1. **OOP s Basics & Principles.**
2. **Inheritance.**
3. **This**
4. **Super**
5. **Difference between this and super.**
6. **Abstraction**
7. **Encapsulation**
8. **Polymorphism**
9. **Constructor Overloading and Constructor Chaining.**
10. **Final**

**OOPS BASICS**

Small talk is pure object-oriented programming language because everything is in the form of object. The java is not pure object oriented programming Because of primitive data types. For fast executing , Java supports primitive Data types.

Reasons:-

1. It supports primitive Data types to store data.
2. It does not support multiple inheritance using classes.
3. By using Reflection API, we can access private Data of class.
4. We can access object data without creating object of class.

What is Oop?

Principles are rules or guidelines means for running some business smoothly. OoP principles also rules or guide lines for developing a real world object in programming world by achieving security to object data and dynamic binding between HAS-A relation user object & sub type objects in accessing operations from differenct sub types of object.

Examples 1:

1. The object ATM contains cash, as a customer we must not take that cash directly. That cash must be hidden from us from direct access and must allow us to take cahs only via card swipping. This way of accessing object data (ATM cash) is called providing security to object Data.

One of Oop principle “Encapslation” provides guidelines to hide data from direct access from user progrmmer & providing security to obejct data.

1. In ATM, we can perform with draw operation by using any BANK ATM card. Based on card we swipped, ATM automatically reads card information, this card is linked bank operation, out bank A/C Information, bank balance, & then perform with draw operation dynamically from diff banks based in ATM card we swipped. This style of execution is called dynamic binding.

The OOP principle inheirtance, polymorphism will provide suggestions to achieve dynaic binding between user object and sybtype objects.

**OOP Principles**: - The OOP principles are

1.Encapsulation

2. Inheritance.

3.Polymorphism.

If we don’t implement Oop principles in project then project is not secure and no scalable.

**Object Based Programming Language:** A language that has in built objects and that does not support class based programming & does not support creating out own custom objects and does not support OOP principles development is called OBPL.

Object Oriented Programming Language:- a language that supports class based programming and supports creating our own custom object & also supprots all oop principles development is called OOPL.

Example: Java,c#,.NET

**INHERITANCE**

**2.Inheritance**:-

The Inheritance is One of Oop principal.

The Inheritance is a mechanism/technique In which one class will acquire the properties and behaviors of the parent class.

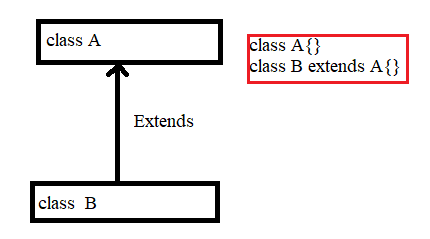
The Inheirtance is a deriving/creating a new class from existing one.

The Inheritance can be implemented in java using below two key words.

1. Extends.
2. Implementation.

**A.Extends:-**The “extends” keyword is used for developing inheritance between two classes or two interfaces.

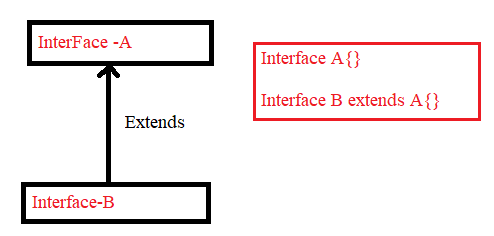
Example:1



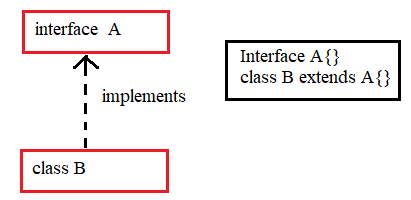
The class that is placed right side of extends keyword is called **super class, parent class or Base class.** The super class is already defined class.

The class which is placed left side of extends keyword is called **subclass, child class or derived class**. The sub class is newly defining class.

Example:2



**B.Implements:-**  It is used for developing inheritance between interface and class.



The class that is placed before “implements” keyword is called **implementation class.**

**Advantages of Inheritances:-**

1. Code is reduced.
2. Code reusability is incresed.
3. We can override existing functionality.
4. We can extend existing functionality.
5. We can implement new functionality.
6. We will get runtime polymorphism.

**2.1.Static members control flow with inheritance:**

When a sub class is loaded, first all its super classes are loaded then subclass is loaded. Then after:

1. All SVs memory is allocated with default values from root super class to current subclass.
2. Then SVs initialization and SBs execution will be done in the order they placed from root super class to current sub class.
3. Finally MM is executed from current sub class. If subclass does not have main method, JVM executes main method from super class, if super class does not have main method, JVM throws exception:Main method not found.
4. If we invoke static methods, JVM executes static methods from current loaded sub class, if it is not available in sub class, it is executed from super class.

**Example:**

class Clerk extends BankAccount

{

static int b=m2();

static int m2(){

System.out.println("B SV");

return 20;

}

static{

System.out.println("B SB");

}

public static void main(String[] args)

{

System.out.println("B main");

System.out.println("B main a:"+ a);

System.out.println("B main b:"+b);

}

}

//Encapsulated class

class BankAccount

{

static int a=m1();

static int m1(){

System.out.println(" A SV");

return 10;

}

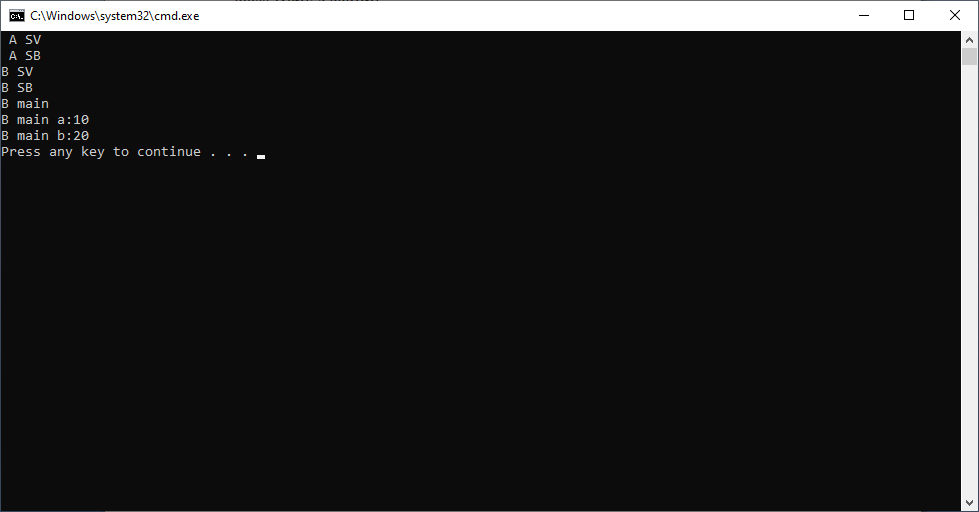
static{

System.out.println(" A SB");

}

}

Output:



Example:2

class BankAccount

{

static int a =10;

static{

System.out.println(" In A SB");

System.out.println("a:"+a);

//System.out.println("b:"+b);

System.out.println("b:"+Clerk.b);

System.out.println("b:"+Clerk.getB());

}

}

//Encapsulated class

class Clerk extends BankAccount

{

static int b=20;

static {

System.out.println("In B SB");

System.out.println("a:"+a);

System.out.println("b:"+b);

System.out.println("b:"+getB());

}

static int getB()

{

return b;

}

public static void main(String arg[])

{

System.out.println("In B main");

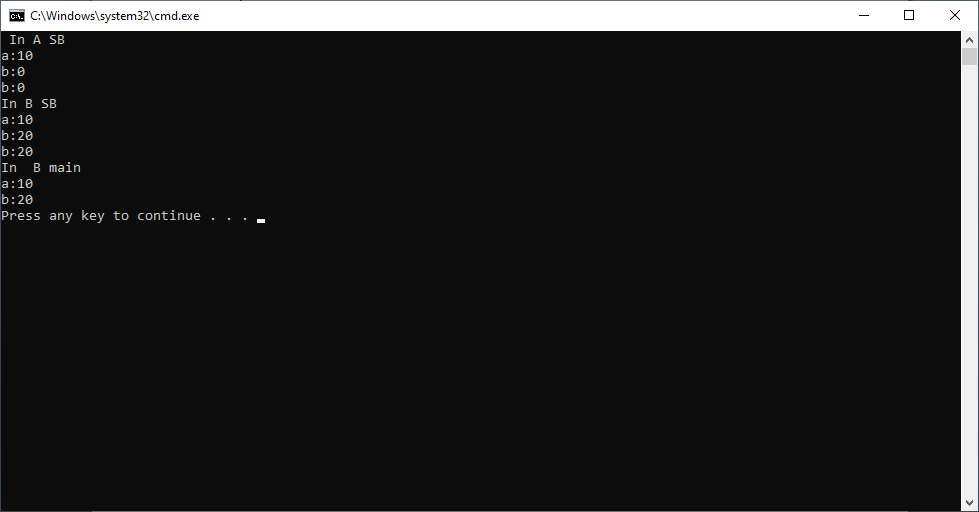
System.out.println("a:"+a);

System.out.println("b:"+b);

}

}

**Output:**

****

**2.2.Non-Static members execution flow with inheritance:**

When subclass object is created, super class non-static variables memory is also created and included in subclass object memory. For initializing super class non-static variables memory, sub class constructor will invoke super class constructor by using super(). In this chain of constructor calls, once the control reached to java.lang.Object class constructor.

1. All NSVs memory will be allocated from super class to sub class in the order they are defined from top to bottom.
2. Then NSVs initialization, NSBs and invoked constructor execution will be done from super class to sub class.
3. If we invoke any non-static method, it will be executed from current sub class; if it is not available in sub class then it is executed from super class.

Example:

class BankAccount

{

Int x=m1();

int m1()

{

System.out.println(" A NSV");

return 10;

}

{

System.out.println("A NSB");

}

BankAccount()

{

System.out.println(" A Constructor");

}

}

//Encapsulated class

class Clerk extends BankAccount

{

int y= m2();

int m2(){

System.out.println(" B NSV");

return 20;

}

{

System.out.println(" B NSB");

}

Clerk()

{

System.out.println(" Clerk Constructor");

}

public static void main(String arg[])

{

System.out.println(" B main");

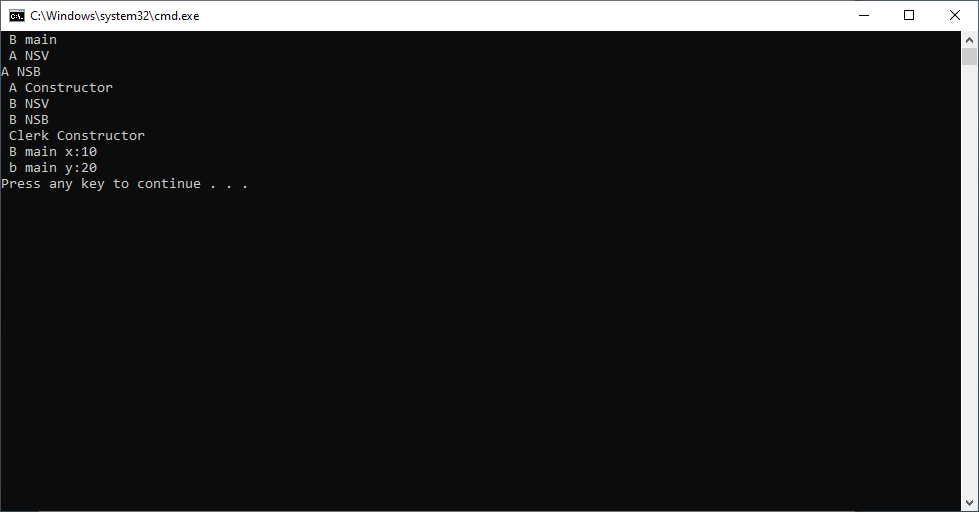
Clerk b1=new Clerk();

System.out.println(" B main x:"+b1.x);

System.out.println(" b main y:"+ b1.y);

}

}



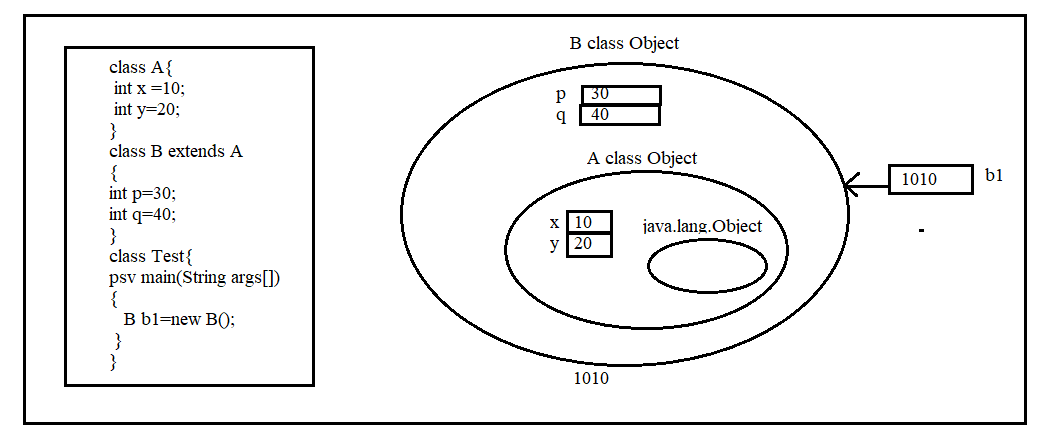
Note:-

1. When U compile the sub class, first parent class is compiled and then sub class is compiled.
2. When you execute the sub class, first parent class is loaded and then sub class is loaded in to JVM.

**2.3.Subclass object structure:-**  when sub class object is created,no super class object is not created. A sub class object will contain the memory for to non-static fields of its own class plus all fields of its parent class, grand parent class, all the way up to root class java.lang.Object . The space for non-static fields are created from java.lang.Object to current sub class.

Ex:-

Java.lang.Object 🡪 A 🡪 B



**2.3.1.Can we access all members of super class in sub class?**

No we can access only visible members. Visible members means non-private members. If we try to access private members of super class in sub class,we will get CE.

* A private member is always no visible.
* Default member is visible in same package sub class & not visible in another package sub class.
* Protected & public members are always visible to sub class either in same package or in another package.
* The super class object can’t get non-static members of sub class.
* The super class object can access static members of sub class using sub class Name.

Example:

class BankAccount

{

private int x=20;

}

//Encapsulated class

class Clerk extends BankAccount

{

public static void main(String arg[])

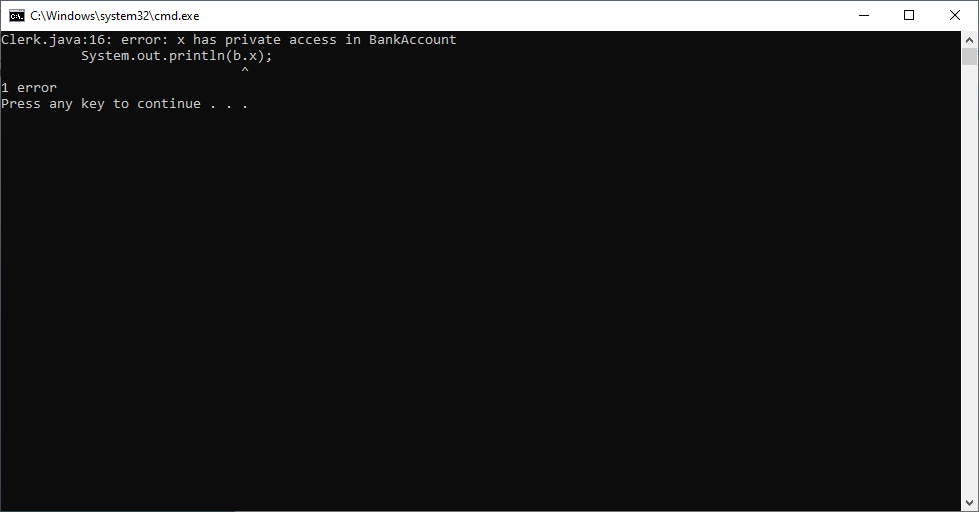
{

BankAccount b=new BankAccount();

System.out.println(b.x);

}

}



When we invoke method by using sub class reference, first that method is searched in subclass, If not available in sub class then method is executed from super class.

The JVM first tries to execute main method from sub class, if not available in sub class then JVM executes it from super class, but not from both classes. If main method is not available in both classes, we get Error: main method is not found.

Example:1

class BankAccount

{

private int x=20;

private int y=30;

public void display1()

{

System.out.println(x);

}

public void display2()

{

System.out.println("Method of super class");

}

}

//Encapsulated class

class Clerk extends BankAccount

{

public void display1()

{

System.out.println(" display in sub class");

}

public static void main(String arg[])

{

Clerk c1=new Clerk();

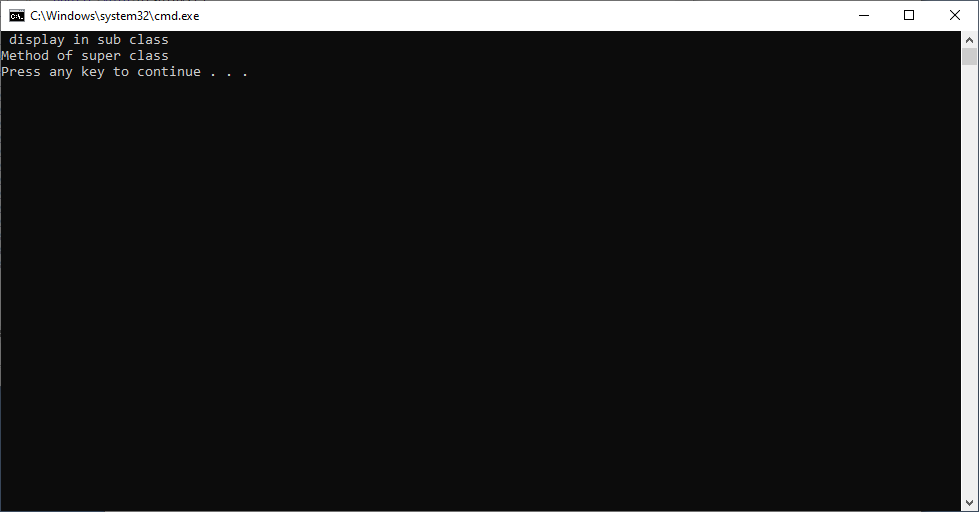
c1.display1();

c1.display2();

}

}

Output:-



Example:2

class BankAccount

{

private int x=20;

private int y=30;

public void display1()

{

System.out.println(x);

}

public void display2()

{

System.out.println(y);

}

public static void main(String arg[])

{

BankAccount b2=new BankAccount();

b2.display1();

b2.display2();

}

}

//Encapsulated class

class Clerk extends BankAccount

{

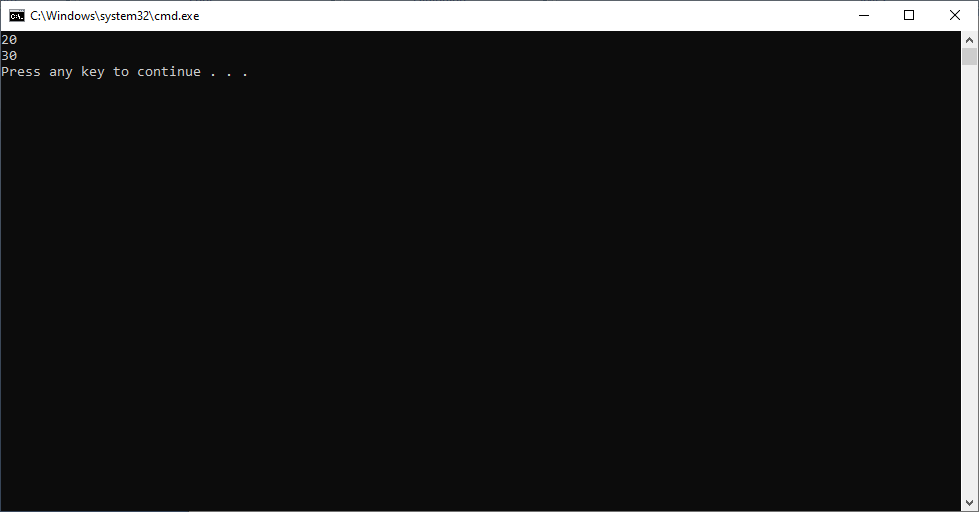
public void display1()

{

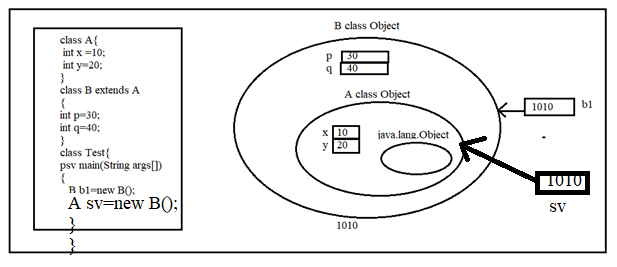
System.out.println(" display in sub class");

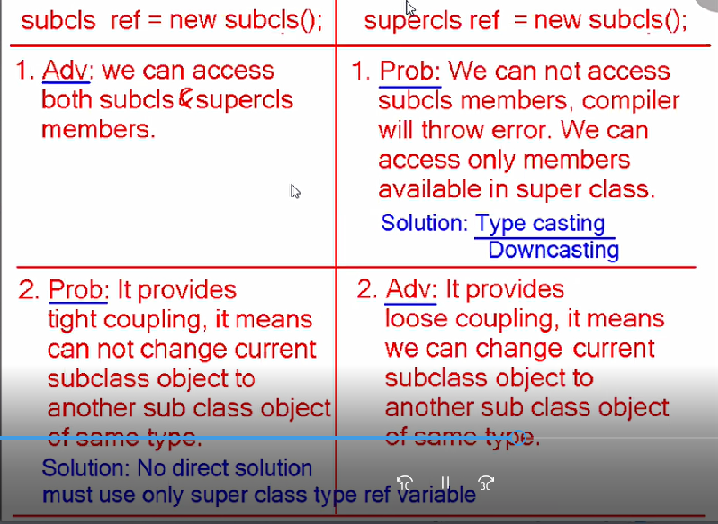
}

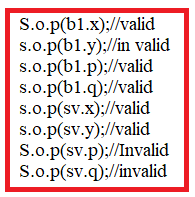
}



**2.4.** Diff between storing sub class object ref in Subclass variable and storing sub class object ref in super class variable.







**2.5.UpCasting:-** Storing subclass object reference in superclass variable is called Upcasting. The upcasting is for achieving the loose coupling. It is **implict refence type conversin**.

Rule:-

Super class variable =new subclassName();

The parent and child relation ship must be between superclass and subclass. Otherwise compiler gives CE.

Ex:-

Class A{}

Class B extends A{}

Class C {}

Class D { A a1=new B(); //valid This is upcasting.

A a2= new C();// invalid

}

Problem: we can’t access sub class object members.

Solution: Downcasting.

Note:- It is not possible to store the super class object reference in sub class variable. It leads to CE: **Incompatable types**.

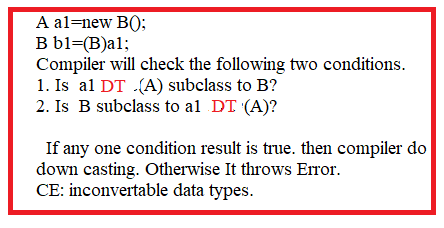
B b1=new A(); // Invalid CE: Incompatable type.

**2.6.DownCasting:-** Converting subclass object reference from superclass to its owntype using cast operator is called downcasting. It is **explicit reference type conversion**.

Syntax:

subclassName var=(subclassName) supeclassvarName;

Rule in using Down casting:- The cast operator type and source type should have inheritance relation(IS-A) . Otherwise It leads to CE: inconvertable type.



Example:

class BankAccount

{

private int x=20;

private int y=30;

public void display1()

{

System.out.println(x);

}

public void display2()

{

System.out.println(y);

}

}

//Encapsulated class

class Clerk extends BankAccount

{

public void display3()

{

System.out.println(" display in sub class");

}

public static void main(String arg[])

{

BankAccount b2=new Clerk();

System.out.println("==================== UpCasting===========================");

b2.display1();

b2.display2();

// b2.display3(); It leads to CE.

Clerk c2=(Clerk)b2;

System.out.println("======================DownCasting=========================");

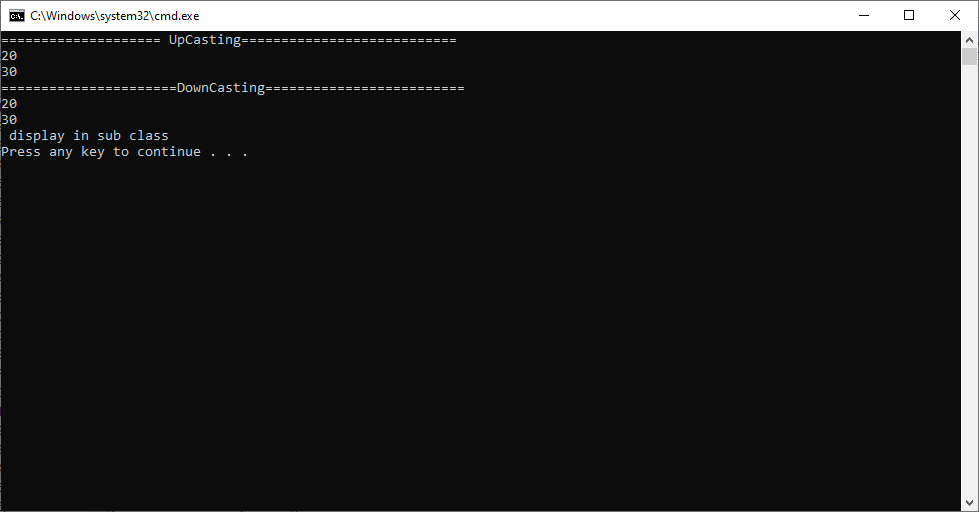
c2.display1();

c2.display2();

c2.display3();

}

}



**Example3:**

class ICICI

{

}

class Bank

{

}

class BankAccount extends Bank

{

}

class Clerk extends BankAccount

{

public static void main(String arg[])

{

Clerk c1=new Clerk();

BankAccount b1=c1;

Bank b11=c1;

Object o1=c1;

b11=b1;

//c1=new ICICI();// CE: Incompatable data types.

// c1=(Clerk)new ICICI(); CE: Incompatable data types.

//Upcasting completed.

c1=(Clerk)b1;

b1=(BankAccount)b11;

o1=(Bank)b1;

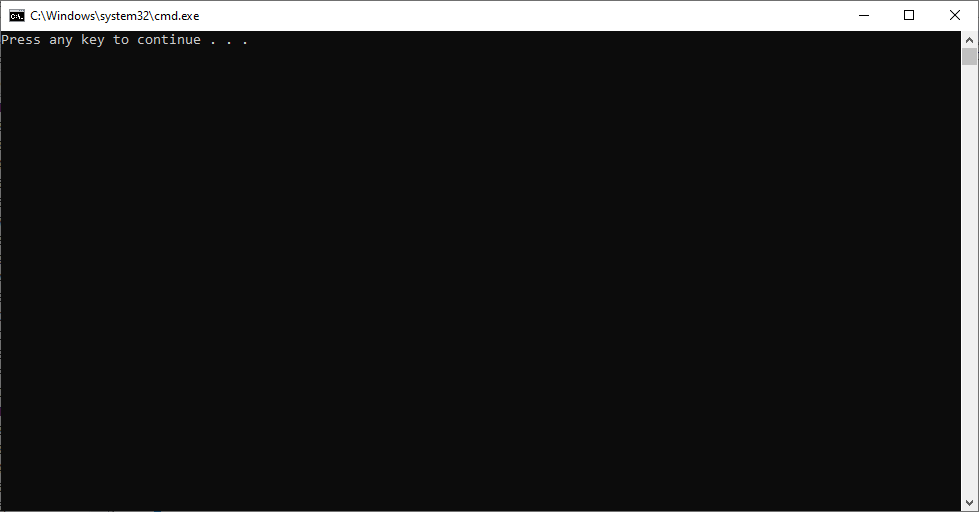
c1=(Clerk)b11;

b1=(BankAccount)b11;

//Down casting completed.

}

}



**2.7.java.lang.classCastException:-** In casting, compiler can’t identify object coming from source variable, because it checks only source variable type and cast operator type has IS-A relation or not. So when source variable type is super class type and cast operator is sub class type compiler always compile this casting.

In casting , the object coming from source variable,if it is not compatable with cast operator type JVM throws “class Cast Exception”.

Example:-

Class A{}

Class B extends A{}

Class C{}

Object o1= new A();

C c1=(C) o1; // This statement does not leads to CE.

// It leads to RE: java.lang.classCastException.

**2.8. instaceof:-** In Java, **instanceof**is an operator which is used to check object reference. It checks whether the reference of an object belongs to the provided type or not. It returns either **true or false**, if an object reference is of specified type or object reference has is-A relation with specified type(class) then it return true otherwise false.

Syntax:

(object)instanceof(type)

Object is object reference variable.

Type is variable name or interface name.

Note:- we can avoid java.lang.classCastException using “instanceof” operator.

Example:-

class ICICI

{

}

class Bank

{

}

class BankAccount extends Bank

{

}

class Clerk extends BankAccount

{

public static void main(String arg[])

{

Clerk c1=new Clerk();

System.out.println(c1 instanceof Clerk);

System.out.println(c1 instanceof BankAccount);

System.out.println(c1 instanceof Bank);

System.out.println(c1 instanceof Object);

BankAccount b1=new BankAccount();

System.out.println(b1 instanceof Clerk);

System.out.println(b1 instanceof BankAccount);

System.out.println(b1 instanceof Bank);

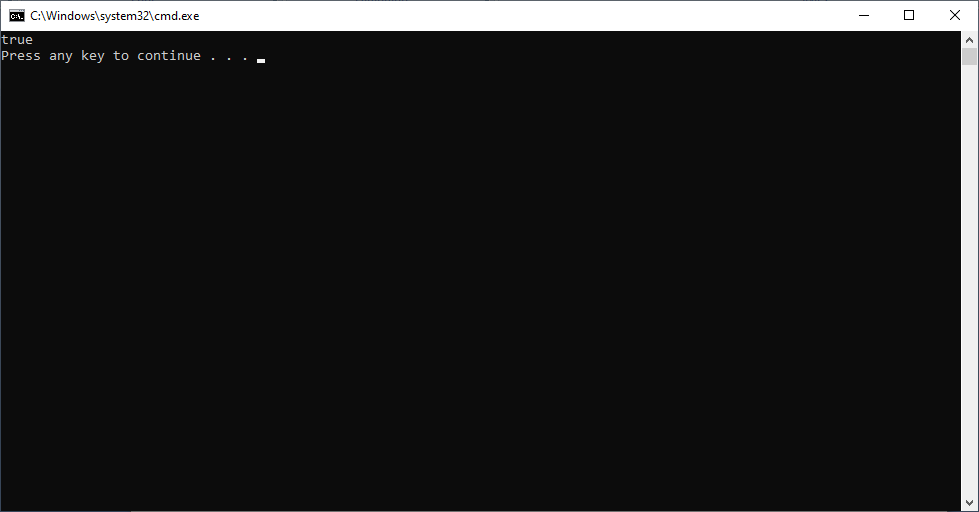
System.out.println(b1 instanceof Object);

//System.out.println(new ICICI() instanceof Bank); It Leads to CE: incompatability type.

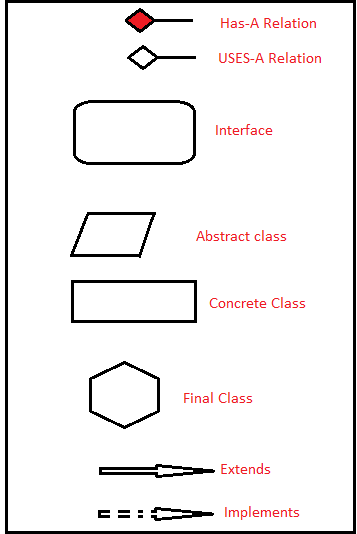
System.out.println(null instanceof Bank);

}

}



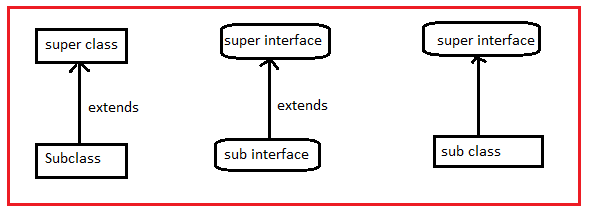
**2.9.Inheritance Types:-** 1.UML(Unified Modeling Language):- It is used for designing OOP based projects. It has different notations and diagrams to represent OOP members and their relations.



**Types of Inheritance:-** There are 5 types of inheritance

1. Single Level
2. Multi Level
3. Hierarchical
4. Hybrid
5. Multile Interfaces inheritance.

**1.Single Level Inheritance:-** If two classes or two interfaces or class and interface participating in an inheritance, it is called single level.



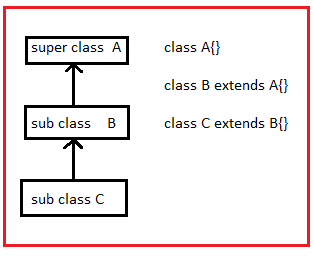
Extends means reuse.

Implements means forcing to implement the body of abstract methods..

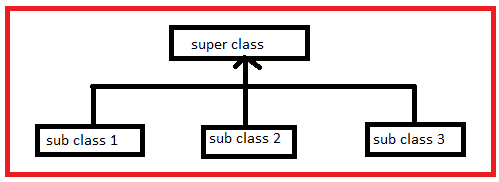
Implements is not allowed between two interfaces, because interface can’t implement method.

Note:- The default interface of java is single level inheritance.

**2.Multilevel Inheritance**:- If more than two classes participates in inheritance relation vertically we called it as multilevel inheritance.

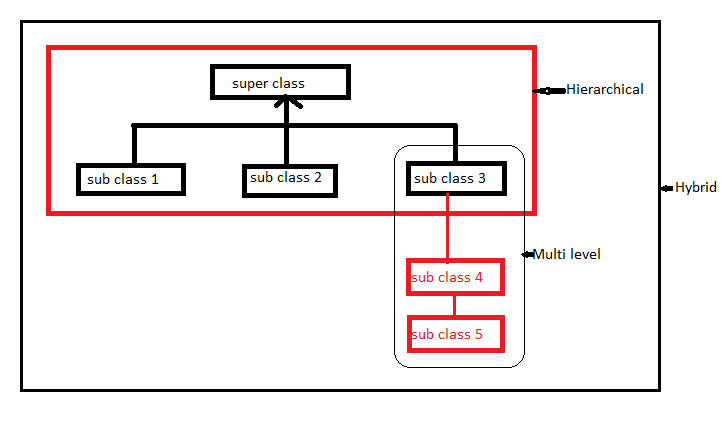


**3.Hierarchical inheritance:-** If we derive multiple subclasses from single super class we call it as hierarchical inheritance.

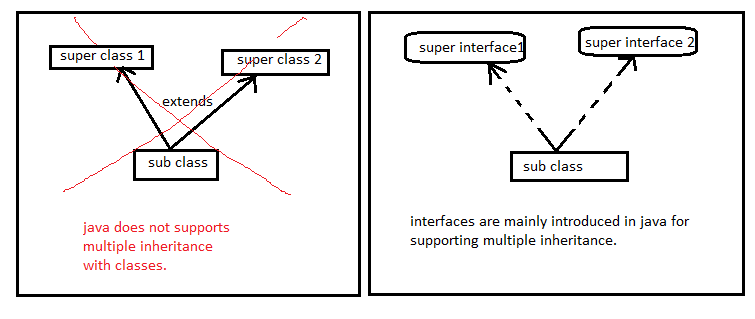


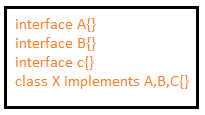
**4.Hybrid Inheritance:-**  Developing an inheritance by combining other types of inheritances is called hybrid inheritance.

In projects, defaultly we develop the hybrid inheritance.



**5.Multiple Inheritance:-** Deriving a subclass from multiple super classes is called multiple inheritance. Java does not support multiple inheritance with classes, but it supports with interfaces.

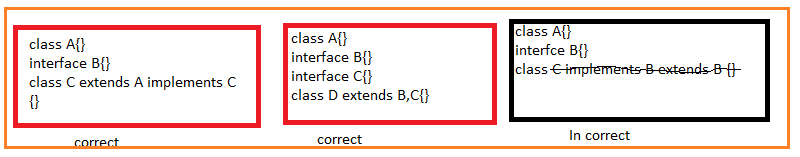




Q. can we derive a class from single class and multiple interfaces?

A. Yes.

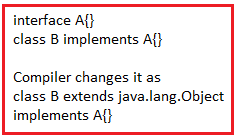
Example:



Q. when we derive a class from an interface, will it also be derived from any other class?

A. Yes, It will also be derived from java.lang.Object.

Example:



Q) we know interface is not sub class of object class,will below casting operation correct?

A. Object obj=new B();

A a=(A)obj;

Above two statements are correct,because every interface subclass is subclass of java.lang.Object so compiler will consider coming object is subclass of interface A. Hence it compiles above casting statement. If the coming object from obj variable is not sub class of interface A, JVM throws Class cast Exception.

**Case#1** Both interfaces has method with same prototype.

Interface f1{ void m1();}

Interface f2{void m2();}

Yes inheritance is possible.

Example :

Class C implements f1,f2{ public void m1(){}}

**Case #2** Both interfaces has method with same name but with different parameters.

Interface f1 {void m1(int a);}

Interface f2 { void m1(long b);}

Yes inheritance is possible.

Example:

Class A implements f1,f2

{

Void m1(int a){}

Void m2(long b){}

}

Case#3: both interfaces have method with same signature, but with different return type.

3.1 compatible referenced return types(super and subclasses)

Interface f1{ object m1();}

Interface f2 {String m1();}

Inheritance is possible and we should override the String return type method as per covariant returns. If we override object return type method it leads to CE.

Example:

Class A implements f1,f2

{

String m1(){ return “ “;}

}

3.2 Incompatible referenced return types(siblings)

Interface f1{ integer m1();}

Interface f2 { String 2();}

Inheritance is not possible.

Case#4: Both class and interface have method with same prototype.

Class A { public void m1(){}}

Interface f1 { void m1(){}}

Inheritance is possible and we no need to implement m1() method from interface as it is inheriting from A class with same prototype.

Case#5: Both class and interface has method with same signature but with different Access modifier i.e; class method is not public.

Class A{ void m1(){}}

Interface f1 {void m1();}

Inheritance is possible but we must override m1() method with “public” keyword. Then to execute m1() method from super class we must call it by using “super.m1();”.

**THIS**

**3.This:** -

**3.1.Variable shadowing:-**Creating a variable inside a method and inside a class with the same name is called variable shadowing.Here the local variable becomes shadow to class level variable. It means when we access variable , its value is read and modify from the local variable.

Example:-

import java.io.Console;

class sample

{

int x=10;

public void abc(){

int x=20;

System.out.println(x);

}

public static void main(String[] args)

{

sample s=new sample();

s.abc();

}

}

**Output: 20**

**Note:-**

**1.** if variable inside a non-static method and inside a class has same name then compiler will not prefix the this(keyword) to local variable. In varibale shadowing, if we want access non-static variable of class and differentiate it from local variable we must use **this** key word explictily.

Example:

import java.io.Console;

class sample

{

int x=10;

public void abc(){

int x=20;

System.out.println(x);

System.out.println(this.x);

}

public static void main(String[] args)

{

sample s=new sample();

s.abc();

}

}

Output:

20

10

**Note:-2**

import java.io.Console;

class sample

{

static int x=10;

public static void abc(){

int x=20;

System.out.println(x);

System.out.println(this.x);

}

public static void main(String[] args)

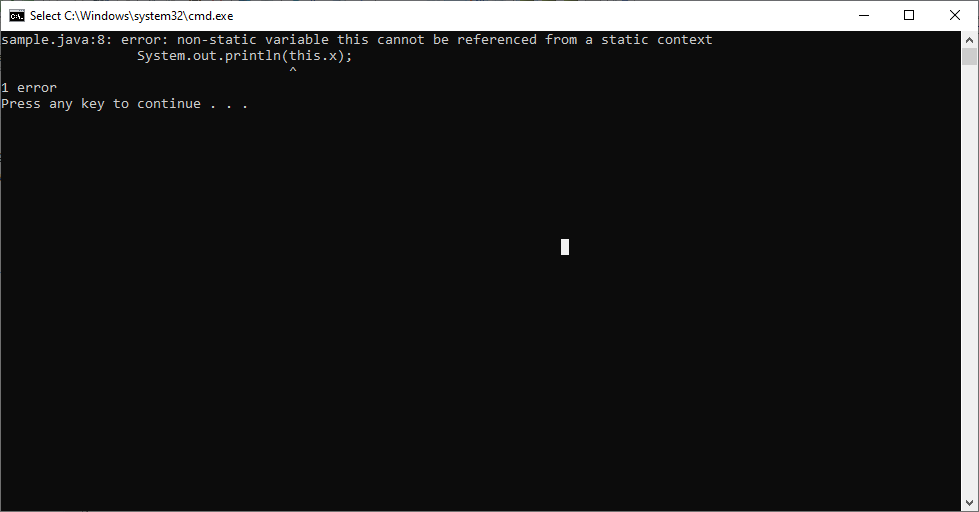
{

sample s=new sample();

sample.abc();

}

}



The compiler will not pass this(keyword) to static method as argument by default. There It is not available inside the static method. But this(keyword) has been used in variable shadowing concept. That’s why the problem has occurred.

Q.) How to differentiate the local variable of method and static variable of class inside a static method?

A) The class name is used to differentiate the static variable of class from local varible of method inside a static metod.

import java.io.Console;

class sample

{

static int x=10;

public static void abc(){

int x=20;

System.out.println(x);

//System.out.println(this.x);// It leads to CE.

System.out.println(sample.x);

}

public static void main(String[] args)

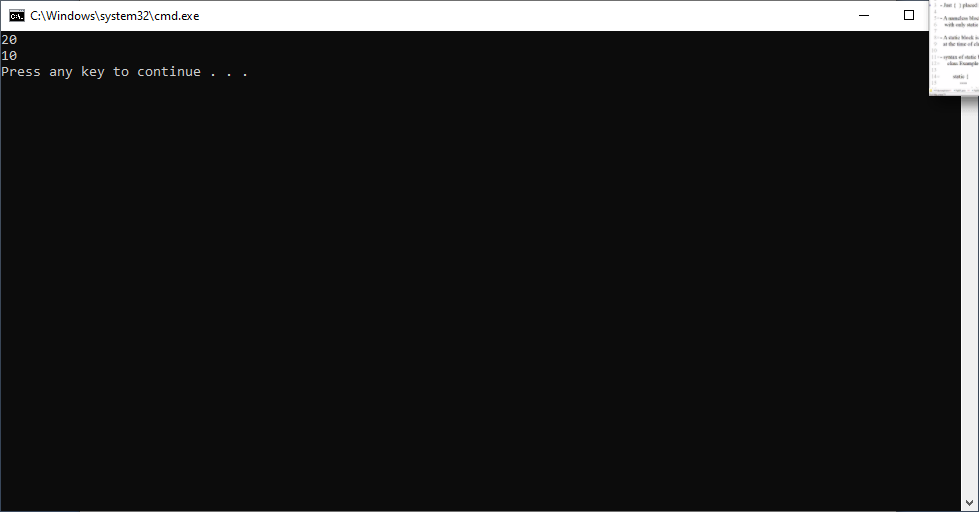
{

sample s=new sample();

sample.abc();

}

}



**‘this’ is keyword and It stores the only current class object reference.**

So if u store another class object reference in ‘this’ in current class , we will get compile error (incompatability error).(i.e) we can ‘t change the value of ‘this’ in current class.

Example:

import java.io.Console;

class Example

{

}

class sample

{

public static void main(String[] args)

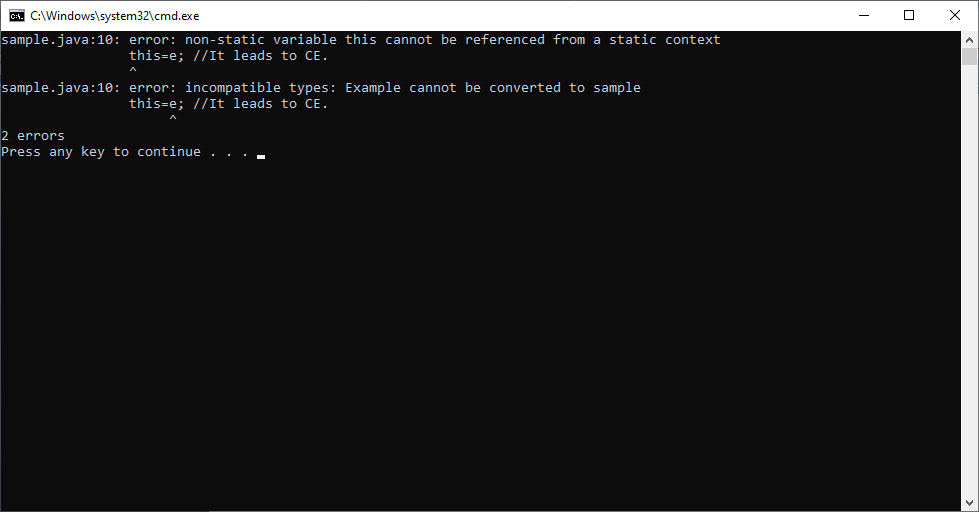
{

Example e= new Example();

this=e; //It leads to CE.

}

}

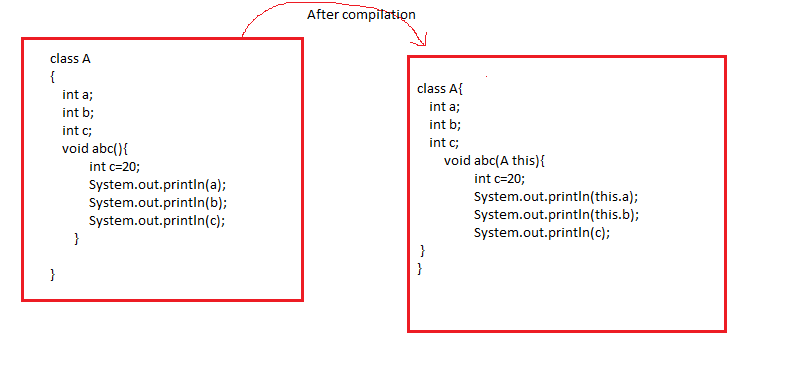


Therefore we can say **that ‘this’ is final parameter referenced variable of current class.**

2.The compiler will place ‘this’ variable declaration as first parameter in every **constructor and every non-static metod**.

3. The compiler will prefix ‘this’ to every class vairable inside a constructor and method.

Note : if there is any local variable or parameter defined in the current method with same non-static variable name compiler does not place this keyword. In this case developer must place ‘this’ word explicitly.



4. At Execution time, the JVM

4.1. creates a memory to ‘this’ variable.

4.2. stores the current object reference in ‘ this’ variable.

5. The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

Example:

import java.io.Console;

class sample

{

int x=10;

int y=20;

public void abc(){

int x=50;

System.out.println("Local Variable:"+ x);

System.out.println("class Variable:"+ this.x);

}

public static void main(String[] args)

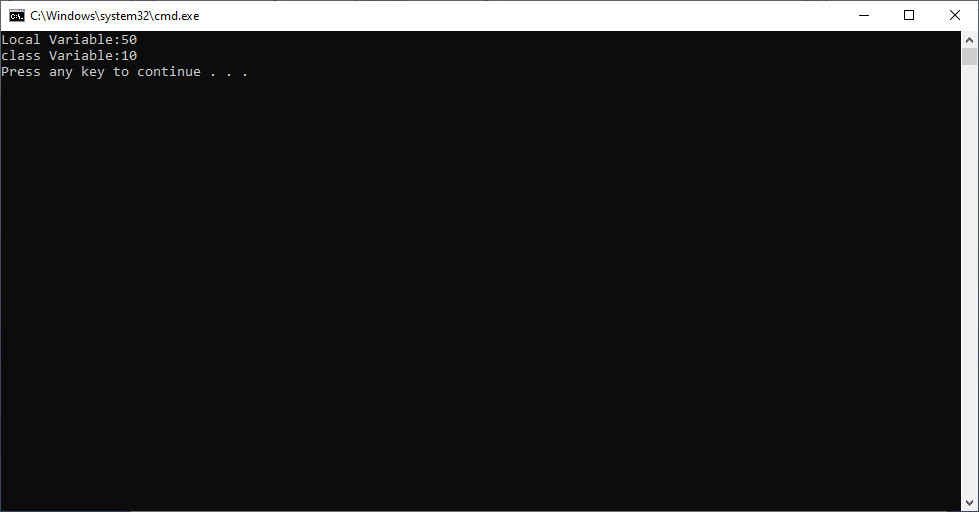
{

sample e= new sample();

e.abc();

}

}



Note: - The ‘this’ variable is not used in static methods . Because it is non-static variable. It u use it leads CE.

Example:

import java.io.Console;

class sample

{

int x=10;

int y=20;

public static void abc(){

int x=50;

System.out.println("Local Variable:"+ x);

System.out.println("class Variable:"+ this.x);

}

public static void main(String[] args)

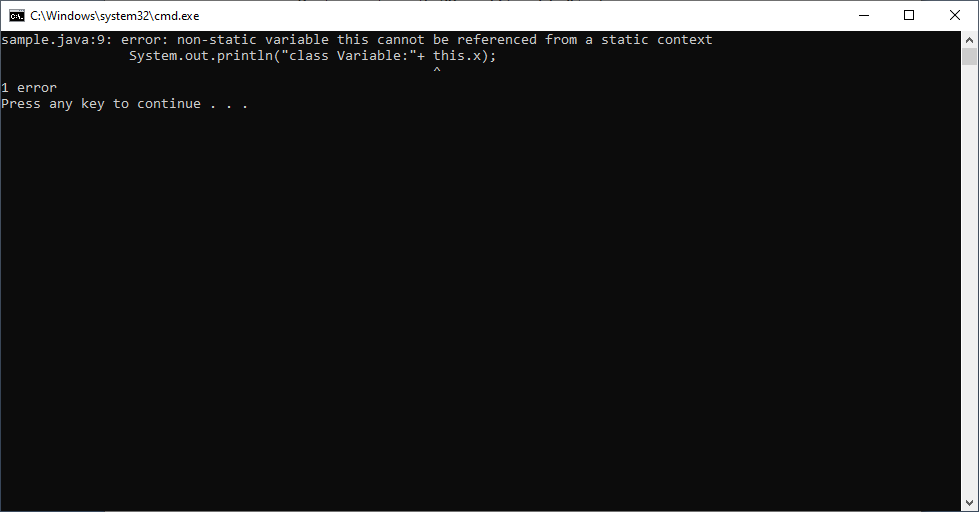
{

sample e= new sample();

e.abc();

}

}



**SUPER**

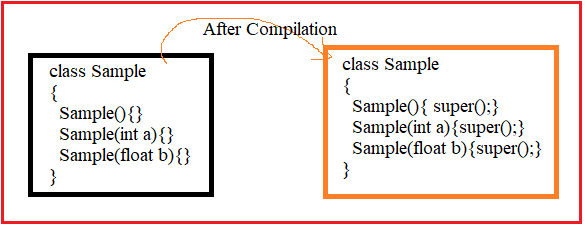
**4.Def1:-** The super is keyword. Like “this”, It is not variable. It does not has “physical Declaration”.

The super keyword is used to access super class variables, methods and constructor from sub class members. The super keyword has two forms.

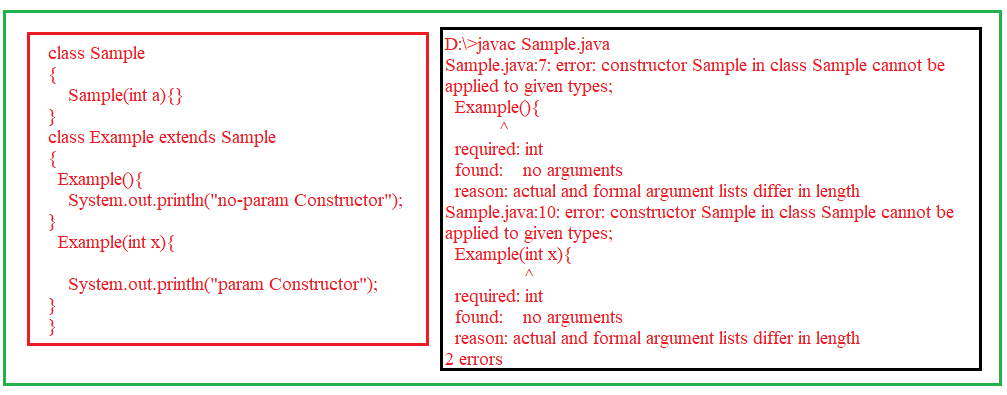
1. Super()
2. Super.

**4.1.super():-**  The super() is used to invoke super class constructor from only all sub class constructors to provide a memory for super class fields in subclass object and further initialize them with initialization logic given by super class developer.

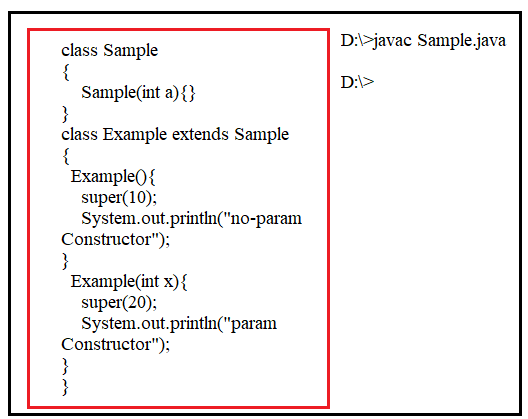
1.1) By default, compiler will place super() call with out arguments as first statement in all constructors of sub class. So by default zero-param constructor will be executed from super class.



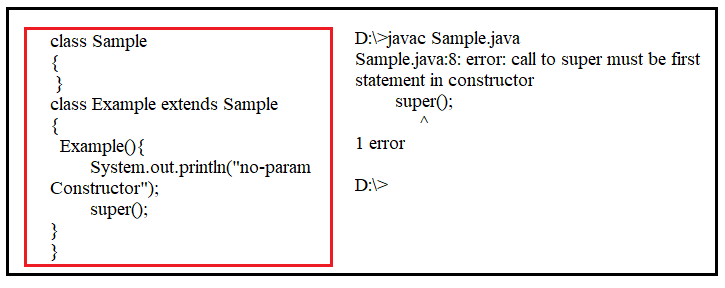
1.2) If super class has only param constructor, we must call that super class param constrctor explicitly in sub class all constructors by placing super(arg); here argument type should be super class constructor parameter matching type. Otherwise compiler throw CE.



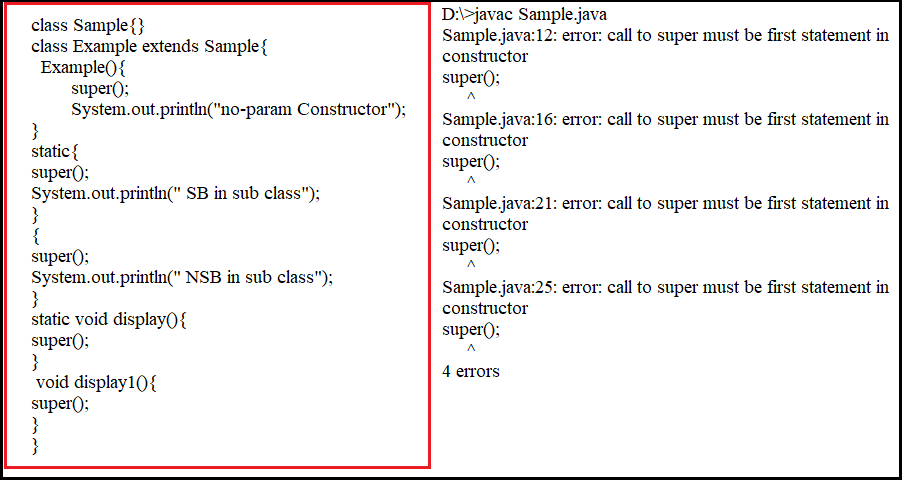
Correct program



1.3) If programmer writes super call(with param/without param) statement in subclass constructor, It should be in first line otherwise it leads to CE.



1.4. The super() call statement must not be written in static block,static method, non-static block and non-static method.



1.5. The **private,protected or public** is allowed to constructors. The default accessibility modifier of constructor is “default”. If class has private access constructor it is never visible , we can not create sub class.

1.6.**we can initialize all or some variables of super class while creating the sub class object.**

class Example

{

int a;

static int b;

Example(int x, int y)

{

a=x;

b=y;

System.out.println("Super class variables initilized from sub class");

}

}

class Sample extends Example

{

Sample()

{

super(20,10);

}

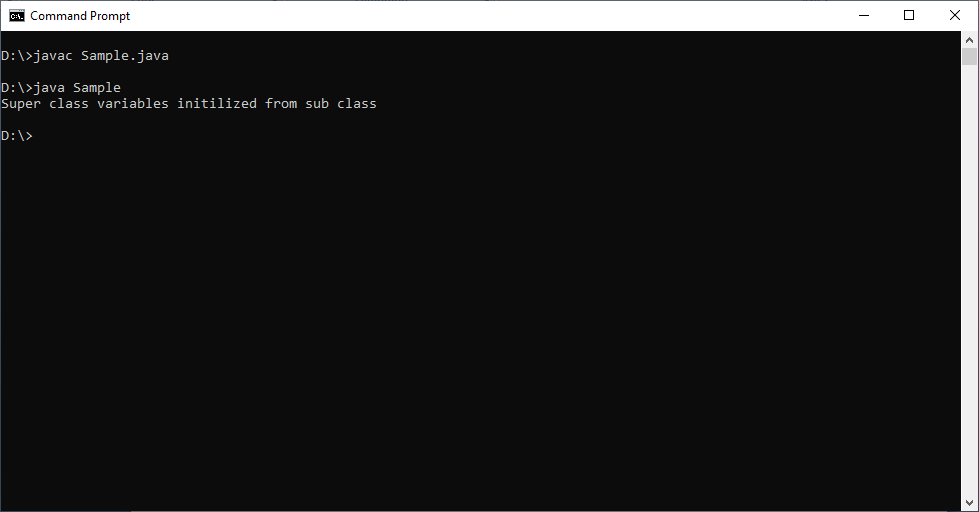
public static void main(String args[])

{

Sample s1=new Sample();

}

}



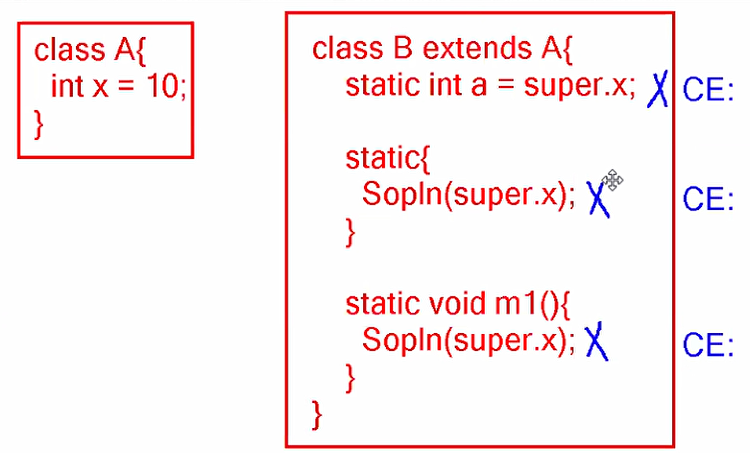
**4.2.super.:- “super.”**  Is used to access super class variables and methods from sub class members or constructors.

Syntax:- super.member;

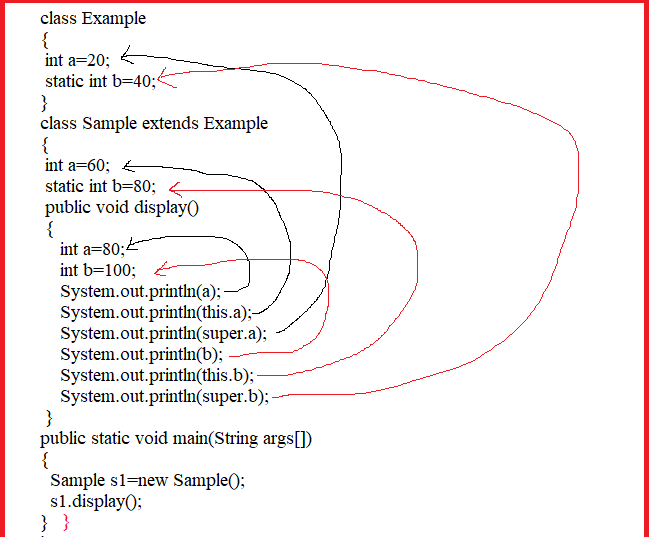
The member can be either an static/instance variable or a method.

Note:- we are not allowed to user super keyword in static members.

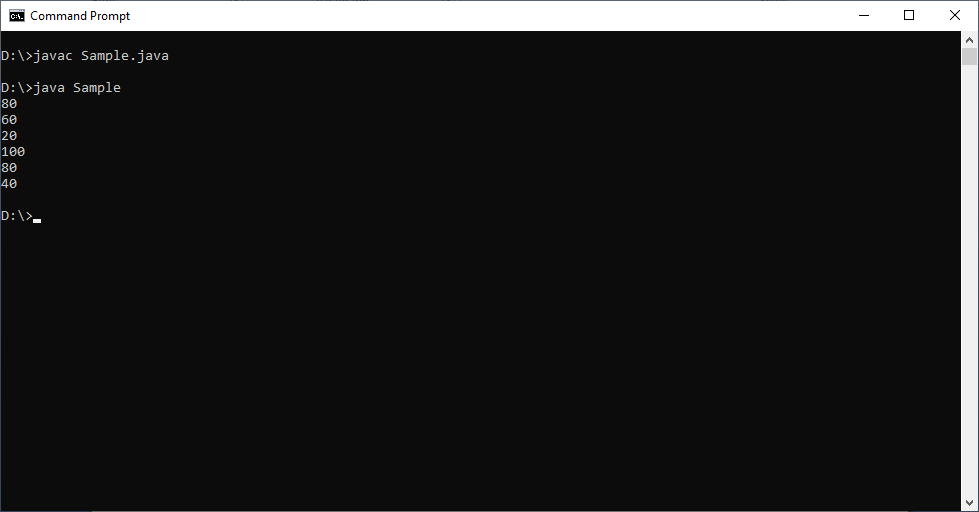
Example:1



Example:2



Output:



Note:- If child class member name is not equal to super class member name then no need to use “super.” Then super class member can directly be accessed in sub class.

If child class member name is equal to super class member name then we should use “super .” to differentiate super class member from child class member.

Example:

class Example

{

int a=20;

int b=80;

}

class Sample extends Example

{

int a=60;

public void display()

{

System.out.println(a);

System.out.println(b);

System.out.println(super.a);

}

public static void main(String args[])

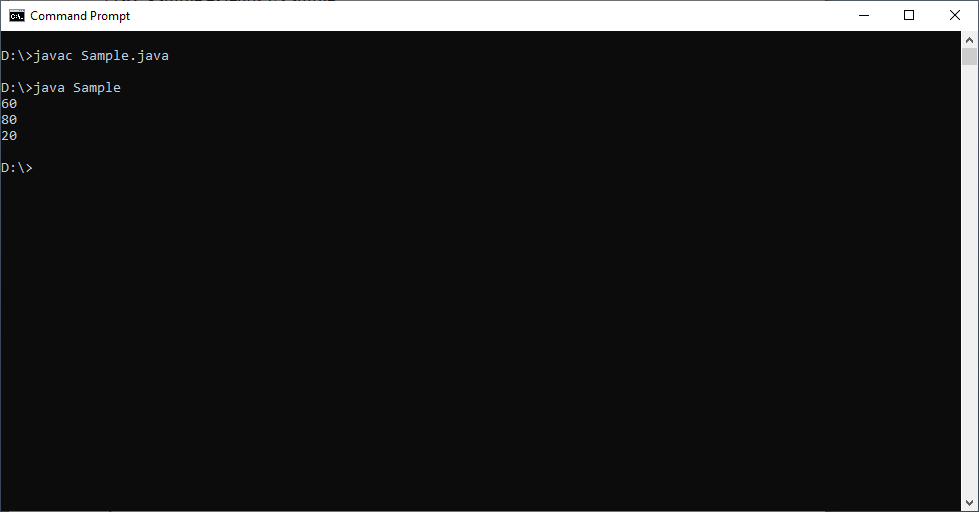
{

Sample s1=new Sample();

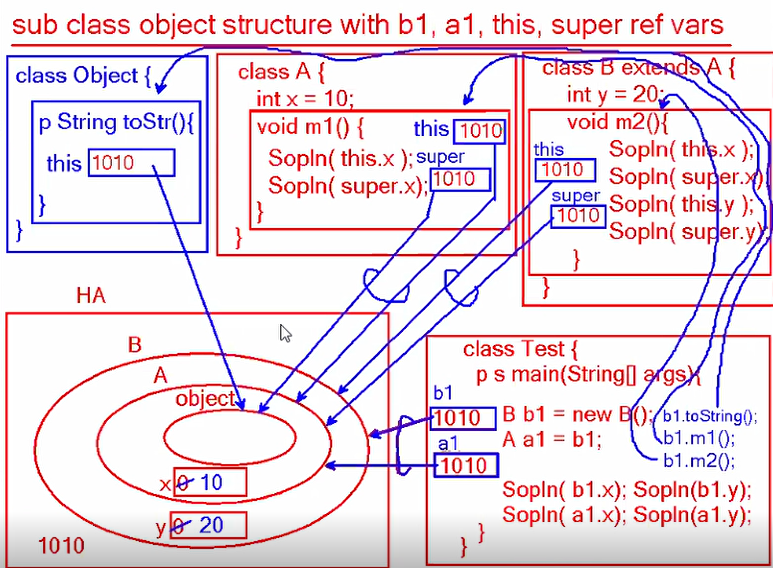
s1.display();

}

}



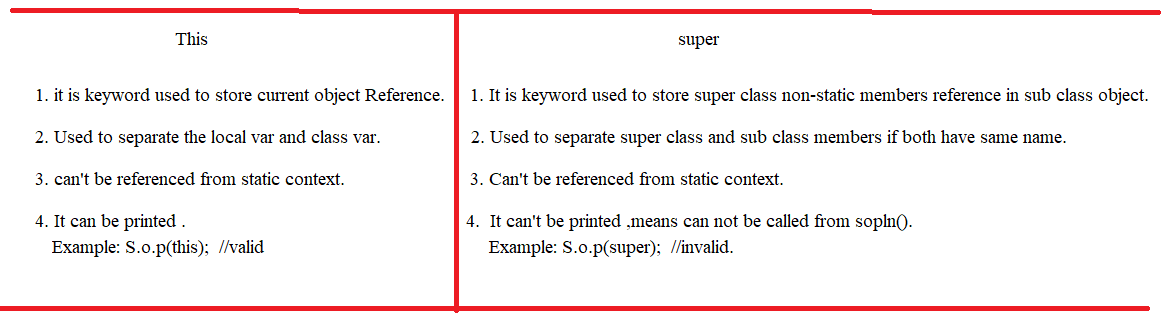
Example:



**4.3. Variable Hiding:-**  Creating a variable in subclass with super class variable name is called variable hiding. Here subclass variable is hiding the super class variable. This variable hiding is solved by “super.**”.**

**Difference Between this and super**

**5.Diff between this and super**.



Q) Why we can not print super keyword like this keyword?

A) Because super class non-static members memory does not have its own hash code and also its memory is part of subclass object.

**ABSTRACTION**

Abstraction is object modeling principle. Abstraction will come into picture at project design phase where as OOP principles encapsulation,inheritance,polymorphism will come into picture in project development phase.

**Def1:** In General abstraction means hiding an object operation implementation details from the user providing only necessary details to access the object operation is called abstraction.

Example 1: In ATM machine with draw operation details are abstracted- It means we do not know how card is reading, PIN is validating , and cash is counting and pushing out from ATM machine.

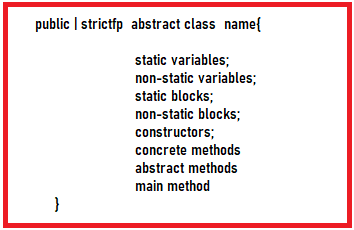
Example 2: out body digestion system is abstracted – means we do not know how food is digested, converting into energy and blood.

**Def2:** Abstraction is contract document between object user and object implementer/object developer. This document will contains list of operations declarations using abstract methods.

This abstraction is developed by either abstract and interface keywords.

**1.Abstract class:-** A class that is declared with abstract modifier is called abstract class. The abstract class is partially implemented class Because it contains both abstract methods and concrete methods.

Syntax:



Example:-1

public abstract class Sample {

staticint a;

int b;

int x;

static{

a=20;

System.out.println("Static block:"+ a);

}

{

b=50;

System.out.println("Non-static block:"+b);

}

public static void display()

{

System.out.println("Static method:"+ a);

}

public void display1(){

System.out.println("non-static method:"+ b);

}

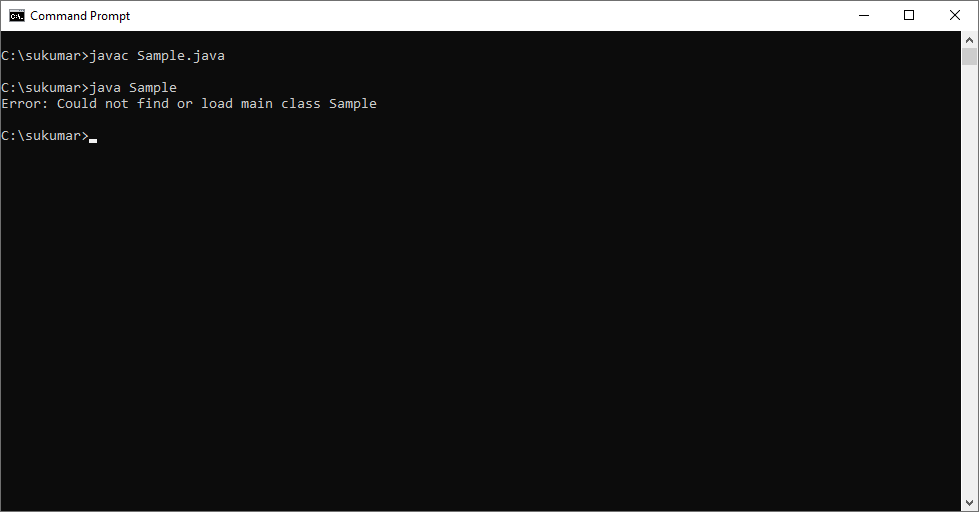
public abstract void display2();

public Sample(){ x=50;}

public static void main(String args[]) {

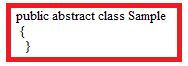
}

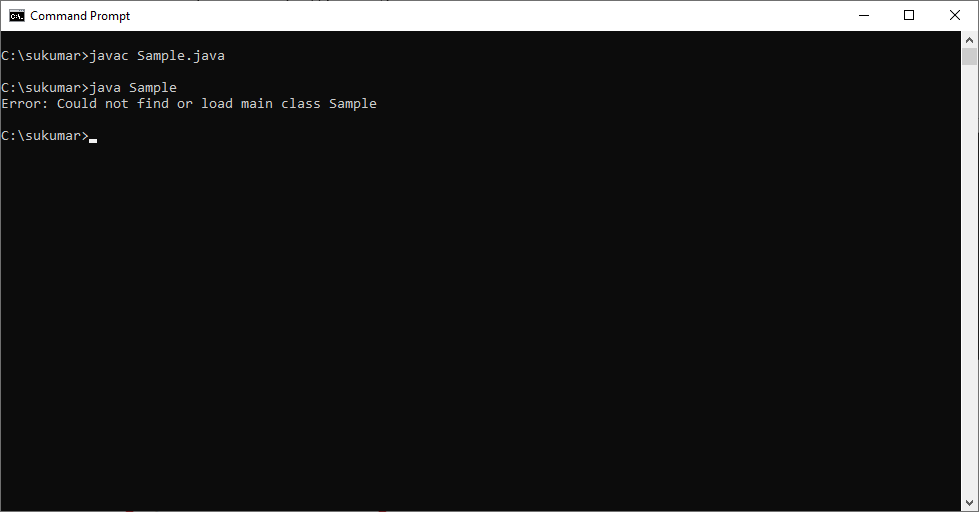
}



The abstract class can be compiled. Even abstract class has main(),That abstract class can’t be loaded.

🡪We can create empty abstract class. It can also be compiled.





🡪We can’t instantiate the Abstract class.

abstract class Example{

}

public class Sample {

public static void main(String args[])

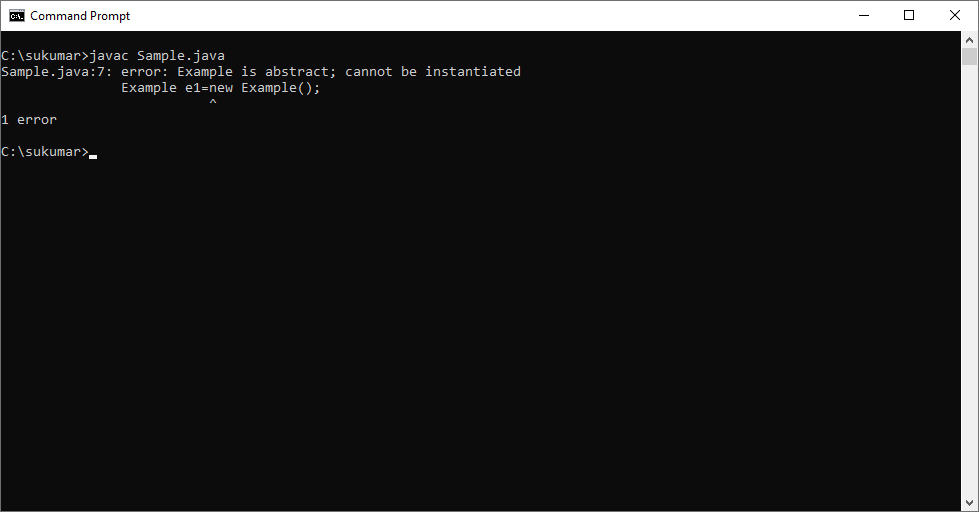
{

Example e1=new Example(); //It leads to CE: abstract class can not be initialized.

System.out.println("Sukumar");

}

}



Note:- The non-static members get memory when its concrete subclass object is created.

When we find an object with multiple sub types, and among its operations some of the operations we want to implement and some of other operations we want implement by sub type object programmer then we must create this object using abstract class.

We will implement the operations whose logic is common to all sub types in abstract class. We won’t implement the operations whose logic is changing from on sub type to another sub type in abstract class. These are implemented by sub type object programmer in sub classes.

Real Time Example:-

Bus object.

It has sub type objects RedBus,Express.

It has operations engine(),wheels(),breaks(), etc…

Among these operations wheels() and breaks() methods can be implemented common to all buses, but engine() method implementation is changed to all busses. Because every bus will have different capacity engine.

Then we must create the object Bus using abstract class by declaring

Engine() method as abstract method.

Breaks() and wheels() method as concrete method.

abstract class Bus{

public void wheels() {

System.out.println("Bus Runs on 6 wheels");

}

public void breaks() {

System.out.println("Breaks applied , bus stopped");

}

public abstract void engine();

}

classRedBus extends Bus{

public void engine() {

System.out.println("40KMPH capacity");

}

}

class Express extends Bus{

public void engine() {

System.out.println("60KMPH capacity");

}

}

public class Sample {

public static void main(String[] args) {

RedBus r1=new RedBus();

Express e1=new Express();

System.out.println("============Red Bus=============");

r1.wheels();

r1.breaks();

r1.engine();

System.out.println("===========Express===============");

e1.wheels();

e1.breaks();

e1.engine();

}

}

Output:-

---------

============Red Bus=============

Bus Runs on 6 wheels

Breaks applied , bus stopped

40KMPH capacity

===========Express===============

Bus Runs on 6 wheels

Breaks applied , bus stopped

60KMPH capacity

**2.Abstract Methods:-** The method that has only prototype and does not have body is called abstract method. Abstract method should contain modifier abstract and should ends with semi colon.

Example:

1. public abstract void m1();
2. public abstract int m1(int a ,int b);

Basically, in projects abstract method protype are defined by superclass developer and they are implemented (methodbody and logic) by sub class developer.

**ENCAPSULATION**

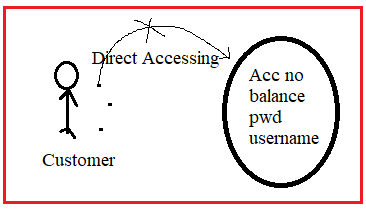
Closing data with some material shield is called encapsulation. We can’t get data with out breaking shield.

Example:

Capsule.



We should not allow customer to direct access object for either storing value in variable and reading value from variable.



By providing direct accessing , there are problems

* Any one can access object data(Loose Data security)
* Wrong values can be stored in variable.

**Definition:1** The process of wrapping up of variables & methods as one uinit by using class & operating those variables only via methods either for reading or storing values is called encapsulation.

By developing encapsulation , we hide data of one object from direct access from other objects and providing access via publicly available getter ,setter and normal methods with proper validation.

**Definition2:**  hiding data by declaring fields as rivate & provide their accessing via public setter ,getter and normal metohds is called Encapsulation.

Advantages:-

1. We can stop storing wrong values.
2. We can stop unauthorized people accessing object data.
3. We can modify data with out effecting other parts in project.

**Develop Encapsulation for Bank Object for balance Variable:**

class Bank{

private double balance; //data hiding.

Public void setBalance(double balance) throws illegalArgumentException

{

If(balance==0)

Throw new illegalArgumentException(“Don’t pass zero”);

If (balance<0 )

Throw new illegalArgumentException(“Don’t pass –ve amount”);

This.balance=balance;

}

Public double getbalance()

{ return balance;}

}

Class Bank

{

Psv main(string arg[])

{

Scanner scn= new Scanner(system.in);

s.o.p(“enter balance”);

double balance= scn.nextDouble();

BankAccount acc= new BankAccount();

Acc1.setBalance(balance);

S.o.p(“CurrentBalance:”+acc1.getBalance());

}

}

Balance is hided by declaring as private. This is called data-hiding. This data (balance) accessibility is only provided through setter & getter methods.

Hiding & providing accessiblity of data through method is called encapsulation.

Example1:-

Class bank

{

Double balance;

}

This class does not have encapsulation because balance can be accessed directly.

Example2:-

Clas bank

{

Private double balance;

}

This calss have only data hiding but not encapsulation. Because balance has been declared as private.

Example3:

Class Bank{

Private double balance;

Public void setBal(int x)

{

Balance=x;

}

Public double getBal()

{

Return balance;

}

}

This class have both data hiding and encapsulation.

Example4:

Class bank

{

Double balance;

Public void setBal(int x)

{

Balance=x;

}

Public double getBal()

{

Return balance;

}

}

This class does not have both data hiding & encapsulation.

**Types of Encapsulation:**  There are two types of Encapsulations.

1. Weak Encapsulation:- If class contain any property access level other than private & method access level other than public that class is called weak encapsulatd class.

Example:

Class Bank{

Int bal;

Private string name;

}

1. Strong Encapsulation:- The class that contains all property access levels as private & every method as public . Then that class is called strong encapsulation class.

Example:

Class Bank{

Private int bal;

Private String name;

}

**Example:-** The following program explains developing a class by following encapsulation principle-

class Clerk

{

public static void main(String[] args)

{

BankAccount acc1=new BankAccount();

acc1.setBalance(5000);

System.out.println(acc1.getBalance());

acc1.setBalance(-5000);

System.out.println(acc1.getBalance());

}

}

//Encapsulated class

class BankAccount

{

private double balance;

public void setBalance(double balance)

{

if(balance<=0)

{

System.out.println("Do not enter -ve amount");

}

else

{

this.balance=balance;

}

}

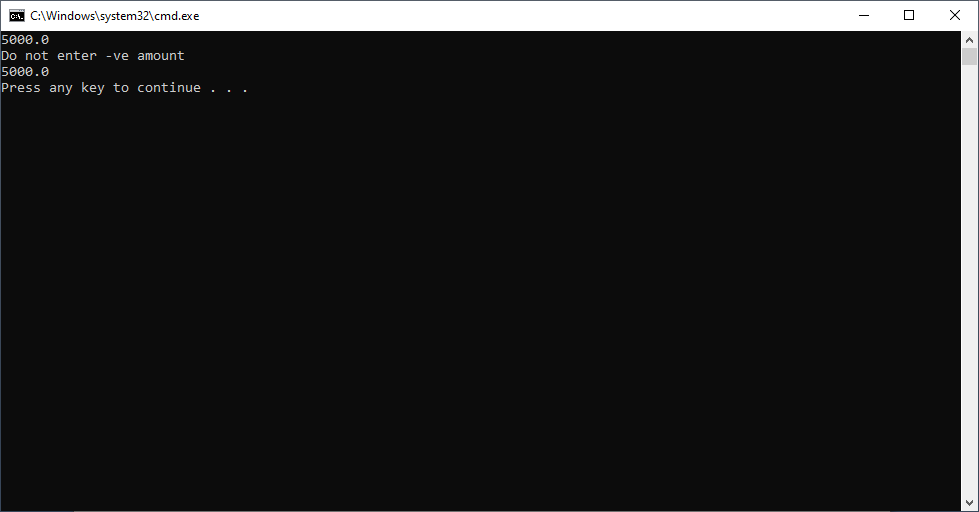
public double getBalance(){

return balance;

}

}

Output:



**POLYMORPHISM**

**1.polymorphism**:-

Poly means --- many/multiple

Morphism means – forms

Polymorphism means “many forms”.

**1.1.General Definition:-**In general as per real-world, polymorphism means object exhibiting multple behaviours for same operation based on situations.

Example:-

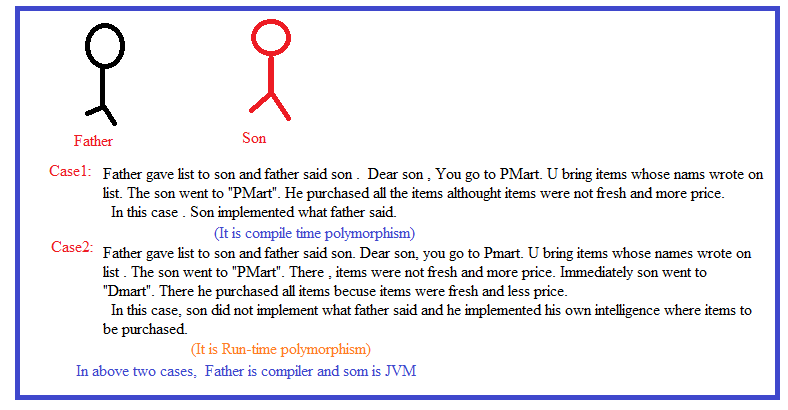
1. person exhibit sad feeling for bad news and same person exhibit happy feeling for good news.
2. Student exhibit good behaviour infront of teacher and same student exhibit other behaviour in the absence of faculty.

**1.2.Technical Defintion:-** providing multiple implementations to same operation based on subtype or based on different type of input values is called polymorphism.

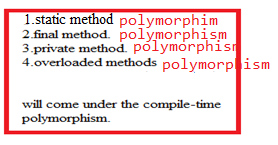
**1.3.Types of polymorphism:-** Java supports two types of polymorphisms

a.Compile time polymorphism/static binding/Early binding.

b.Run-time polymorphism/Dynamic Binding/Late Binding.



**1.3.1)Compile time polymorphism**:- At the time of compilation, the compiler determines a method definition to which method invoking statement will be linked . The same method definition is executed by JVM . This is called compile time polymorphism.



**At the time of compilation, the compiler makes linking based on referenced variable type but not on reference or reference type stored in referenced variable.**

**1.3.2 Run-time polymorphism:-** At the time of compilation, compiler link the method definition with method invoking statement. But JVM will not executed that method. Instead , the same method is executed by JVM from the sub class based on the object stored in referenced variable is called Run-time polymorphism.

**(or)**

The process of executing an invoked method from different sub classes based on object stored in the referenced variable is called run-time polymorphism.



So run time polymorphism is only implemented through

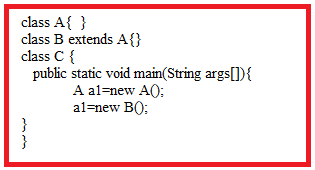
1.Upcasting.

2.Method overriding.

If method is called without upcasting or if method is not overridden then that mehtods polymorphism is always compile time polymorphism.

**2.polymorphism with variables:-** A variable is called polymorphic if it refers to different values under different conditions. Object variables(instance variables) represents the behavior of polymorphic variables in java. It is because object variables of class can refer to objects of its class as well as objects of its sub classes.

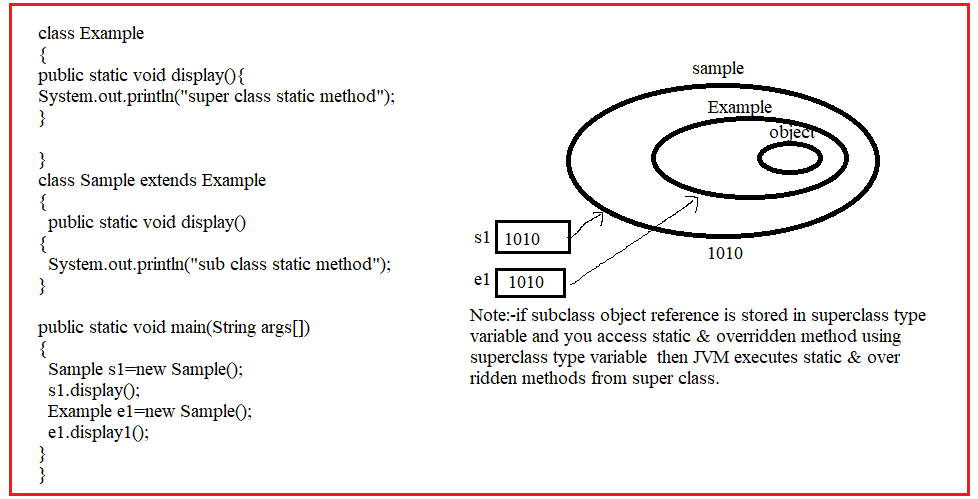
Example:

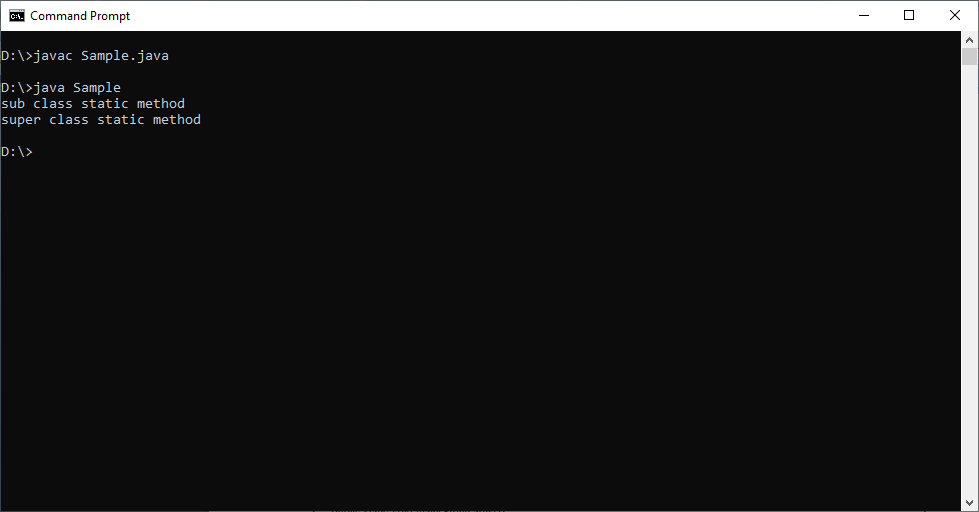


**3.polymorphism with static methods/Method hiding:-** Redefining super class static method in subclass with same prototype for providing new implementation logic as per sub class needs is called “method hiding”.

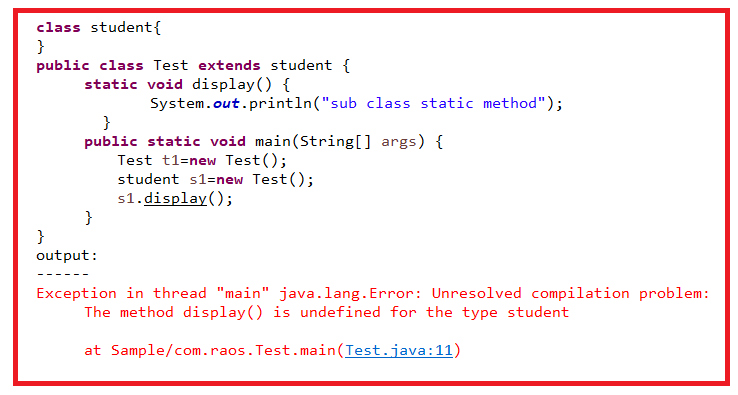
We can declare static methods with same signature in the subclass, but it is not considered overriding. Therefore static method overriding is not possible.

Example:-





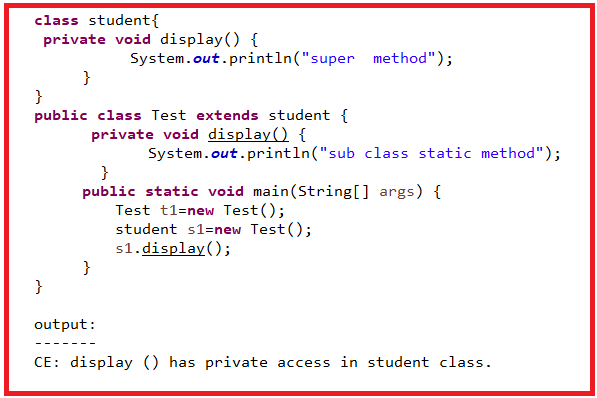
Example:2



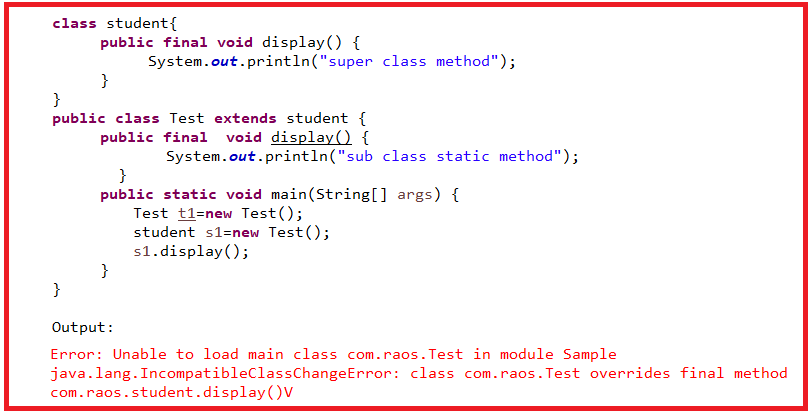
**4.polymorphism with private variable:-** private methods in java are not visible to any other class which limits their scope to the class in which they are declared.

Therefore we can’t override private method in java.

Example:1



**5.polymorphism with final methods:-** The method which is declared using final keyword is called Final method. The final method can’t be overridden in sub class. It is inherited in sub class. Therefore we can’t override final method in java.



6. **Method Overloading:-**  Defining multiple methods with same name but with different parameters type | list| order is called “Method Overloading”. We will overload method for executing same logic with different type of values.

Example:1

Void display()

Void display(int a)

Where parameter data types and parameters list of two overloaded methods is different.

Example:2

Void display(int a, int b);

Int display(int a, int b , int c);

Where parameter list of two overloaded methods is different.

Example:3

Void display(int a,float b);

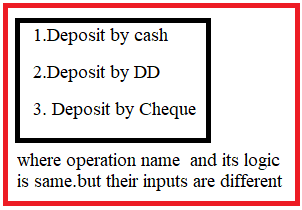
Int display(float b, int a);

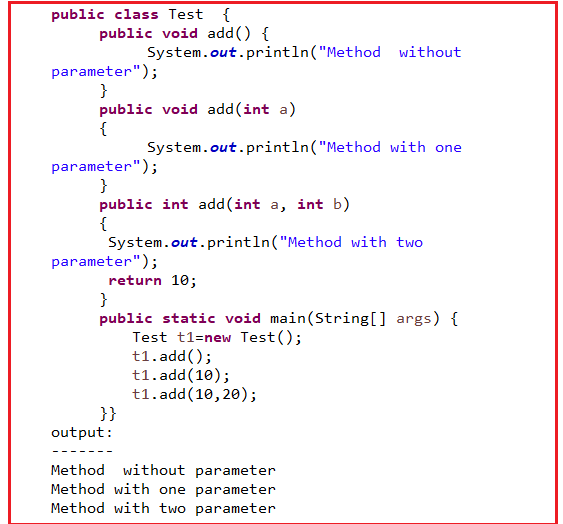
Where parameter order of two overloaded methods is different.

Note:- we can overload a method in same class or in its subclass because overloading method has different signature .

The overloaded methods are differentiated based on parameter types, parameters order or parameter list.

RealTime Example:

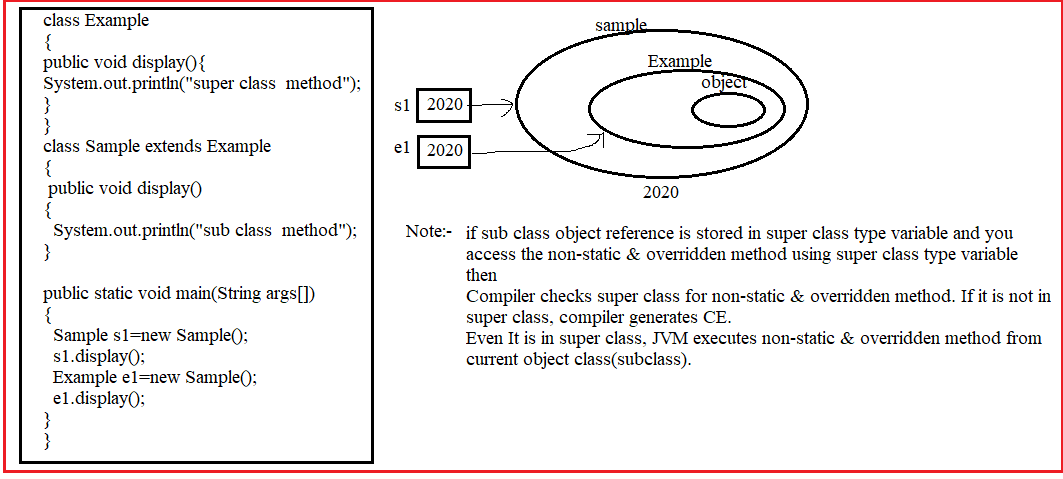




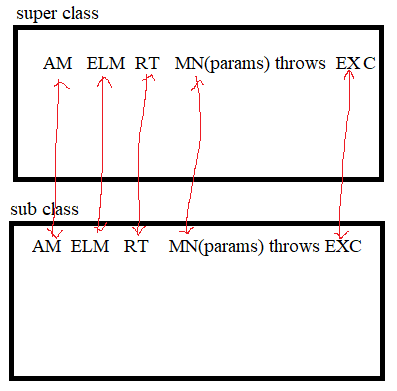
7. **Method overriding:-** Redefining super class non-static method in subclass with same prototype for providing new implementation logic as per sub class needs is called “method overriding”.

7.1.Polymorphsim with non-static methods:

Example:



**Rules for method hiding and method overriding**:-



Rule1:- The subclass overridden method /hiding method should has same exception of super class method or sub type exception of super class method exception.This rule should came from java5 onwards.

Before java5, The sub type exeception of super class method exception is not allowed.

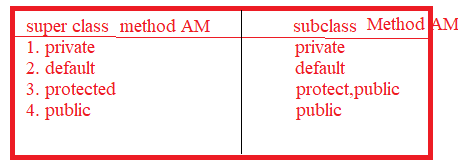
Rule2:- The method name should be same.

Rule3:- The parameters list,parameters type and parameters type should also be same.

Rule4:- The sub class over ridden method should has same RT of super class or sub type of super class method RT. This rule came from java5 onwards.

Before java5, The sub type of super class method RT is not allowed.

Rule5:- Access modifiers.



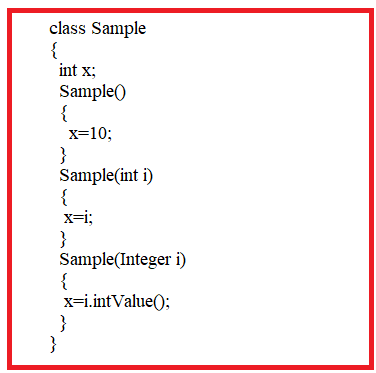
**CONSTRUCTOR OVERLOADING AND CONSTRUCTOR CHANINING**

**1.Constructor Overloading:-** Defining multiple constructors in a class with different parameters type | list | order is called constructor overloading.

In Below two cases we should overload constructor.

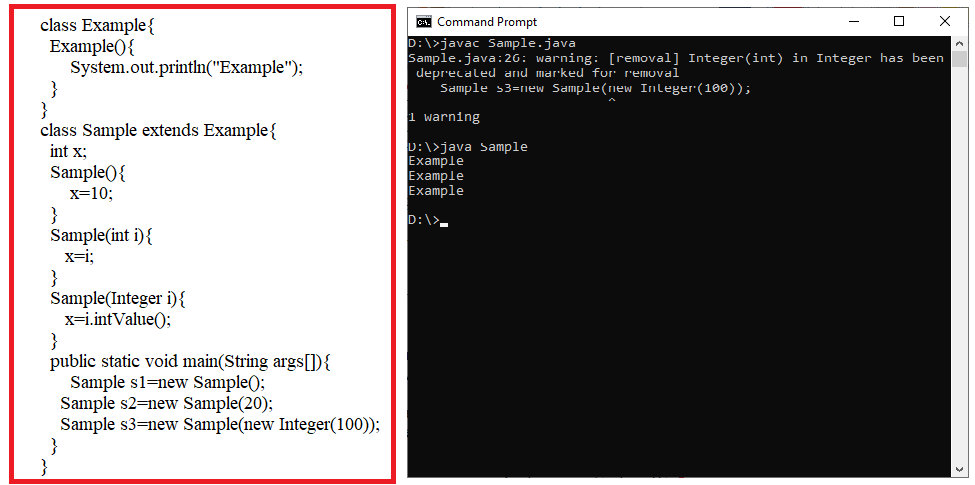
1. To define different object initialization logic.
2. To execute same initialization logic by different types of input values.

Example:



Note:- Java compiler place super() call as first statement in all constructors(parameterized and non-parameterized constructors)

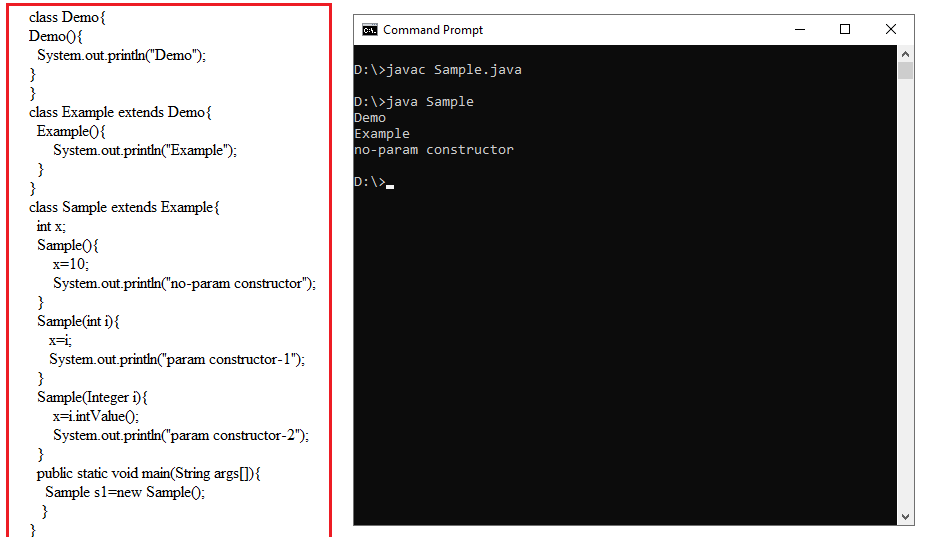
Example:-



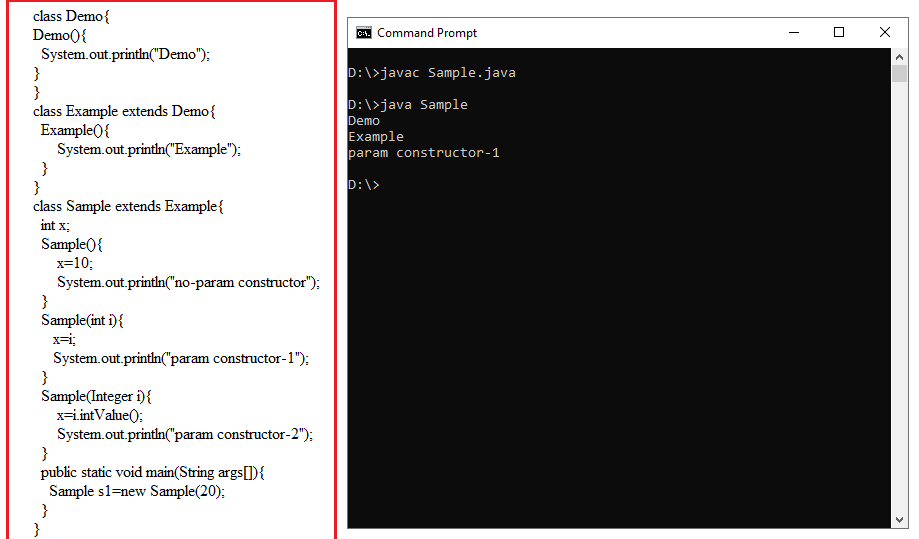
**2.Constructor chaining:-** calling one constructor from other constructor by using super() and this() with out creating other new object is called constructor chaining.

**2.1.**The subclass constructors are chained with super class constructors by using super().

Example:-1



Example:-2



2.2.Subclass overloaded constructors are chained by using this().

**Rules related to this():**

* This() call must be written as first statement in constructor.
* This() and super() calls can not be placed in same constructor,because both must be placed as first statement in constructor.
* Multiple super() and this() can not be written in same constructor, it leads CE.
* In an object creation we are not allowed to call same constructor more than once,it Leads to : **“recursive constructor invocation”.**

Example:

Class sample{

Sample()

{

This();

---

}

}

* This() should not be placed in all constructors. In at least one constructor , we should not write this() otherwise it leads to CE**: “recursive constructor invocation”.**

Example:

Class Sample{

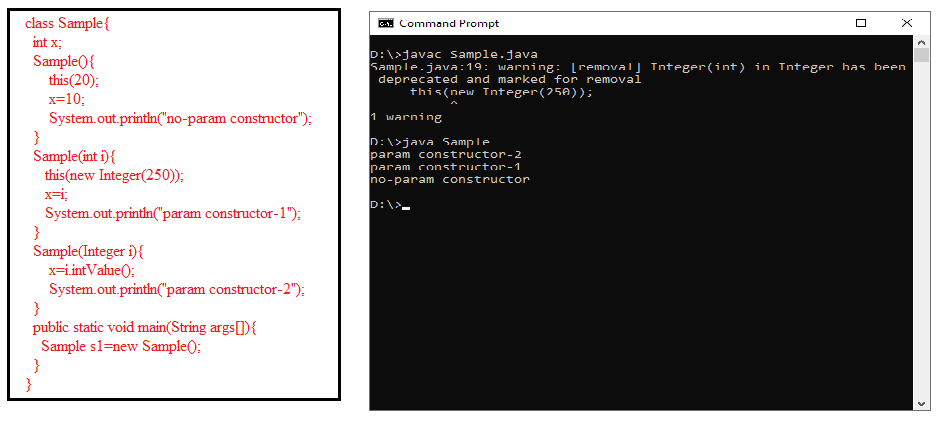
Sample(){ this(20);}

Sample(int a){ this(30.3f);}

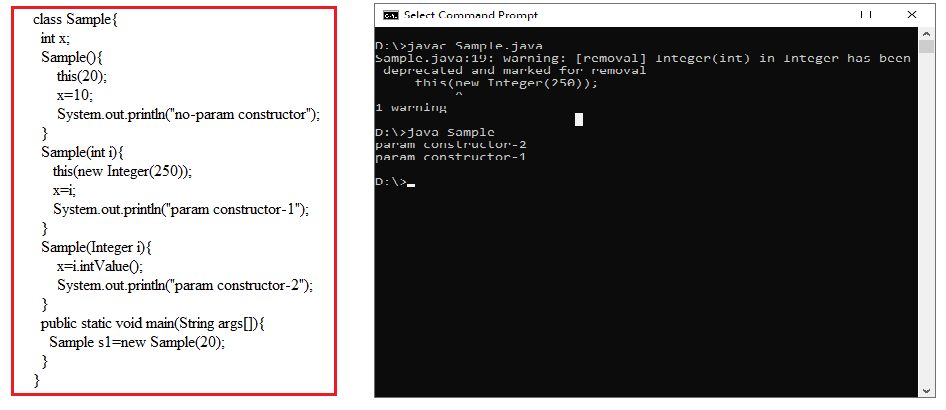
Sample(float f){ this();}

}

Example:1



Example:2



Note:-1 In constructor chaining,

The programmer does not write parameterized/non-parametrised super() call statement in all overloaded constructor and all overloaded constructors have this() except last overloaded constructor. **In an object creation, only non-parametrized super() call is invoked only once for entire constructor chain**.

Note:-2 In constructor chaining,

The programmer write parameterized super()call statement in any one overloaded constructor. In an object creation , only paramerized super() call is invoked only once for entire constructor chain.



**Final**

The keyword final is a modifier keyword. It is used for creating constant member.

The final keyword can be applied to

1. Static variable,non-static variable, local variable and parameter
2. Method
3. Class.
4. Inner class only.

The final keyword can not be applied to

a. module

b. package

c. import

d. interface

e. abstract class

f. block

g. constructor

f.enum because enum is already final.

**1.Final Variable:-** The final keyword can be applied to static,non-static,local and parameter variable. The final variable value can’t be modified.

Syntax1:- final datatype var\_name=value;

Syntax2:- final datatype var\_name;

Var\_name=value;

Example1:-

class Clerk extends BankAccount

{

public static void main(String arg[])

{

final int x=10;

static final int y;

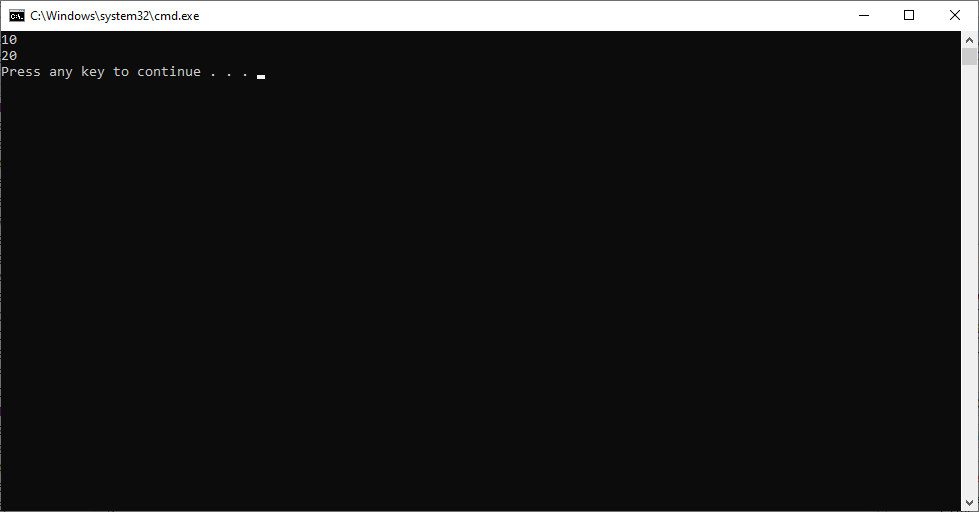
y=20;

System.out.println(x);

System.out.println(y);

}

}



Example2:

class Clerk extends BankAccount

{

public static void main(String arg[])

{

final int x=10;

final int y;

y=20;

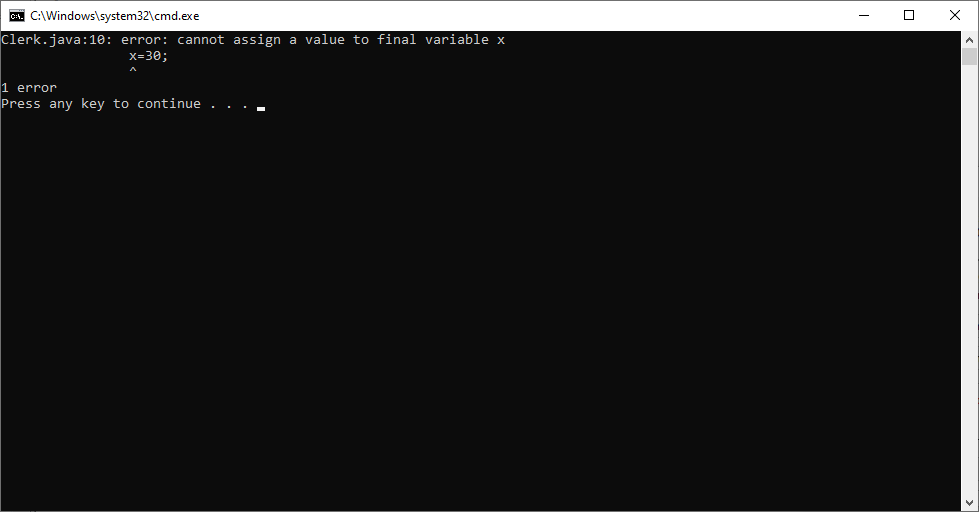
x=30;

System.out.println(x);

System.out.println(y);

}

}



Example:3

class Clerk extends BankAccount

{

int a;

int b;

public static void main(String arg[])

{

final Clerk c=new Clerk();

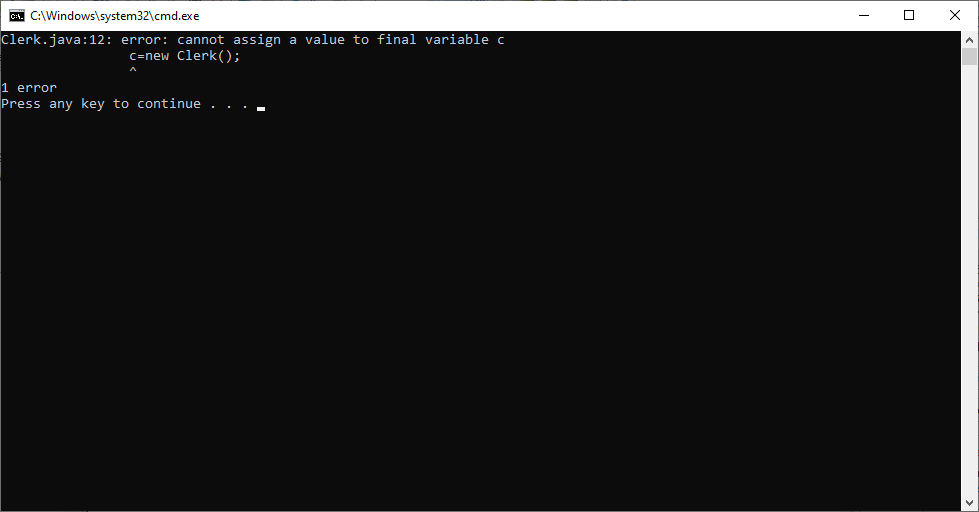
c.a=20;

c.b=30;

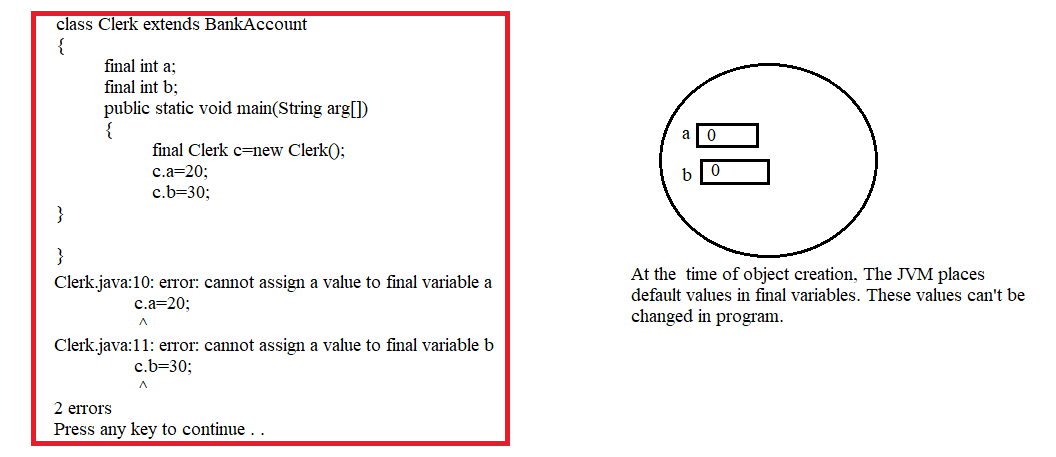
c=new Clerk();

}

}



Example:4



Example:5

Note:- **The final static variable must be initialized in declaration line or any any one of static blocks. non-static variable must be initialized in declaration line, any one of non –static blocks or constructor. Other wise we get compile time error.**

class Clerk extends BankAccount

{

static final int a;

final int b;

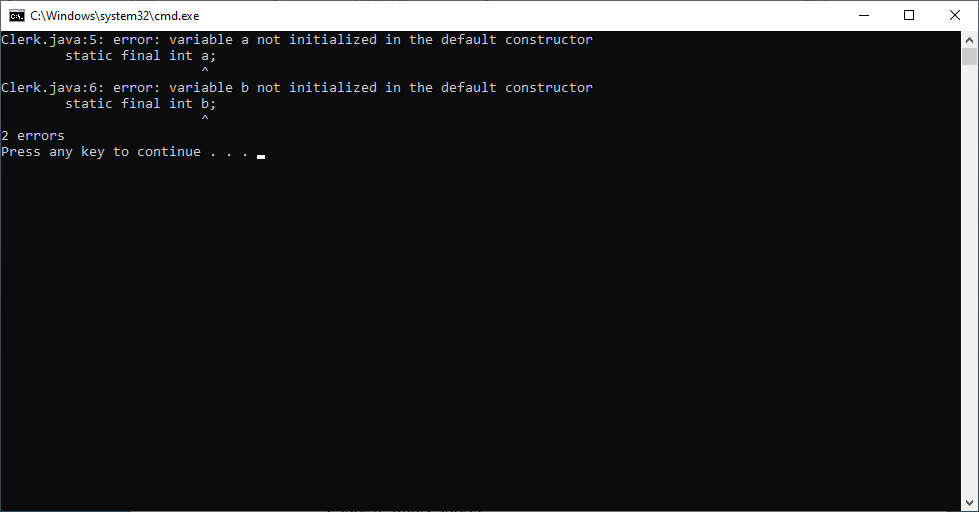
public static void main(String arg[])

{

Clerk c=new Clerk();

}

}



Example:6

class Clerk extends BankAccount

{

static final int a=100;

static final int b;

final int d;

static{ b=40;}

{ d=60;}

public static void main(String arg[])

{

final Clerk c=new Clerk();

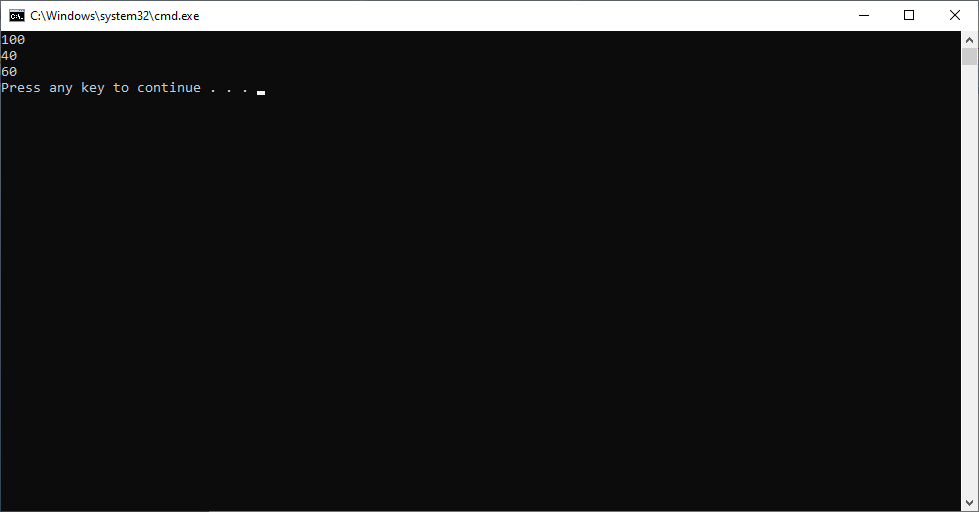
System.out.println(a);

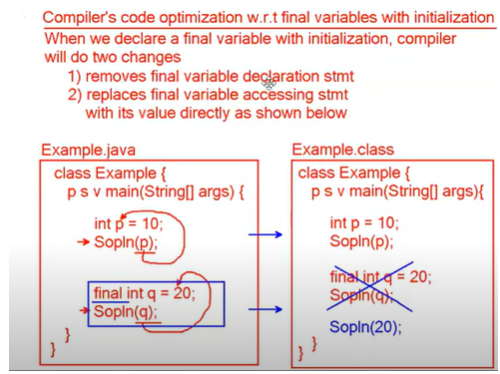
System.out.println(b);

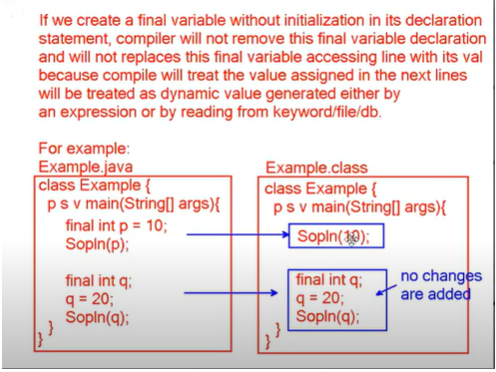
System.out.println(c.d);

}

}

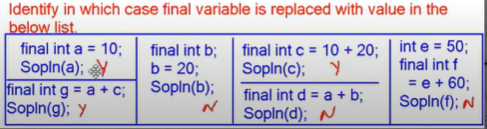








Exercises:



**2.Final Method:-** The method which is declared using final keyword is called Final method. The final method can’t be overridden in sub class. It is inherited in sub class.

The non-final method can be overridden as final method in sub class.

The final method can be overloaded.

The final method can be inherited.

We can declare main method as final.

Example:-

class BankAccount extends Bank

{

public final void display(){System.out.println("Sukumar");}

public final void display(int x){}

public void get(){}

}

class Clerk extends BankAccount

{

// public final void display(){} It leads to CE.

public final void get(){}

public static void main(String arg[])

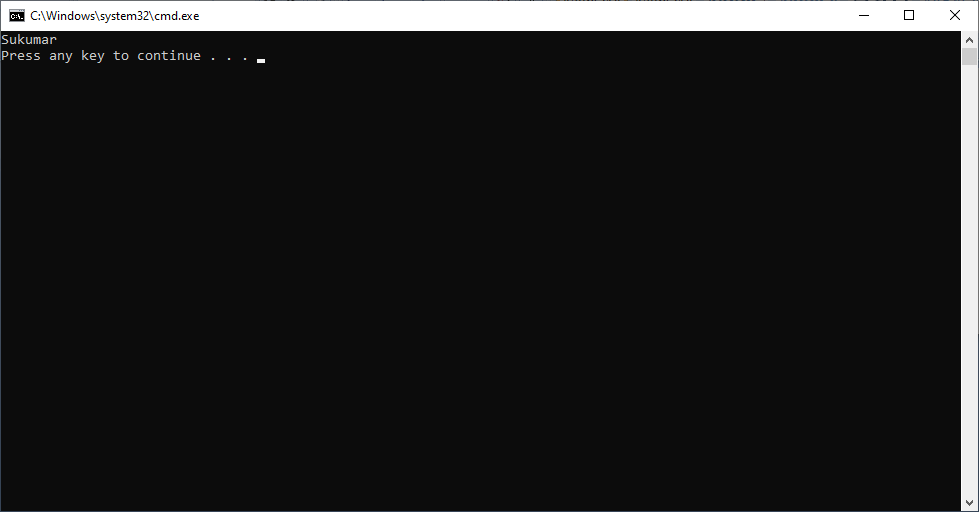
{

BankAccount a=new BankAccount();

a.display();

}

}



**3.Final Class:-** The final class is a class that is declared with the final keyword. Subclasses can't inherit a final class or a final class cannot be inherited by any subclass. So, we can restrict class inheritance by making use of a final class.

-> We don’t’ want to extend functionality of all methods in class. Then why we should declared all methods as final methods. Instead of that Declaring class as final is good.

-> class is final. But Its members may be final and may not be final.

-> we can instantiate final class.

Example1 :-

final class BankAccount

{

}

class Clerk extends BankAccount

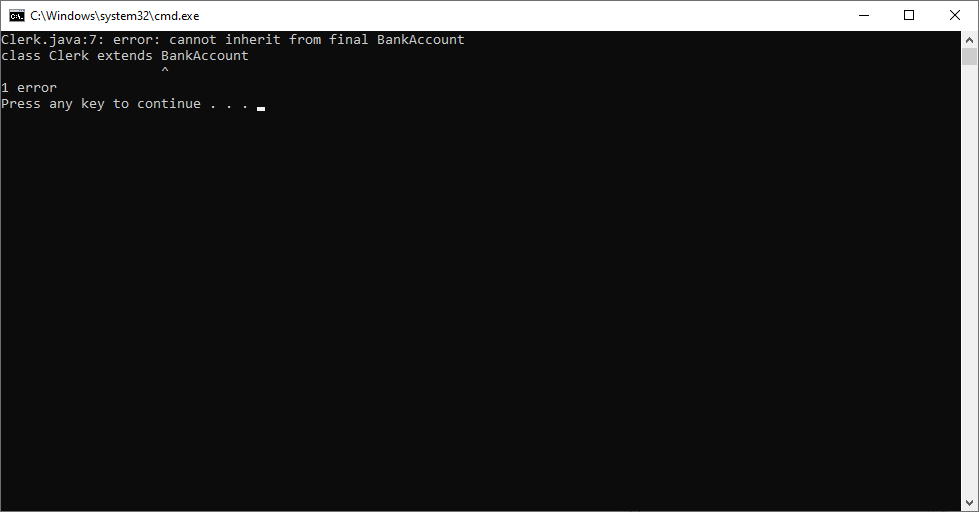
{

public static void main(String arg[])

{

}

}



Example2:-

final class BankAccount

{

final int x=20;

public void abc(){System.out.println("sukumar");}

public final void abc1(){System.out.println("veena");}

}

class Clerk

{

public static void main(String arg[])

{

BankAccount b1=new BankAccount();

System.out.println(b1.x);

b1.abc();

b1.abc1();

}

}