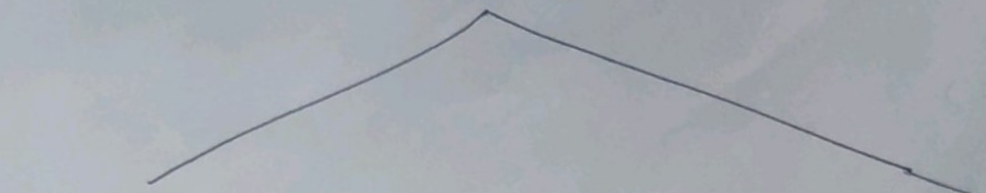
POLYMORPHISM



COMPILE TIME

- EARLY BINDING
- STATIC BINDING
- METHOD OVERLOADING

RUN TIME

- LATE BINDING
- DYNAMIC BINDING
- METHOD OVERRIDING

28.03.23

An object showing different behaviour in its life cycle is known as polymorphism.

There are 2 types of polymorphism

* compile time
* Run-time

COMPILE TIME POLYMORPHISM:

* COMPILE Time polymorphism the method declaration will binds with method definition by the compiler at compile time.

* The compile time polymorphism is also known as early binding (or) static binding. Before the execution the method declaration will binds with method definition hence it is called as early binding.

* once the declaration is binded with a definition it cannot be rebinded hence

it is called as static binding.

The best example to compile time polymorphism is method overloading.

RUN-TIME POLYMORPHISM:

* In run-time polymorphism the method declaration will binds with method definition by the JVM at runtime.

* TO achieve runtime polymorphism, the following things are mandatory.

  * Inheritance
  * Method overriding
  * Upcasting.

* Runtime polymorphism it is also known as late binding (or) dynamic binding.

* At the time of execution the method declaration will binding with method definition, hence it is called as late binding.

* Once the declaration is binded with definition it be rebinded hence it is called as dynamic binding.

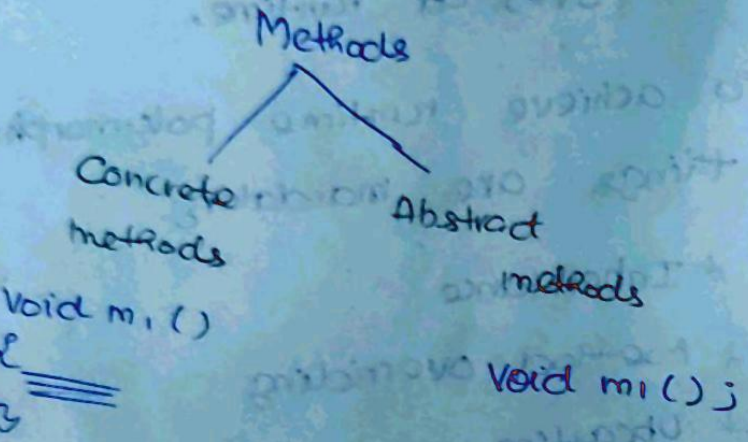
ABSTRACT CLASS

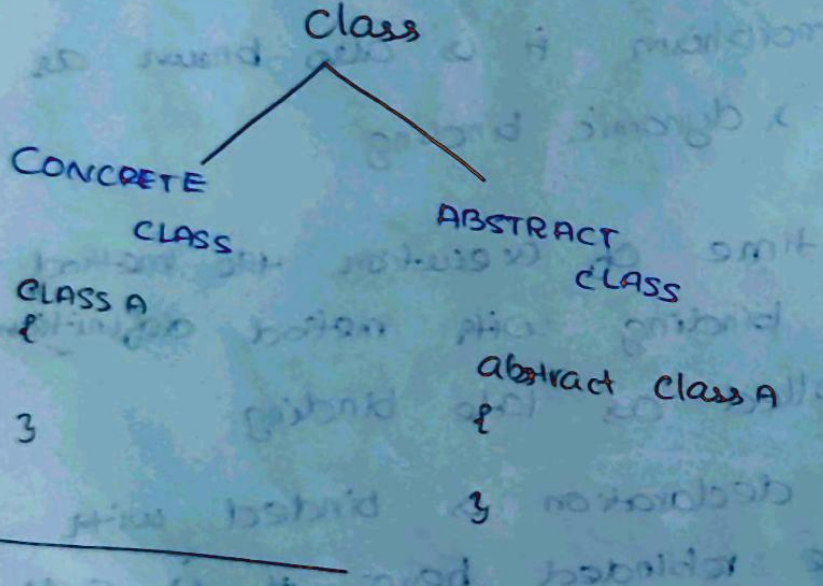
modifier returntype methodname (arguments)

↘ Method declaration

```
{
    return value;
}
```

→ method definition.





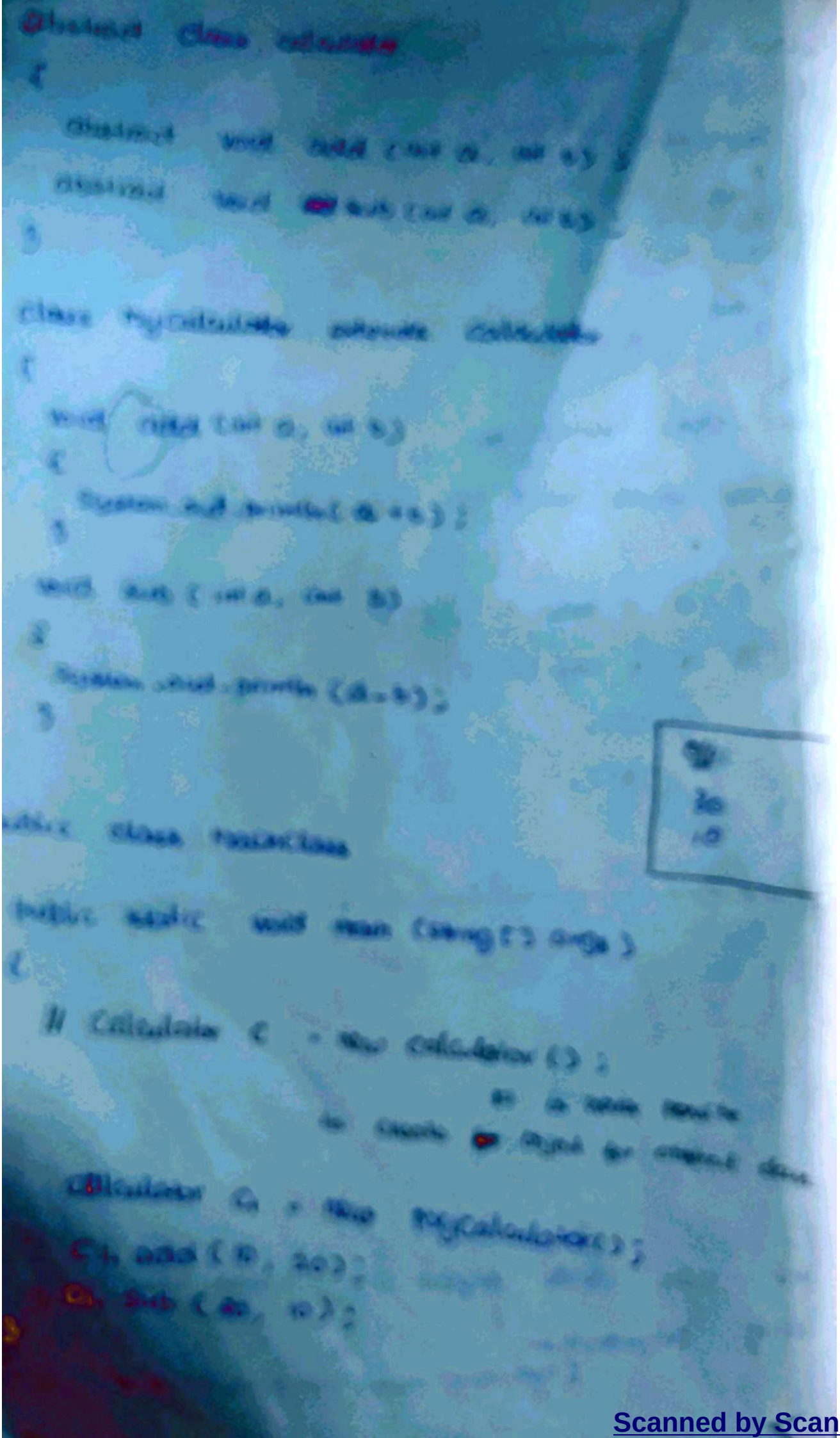

```
abstract class A
{
    abstract void m1();
    abstract void m2();
}
```

```
class B extends A
{
    void m1()
    {
    ==
    }
    void m2()
    {
    ==
    }
}
public class mainclass
{
  public static void main(String[] args)
  {
      A a1 = new A(); X
      A a1 = new B();
        a1.m1();
        a1.m2();
  }
}
```

---

we can create object for abstract class by using polymorphism.

[Inheritance, method overriding, upcasting]



NOTE :

* If method contains both declaration & definition then it is CONCRETE METHOD.

* If a method contains only the declaration then it is ABSTRACT METHOD. The abstract method must be defined with ABSTRACT keyword.

* If a class contains only concrete methods, then it is CONCRETE CLASS.

* Any class if it is defined with ABSTRACT keyword, then it is ABSTRACT CLASS. Inside the Abstract class we can have only concrete methods (or) only abstract methods (or) both.

→* It is never possible to create the object for abstract class.

→* We can inherit from an abstract class by using extends keyword.

→* While inheriting from an abstract class, we must override all the inherited abstract method (or else) we must define the class as abstract.

→* We can create the obj for subclass we can store it in abstract class reference variable (upcasting).

* The following keyword combinations are not possible. private abstract, static abstract, final abstract.

When to define the method as abstract?

Whenever we want to override, the method in the subclass mandatorily, then we can define those methods as "ABSTRACT"

When to define the class as abstract?

Whenever a class contains single abstract method, we can define the class as abstract.

Can we define main method as abstract?

Because main method is static.

Concrete class	Abstract class
It is possible to create an object	It is never possible to create an object.

One and only difference

Can we have an Constructor inside an abstract class?

Yes. We cannot create object, but it can be used to achieve constructor chaining.

```
public ~~class~~ abstract class Account
{
    int accNo;
    double accbal;

    account (int accno, double accbal)
    {
      system.out.println ("Account created successfully...");
      this.accno = accno;
      this.accbal = acc bal;
    }

    abstract void deposit (double amt);
    abstract void withdraw (double amt);
    final void balanceenquiry()
    {
      S.O.Pln ("your account balance...");
    }
}
```

29.03.2

Interface

Interface Interfacename

{

→ data members are static and final.

→ function members are abstract

→ no blocks / constructors / concrete methods

→ default access is public

}

```
interface A
{
    int x = 10;
    void m1();
    void m2();
}

Class B implements A
{
 Public void m1()
 {
 }

 public void m2()
 {
 }

 S.O.Ptln
}
```

```
public class mainclass
{
  public static void main (String [] args)
  {
    S.O.ptln (A.x)
    A a1 = new A(); X
    A a1 = new B()
      a1. m1();
      a1 . m2();
  }
}
```

eg.

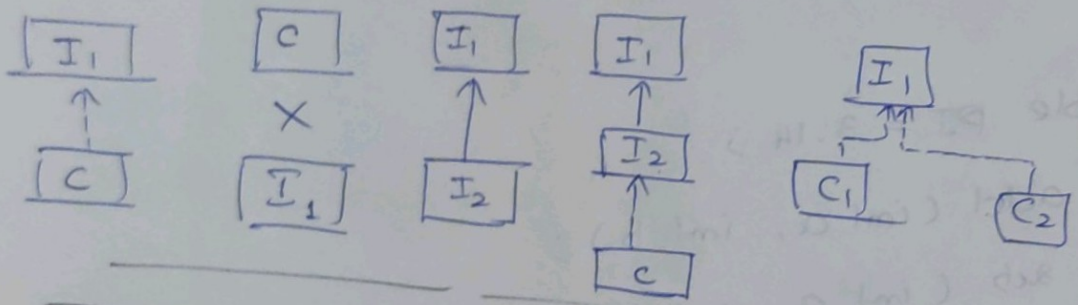
```
interface calculator
{
  double PI = 3.14 ;
  int add (int a, int b) ;
  int sub ( int a, int b);
}

Class Mycalculator implements calculator
{
  public int add (int a, int b)
  {
    return a+b ;
  }
  public int sub (int a, int b)
  {
    return a-b ;
  }
```

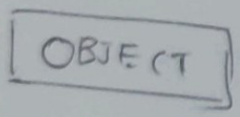
```
public class mainclass
{
  public static void main (String [] args)
  {
    System.out.println ("pi value = " + PI );

    calculator c1 = new mycalculator(); //upcasting
s.o.pln(c1.add (10, 20));

    System.out.println ( c1.sub ( 20, 10 ) );
  }
}
```

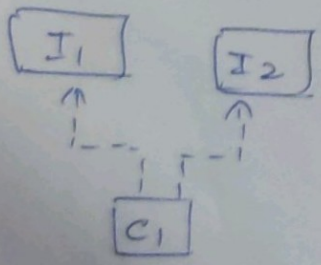
O/P

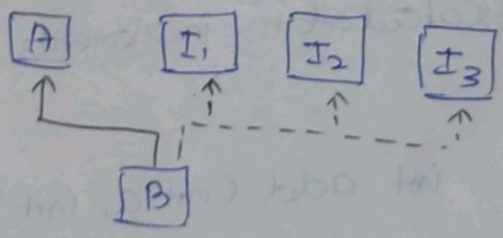
30

10









Class C1 implements I1, I2

Class B extends A
implements I1, I2, I3.....

∴ Java supports multiple inheritance through interface. It figurectly all the three problems.

INTERFACE :

* Interface is a definition block which is used to achieve pure abstraction.

* We can define an interface by using "Interface" keyword".

④* Inside the interface, the data members are static and final [global constants], The function members are by default abstract, we can't define blocks (or) constructors (or) concrete methods inside the interface.

③* The default access inside the interface is public.

* We can access datamembers of an interface by static reference syntax. (i.e)

Interface name . data member

* It is never possible to create an object for interface.

* We can inherit from an interface by implements keyword.

* When we are inheriting from an interface we must override all the abstract methods (or else) we must define the implementation class as "Abstract".

* We can create the object of implementation class. We can store it in interface reference variable.

* We can inherit from interface to an interface by extends keyword.

* It is never possible to inherit from clause to an interface.

(*) ⇒ Java supports multiple inheritance through interfaces.

* A class can extend from only one class but it can implement from multiple interfaces.

Advantages of Interfaces:

* We can achieve pure abstraction
* We can achieve multiple inheritance.
* We can achieve generalization.

What is generalization?

~~covering~~ ~~Subclass type~~

subclass type into super class type

ABSTRACTION

* Hiding the implementation details & showing only necessary behaviours is known as Abstraction.

* We can achieve abstraction either by an abstract class (or) by using interface.

* We use interfaces to achieve pure abstraction.

* We can achieve abstraction by following steps:

Step 1 → Generalize the subclass behaviour in an interface.

Step 2 → Provide the implementation in implementation classes.

Step 3 → Create an object of implementation class and access through interface reference variable.

---

* By using Abstraction, we can achieve loose coupling in an application.

* loose coupling is a way of programming in which if we do the changes

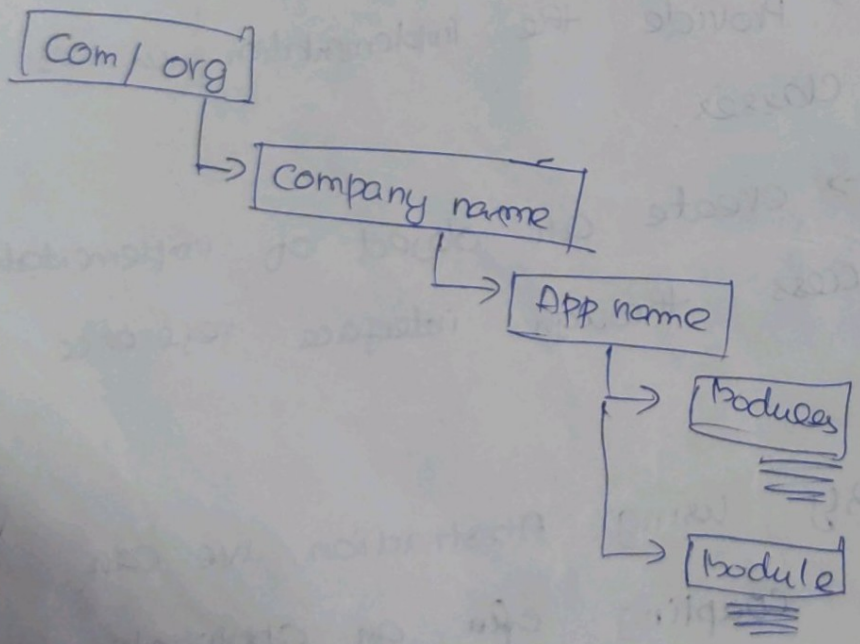
with respect to one layer of the application. It will not have much impact on the other layer of the application.

JAVA PACKAGES

* Package is a folder which is used to store java and java related files.

* We use packages to develop an application in an organized way.

* Each and every java file first line must be package declaration.




```
package com.google.gmail.login;
public class User {
}
```

* If you want to use a class within a package, directly we can use.

* If you want to use the class outside the package, we must use fully qualified class name. (class name along with package structure)

```
package com.google.gmail.login;
public class user {
        com.google.gmail.login.user ui;
}
```

MODULE CLASS:

```
package com.google.gmail.login
public class login
{
    user ui;
}
```

Message class

```
package com.google.gmail.~~login~~inbox;
{
  public class Message
  {
  com.google.login.user ui;
}
```

If you want to use a class outside the package, we can also write import statement.

The import statement must be written after ~~before~~ the package declaration & before the class.

```
package com.google.gmail.login

import com.google.gmail.login.user ;

public class message
{
   user u1;
}
```

if you want to import all the classes which is present in a package, we can use *

```
import com.google.gmail.login ;
```

The java application must be converted into a jar files.

JAR stands for Java archieve.

JAR is a compressed folder which usually contains .class file and resources.

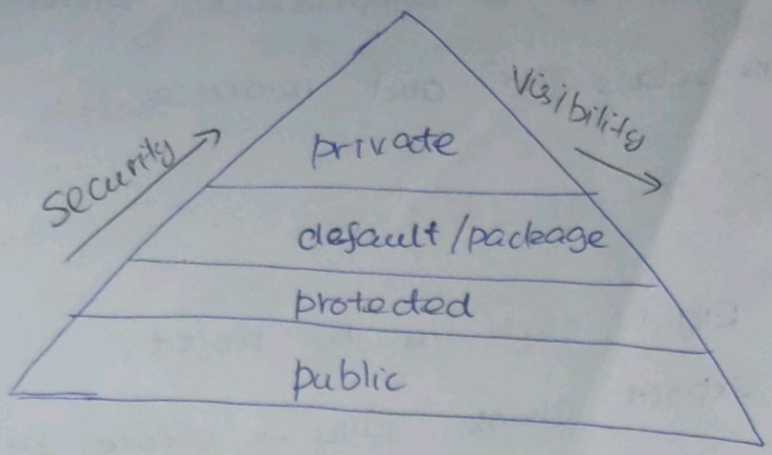
How to create a jar file?

Right click on the project.
export Choose java → choose jar file.
next → Browse the jar file destination and give the name.
Click on finish

How to use jar file in the project?

Right click on the project → Properties
Seled java build path → selected libraries
Choose class path → add external jars
Browse the jar file and apply and close.

ENCAPSULATION



Pab 1

Pab 2

```
class sample1
{
  private int p=10;
      int q=20;
  protected int r=30;
  public int s=40;
}
```

p	q	r	s
✓	✓	✓	✓

extends / inheritance

```
class sample3
{
}
```

p	q	r	s
x	x	✓	✓

```
class sample2
{
}
```

p	q	r	s
x	✓	✓	✓

```
class sample 4
{
}
```

p	q	r	s
x	x	x	✓

) access

) not access

```
package pak1;

public class sample1
{
  private int p = 10;
          int q = 20;
  protected int r = 30;
  public int s = 40;

  void m1()
  {
    S.O.println(p);
    S.O.println(q);
    S.O.println(r);
    S.O.println(s);
  }
}

package pak1;

public class sample2
{
  void m2()
  {
    sample1 s1 = new sample1()
      S.O.println(s1.q);
      S.O.println(s1.r);
      S.O.println(s1.s);
  }
}
```

```
package pak2;

import pak1.sample1;

public class sample3 extends sample1
{ void m3(){
// s.o.ptln(p);
// s.o.ptln(q);
System.out.println(r);
System.out.println(s);
}
}

package pak2;

import pak1.sample1;

public class sample4
{
  void m4()
  { sample1 s1 = new sample1();
    // s.o.ptln(p);
    // s.o.ptln(q);
    // s.o.ptln(r);
    System.out.println (s1.s);
  }
}
```

A/S' Difference b/w Abstract class & interface.

NOTE:

* Wrapping the data inside some definition block is known as encapsulation (or) protecting the members of a class by using some access specifiers is known as encapsulation.

* There are four access specifiers in java,

1. Private
2. default / package
3. Protected
4. Public.

Private:

The private members can be accessed only within the class, It has high-security and less visibility.

Default: The default or package members can be accessed within a class & within a package, if you don't mention any access specifier then the access will be default (or) package.

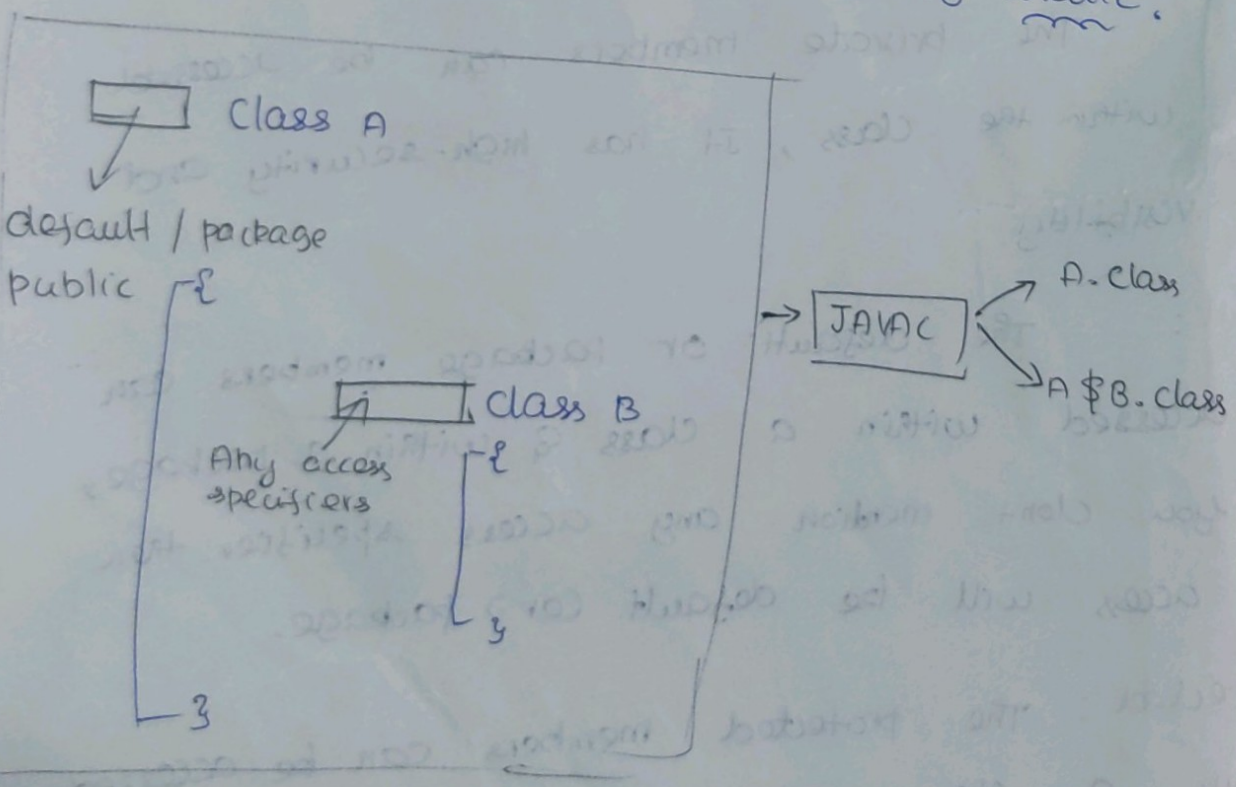
Protected: The protected members can be accessed within a class, within a package and also outside the package through inheritance.

PUBLIC : The public members can be accessed from anywhere, It has high visibility & less security.

Rules of Encapsulation :

Rule 1 : We can define a class inside a class that is known as nested class (or) inner class. The outer class can have only two access specifiers default / package (or) public.

But the inner class can have all the four access specifiers including static.

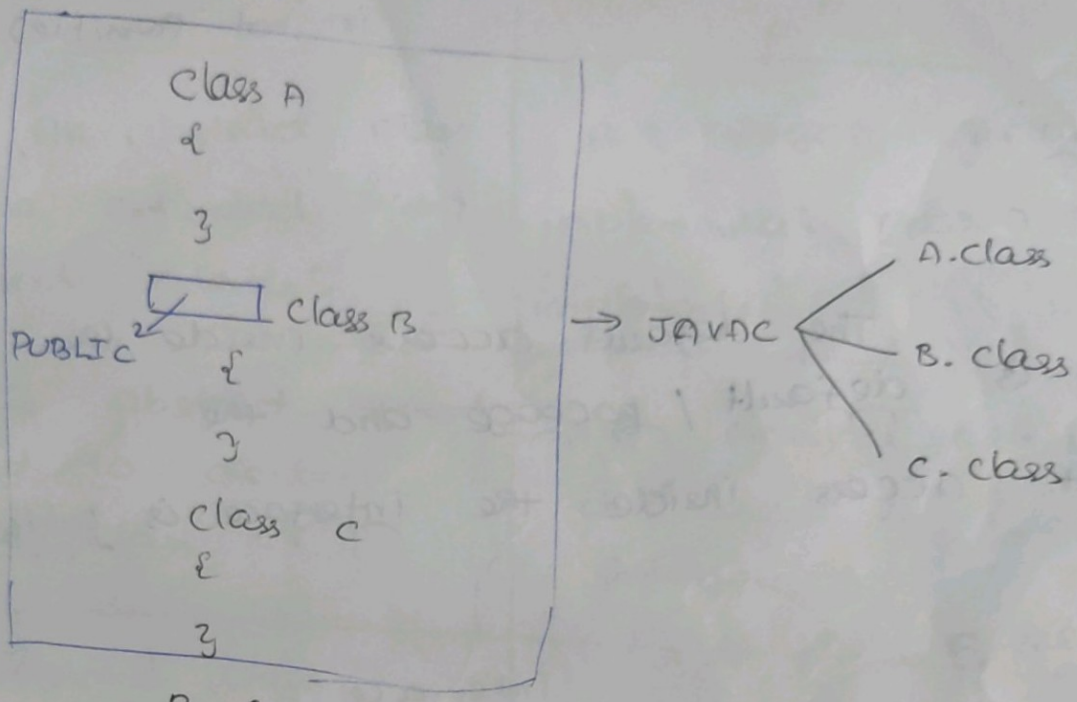


① nested / inner class

RULE NO: 2

FILENAME

In a file we can define "n" number of classes and the filename must be same as public class name.



NOTE:

It is never possible to define more then one public class in a file.

RULE NO: 3 [CONSTRUCTOR]

* The default constructor access is always same as class access.

* If we define our constructor, we can provide all the four access.

```
Class A
{
    A()
    {
    }
}
```

FOUR ACCESS

* If we define the constructor as private we can't create the object outside the class.

* If we define the constructor as private we can't achieve inheritance. (Constructor chaining is not possible).

RULE NO : 4

[default access]

The default access inside the class is default / package and the default access inside the interface is public.

```
Class A
{
    int x = 10;
}
```

default / package

```
interface A
{
    int x = 10;
}
```

public.

RULE NO : 5

While doing method overriding as a programmer, we can increase the visibility, but we can't reduce the visibility.

```
interface A
{
    void m1();
}
Class B implements A
{
    public void m1()
    {
    }
}
```

ABSTRACT CLASS	INTERFACE
③ Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
② An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
① The abstract keyword is used to declare abstract class	② The interface keyword is used to declare interface.
⑤ Abstract class can contains both Abstract & non-abstract	Interface only contains abstract methods
④ It is never possible to create object for interface.	It is never possible to create object for abstract class
A class can inherit from only one class.	A class can inherit from only one class but inherit from multiple interface.
We can achieve pure or partial abstraction on abstraction class	We can achieve pure abstraction using interface.

① Keyword
② extends
③ multiple inheritance
④ Object
⑤ contains
⑥ Abstraction

31-03

JAVA - BEANS [DTO]

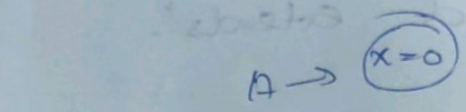
→ Private data members

→ Public default constructor

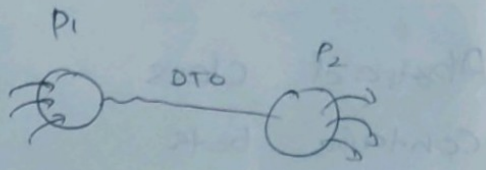
→ public Getters & setters

→ No other methods

```
public class A
{
   private int x ;
   public A() { } ;
   Public int getx()
     {
       return x ;
     }
   public void setx (int x)
     {
       this.x = x ;
     }
}
```



A a1 = new A()



```
package pab1
{
public class employee
{
private int id ;
private String name ;
private double salary ;
```

```
public Employee () {}

public int getid ()
 {
    return id;
 }
public void setid (int id)
 {
   this.id = id;
 }

public String getname ()
 {
    return name;
 }

public void setname( String name)
 {
    this.name = name;
 }

public double getsalary ()
 {
    ~~this.salary = sa~~
    return salary;
 }

public void setsalary( double salary )
 {
   this.salary = salary;
 }
```

(or else)

```
public void setsalary (double salary)
{
    if (salary > 0.0)
        this.salary = salary
    else
        this.salary = 5000.0;
}
}
```

```
Package pab1
public class Mainclass
{
  Public static void main (String [] args)
  {
    employee emp1 = new employee();
    emp1.setid (123);
    emp1.set name ("venki");
    emp1.set salary (15000);

    S.O.pln (emp1.getid());
    S.O.pln (emp1.getname());
    S.O.pln (emp1.getsalary());
  }
}
```

O/P:

123
Venki
15000

NOTE :

* It is a class with private data members, public default constructor, public getters & setters and no other methods, other than getters & setters.

* Java beans are very good example for encapsulation.

* Java beans are mainly used to transfer the data from one program to other program.

* Hence they called as data transfer objects [DTO].

Singleton :

It is a design pattern which will allow us to create a single object of a class.

We can achieve singleton by following steps.

1. We should define constructor as private.
2. We must create only one object within a class
3. We must create a static method which returns same reference.

```
package pab1

public class ~~main~~ calculator
{
   public static calculator calci
                    = new calculator();
   private calculator()
   {
     S.o.ptln("calculator object is created");
   }
   public static calculator getcalci()
   {
    return calci;
   }
}

package pab1

public class mainclass1
{
   public static void main(string[] args)
   {
     calculator c1 = calculator.getcalci();
     calculator c2 = calculator.getcalci();
   }
}
```

o/p:

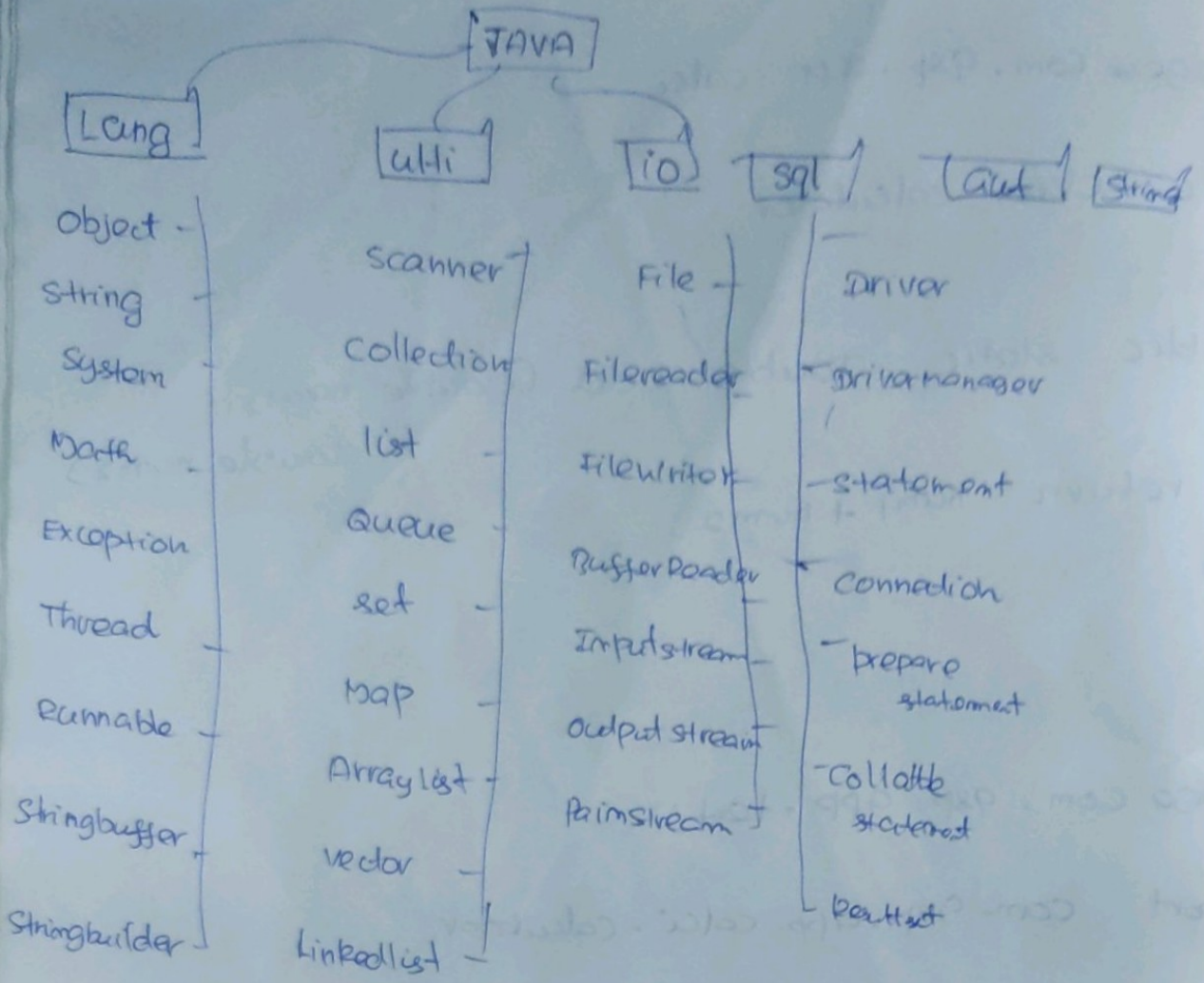
calculator object is created..

Add of two no:

```
package com.gsp.app.calci

public class calculator
{
  public static double add (double num1,
                            double num2)
  {
    return num1 + num2;
  }
}
```

```
package com.gsp.app.test;

import com.gsp.app.calci.calculator

public class test
{
  public static void main (String[] args)
  {
    S.O.ph (caclator.add (a+b));
  }
}
```

JAVA - LIBRARIES



~~public class Calculator~~

{

~~public static void~~ =

Math class.

It is final class present in java.lang package (we can't inherit).

Math class constructor is private (we can't create object)

Math class contains global constants (E and PI)

In math class all the methods are static.

Final keyword:

If we define variable as final we can't reassign.

If we define method as final we can't override.

If we define class as final we can't inherit.

```
public class Mainclass
{
 public static void main (String[] args)
 {
   Scanner sc1 = new Scanner (System.in)
   System.out.println (" Enter Id : ");
   int id = sc1.nextInt ();
   S.o.pln ("Enter name : ");
   String name = sc1.next ();
   S.o.pln ("Enter salary : ");
   double salary = sc1.nextdouble ();

   S.o.pln ( id );
   S.o.pln ( name );
   S.o.pln (salary );
 }
}
```

04-04-23

Object class:

→ It is a root class present in java language package

→ It contains public default constructor

→ It's non-static methods are

- → tostring()
- → hashcode()
- → Equals()
- → clone()
- → finalize()

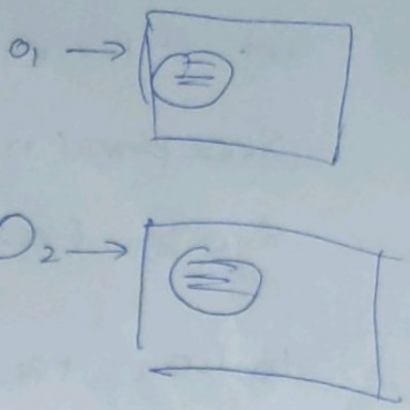
threads:
- → wait()
- → wait(long)
- → wait(long int)
- → notify
- → notifyall()

→ getclass()

```
object o1 = new object();
object o2 = new object();
{
  String s1 = o1.tostring()
  S.o.println(s1); // java.lang.object@xxxx
```

→ S.o.println(o2.tostring()); // java.lang

```
int n1 = o1.hashcode();
s.o.pln (n1); // 1234
s.o.pln (o2.hashcode()); // 4569

boolean res = o1.equals(o2);
s.o.pln(res) // false
s.o.pln (o2.equals(o1)); // false
```



---

eg:

```
package objectclassDemo;

public class mainclass
{
 public static void main(String[] args)
 {
    object o1 = new object();
    object o2 = new object();

 // explicit call to tostring
    string s1 = o1.tostring();
    S.O.pln(s1);
    S.O.pln (o2.tostring());

// implict call to tostring

    S.O.pln (o2);
    S.O.pln (new object());
```

```
int n1 = o1.hashcode();
S.O.pln (n1);
S.O.pln (o2.hashcode());

boolean res = o1.equals(o2);
S.O.pln (res);
```

O/P:

```
java.lang.object@5a7
java.lang.object@76898
java.lang.obj@76 5287
1518467820
1354589220
false
false
true
```

TOString:

* It returns string representation of objects.

↓

[Fully qualified classname @ ~~hashcode~~ address)

* To string method can be called implicitly as well as explicitly.

* When we try to print the reference variable ~~add~~ or an object to string will be called implicitly.

Syntax:

```
public string tostring()
```

Hashcode: It returns hashcode of an object

hashcode is a unique integer number which is calculated based on address of an object.

Syntax: public int hashCode()

equals():

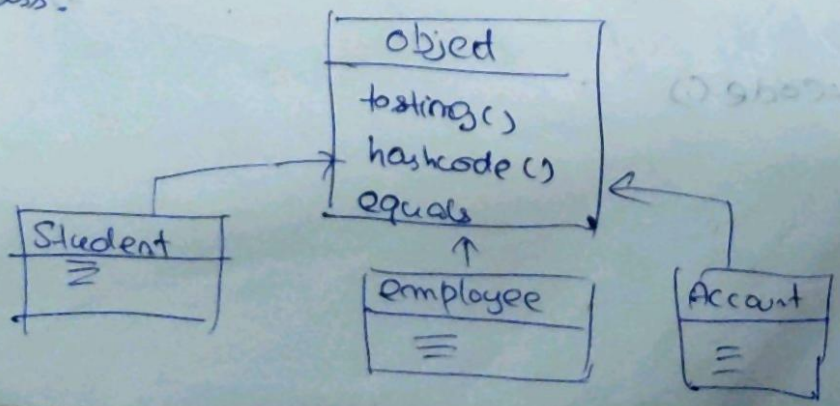
It compares the object based on address. If addresses are equal it returns true or else it returns false.

Syntax: public boolean equals(object o1)

NOTE:

to string(), hashcode() and equals() they are non-static method of object class.

We can call the methods by creating an object of object class or by creating an object of any class. because every class in java is a subclass of object class.



* By default tostring(), hashcode() and equals() methods will work based on address.

* we can override to string, hashcode and equals() method based on our requirements in subclass.

```
Public class student
{
    int sid;
    String sname;
    double rating;
    public student (int sid, string sname,
                                double rating)
    {
        this.sid = sid;
        this.sname = sname;
        this.rating = rating;
    }

    public string tostring()
    {
     return ["Id = "+sid+", name = "+sname", rating = "+rating"];
    }

    public int hash_code()
    {
     return sid;
    }
```

```
public boolean equals (object o1)
{
  if (this.hashcode == o1.hashcode());
      return true;
else
      return false;
}
}
```

```
package objectclassdemo;

import com.spiders.app.student;

public class Mainclass2
{
  public static void main(String[] args)
  {
    student std1 = new student(123, "John cena",
                               4.0);
    student std2 = new student(456, "Ganesh",
                               0.0);
    Student std3 = new student(123, "JohnCena",
                               4.0);
    System.out.println(std1.toString());
    System.out.println(std2);

    System.out.println(std1.hashcode());
    System.out.println(std2.hashcode());

    System.out.println(std1.equals(std2));
    System.out.println(std1.equals(std3));
  }
}
```

O/P: Write override.

[id = 123, name = John cena, rating = 4.0]
[id = 456, name = Gandhi, rating = 0.0]
123
456
false
true

O/P: hiHead overide

com.qspiders.app.student@7534425b

com.qspiders.app.student@35296b54

155532798

3793611743 13

false

false

```
public class student

{
    int sid;
    String sname;
    double rating )

    public student (int sid, string sname,
                                    double rating)
    {
      this.sid = sid;
      this.sname = sname;
      this.rating = rating;
    }

  public student getstudentClone()
  {
    student ref = null;
    try
    {
      ref = (student) this.clone();
    }
    Catch ( CloneNotSupportedException e)
    {
      e.printStackTrace();
    }
    return ref;
  }
```

```
public void finalize()
{
  System.out.println("I am in finalize");
}
}

public string tostring()
{
return "[" Id = "+sid+" , Name = "+sname", rating = "+srating+"]";
}
public int hashcode()
{
    return sid;
}

public boolean equals(Object o1)
{
if (this.hashcode == o1.hashcode());
      return true;
else
      return false;
}
```

06.05.23

STRING

```
package stringclassDemo;

public class Mainclass1
{
  public static void main (string[]args )
  { s.o.pr("*****")
    string s1 = new string ();
    string s2 = new string("abc");
    char[] ch = ['a', 'b', 'c' ];
    String s3 = new string [ch);

    System.out.println(s1);
    System. out. println (s2);
    System. out. println(s3);

    system.out.println(s1.hashcode());
    S.O.println (s2.hashcode());
    S.O.println (s3.hashcode());

    S.O.println (s2. equals (s3));
}
```

O/P

```
*****
abc
abc
0
97895
97895
true
```

```
package stringclassDemo;

public class mainclass2
{
    public static void main (String[] args)
    {
        String s1 = "java";
        String s2 = "java";
        String s3 = new String ("java");
        String s4 = new String ("java");
        S.O.p.hm (s1 == s2);
        S.O.p.hm (s3 == s4);
        S.O.p.hm (s1.equals(s2));
        S.O.p.hm (s3.equals(s4));
    }
}
```

O/P

true
false
true
true

String:

- It is a final class present in java lang package.
  - final class → We can't inherit
- It is immutable class
  - immutable class → We can't change
- It is thread safe

STRING	STRING BUFFER	STRING BUILDER
* It is a final class present in java.lang package.	* It is a final class present in java.lang package.	* It is final class present in java.lang package.
* It is immutable class	* It is mutable class.	* It is mutable class.
* It is thread safe	* It is thread safe.	* It is not thread safe.
* In string, tostring(), hashcode(), equals() are overrided.	* only tostring() is overrided. (one & only difference.)	* only tostring() is overrided.
* there are two ways we can create a string object ① by literals ② by new() operator.	* We can create the object only by new operator	* We can create the object only by new operator.
* String is a comparable type	* It is comparable type	* It is comparable type.
[append, insert, reverse, delete] It can be used only in StringBuffer & StringBuilder.		

03.04.23

```
Public class mainclass
{
  public static void main (String[] args)
  {
    StringBuffer sb1 = new StringBuffer ("abc");
    StringBuffer sb2 = new StringBuffer ("abc");

    System.out.println(sb1);
    System.out.println (sb2);
    System.out.println (sb1.hashcode());
    System.out.println (sb2.hashcode());
    System.out.println (sb1.equals (sb2));

    sb1.append ("xyz");
    System.out.println (sb1);

    sb1.insert (3, "pqr");
    System.out.println (sb1);

    sb1.delete (2,5);
    System.out.println (sb1);

    sb2.reverse ();
    System.out.println (sb2);
  }
}
```

o/p:

```
abc
abc
5336427658
1987532449
false
abcxyz
abcpqrxyz
abxyz
cba
```



```
package StringclassDemo;
import java.util.Scanner;

public class mainclass1
{
  public static void main (String[] args)
  {
    Scanner sc1 = new Scanner (System.in);
    System.out.println("Enter string:");

    String s1 = sc1.next();
    StringBuffer sb1 = new StringBuffer (s1);
                  // converting string to stringbuffer.
    sb1.reverse();

    String s2 = new String (sb1); // converting
                                  stringbuffer to string.
    if (s1.equals (s2))
      System.out.println ("palindrome");
    else
      S.O.Pln ("not");
  }
}
```

ARRAYS

→ Set of homogeneous elements.

```
int x = 10 ;
int x = 20 ; ⇒  int [] arr = {10, 20, 30} ;  in c
```

```
Int [] arr = new int [3] ;
int arr[] = new int [3]
```

0	1	2
10	20	30

Syntax:

```
datatype[] var = new datatype[size]
```

↳ 0 / positive

```
arr[0] = 10 ;
arr [1] = 20 ;
arr [2] = 30 ;
// arr [3] = 40 // ArrayIndexoutofBoundException.
```

```
for (int i=0 ; i< arr.length ; i++)
{
  S.o.pln (arr[i]) ;
}
```

→ index = i
element = arr[i]

foreach:

```
for ( int x : arr)
{
 S.o.pln (x) ;
}
```

⇒ only element.


```
package arrayDemo ;

public class baimclass
{
  public static void main (String[] args)
  {  for (int[] arr = {2, 4, 6, 2, 7, 9} ;
    for (int i=0 ; i< arr.length ; i++)
    {
      if (i%2==0)
        S.o.pln (i + "--->" + arr[i]) ;
    }
    S.o.pln ("*****") ;
    for(int ele : arr)
    {
      if (ele%2==0)
        S.o.pln (ele) ;
    }
  }
}
```

o/p:

```
0 → 2
2 → 6
4 → 2
6 → 9
*****
2
4
6
2
```

```
package arraysDemo ;
import java.util.Scanner ;
public class mainclass
{
  public static void main (String[] args)
```

```
{
    Scanner sc1 = new scanner ( system.in );
    S.o.ptn("How many elements you want");
    int singe = sc1.nextint ();

    int[] arr = new int[singe];

    for (int i=0; i<arr.length ; i++)
    {
     S.o.ptn ("Enter "+ (i+1)+"domant");
     arr[i] = sc1.nextInt ();
    }

  S.o.ptn ("-----");
  for (int ele : arr)
  {
    S.o.ptn (ele);
  }
}
}
```

o/p:

```
How many you want
2
10
20
Enter 1 elamat
10
Enter 2 elamt
20
-------
10
20
```

③

```
package arrays Demo;

public class student
{
    int id;
    string name;
    double[] marks;

    public student (int id, string name, double[]marks
    {
      this.id = id;
      this.name = name;
      this.marks = marks;
    }

    public double total ()
    {
      double sum = 0.0;
      for (double mark : marks)
      {
        sum += mark;
      }
      return sum;
    }

    public double average()
    {
      return total()/ marks.length;
    }

    public string ispass()
    {
      String result =
```

```
for (double mark : marks)
{
  if (mark < 35.0)
  {
    result = "fail";
    break;
  }
}
return result;
}
}
```

```
package arrayDemo;

public class mainclass1
{
  public static void main(string[]args)
  {
   Studat std1 = new Student (123, "Rajnit",
            new double[] {100.0, 100.0, 35.0};

   System.out.println(std1.total());
   system.out.println(std1.average());
   System.out.println(std1.ispax());
  }
}
```

o/p:


```
package arrayDemo;

public class mainclass
{
  public static void main (string[] args)
  {
    String s1 = "madam";
    char[] ch = s1.tocharArray();
    boolean res = true;
    int i=0; j=ch.length-1;
    while (i<=j)
    {
      if (ch[i]!=ch[j])
      {
        res = false;
        break;
      }
      i++;
      j--;
    }
    if (res)
      s.o.p.ln ("palindrome")
    else
      s.o.p.ln ("not")
  }
}
```

```
for (Student std : students)
{
 s.o.p(std.id + "\t" + std.name + "\t" + std.isPass()
  + "\t" + std.average() + "\t" + std.total());
```

Exception handling:

Exception is an unexpected event which makes program to terminate abnormally.

There are 3 types of termination

1. Normal termination
2. Force termination (System.exit(0))
3. Abnormal termination. (Exception).

even there is an exception - we can continue execution by handling exception.

We can handle exception by using try - catch.

In try we must write the code which causes the ~~exception~~ exception.

→ In catch we must solution code.

try must be followed by catch or finally.

catch block will executes only when exception in try.

Finally block will executes always.

If try followed by catch program will continue the exception,

→ if try is followed by finally, finally will executes but program will terminates abnormaly

→ If we do force termination ~~finally~~ finally will not executes.

Difference b/w Error and Exception:

→ Error is class present in java.lang

Exception is class present in java.lang

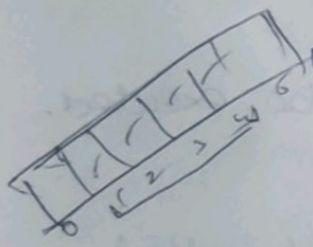
→ Error will occur because of lack of system resources.

Exception will occurs because of unexcepted event.

→ Errors are irrevokable.

Exception are revokeable.

→ Ex for errors : StackOverFlowError...



Ex for Exception : ArithmeticException,

CloneNotSupportedException.

Diff b/w checked Exception and unchecked Exception :

→ Exceptions Identified by compiler is known as Checked Exception.

Exceptions not Identified by compiler is known as unchecked Exception.

→ All Checked Exception are subclass of exception.

All unchecked ~~Exp~~ exception are sub class of RuntimeException

Ex :

Ex :

Types of Exception :

① Checked / caught / compiletime Exception

② unchecked / uncaught / RuntimeException

Try with multiple catch block:

→ a try can have multiple catch but only one finally.

→ only one catch block will be executed.

→ multiple exceptions can be handled in same catch

→ super class catch can handle sub class Exception.

→ catch can be written up to throwable

→ while defending multiple catch we should have from sub class to super class.

Default Exception Handling in JAVA:

* If we are not handling Exception JVM will handle the Exception.

* If JVM is handling Exception it will terminate the program execution and also prints stackTrace.

* We can print stack Trace by using printStackTrace().

How create userdefined / customized exception?

--> by extending any of the Exception class.

Throw and throws

→ throw is a keyword which is used to throw an object which is of type Throwable

→ throw is generally used with userdefined / customized Exception.

→ We can throw only one objed at a time.

→ throw must be used in definition block

→ Syntax: throw new ClassName();

→ throws is a keyword which is used to pass the exception to called.

→ throws is generally used with checked exceptions.

→ throws can be used to pass multiple exceptions to caller.

→ throws can be used in declaration part.

Syntax: throws ClassName1, ClassName2....

Threads:

* Thread is an execution Instance, It will have its own path of execution.

* Thread is a light weight process.

* Threads are used for parallel execution.

* Java supports multi threading.

In java we can create thread in two ways

① extending from thread class.

② Implementing from runnable interface.

Difference b/w start() & run()

Run()

→ It is a abstract method of runnable interface.

→ If we call run() execution is sequential

→ If we call run() execution is on same stack frame.

start():

It is non-static method of thread class.

If we call start() execution is parallel

If we call start() execution is on different stack frame.

Types of threads:

① User threads

② System threads:

Thread sheduler

Garbage collector

thread main.

Thread class :

→ thread is a class present in java.lang

→ thread is having overloaded constructor.

thread ()

thread (string name)

Thread (runnable r)

Thread (runnable r, string name)

---> Thread properties are

Id --> desaultly given by JVM.

Name → desaultly given by JVM.

priority → is b/w 1-10 and default is 5

---> Its methods are :

start(), stop(), pause(), resume(), getId(), getname(), getpriority(), setName(), setpriority(), sleep(long), sleep(long, int), currentThread()..

20.04.23

Collection

* Collection is an API / Framework which helps us to store a group of objects.

* The collection framework is introduced from JDK 1.2V

* The collection framework is implemented based on some data structure, Hence we get special methods.

* Collection is dynamic in nature.

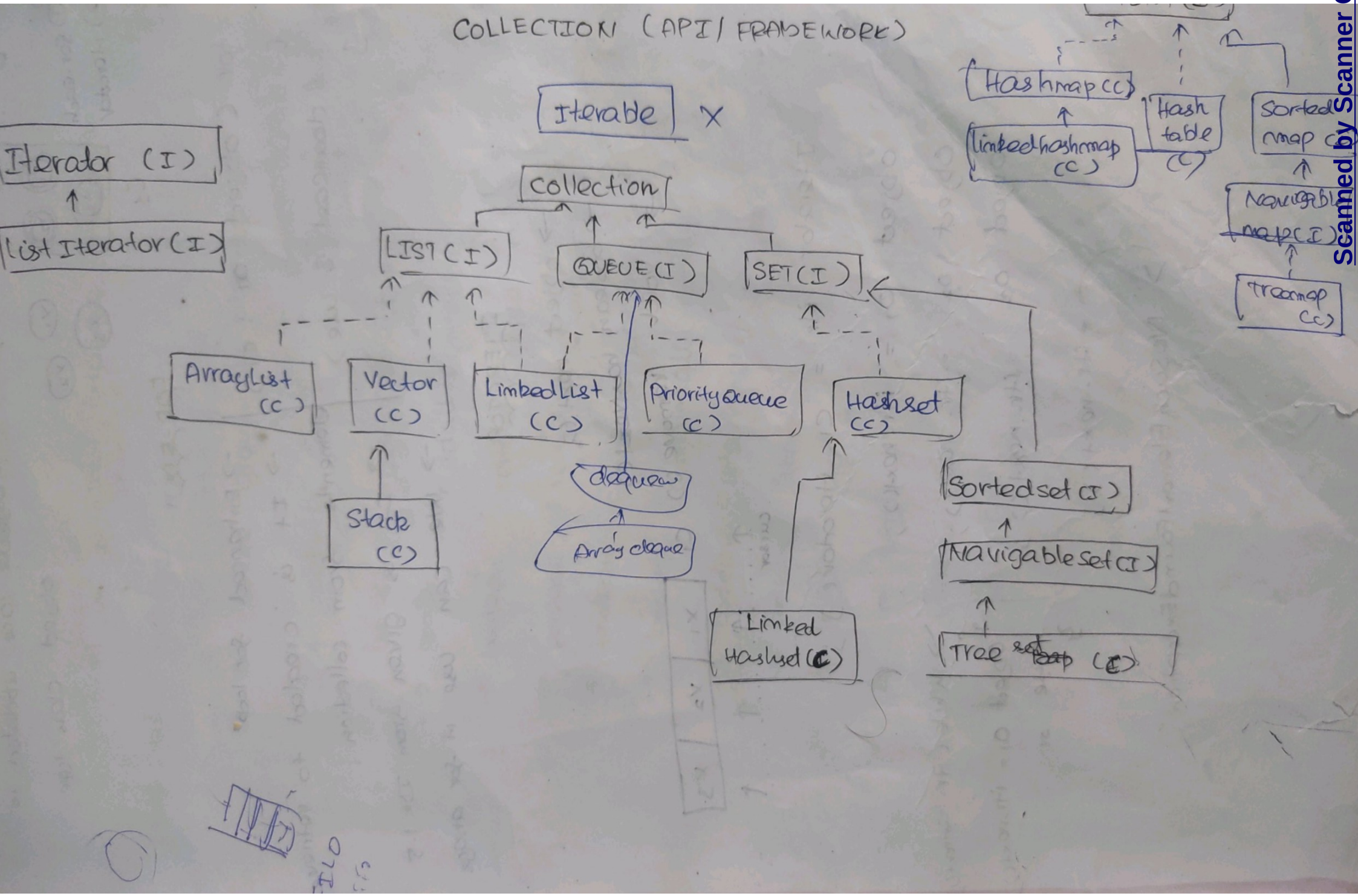
* Collections can store homogeneous as well as heterogeneous elements.

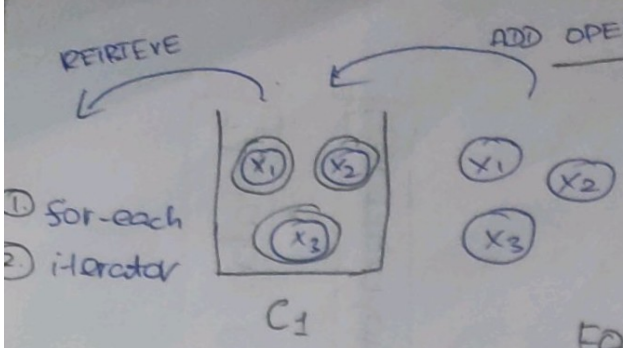
* All the collection classes and interfaces present in java.util package.

API:

Application programming interface which contains interfaces and classes which is used to perform some particular tasks.

COLLECTION (API / FRAMEWORK)





→ objects are updated to object class type.

① for-each
② iterator

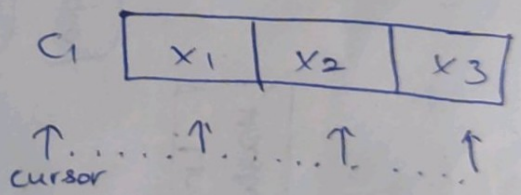
FOR-EACH

```
for ( object o1 : c1 )
{
   // downcast & use )
}
```

→ Enhanced for loop
→ It is created to retrieve elements from collection.
→ It is given from JDK 1.5
→ We can use it for arrays

ITERATOR

→ object next ().
→ boolean hasNext()
→ void remove()




```
Iterator itr = c1.iterator();

object o1 = itr.next();
Object o2 = itr.next();
Object o3 = itr.next();
```

```
while (itr.hasNext())
{
  Object o1 = itr.next();
  // use o1
}
```

= itr.next();

↳ "NoSuchElementFoundException"

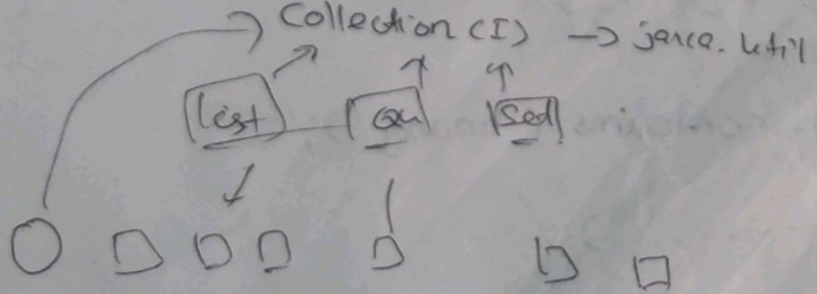
COLLECTION (I) :

It is a root interface present in java.util package.

It store group of objects.

It's methods are :

- → boolean add (object o1)
- → boolean addall (collection c1)
- → boolean remove (object o1)
- → boolean removeall (collection c1)
- → boolean contains (object o1)
- → boolean containsAll (collection c1)
- → bodean retainAll (collection c1)
- → boolean isEmpty ()
- → void clear ()
- → int size()
- → Iterator iterator();
- → object [] toArray();



eg. Run-time poly

Abstraction.

```
package CollectionDemo;

import java.util.ArrayList;
import java.util.collection;
import java.util.Iterator;

public class collectionExample
{
  public static void main (String [] args)
  {
    S.O.ptn ("*******");

    collection c1 = new ArrayList();

    c1.add("vikram");
    c1.add (new string ("manoj"));
    c1.add (new mobile());  c1.add (new StringBuffer
                                    ("selva"));
    // c1.clear();

    S.O.ptn (c1.size());
    S.O.ptn (c1.IsEmpty());

  // c1.remove ("vikram");
  // S.O.ptn (c1.size());

    S.O.ptn (c1.contains ("manoj");
    S.O.ptn ("------");
```

```
for (object o1 : c1)
{
  s.o.plm (o1);
}
 s.o.plm ("----------");

 Iterator  itr = c1.iterator();
 while ( itr.hasnext())
 {
   s.o.plm ( itr.next());
 }

 s.o.plm ("*****")
}
}
```

O/p:

```
*****
4
false
true
----------
vikram
manoj
selva
collection demo.robot
----------
vikram
manoj
selva
Collection demo.robot.
```

```
package CollectionDemo;

import java.util.Collection;
import java.util.LinkedList;

public class Mainclass2
{
  public static void main(String[] args)
  {
    S.O.pln("*******");

    Collection names = new LinkedList();

    names.add("samantha");
    names.add("thrisha");
    names.add("Nayanthara");
    names.add("sunny");

    for(object o1 : names)
    {
      String name = (String)o1; // downcasting
      S.O.pln("name"+"t"+name.length());
    }

    S.O.pln("---------");

    for(object o1 : names)
    {
      String name = (String)o1;
      if(name.contains("s"))
      {
        S.O.pln(name);
      }
```

```
        S.O.println("* **** ");
    }
}
```

o/p

```
Samantha    8
Thrisha     7
Nagamthara  10
Sunny       5
Samantha
Sunny
```

GENERICS

* It makes colledion as homogeneous.
* It is introduced from jdk 1.6 ~~Collection~~

⇒* Collection<String> = new linkedList<String>();

if It is Jdk 1.7

⇒Collection<String> = new LinkedList<>();

*Generics is purely for class not primitive data types.

```
package collectionDemo;

import java.util.ArrayList;

public class Mainclass3
{
 public static void main(String[] args)
 {
  S.O.pln("*******");

  // collection<String> names = new LinkedList<String>()

  Collection<String> names = new LinkedList<>();

  names.add("samantha");
  names.add("Thrisha");
  names.add("Nayanthara");
  names.add("sunny");

  for(String name : names)
  {
   S.O.pln(name + "lt" + name.length());
  }
  S.O.pln("*****");
```

O/P:

****

samantha 8

thrisha 7

Nayanthara 10

sunny 5

*****

Wrapper Class

boolean → Boolean
char → Character
byte → Byte
int → Integer
short → Short
long → Long
float → Float
double → Double

→ are final class present in java.lang package
→ are immutable
→ they contain overloaded constructors.
→ tostring(), hashcode(), equals() are overrided.

⇒ AutoBoxing (converting primitive datatype to object)

```
Integer I1 = new Integer(50);
Integer I2 = 50;
```

⇒ Auto Unboxing (converting object type to primitive type)

```
int y = I1;
```

Parsing : (converting string to primitive)

```
String s1 = "1258";
int x = Integer.parseInt(s1);
double y = Double.parseDouble(s1);
```

21.04.23

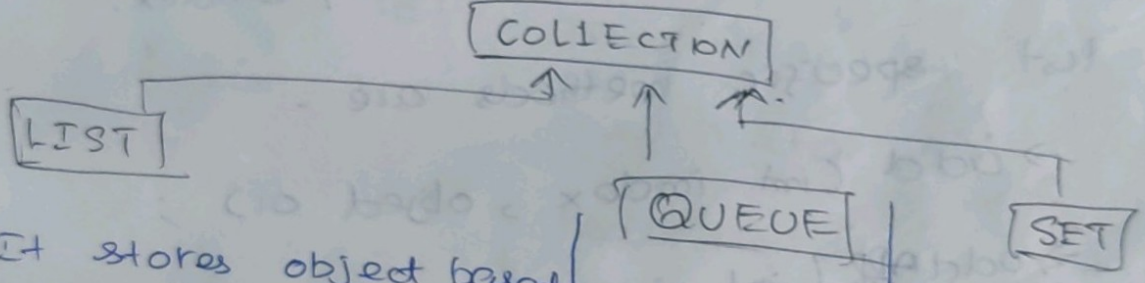
```
package collegeDemo;

import java.util.Collection;
import java.util.Vector;

Public class MainClass4
{
  public static void main(String[] args)
  {
    Collection<Integer> nums = new Vector<>();
    nums.add(10); // auto boxing
    nums.add(20);
    nums.add(30);
    nums.add(40);

    for(int num : nums) // auto unboxing
    {
      S.o.println(num);
    }

    // parsing
    // String s1 = "1234y"; // Number format Exception
```

```
String s1 = "123";
int x = Integer.parseInt(s1);
S.o.p(x);
double d = Double.parseDouble(s1);
S.o.p(d);
}
}
```

op:
10
20
30
40
123
123.0

21.04.23

Types of Collection



* It stores object based on index.
* It allows duplicates.
* It allows null values.
* It preserves insertion order.
* Retrieval can be done by for-each / Iterator / Index.

* It stores group of objects based on hashcode.
* It won't allow duplicates.
* It allows only one null value.
* It won't preserve insertion order.
* Retrieval for-each / Iterator.
* for searching operation

List :

It is type of collection it stores group of objects based on index.

It is a sub interface of collection present in java.util package.

It allows duplicates.

It allows null values.

It preserves Insertion order.

retrieval - foreach / Iterator / Index.

list specific methods are.

→ add (int index, object o1);

→ addAll (int index, collection c1);

→ remove (int index);

→ get (int index);

→ set (int index, object o1) // replace.

→ sublist (int start, int end);

→ listIterator ()

---> Implementation Classes are ArrayList, Vector, LinkedList, Stack.

```
package collectionDemo;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;

public class Mainclass3
{
  public static void main(String[] args)
  {
    // List list = new ArrayList();
    // List list = new Vector();
       List list = new LinkedList();
       list.add("Alpha");
       list.add("Beta");
       list.add("Muthusamy");
       list.add("Muthusamy");
       list.add(new mobile());
       list.add(new StringBuffer("sb"));
       list.add(null);
```

// Index based operation

```
list.add(1,"Gama");
list.remove(2);
S.O.pln(list.get(4));
list.set(1,"Hangama");
S.O.pln(list.sublist(1,5));
S.O.pln(list.indexof("Pathusamy"));
S.O.pln(list.lastIndexof("Pathusamy"));
S.O.pln("---------");
for(int i=0;i<list.Size();i++)
    S.O.pln(i+"-->"+list.get(i));
}
}
```

o/p:

CollectionDemo - bobile@sacf954

[Hangama, Butkusamy, Butkusamy, CollectionDemo
2 bobile@sacf9800]
3

---

0 -> Alpha

1 -> Hangama

2 -> Butkusamy

3 -> Butkusamy

4 -> CollectionDemo: bobile@sacf9800

5 -> sb

6 -> null

Iterador	List Iterator
Iterator is an interface present in java.util package.	List Iterator is a sub-interface ~~present in~~ of iterator present in java.util package.
iterator is used to retrieve the elements of collection.	list Iterator is used to retrieve the elements of list type of collection.
If we use iterator, the cursor can travels only in	If we use list iterator the cursor can travels

both forward and backward direction

both forward and backward direction

```
package collectionDemo;

import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Mainclass5
{
  public static void main(String[] args)
  {
    Scanner sc1 = new Scanner(System.in);
    List<String> Stations = Arrays.asList("TAM",
        "SAN","CHOM","PALL","GUI","MAIN","BEA");
    for (int i=0; i<Station.size(); i++)
    {
      S.O.println(i+"-->"+Stations.get(i));
    }
    S.O.println("----------");
    S.O.println("Enter Source station:");
    String src = sc1.next().toUpperCase();
    S.O.println("Enter destination:");
```

```
String dest = sc1.next().toUppercase();

int srcIndex = -1;
int destIndex = -1;

if (stations.contains(src))
{
  srcIndex = stations.indexOf(src);
}

if (stations.contains(dest))
{
  destIndex = stations.indexOf(dest);
}

if (!(srcIndex < 0 && destIndex < 0))
{
   int diff = Data.abs(destIndex -
                       srcIndex);
   System.out.println("Amount =" + (diff*5));
}
 else
 {
   S.O.println("Invalid station Name..!!!");
 }
}
}
```

O/P:

0 -> TAM
1 -> SAN
2 -> CHEM
3 -> PALL
4 -> GUID
5 -> SAI
6 -> MAM
7 -> BEA

Enter source station:
bea
Enter Destination:
tann
Amount = 35

```
package CollectionDemo;

import java.util.Arrays;
import java.util.List;
import java.util.ListIterator;

public class Mainclass3
{
    public static void main(String[] args)
    {
        List<String> days = Arrays.asList("SUN",
                "MON","TUE","WED","THU","FRI","SAT");

        ListIterator itr = days.ListIterator();

        while(itr.hasNext())
        {
            S.O.P(itr.next()+" ");
        }
```

```
S.O.println("\n-------");
while(itr.hasPrevious())
{
  S.O.println(itr.previous()+" ");
}
}
}
```

```
SUN MON TUE WED THU FRI SAT
---------------------------
SAT FRI THU WED TUE MON SUN
```

Array List:

It is an implementation class of list present in java.util package.

It is implemented based on Dynamic/resizable/growable array data-structure.

It implements 3 marker interfaces RandomAccess, Cloneable, Serializable.

It's overloaded constructors are:

ArrayList()

ArrayList(int capacity)

ArrayList(Collection c)

--> default initial capacity of Arraylist is 10

--> It grows with {new capacity = old capacity * 3/2 + 1 }.

--> It is not preferable for insertion / deletion operation.

--> We can use Arraylist for retrival operation.

Vector:

It is an implementation class of list present in java.util package.

It is implemented based on Dynamic / resizable / growable array data-structure.

It implements 3 marker interfaces RandomAccess, Cloneable, Serializable.

It is legacy class (It is present from JDK 1.0).

Vector class is thread safe (Methods are synchronized)

It is overloaded constructors are.

vector()

~~vect~~ vector(int capacity)

~~Vector(Collection c)~~

Vector(int capacity)

Vector(Collection c)

Vector(int initialcapacity, int inc)

default initial capacity of vector is 10

It grows with (nc + oc * 2).

It is not preferable for insertion/deletion operation.

We use vector if we need thread safety.

Its subclass is stack (LIFO) push() and pop() methods we can use on stack.

Its methods are addElement(object o), removeElement(object o), capacity() ....

Linked list:

It is an implementation class of list and queue present in java.util package.

It is implemented based on ~~linked~~ doubly-linked list data structure.

It implements 2 marker interfaces Clonable, serializable.

Its overloaded constructors are:

linkedList()

linkedList(Collection c).

No default initial capacity in linkedlist

It is preferable for insertion/deletion operation.

D/B Array & Array list?

→ don't use linked list for

→ Linked list specific methods are:

* addFirst(Object o)
* addLast(Object o)
* removeFirst()
* removeLast()
* getFirst()
* getLast().

Set:

* Set is type of collection which stores objects based on hashcode.

* It is a sub interface of collection present in java.util package.

- It won't allow duplicates.
- It allow only one null value.
- It won't preserve insertion order.
- Retrieval can be done by foreach / iterator.
- We use set of searching operation
- No set specific methods.
- implementation classes are hashset, linkedhashset, treeset.

note: If we want to store our objects in set without any duplicates, we must write hashcode() and

DB Array list? Linked list?

lashset :

It is a implement class of set present in java.util.

It is implemented based on HashTable data-structure.

Hashset overloaded constructors are:

→ Hashset()
→ Hashset(int capacity)
→ HashSet(Collection c,)
→ Hashset(int capacity, float loadFactor)
→ default capacity of Hashset is 16 and loadfactor is 0.75 {16*1.0 = 16
32*1.0 = 32}

Linked HashSet:

It is a sub class ~~present~~ of Hashset present in java.util.

It is implemented based on Hybrid data-structure [doublyLinkedList + HashTable].

It preserves insertion order.

Tree Set :

→ It is an implementation class of Set present in java.util package.

* It stores object in sorted order.
* TreeSet implemented based on balanced tree data-structure.
* It stores only comparable type object
* It stores only homogeneous elements.
* It won't allow duplicates.
* It won't allow null values.
* It won't preserve insertion order.
* retrieval by foreach / Iterator.
* Its overloaded constructors are:
  - → Treeset()
  - → Treeset(Collection c)
  - → Treeset(SortedSet s)
  - → Treeset(Comparator c).