interface & class using a dotted arrow as shown in below diagram.

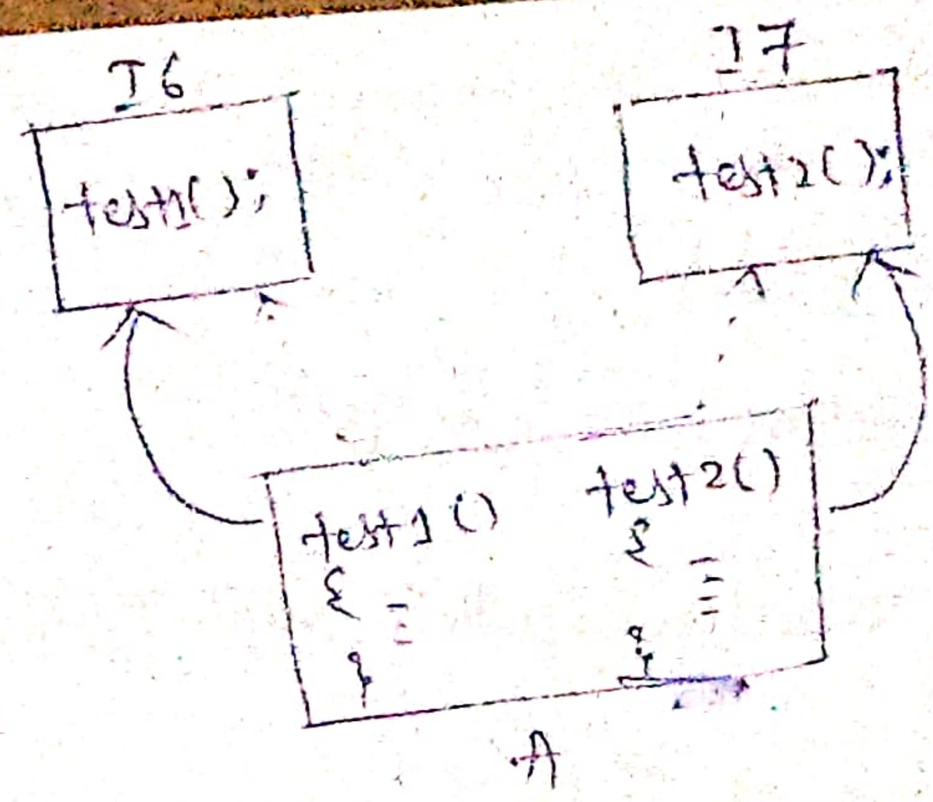


```
ex: interface I1
    {
      Void a1();
    }
    interface I2 extends I1
    {
      Void a2();
    }
abstract class A implements I2
{
  public Void a1()
  {
    S.o.pln("a1() implemented");
  }
}
Class B extends A
{
  public Void a2()
  {
    S.o.pln("a2() implemented");
  }
}
class Driver1
{
  P.S.V.m( )
  {
    B obj1 = new B();
      obj1.a1(); // a1() implemented
      obj2.a2(); // a2() implemented
    A obj2 = obj1; // Up Casting
      obj2.a1(); // a1() implemented
      obj2.a2(); // a2() implemented
  }
}
```

Note: When a class provides implementation abstract method of interface the access specifier of overriding method must be public.

ex:

```
interface I6
{
    void test1();
}
interface I7
{
    void test2();
}
class A implements I6, I7
{
    public void test1()
    {
        S.o.pln("test1() implemented");
    }
    public void test2()
    {
        S.o.pln("test2() implemented");
    }
}

class Driver2
{
    p.s.v.m()
    {
        A obj1 = new A();
        obj1.test1();
        obj2.test2();

I6      obj2 = obj1; //upcasting
        obj2.test1();
        obj2.test2(); //CTE

I7      obj3 = obj1; //upcasting
        obj3.test2();
        obj3.test1(); //CTE
    }
}
```



* Class can implement an interface & also can extend other class in order to achieve this we must use extends keyword before implements.

ex:
```
abstract class B
{
    abstract public void test();
}
interface I8
{
    public void test2();
}
class C extends B implements I8
{
    public void test1()
    {
        s.o.pln("test1() implements");
    }
    public void test2()
    {
        s.o.pln("test2() implemented");
    }
}
```

Note: We can achieve multiple inheritance using interface & in case of multiple inheritance we can convert the parent type reference to another parent type.

ex:
```
interface I9
{
    void a1();
}
interface I10
{
    void a2();
}
class D implements I9, I10
{
    public void a1()
    {
        s.o.pln("a1() implemented");
    }
    public void a2()
    {
        s.o.pln("a2() implemented");
    }
}
```

```
class Driver3
{
  P s.v.m( )
  {
    D obj1 = new D();
      obj1.a1();
      obj1.a2();

    I9 obj2 = obj1; ||upCasting
      obj2.a1();
    obj2.a2(); || CTE

    I10 obj3 = obj1;
      obj3.a2();
    obj3.a1(); || CTE

|| Converting one parent type ref. to another parent type.
    I9 obj4 = (I9)obj3; || from I10 to I9 type
    I10 obj5 = (I10)obj2; || from I9 to I10 type.
  }
}
```

Object Class :

Object Class is the pre-defined class in java it belongs to java.lang package.

* Object class is considered as super most class for all the classes in java.

Note: In java if extends keyword is not used for a class it by default extends object class.

* Object class belongs to java.lang package, any class which belongs to java.lang package will be by default imported.
* Object class has 1 no argument constructor & 11 non-static methods.

Methods of Object Class :-

1. public String toString()
2. public boolean equals(Object obj)
3. public int hashcode()
4. protected Object clone()
5. protected void finalize() (properties)
6. public final void notify()
7. public final void notifyAll()
8. public final void wait()
9. public final void wait(long timeout)
10. public final void wait(long timeout, int nanos)
11. public final Class getClass()

① toString() method :-

The object class implementation of toString() method returns address/reference of the object in a string format.

whenever the object reference variable is try to be used for printing toString() method of object class will be implicitly automatically called.

Before Compile

```
class A
{
  p.s.v.m()
  {
    A obj=new A();
    S.o.pln(obj);//reference
    S.o.pln(obj.toString());//reference
  }
}
```

After Compile

```
class A
{
  p.s.v.m()
  {
    A obj=new A();
    S.o.pln(obj.toString());
    S.o.pln(obj.toString());
  }
}
```

Purpose of Overriding toString() :

Object of class implementation of toString() method returns address/reference of the object instead of this if we want to get states of the object we should override toString() method.

Without Overriding

```
class Emp
{
  int eid;
  Emp(int eid)
  {
    this.eid=eid;
  }
}

class Driver1
{
  p.s.v.m()
  {
    Emp obj1=new Emp(1);
    S.o.pln(obj1);//Reference
  }
}
```

Without overriding

With overriding

```
class Emp
{
  int eid;
  Emp(int eid)
  {
    this.eid=eid;
  }
  //overriding toString()
  public String toString()
  {
    return "Eid:"+eid;
  }
}
```

o/p

Eid : 1 ← With overriding

(2) equals() method:

Object class implementation of equal() method ~~returns~~ Compares address/reference of two objects if they are same it returns true otherwise it returns false.

ex:
```
class B
{
    p.s.v.m()
    {
        B obj1 = new B();
        B obj2 = new B();
        S.o.pln(obj1.equals(obj2));//false

        B obj3 = new B();
        B obj4 = obj3;
        S.o.pln(obj3.equals(obj4));//true.
    }
}
```

Purpose of Overriding equals() method:

Object class implementation of equals method Compares two Object references/address instead of they if we Want to Compare states (properties) of the object We should override equals method

ex: without overriding
```
class Emp
{
    int eid;
    Emp(int eid)
    {
        this.eid = eid;
    }

    public String toString()
    {
        return "Eid:" + eid;
    }
}
```

With overriding
```
class Emp
{
    int eid;
    Emp(int eid)
    {
        this.eid = eid;
    }
    public String toString()
    {
        return "Eid:" + eid;
    }

    //overriding equals
    public boolean equals(object obj)
    {
        return this.eid == ((Emp)obj).eid;
    }
}
```

downcast

```
class Driver1
{
  p.s.v m()
  {
    Emp obj1=new Emp(1);
    Emp obj2 = new Emp(2);
    S.o.pln (obj1.equals (obj2)); //false
  }
}
```

//Compares object references

without overriding

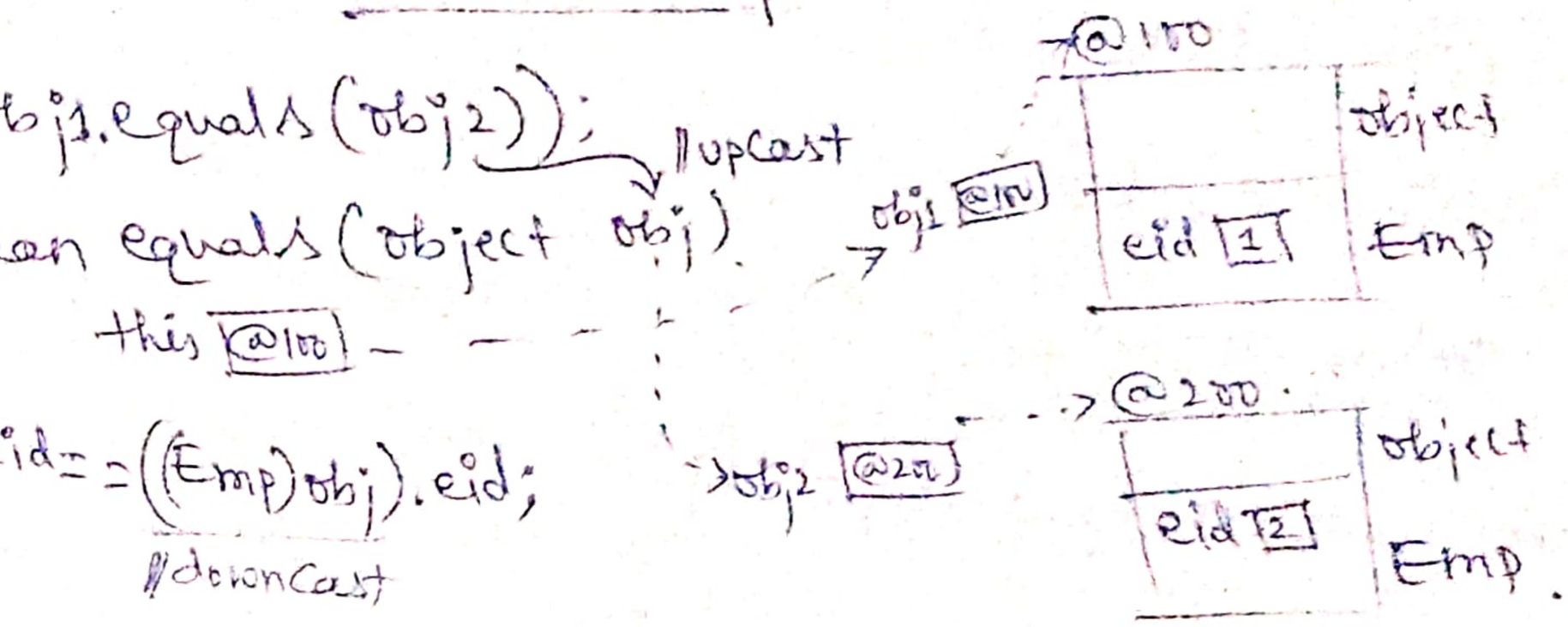
with overriding

Compares eid.

```
/* s.o.pln (obj1.equals (obj2));
public boolean equals (object obj)
{
  return this.eid==((Emp)obj).eid;
}
*/
```

//upcast

//downcast



(3) HashCode() method:-

Hashcode method returns address/ref of the object in integer format.

* Contract b/w equals & HashCode method:

* If equals() method returns true for two object references hashcode() method should return same integers.
* If equals() method returns false for two object references. hashcode() method should return different integers.

ex:

```
class C
{
  P.S.V.m
  {
    C obj1 = new C();
    C obj2 = new C();
    S.o.pln (obj1.equals (obj2)); //false
    S.o.pln (obj1.hashcode());
    S.o.pln (obj2.hashcode());
    C obj3 = new C();
    C obj4 = obj3;
    S.o.pln (obj3.equals (obj4)); //true
    S.o.pln (obj3.hashcode());
    S.o.pln (obj4.hashcode());
  }
}
```

diff integers.

same integers

Purpose of Overriding hashcode() method:
To maintain the Contract b/w equals & hashcode method.

Ex:

```
class Emp
{
    int eid;
    Emp(int eid)
    {
    this.eid = eid;
    }
    public String toString()
    {
        return "Eid:"+eid;
    }
public boolean equals(object obj)
{
    return this.eid == ((Emp)obj).eid;
}
public int hashcode() //overriding
                        hashcode
{
    return eid+321654;
}
}
class Driver 1
{
    P.S.V.m()
    {
        Emp obj1 = new Emp(1);
        Emp obj2 = new Emp(2);
        S.o.pln(obj1.equals(obj2));//true // compares eid.
        S.o.pln(obj1.hashcode()); } Same
        S.o.pln(obj1.hashcode()); } integer.

        Emp obj3 = new Emp(2);
        Emp obj4 = new Emp(3);
        S.o.pln(obj3.equals(obj4)); //false // Compares eid.
        S.o.pln(obj3.hashcode()); } diff
        S.o.pln(obj4.hashcode());
    }
}
```

String :-

String is a pre-defined class in java which belongs to java.lang package.

* It is a non-primitive data type.
* String class extends object class & implements Char Sequence, Comparable, & Serializable interface.
* String is a "final" class.

Can we Override a methods of String Class?

No, String being a final class cannot be inherited
∴ We cannot Override methods of String class.

Note: In order to store String Values we must Create object of String class.

Constructors of String class:

(i) String()
(ii) String(String Literal)
(iii) String(byte [] ref)
(iv) String(char [] ref)
(v) String(String Buffer ref)
(vi) String(String Builder ref)

* We can Create object of String class in two ways

① Directly using '=' (assignment) operator

② using "new" & "Constructor" of String class.

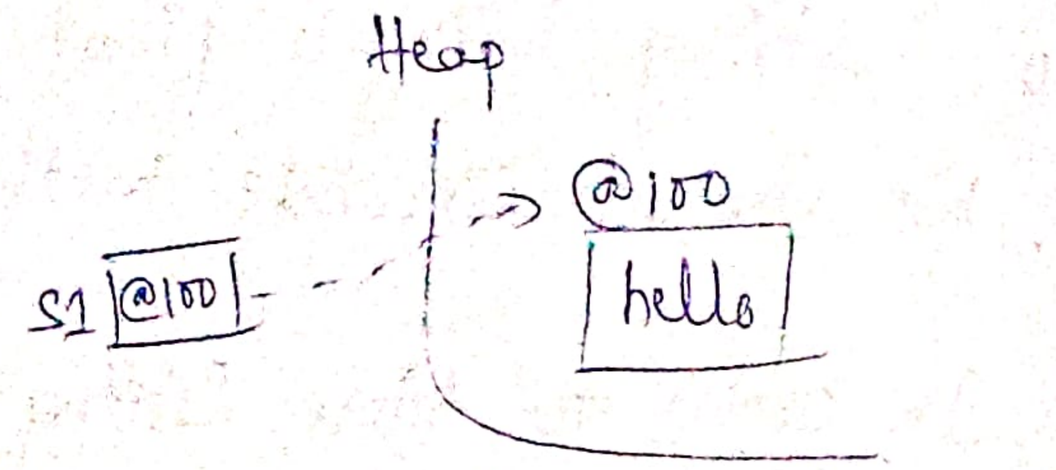
Note: the objects for String class are "immutable" (Can't be modified)

① Creating String Object Using "=" assignment Operator:

Syntax:- String ref_Variable = String_Literal;

ex: String S1 = "hello";

ex: class A
{
p.s.v.m(S[] args)
{
~~S.o.pln(~~
String S1 = "hello";
S.o.pln(S1); //hello
}
}

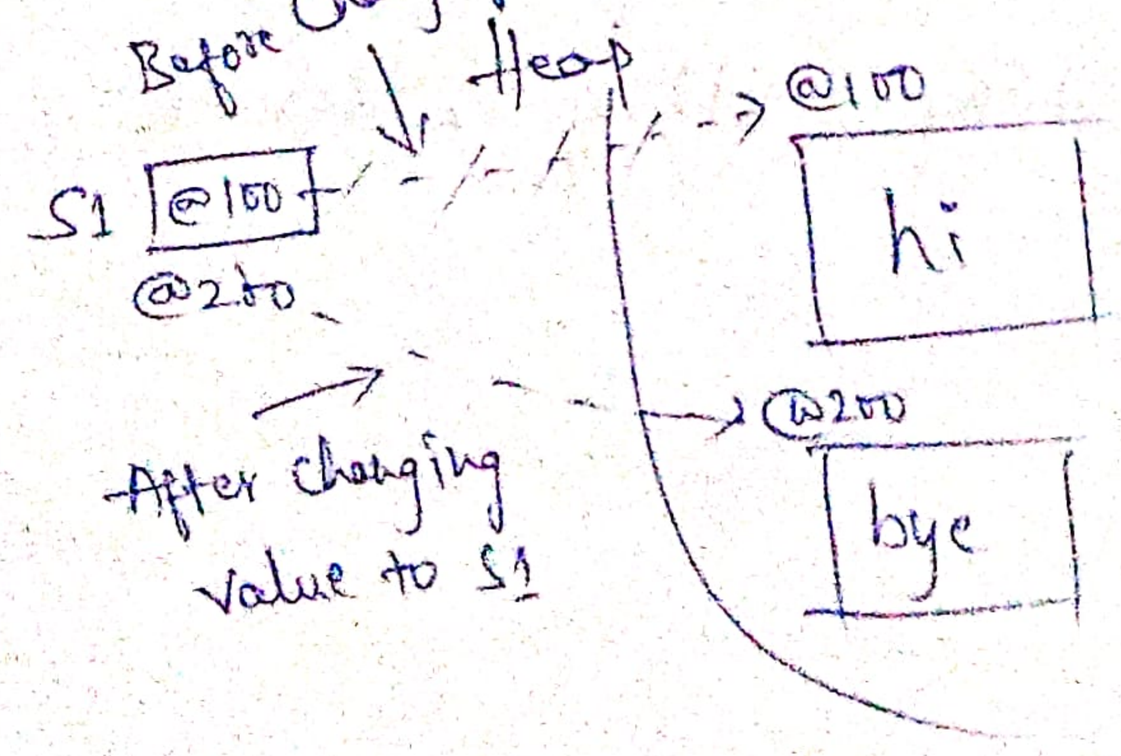


String Constant pool
(part of heap memory)

Note: The objects Created for String class using assignment operator
Will be Created inside "String Constant pool".

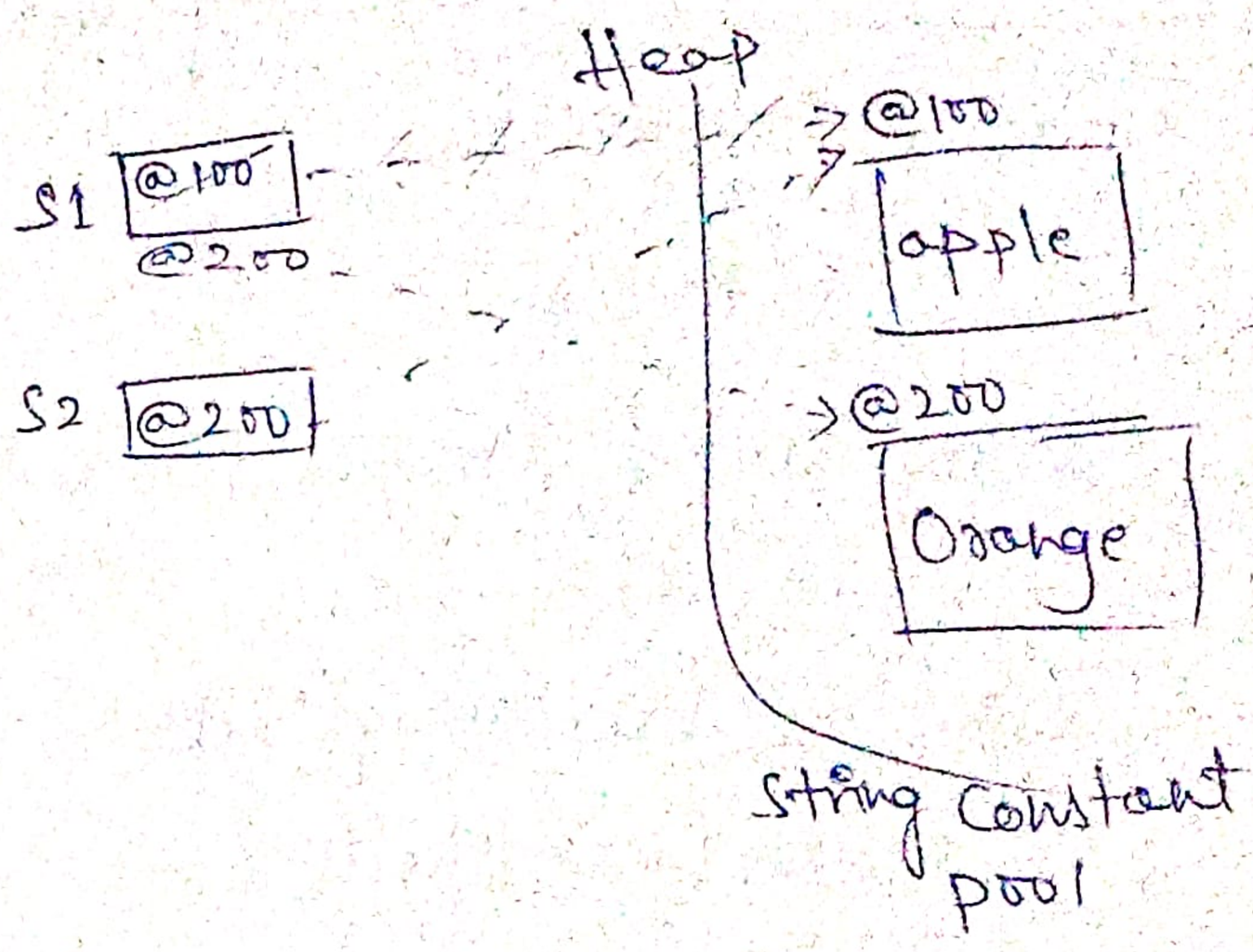
* the object class toString() method is overridden in String class in such a way that it returns value of the object instead of the reference/address.

ex: class A
{
p.s.v.m()
{
String S1 = "hi";
S.o.pln(S1); //hi
S1 = "bye";
S.o.pln(S1); //bye
}
}



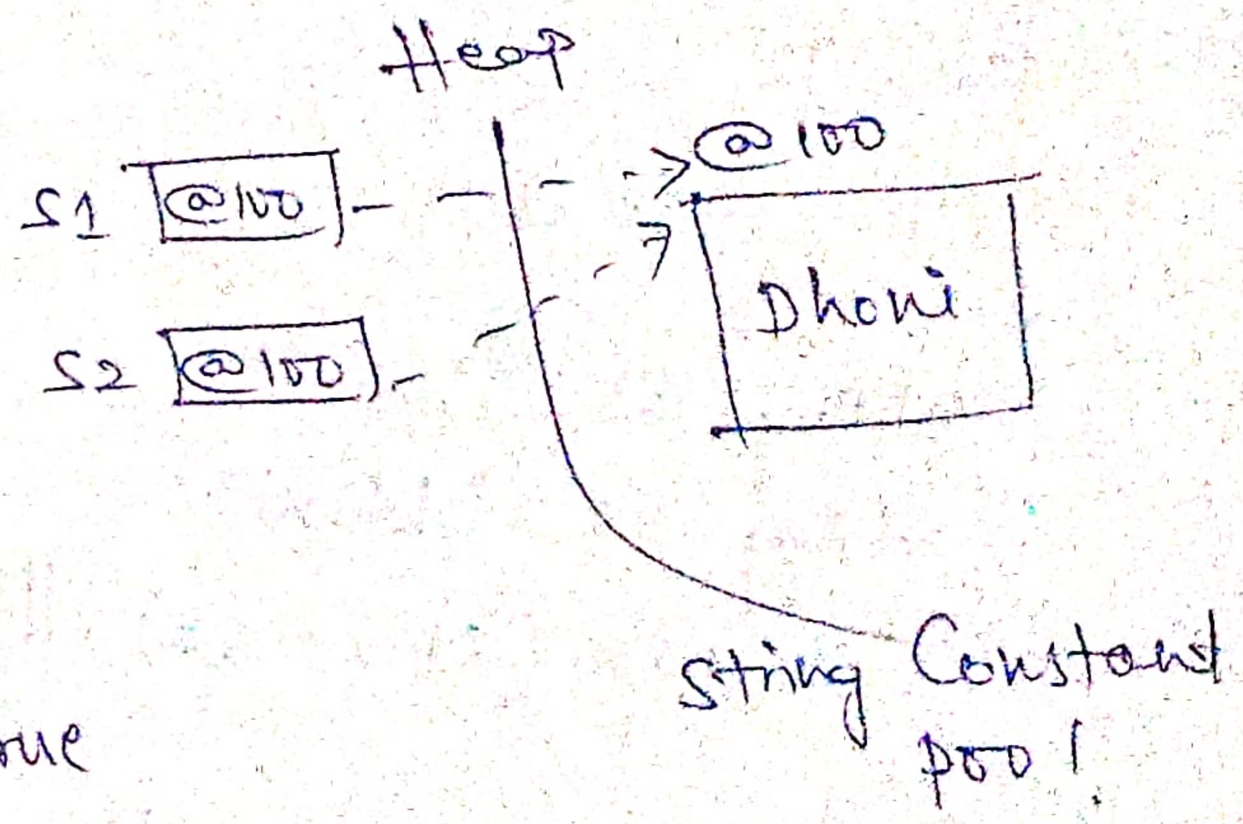
Ex: class A
{
p.s.v.m()
{
String s1 = "apple";
String s2 = s1;
S.o.pln(s1); //apple
S.o.pln(s2); //apple
S.o.pln(s1==s2); //true

s1 = "orange";
S.o.pln(s1); //orange
S.o.pln(s2); //apple
S.o.pln(s1==s2); //false
}
}



Note: Inside String Constant pool we can't have duplicate String objects. (objects with same value).

Ex: class A
{
p.s.v.m()
{
String s1 = "Dhoni";
String s2 = "Dhoni";
S.o.pln(s1==s2); //true
}
}



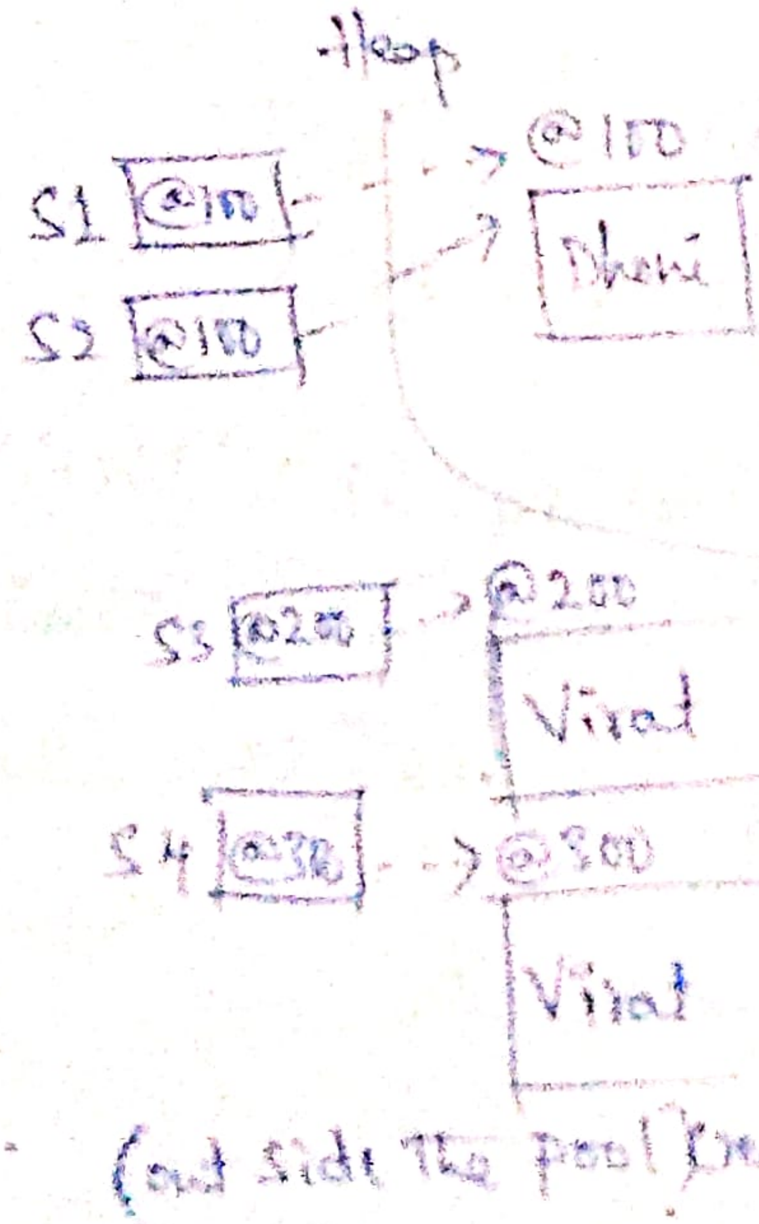
② Creating String Object Using "new" & "Constructor":

ex: class E

```
{
    p.s.v.m()
    {
        String s1=new String();//Creates an empty
                                 string object
        s.o.pln(s1);

        String s2=new String("hello");
        s.o.pln(s2);//hello

        ~~String~~ char[]ch={'a','b','c','d'};
        String s3=new String(ch);
        s.o.pln(s3);//abcd
    }
}
```

ex: class F

```
{
    p.s.v.m()
    {
        String s1="Dhoni";
        String s2="Dhoni";
        s.o.pln(s1==s2);//true

        String s3=new String("Virat");
        String s4=new String("Virat");
        s.o.pln(s3==s4);//false.
    }
}
```

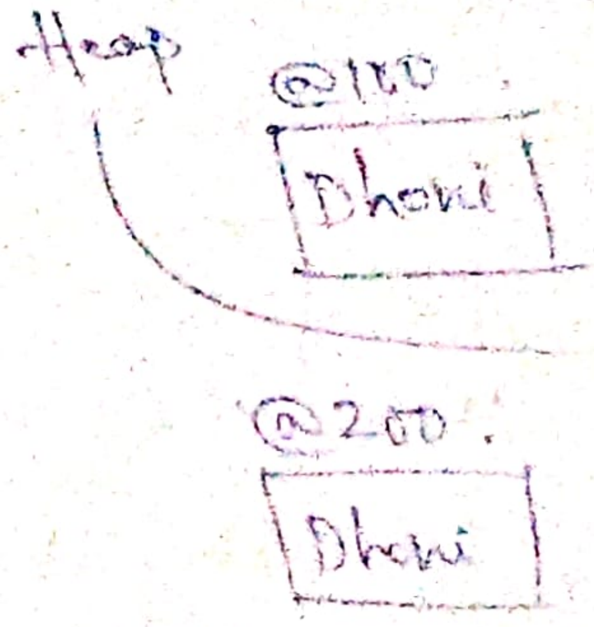


(out side the pool) (created)

Comparing two String Object references

We can compare two string object references using equality operator `==`.

ex: class G
{
p.s.v m()
{
String S1 = "Dhoni";
String S2 = new String("Dhoni");
S.o.pln(S1 == S2); //false
}
}



Comparing two String Values:-

* Using Equals Method:-

Equals() method of object class is overridden in String class in such a way that it compares values of the object instead of the reference.

ex: class H
{
p.s.v m()
{
String S1 = "Dhoni";
String S2 = new String("Dhoni");
S.o.pln(S1.equals(S2)); //true
}
}

* Using equalsIgnoreCase method:

~~class~~ equalsIgnoreCase method is used to compare two string values ignoring their cases.

ex:
```
class A
{
    p.s.v.m()
    {
        String s1 = "Dhoni";
        String s2 = new String("dhoni");
        S.o.pln(s1.equalsIgnoreCase(s2));//true
    }
}
```

* Using CompareTo() method:

$\boxed{\text{public int CompareTo(object obj);}}$

CompareTo is an abstract method of Comparable interface.
Comparable interface belongs to java.lang package.
String class implements Comparable interface & provides implementation for the abstract method (CompareTo()) of Comparable interface.

* CompareTo() returns '0' if both values are same.
* CompareTo() returns '+ve integer' if 1st ~~value~~ string is having higher value compare to 2nd one.
* CompareTo() returns '-ve integer' if 1st string is having lesser value compare to 2nd one.

ex:
```
class A
{    p.s.v.m()
    {
        String s1 = "hello";
        String s2 = new String("hello");
        S.o.pln(s1.CompareTo(s2));//0

        String s3 = "hi";
        String s4 = "Hi";
        S.o.pln(s3.CompareTo(s4));//+ve integer.
    }
```

Converting Primitive Values into String

Primitive data types.
- byte
- short
- int
- long
- float
- double
- char
- boolean

→ String

We can convert any primitive value to String with the help of "ValueOf()" method.

* ValueOf() is a static method of String class. & it is overload (multiple valueof() methods with diff. formal arguments)
* Return type of valueof() method is String.

ex:
```
class K
{
    p.s.v.m()
    {
        int a=12;
        String s1=String.ValueOf(a);
        S.o.pln(s1); // 12 ← but String type.
        S.o.pln(s1+24); // 1224

        double b=12.5;
        String s2=String.valueof(b);
        S.o.pln(s2); // 12.5

        boolean c=false;
        String s3=String.ValueOf(c);
        S.o.pln(s3); // false
    }
}
```

Converting Character Array into String :

We can Convert Character array into String in 2 Ways
(i) Using Constructor of String class which accepts char array
(ii) Using ValueOf() method.

ex:
```
class L
{
    p.s.v.m()
    {
        char[] ch = {'a','b','c','d'};
        String s1 = new String(ch);
        S.o.pln(s1); //abcd.

        String s2 = String.ValueOf(ch);
        S.o.pln(s2); //abcd
    }
}
```

Converting String into Character Array :

We can Convert String into Character Array Using "toCharArray()" method.

* toCharArray() method is a non-Static method.
∴ We must Call it With help of object reference & it returns Character Array.

ex:
```
class M
{
    p.s.v.m()
    {
        String s1 = "qspiders";
        char[] ch = s1.toCharArray();
        for(int i=0; i<ch.length; i++)
        {
            S.o.pln(ch[i]);
        }
    }
}
```

o/p
q
s
p
i
d
e
r
s

To find length of String

* We can find length of the String with the help of non-Static method called "length()".

To extract Character from String:

* We can extract a Character from String with help of "CharAt(int)" method. it accepts index & returns character present at that index.

ex: class N
{
p.s.v.m()
{
String S1 = "Hello";
S.o.pln("length of String is :" + S1.length()); // 5
S.o.pln(S1.CharAt(1)); // e
}
}

* WAP to check how many times lowercase char 'a' is present in the String.

class A
{
p.s.v.m()
{
String S1 = "Malayalam";
char[] ch = S1.toCharArray();
~~char ch = 'a';~~ int Count = 0;
for(int i=0; i<ch.length; i++)
{
if(ch[i] == 'a')
{
Count++;
}
}
S.o.pln("a is present at: " + Count);
}
}

* WAP to Count lowercase Vowels present in the String.

```
class A
{
    p. s. v. m()
    {
        String s1 = "Marvelous";
        char [] ch = s1.toCharArray();
        int Count = 0;
        for(int i=0; i< ch.length; i++)
        {
            if(ch[i]=='a' || 'e' || ch[i]=='e' || ch[i]='o' ----
            {
                Count ++;
            }
        }
        s.o.pln("lower Case Vowels present is "+Count);
    }
}
```

Trim() method:

* trim() method is used to remove the Spaces from beginning & end of the String. it will not affect spaces in ~~the~~ ~~mi~~ Within it.

ex:

```
class Q
{
    p. s. v. m()
    {
        String s1 = "    qspi ders    ";
        s.o.pln(s1.length()); //14
        s1=s1.trim();
        s.o.pln(s1); //qspi ders
        s.o.pln(s1.length()); //9
    }
}
```

* WAP to reverse a string.

```
class A
{
    p.s.v.m()
    {
        String s1 = "Hello";
        String s2 = " ";
        for(int i=0; i<s1.length(); i++)
        {
            s2 = s1.charAt(i)+s2;
        }
        s2 = s2.trim();
        s.o.pln(s2);// OlleH
    }
}
```

Hello
0 1 2 3 4

s2 = s1.charAt(i)+s2
= s1.charAt(0)+s2
= H+" " ← space
s2 = H_

s2 = s1.charAt(1)+s2
s2 = e+H_ = eH_

s2 = s1.charAt(2)+s2
= l+eH_ ⇒ leH_

s2 = s1.charAt(3)+s2
= l+leH_ ⇒ lleH_

s2 = s1.charAt(4)+s2
= O+lleH_ ⇒ OlleH_ ← space

s2 = s2.trim(); ⇐ Space removed
s2 = OlleH

IndexOf() method

* indexof() method is used search for a character (or) string in the given string.
* If the char (or) string to be searched is present it returns index otherwise it returns -1.
* IndexOf() method is an overloaded method.

(i) int indexOf(char ch)
(ii) int indexOf(char ch, ~~start~~ int start_index)
(iii) int indexOf(String s)
(iv) int indexOf(String s, int start_index)

ex: String s1 = "Hello. World";
0 1 2 3 4 5 6 7 8 9 10

s.o.pln(s1.indexOf('r'));// 8
s.o.pln(s1.indexOf('L', 5));// 9 // starts from 5.
s.o.pln(s1.indexOf("wor"));// 6 w starts from 6
s.o.pln(s1.indexof("wor", 8));// -1 starts from 8
∴ Not matching returns -1.

* WAP to check whether lowercase char 'a' is present in the given string or not * using index() method.
* without index() method.

SubString() method

* is used to extract a part of the String from the given String.
* Substring() is Overloaded method.

(i) substring(int Start_index)

* It extract the String from the provided ~~end~~ Start index till the last index of the String.

(ii) String Substring(int Start_index, int end_index)

* It extracts the String from the given Start_index till the given end_index but end_index is exclusive (means last char is not taken).

ex: class W
{ p.s.v.m()
{
String S1 = "Karnataka";
0 1 2 3 4 5 6 7 8

S.o.pln(S1.Substring(3));// nataka
S.o.pln(S1.Substring(0));//Karnataka
S.o.pln(S1.Substring(0,5));// Karna
S.o.pln(S1.Substring(0,9));// Karnataka
S.o.pln(S1.Substring(0,10));// String Index Out of Bound Excep
} }

Split() method :-

* is used to break the given String into multiple Strings.
* It breaks the given String into multiple Strings based the String given as argument. & returns a String Array.

ex: class A
{ p.s.v.m()
{
String s = "hello how are you";
String [] s1 = s.Split(" ");
for(int i=0; i<s1.length; i++)
{
S.o.pln(s1[i]);
}
} }

o/p
hello
How
are
you.

Concat(" ") Method :-

ex:
```
class A
{
    p.s.v.m()
    {
        String s1 = "Hello";
        s1 = s1.Concat("all");
        S.o.pln(s1); // hello all
    }
}
```

String Builder & String Buffer :

String Builder & String Buffer class belongs to java.lang package & these classes are used to create mutable String Object. (content can be changed without creating new object)

String Buffer class :

* It is a thread safe methods of String Buffer class are non-synchronized.
* String builder classes are not thread safe methods of String builder classes are synchronized.

Constructor of String Builder classes

String Builder()
String Builder(String literal)

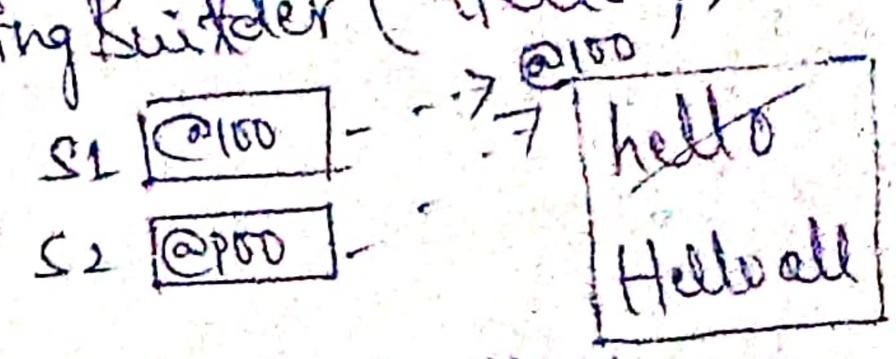
Constructor of String Buffer classes

String Buffer()
String Buffer(String literal).

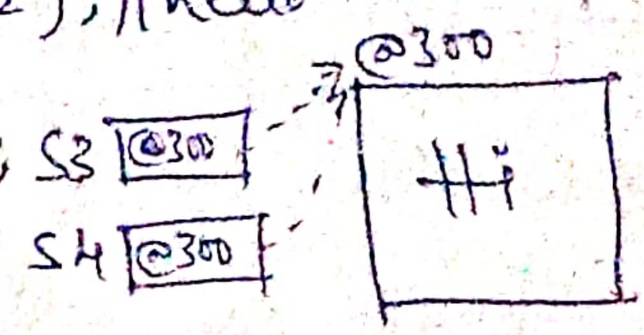
Note. For String builder & String buffer classes we cannot create objects directly using assignment operator.

ex: class A
{
p.s.v.m()
{

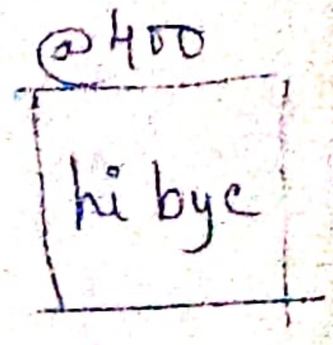
```
StringBuilder s1 = new StringBuilder("Hello");
StringBuilder s2 = s1;
s.o.pln(s1); //hello
s.o.pln(s2); //hello.
s1.append("all");
s.o.pln(s1); //hello all
s.o.pln(s2); //hello all
String s3 = "hi";
String s4 = s3;
s.o.pln(s3); //hi
s.o.pln(s4); //hi
s3.Concat("bye");
s.o.pln(s3); //hi
s.o.pln(s4); //hi
```




```
s.o.pln(s1); //hello
s.o.pln(s2); //hello.
s1.append("all");
s.o.pln(s1); //hello all.
s.o.pln(s2); //hello all.
String s3 = "Hi";
```




```
s.o.pln(s3); //hi
s.o.pln(s4); //hi
s3.Concat("bye");
s.o.pln(s3); //hi
s.o.pln(s4); //hi
```



}

Converting immutable String object into mutable String object

* We can convert immutable String object into mutable with the help of Constructor of String builder (or) String buffer which accepts String.

ex:

```
String s1 = new String("hello");
StringBuilder s2 = new StringBuilder(s1);
s.o.pln(s2); //hello.
```

Converting mutable String object into immutable String object

We can convert mutable string object into immutable with the help of constructors of String class which can accepts StringBuilder & StringBuffer classes.

ex:
```
StringBuffer S1 = new StringBuffer("Hi");
String S2 = new String(S1);
S.o.pln(S2); // hi.
```

Array of Object:

* We can create array to store multiple objects of class as shown in below.

ex:
```
class A
{
    P. S. V. m()
    {
        A obj1 = new A();
        A obj2 = new A();
        A obj3 = new A();
        A obj4 = new A();

        A []arr = new A[4]; // class_name []ref-var = new class name [size];
        arr[0] = obj1;
        arr[1] = obj2;
        arr[2] = obj3;
        arr[3] = obj4;
        S.o.pln(arr); // prints reference of array.
        for(int i=0; i<arr.length; i++)
        {
            S.o.pln(arr[i]); // prints ref. of each objects in array.
        }
    }
}
```

ex:

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

class Driver1
{
    p.s.v.m()
    {
        Student [] s = new student [3];
        Student s1 = new student ("A", 1, 12);
        Student s2 = new Student ("B", 2, 13);
        Student s3 = new student ("C", 3, 14);

        s[0] = s1;
        s[1] = s2;
        s[2] = s3;
    for (int i=0; i< s.length; i++)
    {
        S.o.pln ("Name" + s[i].name);
        S.o.pln ("RollNum:" + s[i].rollNum);
        S.o.pln ("Age" + s[i].age);
    }
    }
}
```

Exception Handling :

The problem (or) abnormally occured during execution of program is called exeption.

* Generally when exception occurs program execution stops. but internally JVM creates an object of throwable type.
* It checks for exception handling mechanism if execution handling mechanism not used program exception stops.
* If exception handling mechanism is used it checks whether mechanism used is capable handling the exception. If yes ⇒ program exception is continues otherwise stops.

" The process of handling the exception with the help of some exception handling mechanism is called as exception handling ".

Exception Handling Mechanism :

* Mechanism that is used to handle the exception is called.
* Java provides following exception handling mechanism.
  1. try - catch block
  2. try with multiple catch block
  3. try - catch - finally block.
* Types of Exception
  1. Checked Exception
  2. Unchecked Exception.

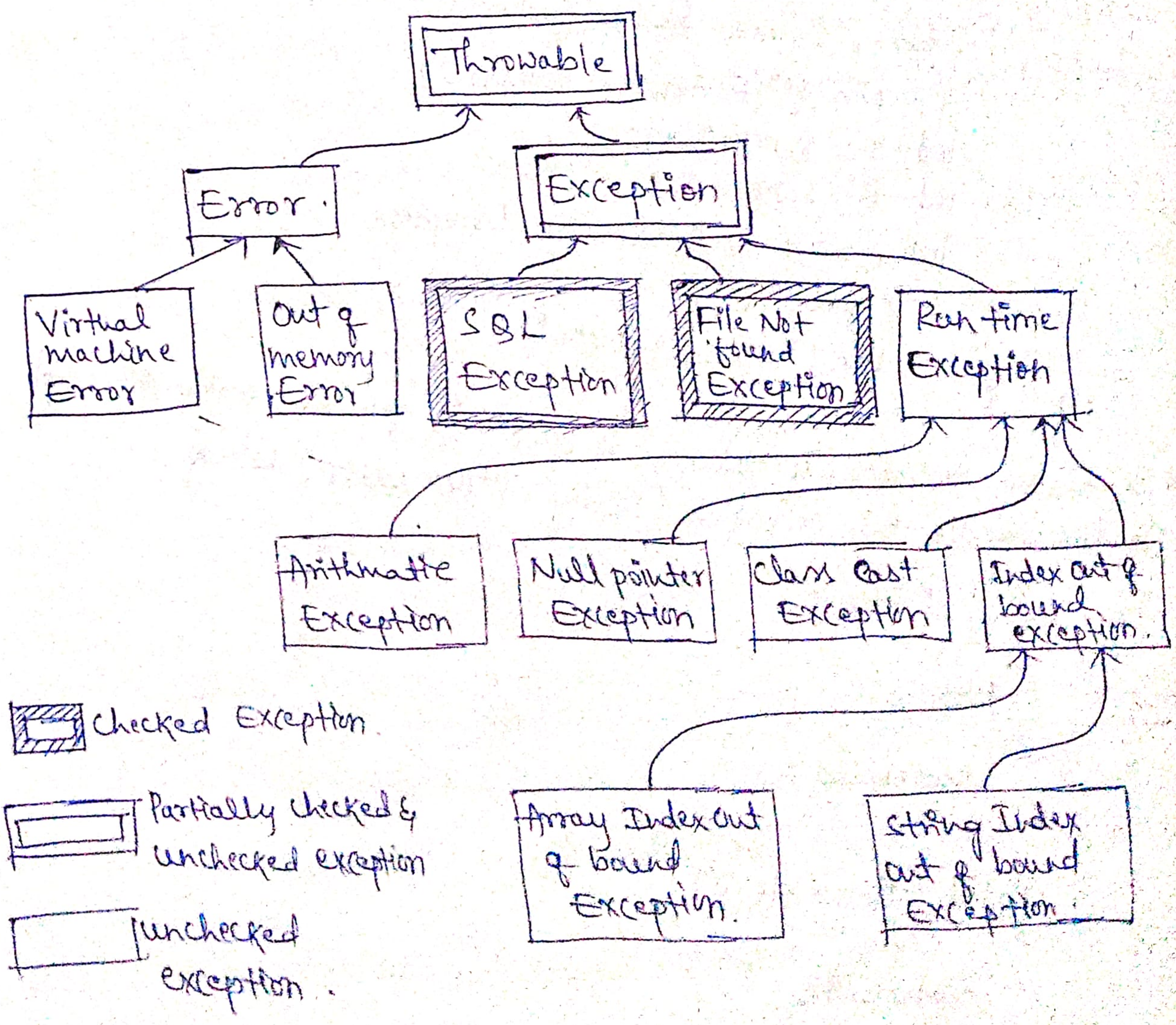
① Checked Exception

* The Compiler ~~aware~~ aware exception are called "Checked exception"
* For checked exception handling (or) declaring exception is mandatory, otherwise Compiler generates Compile Time error.

② Unchecked Exception

* Compiler unaware exception are called "unchecked Exception"
* For unchecked exception handling (or) declaring exception is not mandatory.

Exception Hierarchy



Unchecked Exception:

The Compiler unaware exception are "Unchecked exception".

ex: Run time exception.

Run time Exceptions:

* Run time exception class & all it's sub classes are unchecked exceptions.

ex:

```
import java.Util.Scanner;
class A
{
    P.S.V.m()
    {
        S.O.pln("main() started");
        Scanner s = new Scanner(System.in);
        S.O.pln("Enter two numbers");
        int a = s.nextInt();
        int b = s.nextInt();
        int c = a/b; //Arithmatic Exception.
        S.O.pln(c);
        S.O.pln("main() ended");
    }
}
```

o/p
main() Started
Enter two Numbers
10
0
Arithmatic exception ⇐ error

* In above example if value of b is '0' it generates arithmetic exception. We can handle this exception using "try-catch" block.

try-Catch Block:-

Syntax:

```
try
{
    Statement;
}
Catch(Throwable e)
{
    Statements;
}
```

try block :-

* The statement responsible for exception must be written inside the try block. When exception is generated JVM creates throwable type object & try block throws this object to catch block.

Catch block :-

* is used to catch the thrown throwable type object.
* If catch block has suitable object reference variable declared then exception is caught & handled.
* If there is no suitable object reference variable declared that means exception is not caught & program stops.
* In catch block we will write the handling code (or) alternate statements to be executed when exception is generated.

ex:

```
import java.Util.Scanner;
class A
{
    p.s.v.m()
    {
        S.o.pln("main() started");
        Scanner S = new Scanner(System.in);
        S.o.pln("Enter two numbers");
        int a = S.nextInt();
        int b = S.nextInt();
        try
        {
            int c = a/b;
            S.o.pln(c);
        }
        Catch(ArithmeticException e)
        {
            S.o.pln("ArithmeticException Caught & handled");
        }
        S.o.pln("main() ended");
    }
}
```

o/p

```
main started
Enter two no's
10
0
ArithmeticException Caught
& handled
main() ended.
```

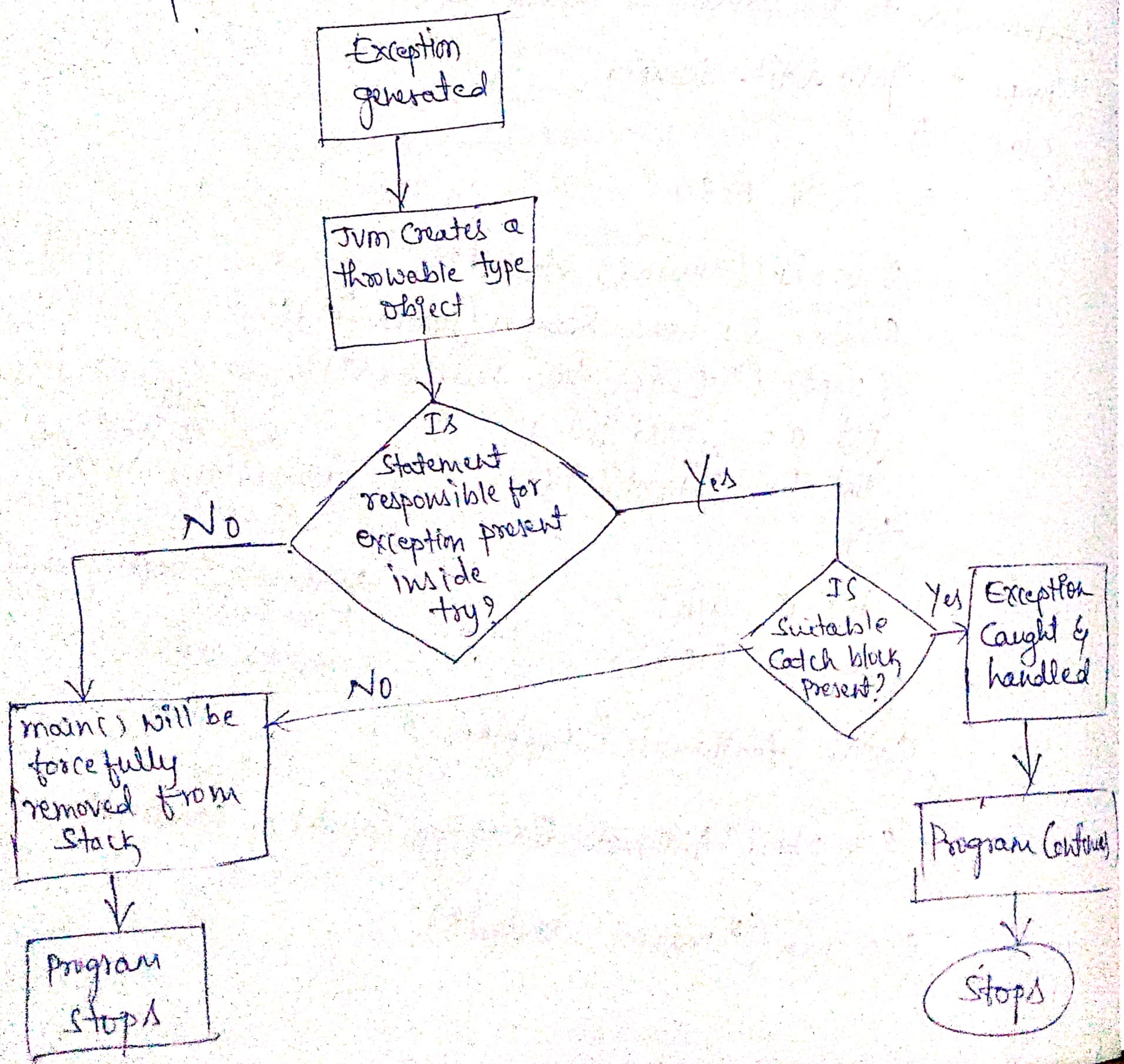
In above example if we declare "NullPointerException" type object ref. in Catch block then program stops (Because there is no suitable Catch block for Arithmetic exception).

ex.
```
Catch (NullPointerException e)
{
    S.o.pln ("Exception Caught & handled");
}
```

o/p
Arithmetic exception ⇐ error

Note:- In the Catch block we can also declare Super class type Object ref. variable in order to handle exception.

ex:
```
Catch (Exception e) //It accepts any exceptions.
{
    S.o.pln ("Exception Caught & handled");
}
```



Multiple Catch Block :-

Syntax:

```
try
{

}
Catch (Throwable e)
{

}
Catch (Throwable e)
{

}
.
.
.
```

* We can use try with multiple catch blocks, Suitable catch block will be check in top to bottom order.

ex: class A

```
{
    p.s.v.m()
    {
        s.o.pln ("Main() Started");
        Scanner s = new Scanner(System.in);
try {   int a = s.nextInt();
        int b = s.nextInt();
        int c = a/b;
        int [] arr = null;
        s.o.pln (arr[0]);
    }
    Catch (Arithmetic Exception e)
    {
        s.o.pln ("Arithmetic Exception Caught & handled");
    }
    Catch (NullPointer Exception e)
    {
        s.o.pln ("Null pointer Exception Caught & handled");
    }
    }
}
```

* In above example there is a possibility of getting either NullPointer (or) Arithmetic exception in order to handle these exceptions we have ~~to~~ used multiple Catch blocks.

<u>Note</u>:- In try with multiple Catch blocks the Catch block with Super class type ref. Variable must be used at the end. otherwise Compiler generates CTE.

<u>Valid</u>

```
try
{
}
Catch (ArithmeticException e)
{
}
Catch (NullPointerException e)
{
}
Catch (Exception e)
{
}
```

<u>Invalid</u>

```
try
{
}
Catch (Exception e)
{
}
Catch (ArithmeticException e)
{
}
Catch (NullPointerException e)
{
}
```

Finally Block:

Finally block gets executed in all following Scenarios.

(i) Exception is not generated:-

```
class A
{
    p. s. v. m()
    {
        try
        {
            int a=10;
            int b=2;
            int c=a/b;
        }
        Catch (Arithmetic Exception e)
        {
            S.o.pln ("Exception Caught & handled");
        }
        finally
        {
            S.o.pln ("finally block executed");
        }
    }
}
```

o/p

Finally block executed.

(ii) Exception generated & Handled:

```
Class A
{
    p. S. V. m()
    {
        try
        {
            int a=10;
            int b=0;
            int c=a/b;
        }
        Catch (Arithmetic Exception e)
        {
            S.o.pln ("Exception Caught & handled");
        }
        finally
        {
            S.o.pln ("finally block executed");
        }
    }
}
```

o/p

Exception Caught & handled
finally block executed.

(iii) Exception generated but not handled

```
class A
{
    p. s. v. m()
    {
        try
        {
            int a=10;
            int b=0;
            int c=a/b;
        }
        Catch (NullPointerException e)
        {
            S.o.pln ("Exception Caught & handled");
        }
        finally
        {
            S.o.pln ("finally block executed");
        }
    }
}
```

o/p

finally block executed

Arithmetic exception ← error

Note: The statements that are mandatory to executed must be written in "finally" block.

ex: ~~clos~~ statement to close database connection,

statement to open & close the resource.

Exception Object Propogation

* The movement of exception object from called method to calling method is called exception object propogation.
* We can handle the exception generated in the called method inside the calling ~~too~~ method as shown in below example

ex:-

```
class A
{
    public static void test()
    {
        S.o.pln("test() started");
        S.o.pln(10/0);
        S.o.pln("test() ended");
    }

    p. s. v. m ()
    {
        S.o.pln("main() started");
        try
        {
            test();
        }
        Catch (Exception e)
        {
            S.o.pln("Exception Caught & handled");
        }
        S.o.pln("main () ended");
    }
}
```

printStackTrace() method :-

* is a non-static method of throwable class & it gives exception details (stackTrace).

ex:-
```
class A
{
    public static void test()
    {
        s.o.pln(10/0);
    }
    p.s.v.m()
    {
        s.o.pln("main() started");
        try
        {
            test();
        }
        catch(ArithmeticException e)
        {
            s.o.pln("Handled");
            e.printStackTrace();
        }
        s.o.pln("main() ended");
    }
}
```

o/p
```
main() started
Java.lang.ArithmeticException <error
Handled
main() ended.
```

Checked Exception :-

Compiler aware exceptions are called checked exceptions, for checked exceptions handling or declaring exception is mandatory

ex:-
```
import java.io.*;
class A
{
    public static void main(String[] args)
    {
        FileOutputStream f1=new FileOutputStream("F:\\p1.txt");
    }
}
```

trying to create one file (.txt)

(i) * In above example statement written as (FileOutputStream f1 = new FileOutputStream ("F:\\P1.txt");) may generate an exception called "FileNotFound" exception.

* File Not found exception is a checked exception. Since it is neither handled (or) declarclared in program compiler generates compile time error.

Handling checked Exception :-

* We can handle the checked exception with help of try & catch block.

ex:

```
import java.io.*;
class A
{
    p.s.v.m()
    {
        try
        {
         FileOutputStream f1=new FileOutputStream ("F:\\P1.txt");
         s.o.pln ("File Created Successfully!!");
        }
        Catch(FileNotFound Exception e)
        {
          s.o.pln ("Failed to Create the file");
          s.o.pln ("Invalid path!!");
        }
    }
}
```

Declaring Exception:

* We can declare the exception using "throws" Keyword.
* "throws" Keyword must be used in the header of the method.
  Using throws Keyword we can declare multiple exceptions.

Syntax: Method header throws Exception1, Excep2 ... ;

ex:
```
import java.io.*;
class A
{
    public static void main(String[] args) throws FileNotFoundException
    {
        ~~FileNotFoundExcep~~
        FileOutputStream f1=new FileOutputStream("F:\\p1.txt");
        S.o.pln("file Created Sucessfully");
    }
}
```

"throw" Keyword:-

* is used to generate exception explicitly (forcefully).

ex:
```
class A
{
    p.s.v.m()
    {
        S.o.pln("main() Started");
        throw new NullPointerException();
    }
}
```

o/p
main() Started
NullPointer Exception.

* "throw" Keyword must be the last statement of the block, if we write any statement after throw in same block it generates Compile time error.

ex:
```
class A
{
    p.s.v.m()
    {
        S.o.pln("main() Started");
        throw new NullPointerException();
        S.o.pln("main() ended"); //unreachable Statement error //CTE
    }
}
```

Custom Exceptions : (User defined Exceptions)

* The exception Created by programmer is called Custom exception (or) User defined exception.
* Steps to Create Custom exception.
  1. Create a class
  2. Make that class extend any class from throwable error.

ex:
```
class myException extends RuntimeException
{

}
class K
{
    p.s.v.m()
    {
        s.o.pln("from main()");
        throw new myException();
    }
}
```

* If Custom Exception class extends any unchecked exception class from throwable hierarchy it acts as unchecked exception.
* If Custom exceptim Class extends any Checked exceptim class from throwable hierarchy it acts as checked exception.
* If Custom exceptim class extends any Partially Checked & Partially unchecked exception class it acts as checked exception.

ex:
```
class ShaadiException extends RuntimeException
{
    ShaadiException(String s) // to display description/msg for
    {                            the exception.
        Super(s);
    }
}
```

```
class Myshaadi
{
  public Static Void ageVerifier(int age)
  {
    if (age<25)
    {
      throw new shaadiException("Control Yourself, Comeback after 25");
    }
    else
    {
      showNextPage();
    }
  }
  public Static Void showNextPage()
  {
    S.o.pln("Welcome to Myshaadi");
    S.o.pln(" Happy Searching");
  }
  public Static Void main(String[] args)
  {
    ageVerifier(21);
  }
}
```

Methods of "Scanner" class :-

nextByte(); ← to receive byte type of data
nextShort(); || to receive Short type of data
nextLong(); || to receive Long type of data
nextFloat(); || to receive float type of data
nextDouble(); || to receive double type of data.

next(); || to receive a String (receives a Single String)
nextLine(); || to receive a Sentance as a String.

nextBoolean(); || to receive boolean type of data.

To receive character :

* We don't have any inbuilt method in Scanner class to receive a character in order to receive character we should first take it as a string using next(); & convert it to character with help of charAt();

ex: import java.util.Scanner;

```
class N
{
  public static void main (String[] args)
  {
    Scanner s=new Scanner(System.in);
    s.o.pln ("Enter a character");
    char s1=s.next().charAt(0);
    s.o.pln ("Entered character is :"+s1);
  }
}
```

Wrapper Classes :

Primitive data types:
- byte
- short
- int
- long
- float
- double
- char
- boolean

Wrapper classes:
- Byte
- Short
- Integer
- Long
- Float
- Double
- Character
- Boolean

classes (or) Non-Primitive data types.

* Wrapper classes are used to Convert primitive Values into objects.

Why should we Convert Primitive Values into Objects ?:-

* The Collection framework of java stores only objects it Can't store primitive Values ∴ it becomes mandatory to Convert primitive values into objects before storing them inside Collection.
* All the Wrapper Classes belongs to java.lang package
* All the Wrapper Classes are final classes.
* All the Wrapper Classes implements Comparable & Serializ interfaces

Boxing :-

* The process of converting primitive value into an object is called boxing.

* We can perform boxing with the help of "Valueof()" method
* Valueof() is a static method present in each & every wrapper class.

ex:-
```
class A
{
    p. s. v. m()
    {
        int a=10;
        Integer b=Integer.Valueof(a); // Boxing
        S.o.pln(b); //10

        double c=12.5;
        Double d=Double.Valueof(c); // Boxing
        S.o.pln(d); // 12.5
    }
}
```

Note: from JDK 1.5 onwards, we can perform boxing without using Value of() method. as shown in below example, this process is called as "autoboxing"

ex:-
```
class B
{
    p. s. v. m()
    {
        int a=16;
        Integer b=a; //Autoboxing
        S.o.pln(b); //16

        boolean c=true;
        Boolean d=c; //Autoboxing
        S.o.pln(d); //true
    }
}
```

UnBoxing

* The process of Converting object into primitive type is Called.

ex: class C

```
{
    p.s.v.m()
    {
        Integer a=new Integer(10);
        S.o.pln(a); //10
        int b=a.intValue(); //Unboxing
        S.o.pln(b); //10

        Double c=new Double(12.6);
        double d=c.doubleValue(); //Unboxing
        S.o.pln(d);
    }
}
```

Note: from JDK 1.5 Onwards we Can Convert an Object into primitive type without using any method, it is known as "Auto Unboxing"

ex: class D

```
{
    p.s.v.m()
    {
        Integer a=new Integer(20);
        int b=a; //Auto UnBoxing
        S.o.pln(b); //20
    }
}
```

Converting String into its respective primitive type

String to byte → Byte.parseByte (String s)
String to short → Short.parseShort (String s)
String to int → Integer.ParseInt (String s)
String to long → Long.parseLong (String s)
String to float → Float.parseFloat (String s)
String to double → Double.parseDouble (String s)
String to boolean → Boolean.parseBoolean (String s)

ex:

```
class E
{
    p.s.v.m()
    {
        String s1 = "125";
        int a = Integer.parseInt(s1);
        S.o.pln(a);

        String s2 = "12.8";
        double b = Double.parseDouble(s2);
        S.o.pln(b);

        String s3 = "true";
        boolean c = Boolean.parseBoolean(s3);
        S.o.pln(c);

        String s4 = "a";
        char ch = s4.charAt(0);
        S.o.pln(ch);
    }
}
```

Collection Framework :

Collection:-

* Representing group of objects as a single entity is called Collection.

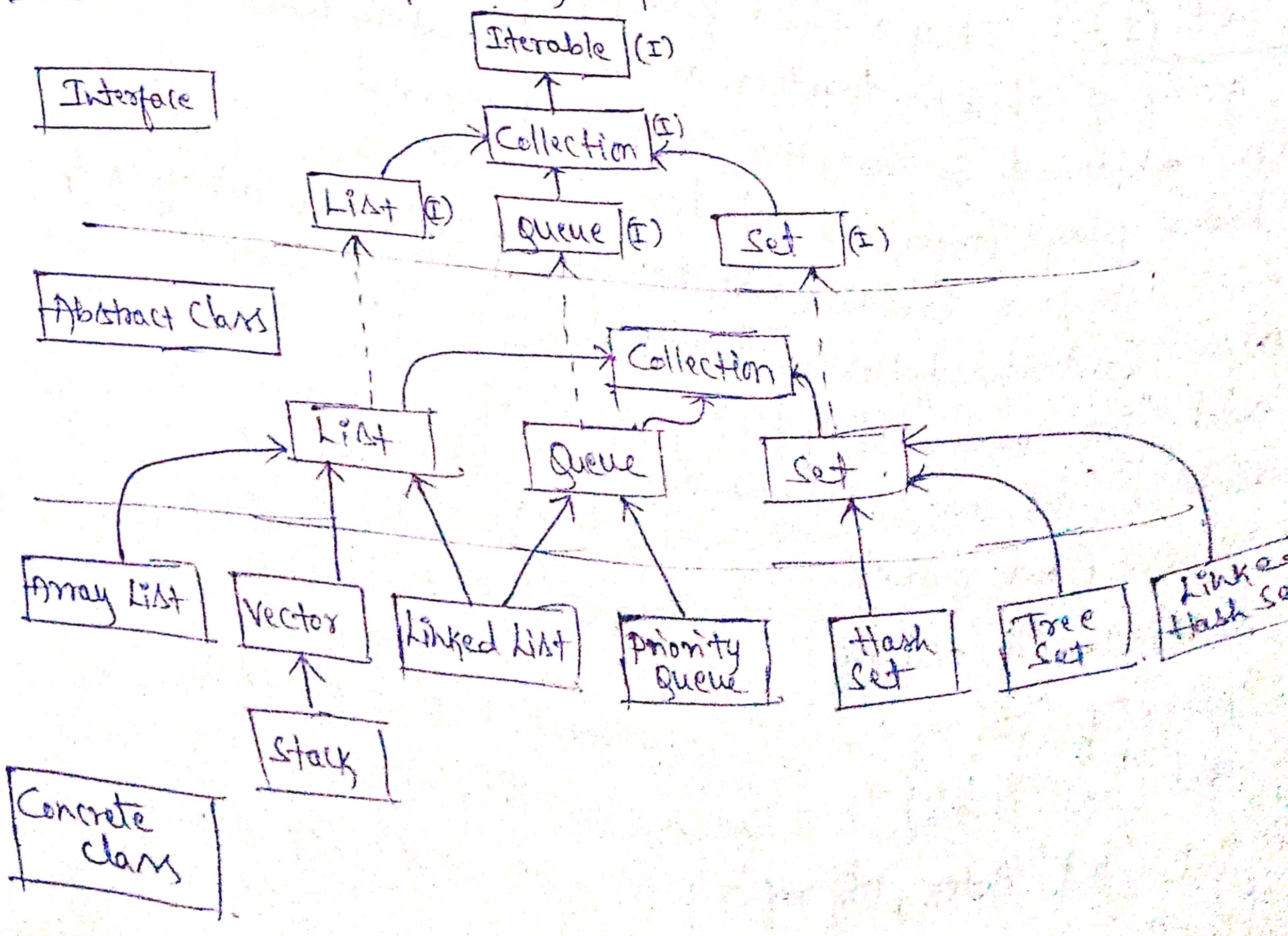
Collection Framework :

* The set of built in classes & interfaces provided by java inorder to represent group of objects as single entity is called Collection framework.

Need of Collection Framework (Advantages of Collection Over Array)

* Collection Can store both homogeneous & hetrogeneous objects.
* We Can increase (or) decrease the size of Collection
* Collection provides many built in methods which reduces programmers efforts.

Note: Collection accepts only objects



Collection (I) :- It identifies most common methods for all Collection objects.

1. isEmpty()
2. add (obj ref)
3. addAll (Collection ref)
4. Size()
5. remove (obj ref)
6. Contains (obj ref)
7. removeAll (Colle^n ref)
8. retainAll (Colle^n ref)
9. ContainsAll (Colle^n ref)
10. Clear()
11. toArray()
12. iterator()

List (I) :- It is a child of Collection. It is used to Store group of objects together as a single entity where duplicates are allowed & insertion order is preserved.

* Index plays important role here.
* List interface Contains all methods of Collection Interface & also Contains following methods.

1. add (int index, obj ref)
2. addAll (int index, Collection ref)
3. remove (int index)
4. lastIndexOf (obj ref)
5. indexOf (obj ref)
6. get (int index)
7. set (int index, obj ref)
8. listIterator()

Array List : It is a Concrete Subclass of List Interface.

Characteristics of Array List :

* Accepts duplicate objects
* it maintains insertion order
* Iteration order predictable
* dynamically growable (or) Shrinkable
* Accepts both homogenous & Hetrogeneous.

Steps to Create Array List

* Import arraylist class from " java.Util. package
* Create an object.

Constructors of Array List

ArrayList()
ArrayList (Collen ref)
ArrayList (int initialcapacity)

ex:
```
import java.Util.ArrayList;
Class A
{
    public Static Void main(String[] args)
    {
        ArrayList a1=new ArrayList();
        S.o.pln (a1.isEmpty());  //true
        a1.add(10); //adds integer value i.e 10.
        a1.add(20);
        a1.add("Hello"); //adds String Hello
        a1.add (true); //adds boolean true
        a1.add (10);
        S.o.pln (a1.isEmpty()); //false
        S.o.pln (a1); // Views arrays list // [10, 20, Hello, true, 10
        S.o.pln (a1.size()); //5
    }
}
```

ex:

```
import java.Util.ArrayList;
class B
{
    p.s.v.m()
    {
        ArrayList a1=new ArrayList();
         a1.add(10);
         a1.add(20);
         a1.add(30);
         S.o.pln(a1); // [10, 20, 30]
        ArrayList a2=new ArrayList();
         a2.add(40);
         a2.add(50);
         S.o.pln(a2); // [40, 50]
        a2.add(a1); //adds a1 arraylist to a2
        S.o.pln(a2.size()); //3
        S.o.pln(a2); // [40, 50, [10, 20, 30]]
                                  Considers as 1 value
        a2.addAll(a1);
        S.o.pln(a2.size()); //5
        S.o.pln(a2); // [40, 50, 10, 20, 30]
    }
}
```

→ Note: We can add elements of one arraylist into another with the help of add() (or) addAll() method.

* add() method adds the arraylist as a single object
* addAll() adds elements of arraylist as seperate objects.

ex: import java.Util.ArrayList;

```
class C
{
    p. s. v. m()
    {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add("Hello");
        a1.add(12.6);
        a1.add(12);
        s.o.pln(a1.get(0)); || gives one element
        for(int i=0; i<a1.size(); i++)
        {
            s.o.pln(a1.get(i));
        }
    }
}
```

Note: We can insert the element into ArrayList in the middle with the help of add() which accepts index as well as object

```
import java.Util.ArrayList;
class A
{
    p. s. v. m()
    {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(10);
        a1.add(10.5);
        a1.add("hello");
        a1.add(false);
        s.o.pln(a1); || [10, 10, 10.5, Hello, false]
        s.o.pln(a1.size()); || 5
        a1.add(3, 100); || adding 100 to 3rd index (or) position
        s.o.pln(a1); || [10, 10, 10.5, 100, Hello, false]
        s.o.pln(a1.size()); || 6
    }
}
```

Removing Element from ArrayList :-

* We can remove an element from arraylist using remove ()
* remove() is an Overloaded method.

```
remove (int index)
remove (Object ref)
```

Ex:

```
import java.Util.ArrayList
class E
{
    p. s. v. m()
    {
        ArrayList a1 = new ArrayList();
        a1.add (10);
        a1.add (20);
        a1.add (30);
        a1.add (40);
        a1.add ("hello");
        a1.add (15.6);
        S.o.pln (a1); // [10, 20, 30, 40, hello, 15.6]
        ~~S.o.pln~~ a1. remove (15.6);
        S.o.pln (a1); // [10, 20, 30, 40, hello]
        a1.remove ("hello");
        S.o.pln (a1); // [10, 20, 30, 40]
        a1.remove (1); //remove (int index) method called // Index 1 Value removed
        S.o.pln (a1); // [10, 30, 40]
        a1. remove (new Integer(30));
        S.o.pln (a1); // [10 40]
    }
}
```

Generic ArrayList:

is used to store homogeneous objects.

Syntax: ArrayList<type> ref_Var = new ArrayList<>();

ex: import java.Util.ArrayList;

```
class F
{
    p.s.v.m()
    {
        ArrayList<Integer> a1 = new ArrayList<>();
        a1.add(10);
        a1.add(10);
        a1.add(100);
        a1.add(200);
        // a1.add("hello"); // CTE
        for(int i=0; i<a1.size(); i++)
        {
            S.o.pln(a1.get(i));
        }
    }
```

for-each (enhanced for loop): used to access the elements from ArrayList

Syntax: for(object ref1: Collection ref2)
{
    Statements;
}

ex: import java.Util.ArrayList;

```
class G
{
    p.s.v.m()
    {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.add("hi");
        a1.add(false);
```

```
        for(object i : a1)
        {
            s.o.pln(i);
        }
    }
}
```

for generic arrayList

ex:

```
import java.Util.ArrayList;
class H
{
    p.s.v.m()
    {
        ArrayList<Integer> a1=new ArrayList<>();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.add(40);
        for(Integer i : a1)
        {
            s.o.pln(i);
        }
    }
}
```

removeAll():-

removeAll() is used to remove elements from the Current arraylist. It remove those elements which are present in the ArrayList given as argument.

ex:

```
import Java.Util.ArrayList;
class J
{
    p.s.v.m()
    {
        ArrayList a1=new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30)
        a1.add(40);
```

```
ArrayList a2 = new ArrayList();
    a2.add(10);
    a2.add(20);
    a2.add(30);
    a2.add(100);
    a2.add(200);
  S.o.pln("ArrayList a1:" +a1);// [10,20,30,40]
  S.o.pln(a2);// [10,20,30,100,200]

    a2.retainAll(a1);
    S.o.pln(a1);// [10,20,30,40]
    S.o.pln(a2);// [10,20,30]
}
```

Sorting ArrayList:

We Can Sort Arraylist with help of Sort Method of Collections class

* "Collections" is a class present in java.Util. package,
  'Sort(List ref)' is a Static method of Collections class.
* It also has another method Called "reverse(list ref)", which can be used to reverse the ArrayList:

ex-
```
import java.Util.ArrayList;
import java.Util.Collections;
class L
{
    P.S.V.M()
    {
        ArrayList a1=new ArrayList();
          a1.add(4);
          a1.add(15);
          a1.add(6);
          a1.add(123);
          a1.add(76);
        S.o.pln(a1);// [4,15,6,123,76]
        Collections.Sort(a1);
        S.o.pln(a1);// [4,6,15,76,123]
        Collections.reverse(a1);
        S.o.pln(a1);// [123,76,15,6,4]
    }
}
```

Note: To sort, elements of ArrayList must be homogeneous & 'Comparable type'.

Comparable Interface:

* It is an interface having a non-static abstract method 'CompareTo(Object ref)'

* The 'Collections.Sort()' uses 'CompareTo()' internally to sort the elements of ArrayList. ∴ the objects present inside the ArrayList must be Comparable type. i.e the class for which the objects are created must implement Comparable interface & provide implementation to CompareTo() method.

Note: All wrapper classes by default implements Comparable interface & provides implementation to CompareTo() method.

ex: Class Emp implements Comparable
{
    String name;
    int eid;
    double Salary;
    Emp(String name, int eid, double Salary)
    {
        this.name = name;
        this.eid = eid;
        this.Salary = Salary;
    }
    //overriding toString() to get name of the employees instead of reference
    // when we try to print view of ArrayList.
    public String toString()
    {
        return name;

```
public int CompareTo(object obj)
{
    if(this.eid == ((Emp)obj).eid)
    {
        return 0;
    }
    else if(this.eid > ((Emp)obj).eid)
    {
        return 1;
    }
    else
    {
        return -1;
    }
}
import java.Util.ArrayList;
import java.Util.Collections;
class Driver1
{
    p.s.v.m()
    {
        ArrayList e=new ArrayList();
        e.add(new Emp("Sanjay",5,50000.0));
        e.add(new Emp("Ravi",2,50000.0));
        e.add(new Emp("Dolly",3,50000.0));
        s.o.pln(e); // [Sanjay, Ravi, Dolly]
        s.o.pln("After Sorting based on eid");
        Collections.Sort(e);
        s.o.pln(e); // [Ravi, Dolly, Sanjay]
    }
}
```

Contains(Object ref):

* It used to check whether given element if present in the Collection or not.

ex:
```
import java.Util.ArrayList;
class M
{
    P.S.V.M()
    {
        ArrayList a1=new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.add(40);
        S.o.pln(a1.Contains(10));//true
    }
}
```

ContainsAll(Collection ref)

* It used to check whether the ArrayList Contains all the elements of given arraylist.

ex.
```
Import java.Util.ArrayList;
class M
{
    P.S.V.M()
    {
        ArrayList a1=new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.add(40);
        ArrayList a2= new ArrayList();
        a2.add(10);
        a2.add(20);
        S.o.pln(a1.Contains(a2));//true
    }
}
```

clear()

* It is used to remove all the elements from the arraylist.

```
import java.Util.ArrayList;
class M
{
    p. s. v. m()
    {
        ArrayList a1=new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.clear();
        s.o.pln(a1.size());//0
    }
}
```

indexOf(object ref)

* It gives the index of given Object (first Occurence).

lastIndexOf(Object ref)

* It gives the index of given Object (last occurence)

ex:

```
import java.Util.ArrayList;
Class M
{
    P. S. V. M()
    {
        ArrayList a1=new ArrayList();
        a1.add(10);      a1.add(40);
        a1.add(20);      a1.add(30);
        a1.add(30);
        s.o.pln(a1.indexOf(30));//2
        s.o.pln(a1.lastIndexOf(30));//4
    }
}
```

Set(int index, Object ref):

* It used to replace the element present at specified index with the given element.

ex:

```
import java.Util.ArrayList;
class M
{
    p.s.v.m()
    {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(20);
        a1.add(30);
        a1.add(40);
        S.o.pln(a1); // [10,20,30,40]
        a1.set(1,100);
        S.o.pln(a1); // [10,100,30,40]
    }
}
```

Emp Pogm

Linked List :-

* Linked list is Concrete Subclass of List Interface.
* It belongs to java.Util package

Characteristics of Linked list

* Maintains the insertion order
* Iteration Order is predictable
* Dynamically growable (or) Shrinkable
* Accept duplicate objects
* Can Store both homogeneous as well as heterogeneous objects.

Steps to Create a Linked list

* import linked list Class from java.Util package
* Create an object using the Constructors.

Constructors of linked list

1. LinkedList()
2. LinkedList(Collection ref)

Linked List Specific Methods :

1. addFirst (Object ref)
2. addLast (Object ref).
3. getFirst()
4. getLast()
5. removeFirst()
6. removeLast()

ex:

```
import java.Util.LinkedList;
class A
{
    p.s.v.m()
    {
        LinkedList l1 = new LinkedList();
        l1.add(10);
        l1.add(20);
        l1.add(30);
        l1.add(30.5);
        l1.add("hello");
        l1.add(20);

        s.o.pln(l1); //[10,20,30,30.5,hello,20]
        s.o.pln(l1.size); //6

        //Accessing elements of linked list
        for(Object i)
        {
            s.o.pln(i);
        }
        l1.addFirst(100);
        l1.addLast(1000);
        s.o.pln(l1); //[100,10,20,30,30.5,hello,20,1000]

        s.o.pln("First Object:" + l1.getFirst()); //100
        s.o.pln("Last Object:" + l1.getLast()); //1000

        l1.removeFirst();
        l1.removeLast();
        s.o.pln(l1); //[10,20,30,30.5,hello,20]
    }
}
```

Constructors of ArrayList

① ArrayList():

* It Creates an empty ArrayList with initial Capacity 10

② ArrayList(int initialCapacity):

* It used to Create an ArrayList with given initial Capacity.

③ ArrayList(Collection ref)

* It is used to Convert any Collection type to arraylist

ex:
```
LinkedList l1 = new LinkedList();
l1.add(10);
l1.add(20);
l1.add(30);
```
Collection ref

```
ArrayList a1 = new ArrayList(l1);
S.o.pln(l1); // [10, 20, 30]
```

Note: The underlying datastructure for ArrayList is Resizable array

```
ArrayList a1 = new ArrayList();
```
Whenever we Create an ArrayList Using no argument Constructor, An empty arraylist object is Created.

Default initial Capacity of this ArrayList is 10.

* Once the maximum Capacity reaches, A bigger ArrayList is going to be Created & all the elements of previous arraylist will be Co to new arraylist & new element also gets added & the reference Variable Changes (refers) to new ArrayList Obje

* What is the new Capacity of ArrayList?

new Capacity = (Current Capacity * 3/2) + 1

ex: new Capacity = $(10 * \frac{3}{2}) + 1$

= 16

ArrayList (int initialCapacity) Constructor

* Suppose we want to Create an Array List to Store 100 objects & if we Create arraylist using no argument Constructor (ArrayList()). First an arraylist with default Capacity 10 will be Created & then each time maximum Capacity reaches new arraylist will be Created. This reduces performance.

∴ It is better to Create arraylist of initial Capacity 100 using ArrayList(int initialCapacity) Constructor.

Diff b/w ArrayList() & Linked List()

ArrayList()	Linked List()
* Internal Datastructure used is ~~called~~ arraylist is resizable (or) growable array	* Internal Datastructure used for linked list is "Doubly linked list"
* Arraylist is recommended with the frequent operation for the Collection is to accessing the elements.	* linked list is recommended from the frequent operation for the Collection is Insertion (or) Deletion of the element from the

Vector :-

* It is a Concrete Subclass of List interface & it belongs to java.util. package.

Characteristics of Vector

* Maintains the insertion order
* Iteration order is predictable
* Dynamically growable (or) Shrinkable
* Accepts duplicate objects
* Can store both homogeneous as well heterogeneous objects.

Note: Vector Came before ArrayList, List, & Collection (I), i.e java 1

After java 1.2 Vector is the Child of Collection (I) & List (I).

∴ it has 3 methods for adding an element.

1. add(object ref) → from Collection (I)
2. add (int index, Object ref) → from List (I)
3. addElement (Object ref) → Vector

for removing

1. remove (object ref) → from Collection (I).
2. remove (int index) → from List (I).
3. removeElement (object ref) → from Vector.
4. clear () → from Collection (I).
5. removeAllElements() → from Vector.

for Accessing elements

1. get (int index) → from List (I)
2. elementAt(

eg import pre.util.Vector;

```
class {
    p s v m ()
    {
        Vector v = new Vector();
        v.addElement(10);
        v.addElement(100);
        v.add(200);
        s.o.pln(v); // [10,100,200]
        v.removeElement(100);
        s.o.pln(v); // [10,200]
        v.removeAllElements();
        s.o.pln(v); // [ ]
        v.add(25);
        v.add(250);
        v.add(2580);
        s.o.pln(v.firstElement()); //25
        s.o.pln(v.lastElement()); //2580
        s.o.pln(v.elementAt(2)); //2580
    }
}
```

* wap to Create an Arraylist store some string & print only the string. starting with lower case 'a' & ending with lowercase 'b'.

```
import java.Util.ArrayList;

class A
{
    public static Void main (String[] args)
    {
        ~~Arraylist a1 = new~~
        ArrayList <String> a1 = new Arraylist <>();
        a1.add ("abhi");
        a1.add ("abcdb");
        a1.add ("Dhoni");
        a1.add ("Jon Snow");

        for(String s : a1)
        {
            if (s.charAt(0)=='a' || s.charAt(s.length()-1)=='b')
            {
                s.o.pln(s);
            }
        }
    }
```

(or)

```
for(int i=0 ; i<a1.size(); i++
{
    String s = a1.get(i);
    if(s.charAt(0)=='a' || s.charAt(s.length()-1
    {
        s.o.pln(s);
    }
}
```

Set

* is an Interface & belongs to java.Util package
* The Implementing Classes of Set are
  * HashSet
  * Tree Set
  * Linked Hash Set

HashSet

* is Implementing Class of Set Interface & it belongs to java.Util pack

Characteristics of Hash Set

* It will not accept duplicate Objects
* It will not maintain insertion order
* We Can't predict the iteration order
* We add (or) remove elements
* It Can accept both homogeneous & heterogeneous Objects

ex:

```
import java.Util.HashSet;
Class A
{
  p.s.v.m()
  {
    Hashset h1 = new Hashset();
    S.o.pln(h1.add(10)); //True
    S.o.pln(h1.add(20)); //True
    S.o.pln(h1.add(20)); // false
    S.o.pln(h1.add(1.5)); //True
    S.o.pln(h1.add("hello")); // true
    S.o.pln(h1); // randomly views the values of HashSet
    S.o.pln(h1.size()); //4
  }
}
```

Note: return type of add() is boolean, it returns true when the object is added to Collection, otherwise it returns false.

* get() method can be used only for list type of Collection. If we want to fetch elements of Hashset one by one we use for each loop.

ex:
```
Hashset<Integer> h1 = new Hashset<>(); // Generic Hashset
h1.add(80);
h1.add(50);
h1.add(20);
h1.add(20);
h1.add(200);
for(Integer i : h1)
{
    S.o.pln(i);
}
```

Sorting elements of Hashset

* We cannot sort hashset directly, if we want to display the elements of the hashset in sorted order we can convert it into ArrayList & sort the ArrayList (or) we can convert it into Treeset.

ex:
```
import java.Util.ArrayList;
import java.Util.Hashset;
import java.Util.Collections;
class A
{
    P.S.V.M()
    {
        Hashset<Integer> h1 = new Hashset<>();
        h1.add(80);
        h1.add(50);
        h1.add(200);
        ArrayList a1 = new ArrayList(h1); // Converting Hashset to ArrayList
        Collections.Sort(a1);
        for(Object i : a1)
        {
            S.o.pln(i);
        }
    }
}
```

Tree Set :

* It an implementing class of Set interface & belongs to java.util

characteristics of Treeset

* It will not accept duplicate objects
* It will not maintain insertion order
* It by defaults stores the elements in ascending order
* We can predict the iteration order.
* We can add or remove objects
* It can accept only homogeneous objects.
(because default it arranges in ascending order)

ex:
```
import java.util.Treeset;
Treeset t1=new Treeset();
t1.add(100);
t1.add(20);
t1.add(30);
t1.add(30); // it will not accept duplicate values
t1.add(300);
S.o.pln(t1); // [20, 30, 100, 300]
```

Linked Hash Set :

* is an implementing class of Set interface & belongs to java.Util package.

characteristics of Linked Hash Set

* It will not accept duplicate objects
* It maintains the insertion order
* We can predict iteration order
* We can add (or) remove elements
* It can accepts homogeneous & heterogeneous objects.

ex:
```
import java.Util.LinkedHashSet;
class A
{
    p.s.v.m()
    {
        LinkedHashSet l1=new LinkedHashSet();
        l1.add(14);
        l1.add(114);
        l1.add(58);
        l1.add(58);
        l1.add(580);
        S.o.pln(l1); // [14, 114, 58, 580]
        S.o.pln(l1.size()); //4
    }
}
```

Note: when we use linkedlist as queue it follows first-in-first out order means the element added first will be removed first.

Removing elements One by One from Queue

```
LinkedList l1 = new LinkedList();
    l1.offer(3);
    l1.offer(30);
    l1.offer(14);
    l1.offer(4);
    l1.offer(45);

while (!(l1.isEmpty()))
{
    S.o.pln(l1.poll()); // 3, 30, 14, 4, 45   ← removing order.
}
S.o.pln(l1); // [ ] — empty
```

(ii) Priority Queue :-

* is an implementing class of Queue interface

Note: It removes the elements based on the priority means ascend (or) descending order.

ex:

```
import java.Util.PriorityQueue;
class A
{
    P.S.V.m()
    {
        PriorityQueue p1 = new PriorityQueue();
        p1.offer(14);
        p1.offer(5);
        p1.offer(140);
        p1.offer(86);
        S.o.pln(p1); //randomly displays.
        S.o.pln(p1.poll()); // 5
        S.o.pln(p1.peek()); // 14.
    }
}
```

Queue :

* is an interface & belongs to java.Util package.
* Implementing classes of Queue are
  (i) linked list
  (ii) Priority Queue

(1) Using linked list as Queue :

* linked list is an implementing class of Queue interface & belongs to java.Util package.
* Important methods
  (i) offer() :-
  * It used to add an element to Queue
  (ii) poll() :-
  * It used to remove an element from queue
  (iii) peek() :-
  * It returns the element which is ready to be removed.

ex:

```
import java.Util.LinkedList;
class A
{
    p.s.v.m()
    {
        LinkedList l1=new LinkedList();
        l1.offer(3);
        l1.offer(30);
        l1.offer(14);
        l1.offer(4);
        l1.offer(45);
        S.o.pln(l1); // [3, 30, 14, 4, 45]
        S.o.pln(l1.poll()); // 3 <= removed
        S.o.pln(l1.peek()); // 30 <= ready to be removed
        S.o.pln(l1); // [30, 14,
```

Note: when we use linkedlist as Queue it follows first-in-first out order means the element added first will be removed first.

Removing elements One by One from Queue

```
LinkedList l1 = new LinkedList();
    l1.offer(3);
    l1.offer(30);
    l1.offer(14);
    l1.offer(4);
    l1.offer(45);
while (!(l1.isEmpty()))
{
    S.o.pln(l1.poll()); // 3, 30, 14, 4, 45   ← removing order
}
S.o.pln(l1); // [ ] — empty
```

(ii) Priority Queue :-

* is an implementing class of Queue interface

Note: It removes the elements based on the priority means ascend (or) descending order

ex

```
import java.Util.PriorityQueue;
class A
{
    P.S.V.m()
    {
        PriorityQueue p1 = new PriorityQueue();
        p1.offer(14);
        p1.offer(5);
        p1.offer(40);
        p1.offer(86);
        S.o.pln(p1); // randomly displays
        S.o.pln(p1.poll()); // 5
        S.o.pln(p1.peek()); // 14
    }
}
```

Removing elements from Priority Queue

```
PriorityQueue p1 = new PriorityQueue();

    p1.offer(14);
    p1.offer(5);
    p1.offer(140);
    p1.offer(86);

    while(!(p1.isEmpty()))
    {
        S.o.println(p1.poll()); // 5, 14, 86, 140
    }
```

removing in Ascending order

Removing elements from priority queue in descending order

```
class MyComparator implements Comparator
{
    public int Compare(Object obj1, Object obj2)
    {
        int i1 = (Integer)obj1;
        int i2 = (Integer)obj2;

        if(i1 == i2)
        {
            return 0;
        }
        else if(i1 > i2)
        {
            return -1;
        }
        else
        {
            return 1;
```

```
import java.Util.PriorityQueue;
class J
{
    p.s.v.m()
    {
        priority. p1 = new PriorityQueue(new MyComparator());
        p1.offer(12);
        p1.offer(122);
        p1.offer(152);
        p1.offer(14);
        p1.offer(6);
        while(!(p1.isEmpty()))
        {
            S.o.pln(p1.poll());
        }
    }
}
```

Comparator :- is an interface which belongs to java.util package

* It has an abstract method.

i.e public int Compare (Object Obj1, Object Obj2)

* Comparator is used for Customized Sorting as shown in below

ex: Sorting elements of ArrayList in descending order of Integer using Comparator.

```
class MyComparator implements Comparator
{
//implemention given to get descending order of integers.
public int Compare (Object Obj1, Object Obj2)
{
    int i1 = (Integer)Obj1;
    int i2 = (Integer) Obj2;
    if (i1 == 2)
    {
        return 0;
    }
    else if (i1 > i2)
    {
        return -1;
    }
    else
    {
        return 1;
    }
}
}
import java.util.ArrayList;
import java.util.Collections;
class H
{
    P s. v. m ()
    {
        ArrayList <Integer> a1 = new ArrayList<>();
        a1.add (14);
        a1.add (44);
        a1.add (7);
        a1.add (77);
        a1.add (17);
        Collections.Sort (a1, new MyComparator());
        S.o.pln (a1); // [44, 77, 17, 14, 7]
```

Map :-

* is an interface in java, which is present in java.Util package
* The implementing classes of Map are
  * HashMap
  * Tree Map
  * Linked Hash Map
* Map is used to store the data in Key-Value pairs

ex: Key → Roll No - Student ← value
Empid - Employee

* Each Key-value pair in the map is called an "entry".

characteristics of Map :-

* Map Cannot have duplicate keys
* Map Can have duplicate Values
* Each Key Can have atmost one Value not more than that

HashMap :-

* is an implementing class of Map interface, it belongs java.Util package.

characterstics of HashMap :

* It Can have heterogeneous Key-Value pairs
* It will not maintain the insertion order
* We Cannot predict the iteration order.

important methods of Map

* put() :-
* putall()
* ContainsKey(key)
* ContainsValue(Value)
* KeySet()
* Values()
* isEmpty()
* EntrySet()
* Size()
* Clear()
* Remove(Key)
* get(Key)

ex import java.Util.HashMap;
class A
{
p s v m ()
{
HashMap h1=new HashMap();
Key h1.put (1, "hello"); Values
h1.put (5, "abc");
h1.put ("hi", 15.5);
h1.put (15, true);
S.o.pln (h1); // {1=hello, hi=15.5, 5=abc, 15=true}
h1.put (6, "hello");
S.o.pln (h1); // {1=hello, hi=15.5, 5=abc, 6=hello, 15=true}
h1.put (5, "xyz");
S.o.pln (h1); // {1=hello, hi=15.5, 5=xyz, 6=hello, 15=true}
}
}

ex class A
{
p s v m ()
{
~~HashMap h1= new Hash~~
HashMap <Integer, String> h1=new HashMap<>();
h1.put (1, "Sandeep");
h1.put (2, "Vicky");
h1.put (3, "Minal");
h1.put (4, "Balaya");
S.o.pln (h1); // {1=Sandeep, 2=Vicky, 3=Minal, 4=Balaya}
S.o.pln (h1.size()); //4
S.o.pln (h1.get(3)); //Minal
S.o.pln (h1.containskey(2)); //true
S.o.pln (h1.containsValue("Sandeep")); //true
h1.remove (3);
S.o.pln (h1); // {1=Sandeep 2=Vikay 4=Balaya}
S.o.pln (h1.size()); //3
h1.clear();
S.o.pln (h1); // {}
}

To extract only Keys from the Map:

* We can extract only Keys from the map with the help of KeySet()
return type of KeySet() is Set

ex:
```
HashMap<Integer, String> h1=new HashMap<>();
h1.put(1, "Sandeep");
h1.put(2, "Vikash");
h1.put(3, "Mrinal");
h1.put(4, "Rahul");
Set s=h1.KeySet();
for(Object i:s)
{
    s.o.pln(i);
}
```

To extract only Values from the Map:

* We can extract only Values from the map with the help of Values()
return type of Values() is Collection.

ex:
```
HashMap<Integer, String> h1=new HashMap<>();
h1.put(1, "Sandeep");
h1.put(2, "V.");
h1.put(3, "M");
h1.put(4, "B");

~~Set~~ Collection c=h1.Values();

for(object i:c)
{
    s.o.pln(i);
}
```

To extract Key-Value pair (Tree) from Map.

* We should use entrySet() in the for each loop as shown in below

ex:
```
HashMap<Integer, String> h1 = new HashMap<>();
h1.put(1, "A");
:
for(Map.Entry<Integer, String> i : h1.entrySet())
{
    S.o.pln(i);
}
```

o/p
1 = A
2 = B

* Each Key-Value pair (entry) in the map is of Map.Entry type

Entry ⇒ is a nested interface, which is present inside Map interface

* Entry interface provides getKey() & getValue() using we can access Keys & Values seperately.

ex:
```
for(Map.Entry<Integer, String> i : h1.entrySet())
{
S.o.pln(i.getKey()+"----->"+i.getValue());
}
```