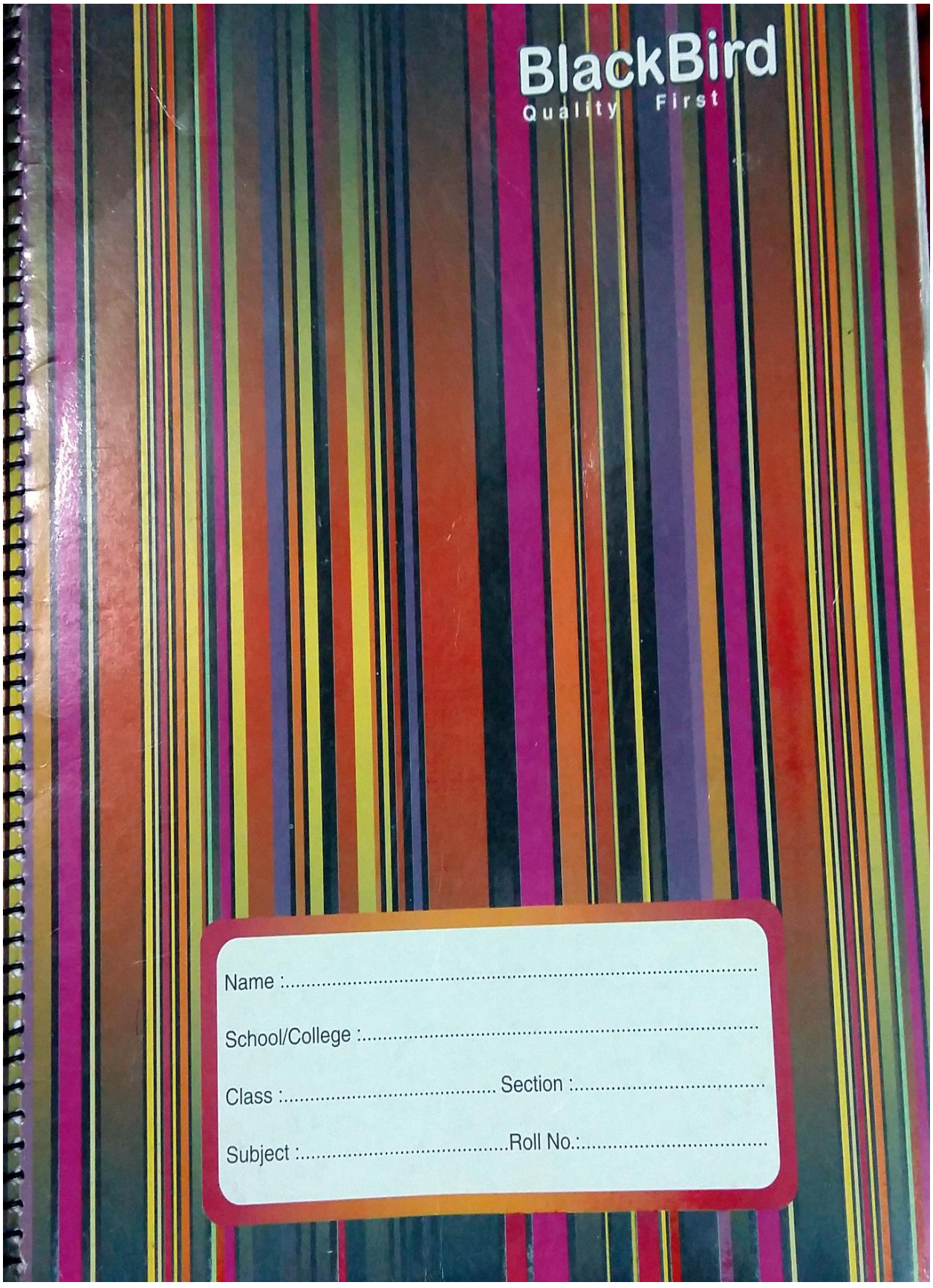

BlackBird
Quality First
Name :.....
School/College :.....
Class :..... Section :.....
Subject :.....Roll No.:.....

OOPS

(I) Data Hiding
(II) Abstraction
(III) Encapsulation
(IV) Tightly encapsulated class

Security

(V) Is a Relationship
(VI) Has a Relationship
(VII) Method signature
(VIII) Overloading
(IX) Overriding
(X) static control flow
(XI) Instance control flow
(XII) Constructors
(XIII) Coupling
(XIV) Cohesion
(XV) Type-casting

Data Hiding

* Outside person can't access our internal data directly or our internal data should not go out directly, this oop feature is nothing but data hiding.

After validation or authentication outside person can access our internal data.

Ex: After providing proper username and password, we can able to access our gmail inbox information.

Ex: Even though we are valid customer of the bank, we can able to access our account information and we can't access others account information.

```
public class Account
{
 private double balance;

 public double getBalance()
  {
   //validation
   return balance;
  }
}
```

By declairing data member (variable) as "private", we can achieve data hiding.

The main advantage of data hiding is security.

*** It is highly recomended to declair data member (variable) as private.

Abstruction:

Hiding internal implementation and just highlight the setup services what we are offering, is a concept of abstraction.

Ex: Through bank atm GUI screen, bank people are highlighting the setup services, what they are offering without highlighting internal implementation.

The main advantages of abstruction are

1. We can achieve security, bez we are not highlighting our internal implementation.

2. Whithout affecting outside person, we can able to perform any type of changes

in our internal system, and hence Enhancement will become easy.

③ It improves maintainability of the application

④ It improves easyness to use our system.

By using interfaces and the abstract classes we can implement abstraction.

☒ Encapsulation:

The process of binding data and corresponding methods into a single unit is nothing but encapsulation.

Ex: class Student {

data members (Variables)

+

behaviours (methods)

}

Every class in java are encapsulation.

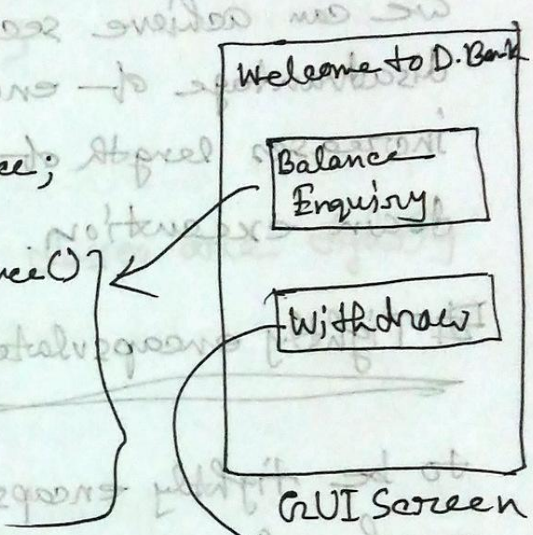
If any component follows data hiding and abstruction, such type of component is set to be encapsulated component.

Encapsulation = Data Hiding + Abstruction

```
public class Account
{
private double balance;

public double getBalance()
 {
  // validate
  return balance;
 }

public void setBalance(double balance)
 {
 // validate
   this.balance = balance;
 }
}
```



The main advantages of encapsulation are, we can achieve security, enhancement will become easy, it improves maintainability of the application.

* The main advantage of encapsulation is we can achieve security but the main disadvantage of encapsulation is it increases length of the code and slows down execution.

Tightly encapsulated class:

A class is set to be tightly encapsulated, if and only if each and every variable declared as private. Whether class contains corresponding getter and setter methods or not, and whether these methods are declared as public or not these thing we are not required to check.

```
public class Account
{
private double balance;

public double getBalance()
  {
    return balance;
  }
}
```

* Which of the following classes are tightly encapsulated:

```
class A {
  private int aj;
}
class B extends A
  {
  int bj;
  }

Class C extends A
  {
  private int cj;
  }
```

```
class A
{
int x = 10;
}
class B extends A
{
private int y = 20;
}
class C extends B
{
private int z = 30;
}
```

*** If the parent class is not tightly encapsulated then no child class is tightly encapsulated.

Is A Relationship:

1. It is also known as Inheritance.
2. The main advantage of is a relationship is reusability.
3. By using "extends" keyword we can implement is a relationship.

```
class P{
public void m1()
{
S.O.p("Parent");
}
}
class C extends P{
public void m2(){
S.O.p("child");
}
}
```

```
class Test{
P S V main(String[] arg){
① P p = new P();
   p.m1(); ✓
   p.m2(); ✗
② C c = new C();
   c.m1(); ✓
   c.m2(); ✓
③ P p = new C();
   p.m1(); ✓
   p.m2(); ✗
④ C c1 = new P();
}
}
```

CE: cannot find symb
symbol: method m2()
location: class P

CE: incompatible ty
found: P
required: C

conclusion:

(I) Whatever methods ~~child~~ parent has by default available to the child and hence on the child reference we can call both parent and child class methods.

(II) Whatever method child has by default not available to the parent and hence on the parent reference we can't call child specific methods.

(III) Parent reference can be used to hold child object, but by using that reference we can't call child specific methods. but we can call the methods present in parent class.

(IV) ~~Child~~ Parent reference can be use to hold child object but child reference can not be used to hold parent object.

Without Inheritance (Loan Module)

class CarLoan { 300 methods }

class HomeLoan { 300 methods }

class PersonalLoan { 300 methods }

total 900 methods

With Inheritance

First we have to find out common methods in all classes, then create a class with those common classes.

```
class Loan {
    250 methods
}

class CarLoan extends Loan {
    50 methods
}

class HomeLoan extend Loan {
    50 methods
}

class PersonalLoan extends Loan {
    50 methods
}
```

total 400 methods

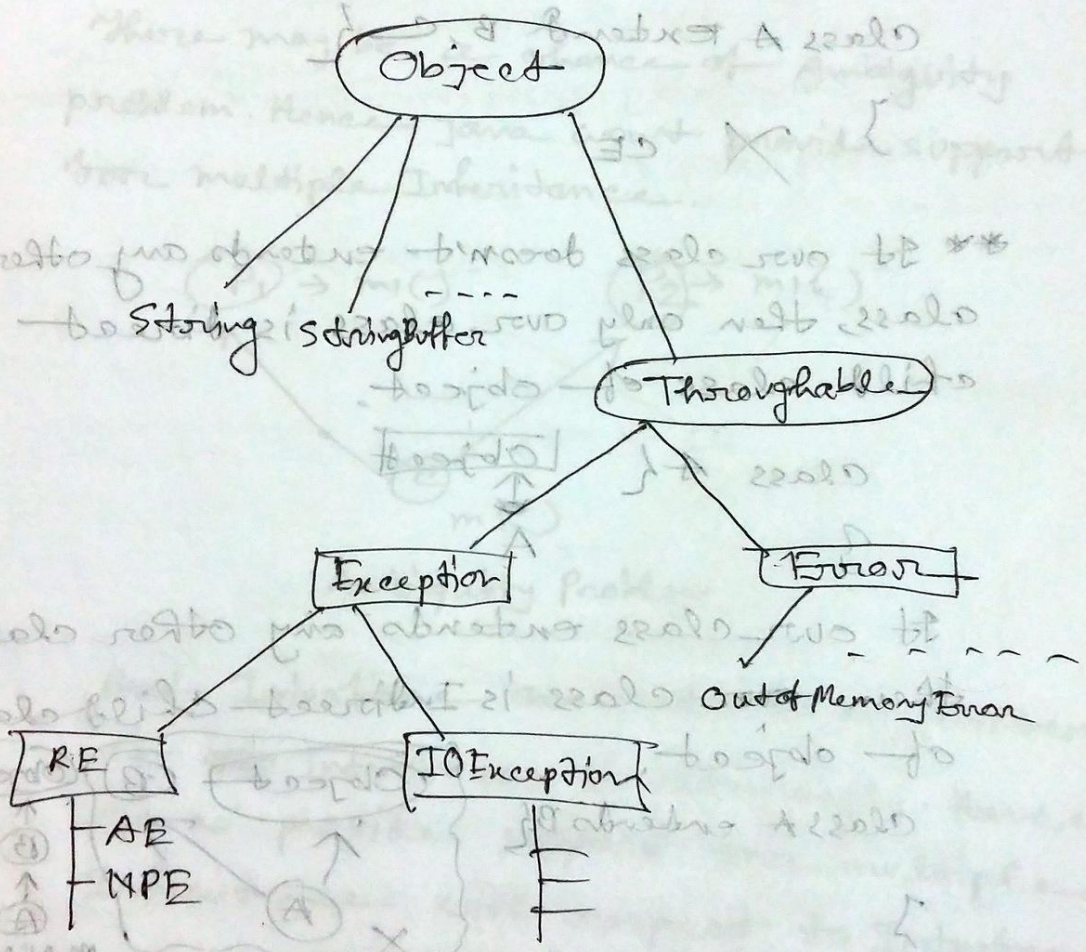
* The most common methods which are applicable for any type of child, we have to define in parent class.

* The specific methods which are applicable for a particular child, we have to define in child class.

* Total Java API is implemented based on Inheritance concept.

* The most common methods which are applicable for any Java object (classes) are defined in "Object" class. Hence every class in Java is a child class of object either directly or indirectly. So that "Object" class methods by default available to every java class without re writting. Due to this "Object" class acts root for all Java classes.

"Throughable" class defines most common methods which are requier for every Exception and Error classes. Hence this class acces root for java Exception Hierarchy.



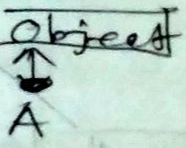
Multiple Inheritance:

A Java class can't extends more than one class at a time. Hence Java won't provide support for multiple Inheritance in classes.

```
class A extends B, C {
}
```
✗ CE

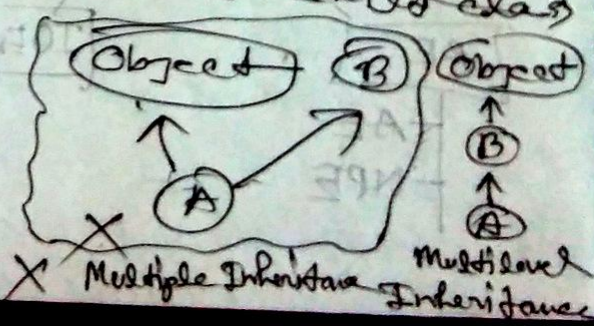
** If our class doesn't extends any other class, then only our class is direct child class of object.

```
class A {
}
```



If our class extends any other class then our class is Indirect child class of object.

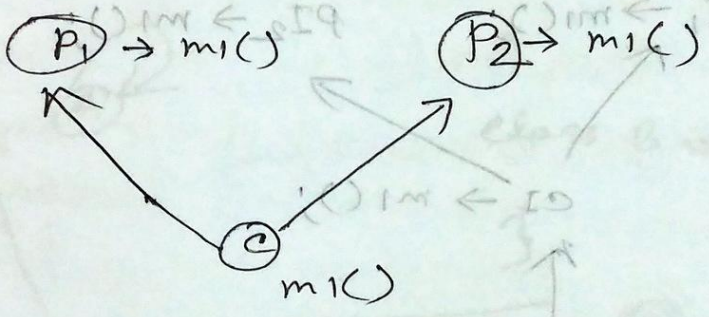
```
class A extends B{
}
```



** Either directly or Indirectly Java won't provide support for Inheritance with respect to classes.

* Why java won't provide support for multiple Inheritance?

There may be a chance of Ambiguity problem. Hence Java won't provide support for multiple Inheritance.



Ambiguity Problem

But Interface can extends any number of Interfaces simultaniously. Hence Java provide support for multiple Inheritance with respect to Interfaces.

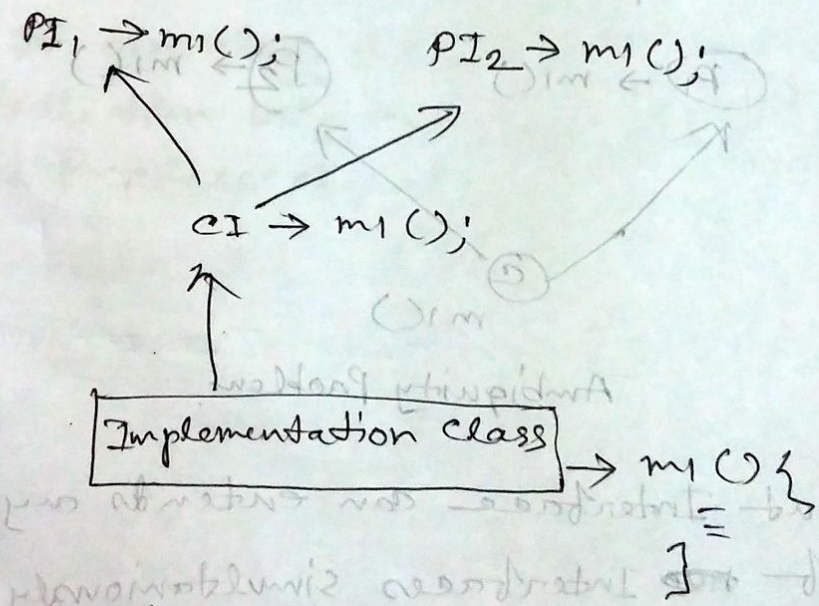
```
interface A {
}
```

```
interface B {
}
```

```
interface C extends A, B {
}
```

✓

* Why ambiguity problem won't be there in interfaces?



Eventhough multiple method declaration class are available but implementation is unique and hence there is no chance of

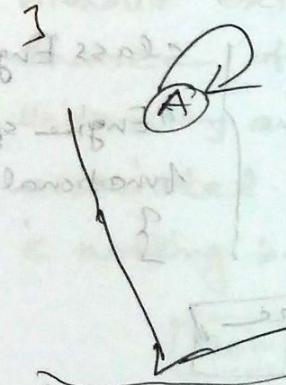
Ambiguity problem in interfaces.

** Strictly speacking through Interfaces we wont get any inheritance.

cyclic Inheritance:

cyclic Inheritance is not allowed in Java, of course it is not required.

```
class A extends A{
}
```




```
class A extends B{
}
class B extends A{
}
```

CE: cyclic Inheritance involving A

Has-A Relationship:

Has a relationship is also known as composition or aggregation.

There is no specific keyword to implement has a relation but most of the time we are depending on new keyword.

The main advantage of has a relationship is reusability of the code.

Ex:

```
class Car{
    Engine E = new Engine();
}
```

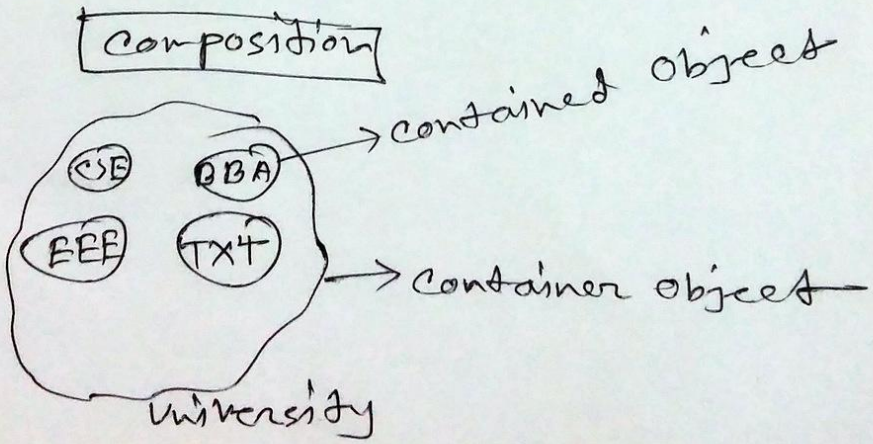
```
class Engine{
    // Engine specific functionality
}
```

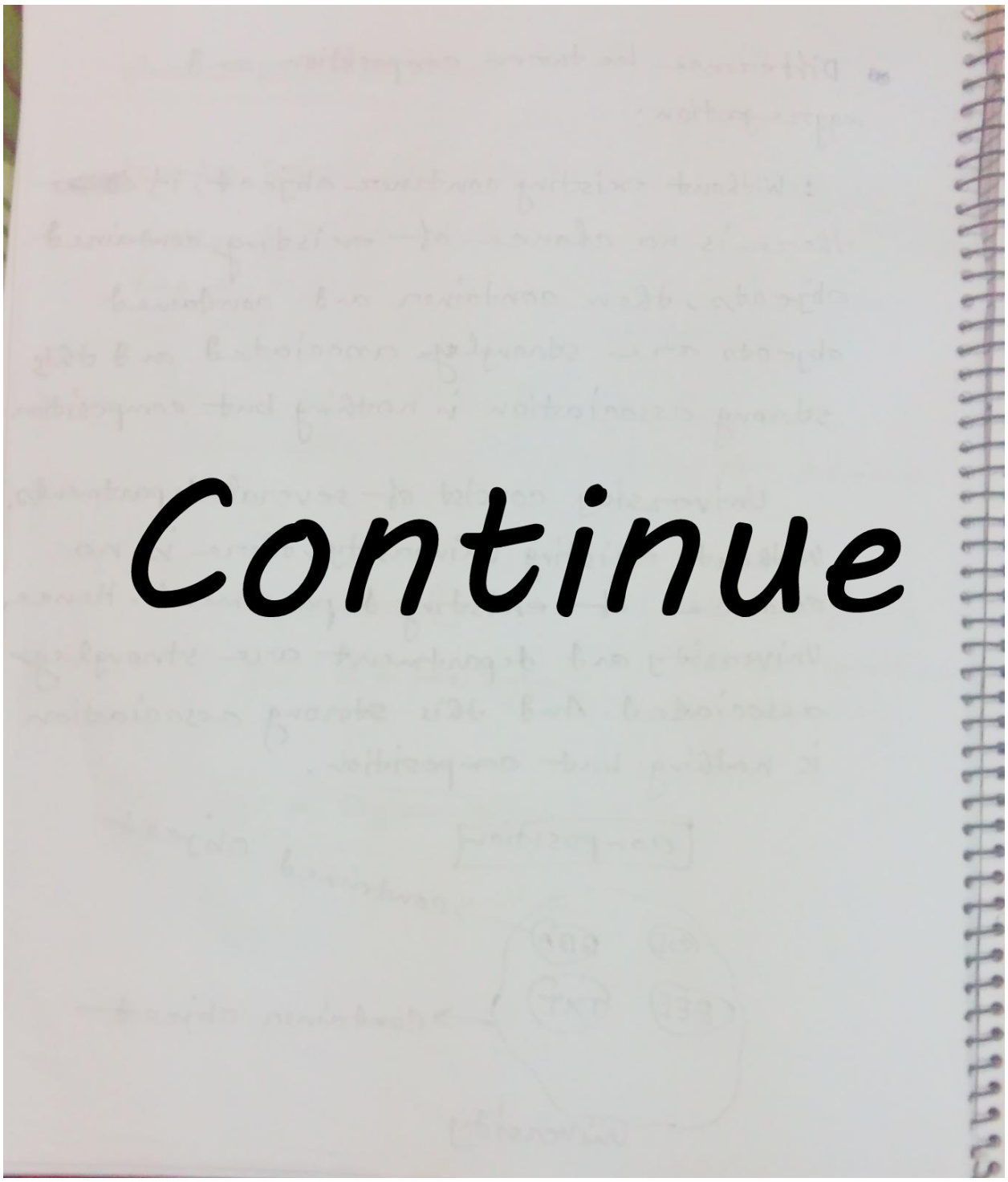
Car has a Engine reference

* Difference between composition and aggregation:

1. Without existing container object, if there is no chance of existing contained objects, then container and contained objects are strongly associated and this strong association is nothing but composition.

University consist of several departments. Without existing university, there is no chance of existing department. Hence University and department are strongly associated. And this strong association is nothing but composition.




Continue

Polymorphism

* One method, in many forms.

two types:

1. Compile time Polymorphism / Static Binding / early binding.
   Ex: Method overloading.
2. Runtime Polymorphism / Dynamic or late binding
   Ex: Method Overriding.

Overloading

Three types:

① Method overloading
② Constructor Overloading
③ Operator Overloading

Method Overloading:

Class contain morethan one method with same name but different number of arguments.

or

more than one then with same name,
- same number of arguments but different type of argument.

```
class Test{
overloaded method { void m1(int a)
                    void m1(int a, int b)

overloaded method { void m2(int a)
                    void m2(char a)
```

This concept is called method overloading concept.

Constructor overloading:

```
class Test {
    Test (int a){
    }
    Test (int a, int b){
    }
    Test (char a){
    }
    p s v main (String[] args){
        new Test (10);
        new Test (10, 20);
        new Test ('a');
```

Operator Overloading:

5 ⊕ 6 = 11 → add

"ab" ⊕ "Bang" = abBangla → concate

One Java class is enough to achieve overloading.

Overriding:

For overriding two classes needed with inheritance relationship.

```
class Parent {
    void marry(){          → overriden method
        S.O.P("Black Girl");
    }
}

class Child extends Parent {
    void marry(){          → overriding method
        S.O.P("White girl");
    }
}
```

1: Method implementation already in parent class, child will implement again.

8 Rules of overriding:

1. Parent class method signature and child class method signature will be same.

2. Parent class method return type and child class method return type at primitive level must be same

3. Class level it is possible to change return type:

```
class Animal { }
class Dog extends Animal { }

class A {
    Animal m1() { }
}
class B extends A {
    Dog m1() { }
}
```

covariant return type concept, added in 1.5 V

4/ If a method declared as final, it is not possible to override.

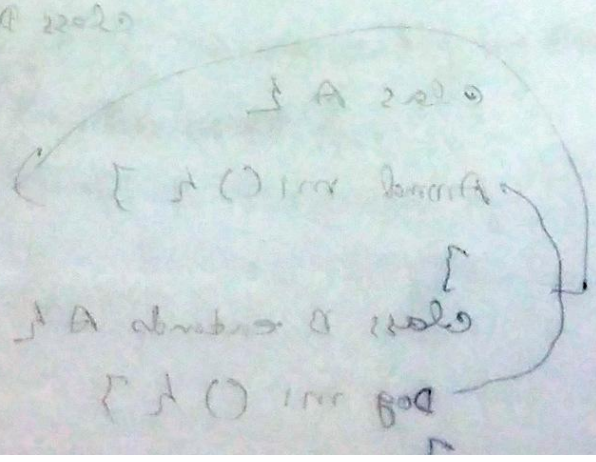
Note: class final → can not Inheritance

method final → override not possible

variable final → it is not possible to change or reassign.

in local variable, only final modifier is possible.

Final class variables are not final by default, but methods are final by default.



51

```
class Parent {
}
class Child extends Parent {
}
```

Parent p = new Parent(); valid

Child c = new Child(); valid

Parent p = new Child(); valid

~~Parent class reffer~~

Child c = new Parent(); Invalid

```
class Parent {
    void m1() { }
}
class Child extends Parent {
    void m1() { }  // overriding
    void m2() { }  // child method

    p s v main(String[] args) {
        Parent p = new Child();
        p.m1();  // compile: Parent runtime: child
        p.m2();  // compile: Parent compiler error
    }
}
```

O/P: child m1 method.

compile time, compiler will check if any m1() method in Parent class, if there then ok. But runtime it will execute child (Object) class method m1();

compile time, compiler will check if any m2() method in Parent, if not then it will give error message

we can call m2 method using p

```
Child c = (Child) p;
    c.m2();
```

```
class Parent {
  static void m1() { s.o.p("Parent m1"); }
}
class Child extends Parent {
  static void m1() { s.o.p ("Child m1"); }
  p.s  v  main(String[] args) {
    Parent p = new Child();
    p.m1();
  }
}
```

O/P: Parent m1

static method is binding with class.
static method is not possible to override.
This method is called method hiding concept.

* Instance methods are bond with object.

6/ private variables and method

Private variable and methods are accesible within the class only

^ ~~If father~~ Parent private not accers by child.

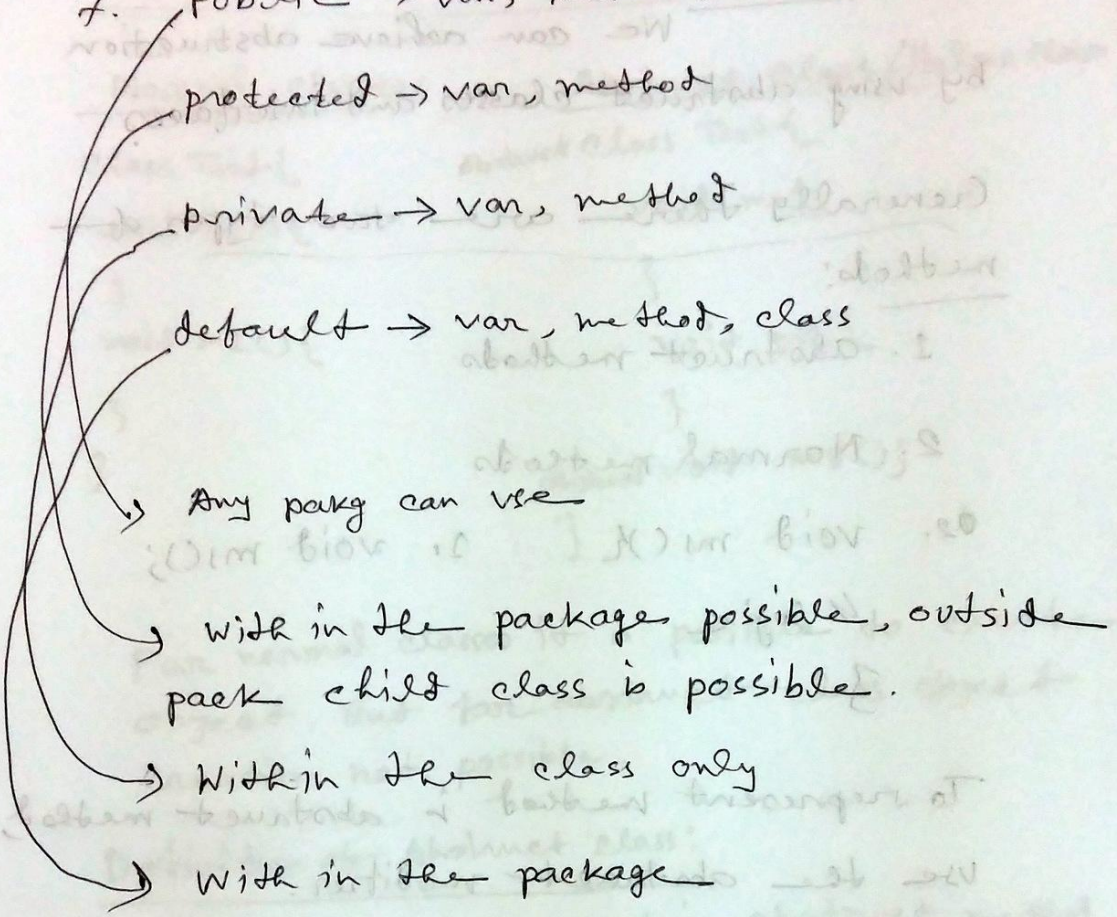
private method override not possible.

7. Public → var, method, class

protected → var, method

private → var, method

default → var, method, class



Abstraction:

We can achieve abstraction by using abstract classes and interfaces.

Generally there are two types of methods:

1. abstract methods
2. Normal methods

```
2. void m1(){
    // logic
   }
```

```
1. void m1();
```

To represent method is abstract method, use the abstract modifier.

```
abstract void m1();
```

This is the syntax of abstract method.

Classes are also two types:

Normal classes:

```
class Test{
void m1(){
}
void m2(){
}
}
```

Abstract class / Helper classes

```
abstract class Test{
    void m1{
    }
    void m2{
    }
abstract void m3();
}
```

For normal classes it is possible to create object. But for abstract class object creation not possible.

Definition of Abstract class:

Abstract class may contain abstract method, may not ~~pos~~ contain abstract method, but it is not possible to create object.

Continue

Core Java Basics

* Class contains five elements

1. Variables
2. Methods
3. Constructor
4. Instance block
5. Static Block

Variables

Local | Instance | static

Local variable:

1. Inside method or constructor or blocks
2. scope of the variable

```
void m1(){
  int a = 10; //Local variable
  system.o.println(a); possible
}
void m2(){
  system.o.println(a); not possible
}
```

3. Memory allocation.

```
void m1(){ // memory allocated when method start
  int a=10; // local variable
  int b=20; //  "      "
} // memory destroyed when method is completed
```

4. Stored memory.

→ Local variable stored in stack memory.

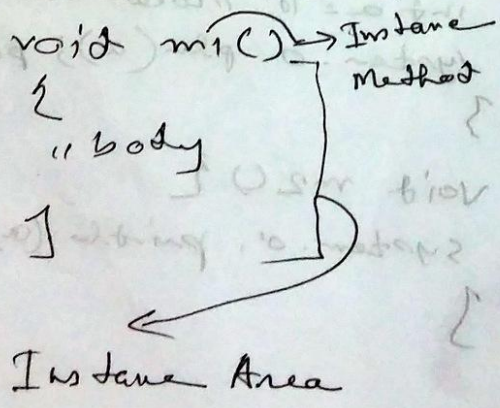
✱ Instance Variables:

Java contains two types of areas:

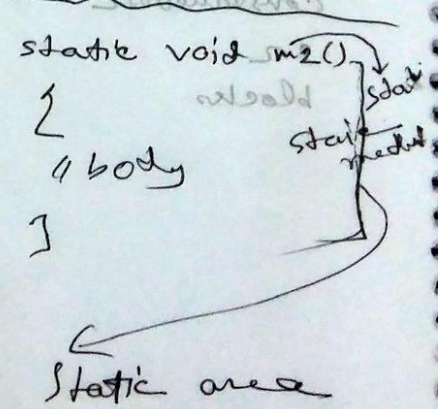
① Instance area.

② static area.

Instance area:



Static Area:



* Instance variable:

① declared inside the class but outside of the method.

```
class Test{
    int a=100; // Instance variable
    int b=200; //  "          "
    public static void main(String[] args){
    }
    void m1(){
    }
}
```

static method → main

static area → main

Instance method → m1

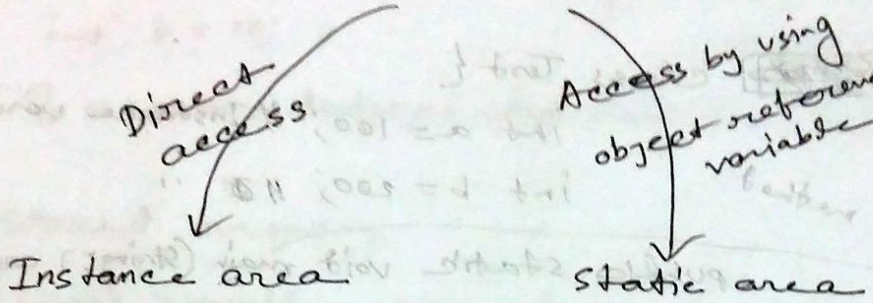
static area → m1

② Scope of the Instance variable:
Inside the class all methods and constructor and blocks

③ Instance variable memory allocation when object is created.

(i) stored in heap memory.

(ii) Access permission:

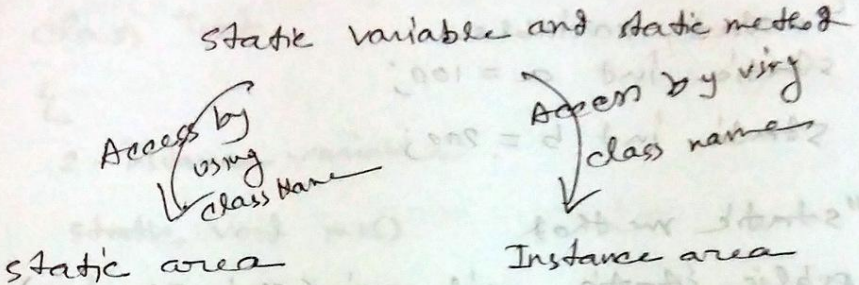



```
class Test {
    int a = 100;  // Instance variable
    int b = 200;

public static void main (String[] args)   // static method
{   // static area
Test t = new Test();
System.out.println(t.a);
                  (t.b);
}

void m1()   // Instance Method
{ // Instance Area
System.out.println(a);
                  (b);
}
}
```

* Static variables:



① declared inside the class but outside the method with static modifier.

② scope is with in the class

③ ·class ~~being~~ loading memory allocate→
·class unloading memory destroyed.

④ stored in non-heap memory.

```
class Test
{
// static variable
static int a = 100;
static int b = 200;

// static method
public static void main(String[] args)
{ // static area

System.out.println(Test.a);
"                  (Test.b);
}
// Instance method
void m1()
{ // Instance area

System.out.println(Test.a);
"                  (Test.b);
}

}
```

Assignment 1 :

```
class Test
{
 2 - instance variable
 static void m1()
 {
  print 2 variable
 }
 static void m()
 {
  print 2 variable
 }
 public static void main (String[] args)
 {
   call m1();
   call m2();
 }
}
```

call static variable in Three ways:

```
class Test
{
static int a = 100;

public s v main (String[] args)
{
System.o.println(Test.a); → 1
  "   "   "   (a); → 2
Test t = new Test();
System.o.println(t.a); → 3
}
}
```

Default values:

byte 0

short 0

int 0

long 0

float 0.0

double 0.0

char single space

boolean false

* For classes default value is null.

Assignment 2:

```
class Test
{
 2- instance variable
 2 - static variable

 void m1()
 {
  print 4 - variables.
 }
 static void m2()
  {
  print 4- variables
 }
public static void main (String[] args)
  {
    call m1();
    call m2();
  }
}
```

class vs object

(i) class is the logical entity.
Object is the physical entity which represent memory.

(ii) class is the blue print, it decide object creation.

(iii) Based on single class, it is possible to create multiple object, but every object need some memory.

```
class Test
{
    int a = 10;
static int b = 20;

    public static void main (String[] args)
    {
    Test t = new Test();

    S.O.P (t.a); O/P: 10
    S.O.P (t.b); O/P: 20

    t.a = 100;
    t.b = 200;

    S.O.P (t.a); O/P: 100
    S.O.P (t.b); O/P: 200

    Test t2 = new Test();
    S.O.P (t2.a); O/P: 10
    S.O.P (t2.b); O/P: 200
    }
}
```

* Types of methods:

two types:

Instance method

```
void m1(){
  //busines logic
}
```

static method

```
static void m2()
{
  //business logic
}
```

Method syntax

Modifier list return-type method name (params-list) throws Exception

Method signature

method name (params-list)

* Three parts of method:

```
void m1() // method declaration
{
  logic here // method Implementation
}
Test t = new Test();
t.m1(); // method calling
```

Example of method:

```
class Test
 {
  void m1(int a, char ch) // local variable
  {
  System.out.println("m1 method");
   "     "  "    (a);
   "     "  "    (ch);
  }
  static void m2(String str, double d)
  {
   System.out.println("m2 method");
     "     "    (str);
     "     "    (d);
  }
 public static void main(String[] args)
 {
  Test t = new Test();
  t.m1(5, 'b');
  Test.m2("abc", 5.6);
 }
}
```

Eum 2:

```
class X { }
class Emp { }
class Y { }
class Student { }

class Test
{
 void m1 (X x, Emp e)
 {
  system.out.println("m1 method");
 }
 static void m2 (Y y, Student s)
 {
  system.out.println("m2 method");
 }
public static void main (String[] args)
 {
  Test t = new Test();

  X x = new X();
  Emp e = new Emp();
  t.m1(x, e); // t.m1(new X(), new Emp());

  Y y = new Y();
  Student s = new Student();
  Test.m2(y, s); // Test.m2(new Y(), new Student())
 }
```

Exp: 3

```
class Test
{
  void m1()
  {
  System.out.println("m1 method");
  }

  void m1()
  {
  System.out.println("m2 method");
  }

 public static void main(String[] args)
  {
    Test t = new Test();
    t.m1();
  }
}
```

O/p: error: method m1() is already defined in class Test.

N.B: Two Methods with same signature are not allowed in java. (Inside a class)

* For java method return type is mandatory.

* Inside method, another method declaration is not allowed in java.

* But inner class are possible.

```
class Test
{
 void m1()
 {
  m2();
  System.o.println("m1 method");
 }
 void m2()
 {
  System.out.println("m2 method");
  m3(5);
 }
 void m3(int a)
 {
  System.out.println("m3 method");
 }
 public static void main(String[] args)
 {
  Test t = new Test();
  t.m1();
 }
}
```

```
class Test
{
    int x = 100;
    int y = 200;

    void add(int a, int b)
    {
     System.out.println(x+y);
                        (a+b);
    }

public static void main(String[] args)
    {
     Test t = new Test();
     t.add(1000, 2000);
    }
}
```

```
    void add (int x, int y)
    {
     System.out.println(x+y);
                        (x+y);
    }
```

x+y → represent local variable

If we want to get instance variable then, we have to use "this" key word

this.x, this.y

* Inside static method we can not use "this" keyword for instance variable

* S.O.P ("Bang" + "Desh");

  O/P: BanglaDesh

  S.O.P(10 + 20);

  O/P: 30

Java is not supporting operator overloading, but only one implicit operator is "+"

Data Types:

```
int a = 10;
```

int --> Data types

a ---> variable name

= ---> assignment operator

10 ---> constant value or literal

; ---> statement terminator

printing variables:

```
int eid = 100;

S.O.P (eid); //valid
S.O.P ('eid'); //Invalid
S.O.P ("eid"); //Invalid
```

Constructor:

Object creation:

① new keyword.

② instance factory method

③ static factory method

④ pattern factory method

⑤ newInstance() method

⑥ clone() method

⑦ Deserialization process.

Using new keyword:

```
class Test
{
}

Test t = new Test();
```

Test --> class name

t --> reference var or object name

= --> Assignment operator

new --> keyword (used to create the object)

Test() --> Constructor

Rules to declare constructor in java:

1. Constructor name and class name must be same.
2. constructor able to take params.
3. constructor are not allowed return types.

```
class Test
{
 void m1()
 {
  System.out.println("m1 method");
 }
 public static void main(String[] args)
 {
  Test t = new Test();
  t.m1();
 }
}
```

```
Test() // Default constructor
{
 // Empty implimatation
}
```

Types of constructor:

1. Default constructor (provided by compiler)
→ Default constructor are always zero argument constructor, with empty implementation.

2. User defined constructor.

Compile time, compiler will check is there is any constructor available or not, if not exist then compiler will generate default constructor

```
class Test
{
 Test()
 {
 System.o.println("0-arg constructor");
 }
 Test(int a)
 {
 System.o.println("1-arg constructor");
 }
 void m1()
 {
 System.o.println("m1 method");
 }
public s v main(String[] args)
 {
 Test t = new Test();
 Test t1 = new Test(10);
  t.m(); t1.m1();
 }
}
```

o/p: 0-arg constructor
1-arg "
m1 method
m1 method

```
class Test
{
 Test(int a)
 {
  System.o.p("1-arg constructor");
 }

P. S. V. main(String[] args){
   Test t = new Test();
   Test t1 = new Test(10);
 }
}
```

compile time error:

If there is no user defined constructor, then only default constructor will generate.

Advantage of constructor :

```
class Emp
{
 int eid;
 string ename;
 float esal;

 void disp()
 {
  system.o.p("emp id=" + eid);
  system.o.p("emp name=" + ename);
  system.o.p("emp salary=" + esal);
 }

 p.s v main(String[] args)
 {
  Emp e = new Emp(); // default constructor
                        executed
  e.disp();
 }
}
```

o/p: 0
null
0.0

But if we add a constructor

```
Emp(int eid, String ename, float esal)
{
  this.eid = eid;
  this.ename = ename;
  this.esal = esal;
}
```

or

```
eid = 100;
ename = "mouduf";
esal = 5000.0f;
```

Then in main method:

```
Emp e = new Emp(1, "mouduf", 5000.0f);
```

or

```
Emp e = new Emp();
```

Two advantages:

1. Constructors are used to write the logics, those logics are executed object creation.

2. Constructors are used to initialized instance variables during object creation.

```
class Emp{
    int eid;
    String ename;
    float esal;
    Emp(int eid, String ename, float esal){
     this.eid = eid; this.ename = ename;
     this.esal = esal;
    }
    void disp(){
    S.O.P("eid:" + eid);
    S.O.P("ename:" + ename);
    S.O.P("esal:" + esal);
    }
    P.S.V main(String[] args){
    Emp e = new Emp(1, "mou", 5.6);
    e.disp();
    Emp e1 = new Emp(2, "tas", 7.8);
    e1.disp();
    }
}
```

* calling constructor:

```
class Test {
    Test(){
      S.O.P("0-arg cons");
    }
    Test(int a){
      S.O.P("1-arg cons");
    }
    Test(int a, int b){
      S.O.P("2-arg cons");
    }
    public static void main(String[] args){
    Test t1 = new Test();
    Test t2 = new Test(10);
    Test t3 = new Test(10,20);
    }
}
```

```
class Test {
  Test() {
    this(10);
    S.O.P("0-arg cons");
  }
  Test(int a) {
    this(10,20);
    S.O.P("1-arg cons");
  }
  Test(int a, int b) {
    S.O.P("2-arg cons");
  }
  public static void main(String[] args) {
    Test t = new Test();
  }
```

same from last example

```
Test () {
    S.O.P ("0-arg cons");
    this(10);
}
```

→ constructor calling inside constructor using "this" keyword must be first statement.

But for methods it could be any statement.

```
Test () {
    this (10);
    this (10,20);   → not first constructor that's why error
    S.O.P ("0-arg cons");
}
```

* one constructor is able to call one constructor at a time.

Formats of object creation:

① Named object approach

Test t = new Test();

② Nameless approach

new Test();

Instance Block:

purpose/or ~~Advantage~~ of constructor:

1. write the logics
2. Initialize instance variable

* purpose of Instance block is same:

1. Write the logic
2. Initialize instance variable.

both are executed after object creation only.

Syntax of Instance Block:

```
{
  # logic Here
}
```

```
class Test{
    Test(){
     S.O.p("0-arg constructor");
    }
    {
     S.O.p("instance Block);
    }
    public static void main(String[] args){
        Test t = new Test();
    }
}
```

During object creation instance block will be executed first, then constructor.

```
class Test{
    Test(){
     S.O.p("0-arg constructor");
    }
    {
     S.O.p("instance Block);
    }
    public static void main(String[] args){
        Test t = new Test();
    }
}
```

During object creation instance block will be executed first, then constructor.

```
class Test{
  Test(){
   S.O.P("0-arg");
  }
  Test(int a){
   S.O.P("1-arg");
  }
  Test(int a,int b){
   S.O.P("2-arg");
  }

  {
   S.O.P("Instance block");
  }
  public static void main(String[] args){
   new Test();
   new Test(10);
   new Test(10,20);
  }
}
```

O/P: Instance block
0-arg
Instance block
1-arg
Instance block
2-arg

```
class Test{
    Test(){
    S.O.p("0-arg");
    }
    {
     S.O.p("Instance block-1");
    }
    {
     S.O.p("Instance block-2");
    }
    P.S.V.main(String[] args){
        new Test();
    }
}
```

Inside class multiple instance block is possible. but execution is top to bottom.

Assignment:

```
class Test {
    Test() {
        this(10);
        S.O.P("0-arg constructor");
    }
    Test(int a){
        S.O.P("1-arg con");
    }
    {
        S.O.P("Instance block");
    }
P.S.V main(String[] args){
    new Test();
}
}
```

O/P: Instance block
1-arg con
0-arg constructor

Instance block creation depends on object creation, but not constructor calling.

Static Block:

Used to write the logics, and executed during .class file loading.

Syntax:

```
static
{
  // logic Here
}
```

```
class Test {
  Test() {
    S.O.P("0-arg cons");
  }
  {
    S.O.P("Instance Block");
  }
  static {
    S.O.P("static Block");
  }
  public static void main(String[] args) {
    new Test();
    new Test(10);
  }
}
```

one more constructor

one more Instance block

one more static Block

Basic OOP of Core Java

Main Building Block of oop

1. Inheritance
2. Polymarphism
3. Abstruction
4. Encapsulation
5. Class
6. Object

Inheritance:

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

class B {
void m1(){ }
void m2(){ }
void m3(){ }
void m4(){ }
}
```

```
class C {
void m1(){ }
void m2(){ }
void m3(){ }
void m4(){ }
void m5(){ }
void m6(){ }
```

Disadvantage:

1. Redundancy (Duplicate)
2. Length of the code

Advantage of Inheritance:

① Reduce Redundancy of the application.

② Reduce length of the code.

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

class B extends A {
    void m3() { }
    void m4() { }
}

class C extends B {
    void m5() { }
    void m6() { }
}
```

Object creation:

* It is possible to create object for both parent and child class.

* If you create parent class object then you will get only parent property.

* If you create child class object, you will get both parent and child class property.

Note 1: If you extending a class, your class become parent, if you are not extend any class, then "Object" will be the parent class.

```
class A  //father of A is "Object"
{

}
class B extend A  // B is "A" class
{

}
```

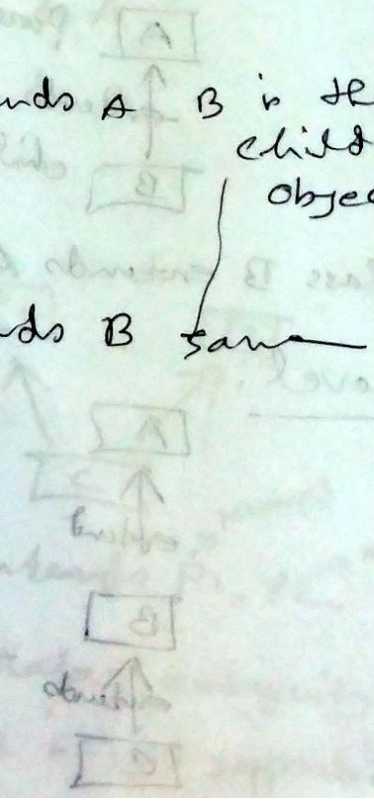
Note 2: The root class of all java classes are "Object" class.

Note 3: Every class of java are direct or indirect child class of "Object" class.

```
class A
{
}
class B extends A
{
}
class C extends B
{
}
```

A is the direct child class of Object

B is the indirect child class of Object

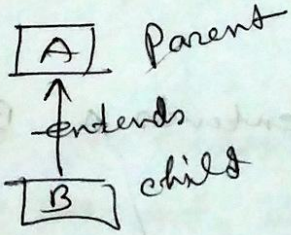


Types of Inheritance:

1. Single
2. Multiple
3. Multilevel
4. Hireachy
5. Hybrid

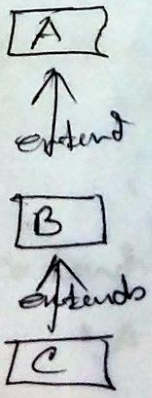
These two are not supported by java

Single:



class B extends A

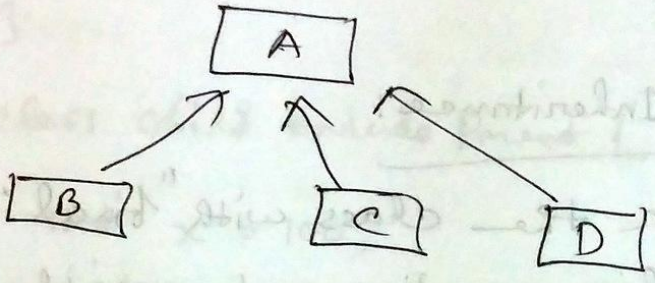
Multilevel:



class C extends B

class B extends A

Hierarchy:

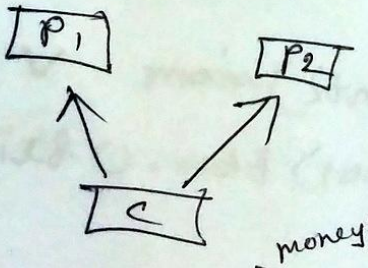


class B extends A

class C extends A

class D extends A

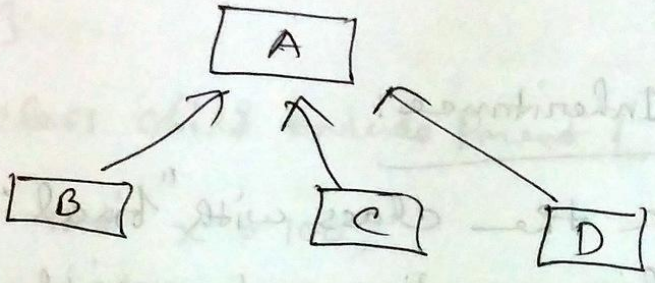
Multiple:



class C extends P1, P2 (P1: money(), P2: money())

It will generate ambiguity problem.
That's why this is not supported by java.

Hierarchy:

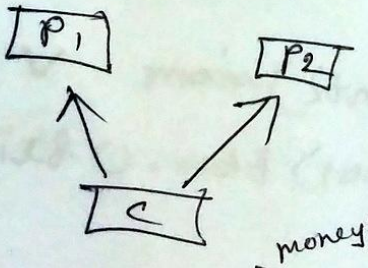


class B extends A

class C extends A

class D extends A

Multiple:



class C extends P1, P2 (P1: money(), P2: money())

It will generate ambiguity problem.
That's why this is not supported by java.

Hybrid: ✗

Multiple + Hierarchy

✗

Prevent Inheritance:

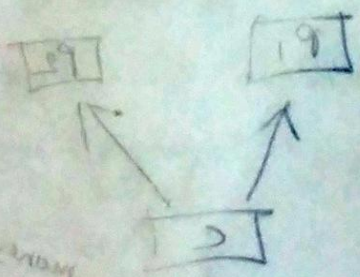
Declair the class with "final" modifier, child class creation not possible

```
final class A{

}

class B extends A{  // not possible

}
```



```
class Parent {
    int a = 10;
    int b = 20;
}

class child extends Parent {
    int x = 100;
    int y = 200;

    void add (int i, int j) {
        s.o.p (i+j);
        s.o.p (x+y);
        s.o.p (a+b);
    }

P.S.V main(String[] args) {
    new child().add(100,200);
}
}
```

```
int a = 10;
int b = 20;

int a = 100;
int b = 200;

(int a, int b)
s.o.p (a+b)
s.o.p (a+b)
s.o.p (a+b)
o/p 30, 30, 30

new child(10,20)

s.o.p (a+b);
s.o.p (this.a + this.b);
s.o.p (super.a + super.b);
```

```
class Test{
  parent variable
  parent methods
  parent constructor
  parent Instance block
  parent static Block
}

class Parent{
  void m1(){System.out.println("parent m1 method");}
}
class child extends Parent{
  void m1(){S.O.P("child m1 method");}
  void m2(){m1();m1();}
P.S.V.main(String[] args){
  child ch = new child();
    ch.m2();
}
}
```

O/P: child m1 method
child m1 method

if we want parent class method, then

```
this.m1(); // this is optional by default it
              calls child class method.
super.m1(); // To call parent class
               method.
```

Parent class constructor:

```
class Parent {
    Parent(){
    S.O.p("parent class 0-arg cons");
    }
}
class child extends Parent {
    child(){ this(10);
    S.O.p("child class 0-arg cons");
    }
    child(int a){ super();
    S.O.p("child class 1-arg cons");
    }
public s v main(String[] args){
    new child();
}}
```

O/P: Parent class 0-arg cons
child class 1-arg cons
child " 0-arg cons

* super() will be the first statement of the constructor.

** **

```
class Parent{
    Parent(){
        S.O.p("Parent 0-arg constructor");
    }
}
class child extends Parent{
    child(){ super(); }
    S.O.p("child class 0-arg con");
}

p.s.v main(String[] args){
    new child();
}
}
```

If we remove Parent class constructor

O/P:
Parent 0-arg constructor
child class 0-arg con

```
* class Parent{
    Parent(int a){
      S.O.p("Parent class 1-arg");
    }
}
class Child extends Parent {
    Child () {
      super(10);
      S.O.p("child class 0-arg");
    }
    p s v main(String[] args) {
      new Child();
    }
}
```

If we replace with "super();"

```
class Parent {
    Parent() {
        S.O.P("Parent 0-arg");
    }
}

class Child extends Parent {
    Child() { // By default "super();"
        S.O.P("Child class 0-arg cons");
    }
    public s v main(String[] args) {
        new Child();
    }
}
```

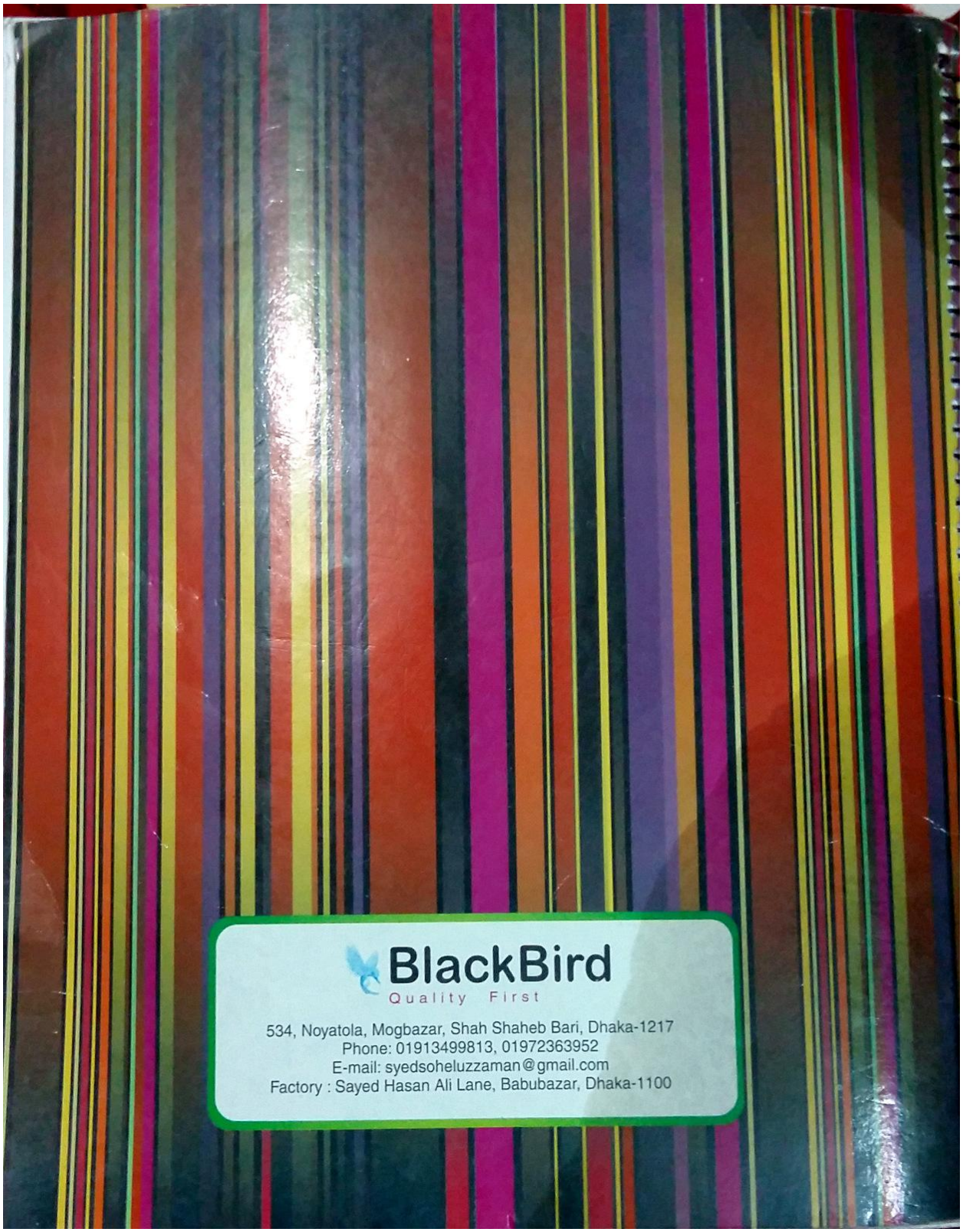
O/P: Parent 0-arg cons
Child 0-arg cons

```
class Parent{
  parent(){
    S.O.P("Parent class o-arg");
  }
}
class child extends Parent{
  /* code is generated by compiler
     child(){
       super();
     } */
  p.s.v main(String[] args){
    new child();
  }
}
```

o/p: 'parent o-arg"

Instance Block:

We know Instance block execute during object creation. First Parent class Instance block then child class instance block will be ~~create~~ execute.


BlackBird
Quality First
534, Noyatola, Mogbazar, Shah Shaheb Bari, Dhaka-1217
Phone: 01913499813, 01972363952
E-mail: syedsoheluzzaman@gmail.com
Factory : Sayed Hasan Ali Lane, Babubazar, Dhaka-1100