**Object & Class**

OOP’s stands for Object Oriented Programming. It is a concept of using objects in programming to implement real-world entities. OOP provides a clear syntax for our code making it easier to execute the code.

**The key concepts in OOP include:**

* Object & Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

 Advantage:

* Troubleshooting
* code reusability
* Productivity
* redundancy,
* Code flexibility

**Class:**

**(i)class is a collection of object.**

**(ii)class is not a real world entity it is just a template or blueprint of object.**

**(iii)class is does not occupy any memory.**

**Ex. Default access modifier(optional) class ClassName**

**{**

**Methods**

**Constructors**

**Fields,Variable**

**Blocks**

**Nested class**

**}**

**Method: A set of code which perform a particular task.**

**Advantage of Method:**

**i).Code Reusability**

**ii).Code Optimization**

**Syntax: accessmodifier return-type methodName(parameter,parameter)**

**{**

**}**

**Object:**

**i)A object is a instance of class.**

**ii)Object is real worls Entity.**

**iii)Object occupy Memory**

**Object consist of:**

**i)Identity-name**

**ii)State/Attribute-color,bread,age**

**iii)Behavior-eat,run**

**New keyword is used to create Object.**

**Ex.**

**Class Animal{**

**String str = “ run”;**

**}**

**Animal a = new Animal();**

**a.run();**

**Ex.of class object Method:**

class Animal{  
 public void eat()  
 {  
 System.*out*.println("I am eating");  
 }  
 public void run()  
 {  
 System.*out*.println("I am running");  
 }  
 class Birds {  
 public void fly() {  
 System.*out*.println("birds is fly");  
  
 }  
 }  
  
 public static void main(String[] args) {  
 Animal a = new Animal();  
 a.eat();  
 a.run();  
 Animal.Birds b = a.new Birds();  
 b.fly();  
  
 }  
}

**1.By reference variable.**

**Ex.** public class Reference {  
 String color;  
 int age;  
  
 public static void main(String[] args) {  
 Reference a = new Reference();  
 a.color = "black";  
 a.age = 24;  
 System.*out*.println(a.color);  
 System.*out*.println(a.age);  
 }  
}

**2.By using Method.**

public class Animals {  
 String color;  
 int age;  
 void findMethod(String color,int age)  
 {  
 this.color=color;  
 this. age= age;  
 }  
 void display()  
 {  
 System.*out*.println(color+" "+age);  
 }  
  
 public static void main(String[] args) {  
 Animals name = new Animals();  
 name.findMethod("black",20);  
 name.display();  
 }  
}

**Constructor:**

**Constructor is a block(similar to method) having same name as that of class name.**

**Constructor does not have any return type,not even void.**

**Only modifier uses in the constructor are Public,private,protected & default(static,final,synchronized etc cannot be used with constructors).**

**It execute automatically when we create an object.**

**Constructor are used to initialize an Object.**

**There are three types:**

**1.Default Constructor.(No argument Constructor)**

**Default constructor to create compiler.**

**When you creating a object then compiler creates one deafult constructor.**

**We are creating constructor then no default constructor to creating a compiler.**

**Ex:**

package oops.com;  
  
class Test{  
 int i;  
 String str;  
  
 public static void main(String[] args) {  
  
 Test a = new Test();  
 System.*out*.println(a.i);  
 System.*out*.println(a.str);  
 }  
}

package oops.com;  
  
class Employee {  
 String name;  
 int empId;  
 //String name="kaushik";  
 //int empId=101;  
  
 public static void main(String[] args) {  
 Employee e1 = new Employee();  
 Employee e2 = new Employee();  
 System.*out*.println(e1.name+" "+e1.empId);  
 System.*out*.println(e2.name+" "+e2.empId);  
 }  
}

**2.No-arg Constructor:**

**User-defined constructor**

**Progamer create a constructor.**

**Ex.** package oops.com;  
  
  
  
class Test1 {  
 Test1()  
 {  
 System.*out*.println("no-arg constructor");  
 }  
  
 public static void main(String[] args) {  
 {  
 Test1 t = new Test1();  
 }  
 }  
}

**3.Parametrized Constructor:**

**User creating constructor**

**Ex:**

package oops.com;  
  
public class Parametrized {  
 Parametrized(int a)  
 {  
   
 System.*out*.println("parametrized constructor");  
 }  
  
 public static void main(String[] args) {  
 Parametrized p = new Parametrized(10);  
 }  
  
}

package oops.com;  
  
class Employee {  
 String name;  
 int empId;  
 Employee(String name,int empId)  
 {  
 this.name= name;  
 this.empId = empId;  
 }  
  
 public static void main(String[] args) {  
 Employee e1 = new Employee("kaushik",101);  
 Employee e2 = new Employee("akash",102);  
 System.*out*.println(e1.name+" "+e1.empId);  
 System.*out*.println(e2.name+" "+e2.empId);  
 }

**String default value is (Null).**

**Integer default value is (0).**

**Inheritance:**

Inheritance is the mechanism in which one class acquires the properties and features of another class.

Inheritance is inheriting the properties of parent class into child class.

The class that inherits the properties is called as a sub-class (child class) while the class from which the property is inherited is called as the super-class (parent class).

A child class inherits properties of parent class with the help of extends keyword.

Iheritance is called (IS-A) Realtionship.

Example:

package oops.com;  
  
public class Animal {  
 void eat()  
 {  
 System.*out*.println("i m eating");  
 }  
}  
class Dog extends Animal{  
 public static void main(String[] args) {  
 Dog d = new Dog();  
 d.eat();  
 }  
}

Advantage:

i. Code Reusability

ii. We can achieve Run time polymorphism using inheritance.

Disadvantange:

* Classes are tightly couple then we can change one class, all classes are change.

**Inheritance can be further divided into the following types:**

* Single level
* Multi-level
* Hierarchical
* Multiple
* Hybrid

Multiple and hybrid inheritance is not directly supported in java, instead it is achieved through the use of interfaces in java.

**i. Single Inheritance**

When a single class inherits the attributes and methods of another class, it is known as single inheritance.



**Example:**

package oops.com;  
  
public class Animals {  
  
 void showA()  
 {  
 System.*out*.println("First method:");  
 }  
}  
  
  
class Lion extends Animals{  
 void showB()  
 {  
 System.*out*.println("second method:");  
 }  
  
 public static void main(String[] args) {  
 Animals a = new Animals();  
 a.showA();  
 Lion l = new Lion();  
 l.showA();  
 l.showB();  
 }  
}

Output:

First method:

First method:

second method:

**ii. Multi-level Inheritance**

When a class 3 inherits attributes and methods from class 2 which in turn inherits its attributes and methods from class 1, it is called a multi-level inheritance.

It forms a child-parent-grandparent (or a parent-child-grandchild) relationship. Meaning that child inherits from the parent while the parent inherits from the grandparent.



**Example:**

package oops.com;  
  
public class A {  
 void showa()  
 {  
 System.*out*.println("A class method:");  
 }  
}  
class B extends A{  
 void showb()  
 {  
 System.*out*.println("B class method:");  
 }  
}  
 class C extends B{  
 void showc()  
 {  
 System.*out*.println("C class method:");  
 }  
  
 public static void main(String[] args) {  
 A a = new A();  
 a.showa();  
 System.*out*.println("----------------");  
 B b = new B();  
 b.showa();  
 b.showb();  
 System.*out*.println("----------------");  
 C c = new C ();  
 c.showa();  
 c.showb();  
 c.showc();  
 }  
 }

Output:

A class method:

----------------

A class method:

B class method:

----------------

A class method:

B class method:

C class method:

**iii. Hierarchical Inheritance**

Hierarchical inheritance is when two or more classes inherit from a single class. This can be easily visualized as a parent with more than one child. Here each child can inherit the properties of a parent.



**Example:**

package oops.com;  
  
public class D {  
 void showd()  
 {  
 System.*out*.println("d class method:");  
 }  
}  
class E extends D{  
 void showe()  
 {  
 System.*out*.println("e class method");  
 }  
}  
class F extends D{  
 void showf()  
 {  
 System.*out*.println("F class method");  
 }  
  
 public static void main(String[] args) {  
 D d = new D();  
 d.showd();  
 System.*out*.println("----------");  
 E e = new E();  
 e.showd();  
 e.showe();  
 System.*out*.println("\_\_\_\_\_\_\_\_\_\_");  
 F f = new F();  
 f.showd();  
 f.showf();  
 }  
}

Output:

d class method:

----------

d class method:

e class method

\_\_\_\_\_\_\_\_\_\_

d class method:

F class method

Important:

Constructor cannot inherit.

Private method cannot inherit.

Realtionship between classes:

Inheritance:

i.(IS-A)

ii.extends

iii.blood relation

iv.tightly coupled

Association:

Association in Java defines the connection between two classes that are set up through their objects. Association manages **one-to-one, one-to-many**, and **many-to-many** relationships.

i.(HAS-A)

ii.reference variable,new

iii.no-blood relation

iv.no tightly coupled

Association has two types:

i.Aggregation: it is a weak bonding.

The [**Aggregation**](https://www.javatpoint.com/aggregation-in-java) association defines the **HAS-A** relationship. Aggregation follows the one-to-one or one-way relationship. If two entities are in the aggregation composition, and one entity fails due to some error, it will not affect the other entity.

ii.Composition:It is a strong bonding.

A restricted form of the **Aggregation** where the entities are strongly dependent on each other. Unlike Aggregation, [Composition](https://www.javatpoint.com/composition-in-java) represents the **part-of** relationship. When there is an aggregation between two entities, the aggregate object can exist without the other entity, but in the case of Composition, the composed object can't exist.

**Polymorphism**

The word polymorphism is derived from the Greek words poly(meaning many), and morph(meaning form).

it is the property of some code to behave differently for different contexts.

**Polymorphism is of two types:**

* Compile time polymorphism
* Run time polymorphism

Method Overloading:

Compile time polymorphism achieve method overloading

Create multiple method having same name

Multiple method inside Same class

Arguments should be different:

-no of argument

-type of argument

-sequence of argument

Ex:

public class Tests {  
 void shows()  
 {  
 System.*out*.println("1");  
 }  
 void shows()  
 {  
 System.*out*.println("2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows();  
 }  
}

Output:

java: method shows() is already defined in class oops.com.Tests

Passed argument:

public class Tests {  
 void shows()  
 {  
 System.*out*.println("1");  
 }  
 void shows(int a)  
 {  
 System.*out*.println("2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows(10);  
 }  
}

output:2

Sequence different:

public class Tests {  
 void shows(int a, String b) {  
 System.*out*.println("output:1");  
 }  
  
 void shows(String b, int a) {  
 System.*out*.println("output:2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows(10, "kaushik");  
 }  
}

Output:1

public class Tests {  
 void shows(int a, String b) {  
 System.*out*.println("output:1");  
 }  
  
 void shows(String b, int a) {  
 System.*out*.println("output:2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows( "kaushik",10);  
 }  
}

Output:2

Type argument:

public class Tests {  
 void shows(int a) {  
 System.*out*.println("output:1");  
 }  
  
 void shows(float a) {  
 System.*out*.println("output:2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows(10.5);  
 }  
}

Output:2

Ouestions:

1.Can we achieve Method Overloading by changing the return type of method only?

Ans.In java, Method Overloading is not possible by changing the return type of the method only because of ambiguity.

Examples:

public class Tests {  
 int shows(int a) {  
 System.*out*.println("output:1");  
 }  
  
 void shows(int a) {  
 System.*out*.println("output:2");  
 }  
  
 public static void main(String[] args) {  
 Tests t = new Tests();  
 t.shows(10);  
 }  
}

Output: java: method shows(int) is already defined in class oops.com.Tests

2.Can we overloaded java main() method?

Ans.yes,we can have any number of main methods in a class by method overloading.

This is because jvm always claas main() method which receives string array as arguments only.

Example:

class Tests{  
 public static void main(String[] args) {  
 System.*out*.println("1");  
 Tests t = new Tests();  
 t.*main*(20);  
 }  
  
 public static void main(int a) {  
 System.*out*.println("2");  
 }  
}

Output:

1

2

Automatic Promotion:

Char->Int-Float-long-double

class tests{  
 void show(int a)  
 {  
 System.*out*.println("1");  
 }  
 void show(String b)  
 {  
 System.*out*.println("2");  
 }  
  
 public static void main(String[] args) {  
 tests t = new tests();  
 t.show('a');  
 }  
}

Output:1

class tests{  
 void show(int a,float b)  
 {  
 System.*out*.println("1");  
 }  
 void show(float b,int a)  
 {  
 System.*out*.println("2");  
 }  
  
 public static void main(String[] args) {  
 tests t = new tests();  
 t.show(10,20);  
 }  
}

Compiler was confused show ambigious error.

Varargs:

class tests{  
 void show(int a)  
 {  
 System.*out*.println("1 argument");  
 }  
 void show(int...a)  
 {  
 System.*out*.println("Varargs method");  
 }  
  
 public static void main(String[] args) {  
 tests t = new tests();  
 t.show(10,20,30);  
 }  
}

Output:Varargs method

class tests{  
 void show(int a)  
 {  
 System.*out*.println("1 argument");  
 }  
 void show(int...a)  
 {  
 System.*out*.println("Varargs method");  
 }  
  
 public static void main(String[] args) {  
 tests t = new tests();  
 t.show();  
 }  
}

Output:Varargs method

class tests{  
 void show(int a)  
 {  
 System.*out*.println("1 argument");  
 }  
 void show(int...a)  
 {  
 System.*out*.println("Varargs method");  
 }  
  
 public static void main(String[] args) {  
 tests t = new tests();  
 t.show(10);  
 }  
}

Output:1 arguments:

* **Compile-time polymorphism:**

Compile time polymorphism, also known as static polymorphism.

Method overloading or operator overloading are examples of compile-time polymorphism.

We can achieve compile time polymorphism with method overloading.

**Example:**

class AddMethods {

static int AddNums(int a, int b) {

return (a+b);

}

static int AddNums(int a, int b, int c) {

return (a+b+c);

}

static double AddNums(double a, double b) {

return (a+b);

}

}

public class Example1 {

public static void main(String[] args) {

System.out.println("Addition of two integers: "+ AddMethods.AddNums(6, 11));

System.out.println("Addition of three integers: "+ AddMethods.AddNums(6, 11, 5));

System.out.println("Addition of two decimal numbers: "+ AddMethods.AddNums(6.3, 2.8));

}

}

Copy

Output:

Addition of two integers: 17

Addition of three integers: 22

Addition of two decimal numbers: 9.1

Copy

Here we have defined three methods with same name, but the parameters passed inside each differ by either the data type or the number of arguments passed. And we get output from each of these methods based on the input provided.

**ii. Run-time polymorphism:**

We can achieve runtime polymorphism with method overiden and dynamic dispatch.

Method Overiding:

Method has Same name

Multiple method inside Different class

Same Agrumnnts:

- no of argument

-type of argument

-sequence of argument

It is supported inheritance.

**Example:**

package oops.com;  
  
public class Bird {  
 void show(){  
 System.*out*.println("1st method");  
 }  
}  
class Parrot extends Bird{  
 void show()  
 {  
 System.*out*.println("2 nd method");  
 }  
  
 public static void main(String[] args) {  
 Bird b = new Bird();  
 b.show();  
 Parrot p = new Parrot();  
 b.show();  
 p.show();  
 }  
}

Output:

1st method

1st method

2 nd method

Same Agrumnnts:

- no of argument

-type of argument

-sequence of argument

Example:

package oops.com;  
  
public class Bird {  
 void show(int a, String b, float c){  
 System.*out*.println("1st method");  
 }  
}  
class Parrot extends Bird{  
 void show(int a, String b, float c)  
 {  
 System.*out*.println("2nd method");  
 }  
  
 public static void main(String[] args) {  
 Bird b = new Bird();  
 b.show(10," ",10.3f);  
 Parrot p = new Parrot();  
  
 p.show(20," ",20.5f);  
 }  
}

Output:

1st method

2nd method

Questions:

1.Do Overiding method must have same name return type(or subtype)?

Ans.From java 5.0 onwards it is possible to have different return type for a overriding method in child class,but child’s return type should be sub –type of parent’s return type.So this is known as Covarient return type.

Example:

package oops.com;  
  
public class Test2 {  
 Object show()  
 {  
 System.*out*.println("1st method");  
 return "";  
 }  
}  
class xyz extends Test2{  
 String show()  
 {  
 System.*out*.println("2nd method");  
 return "";  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

Access modifier should be same:

Child class access modifier big as compared to parent:

Access modifier should be anything but follow below example this rule:

Example:

package oops.com;  
  
public class Test2 {  
 public void show()  
 {  
 System.*out*.println("1st method");  
   
 }  
}  
class xyz extends Test2{  
 public void show()  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

package oops.com;  
  
public class Test2 {  
 void show()  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 public void show()  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Same output

package oops.com;  
  
public class Test2 {  
 public void show()  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show()  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

java: show() in oops.com.xyz cannot override show() in oops.com.Test2

attempting to assign weaker access privileges; was public

Overiding and Exception handling:

Parent class does not throws any exception then child class only throws unchecked exception, but we used checked exception then we provide the error.(Arithmetic Exception is unchecked exception)

(Exception is checked Exception)

Example:

package oops.com;  
  
public class Test2 {  
 void show()  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show() throws ArithmeticException  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

package oops.com;  
  
public class Test2 {  
 void show()  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show() throws Exception  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

java: show() in oops.com.xyz cannot override show() in oops.com.Test2

overridden method does not throw java.lang.Exception

Rule 2:

If we provide a parent class (throws RuntimeException)

then child class throws same exception.

package oops.com;  
  
public class Test2 {  
 void show() throws RuntimeException  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show() throws RuntimeException  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

If we provide same level exception and maybe provide child class(AritmeticException)then compile and run not provide the error.

Example:

package oops.com;  
  
public class Test2 {  
 void show() throws RuntimeException  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show() throws ArithmeticException  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

If we not provide any exception in child class then program compile and run.

package oops.com;  
  
public class Test2 {  
 void show() throws RuntimeException  
 {  
 System.*out*.println("1st method");  
  
 }  
}  
class xyz extends Test2{  
 void show()   
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

1st method

2nd method

But if we provide a parent class ( throws Exception) then program provide the error.

package oops.com;  
  
public class Test2 {  
 void show() throws RuntimeException  
 {  
 System.*out*.println("1st method");  
  
 }  
 }  
}  
class xyz extends Test2{  
 void show() throws Exception  
 {  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
 Test2 t = new Test2();  
 t.show();  
 xyz x= new xyz();  
 x.show();  
 }  
}

Output:

java: class, interface, enum, or record expected

Abstract class always override:

Abstract class does not create object.

Important:

We can call parent class method in overriding method using super keyword:

Example:

package oops.com;  
  
public class Test2 {  
 void show()  
 {  
  
 System.*out*.println("1st method");  
  
 }  
  
}  
class xyz extends Test2{  
 void show()  
 {  
 super.show();  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
  
 xyz x= new xyz();  
 x.show();  
   
  
  
 }  
}

Output:

1st method

2nd method

Super class is used to show the parent class method using (super.method name).

Final,Static,Private method cannot Ovveride.

Example:

package oops.com;  
  
public class Test2 {  
 final void show()  
 {  
  
 System.*out*.println("1st method");  
  
 }  
  
}  
class xyz extends Test2{  
 void show()  
 {  
 super.show();  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
  
 xyz x= new xyz();  
 x.show();  
  
  
  
 }  
}

Output:

java: show() in oops.com.xyz cannot override show() in oops.com.Test2

overridden method is final

Same as using static,final keyword.

Private:

Print child class and not override parent class.

package oops.com;  
  
public class Test2 {  
 private void show()  
 {  
  
 System.*out*.println("1st method");  
  
 }  
  
}  
class xyz extends Test2{  
 void show()  
 {  
  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
  
 xyz x= new xyz();  
 x.show();  
  
  
  
 }  
}

Output:2nd method

If we use synchronized keyword as same as private method.(Another strictfp keyword is same as synchronized keyword)

If we use synchronized key in parent and child class then not affect on overriding.

package oops.com;  
  
public class Test2 {  
 synchronized void show()  
 {  
  
 System.*out*.println("1st method");  
  
 }  
  
}  
class xyz extends Test2{  
 void show()  
 {  
  
 System.*out*.println("2nd method");  
  
 }  
  
 public static void main(String[] args) {  
  
 Test2 t = new Test2();  
 t.show();  
  
 xyz x= new xyz();  
 x.show();  
  
  
  
 }  
}

Output:

1st method

2nd method

**Abstraction**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction is similar to data encapsulation and is very important in OOP.

 Abstraction is implemented using classes and interfaces.

The abstract keyword is used to declare abstract classes.



Example: A driver only knows that pressing the accelerator increases speed or applying brakes stops the car, but they don't know how.

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

* **Abstract method:** can only be used in an abstract class, and it does not have a body.
* **If we create abstract method then compulsory to create abstract class, but abstract class we declare then not compulsory to declare abstract method also if create simple method.**
* **Abstraction are used to method overriding concept.**
* If any class inherit the abstract class then inside abstract class all method then making their body is compulsory,if we not create then provide the error.

Abstraction achieve in two ways:

i.Abstart class(Abstract method)

ii.Interface

* **Abstract Methods:**

Inside the abstract class, when we create an abstract method, we use a semicolon (;) not uses of curly brackets ({}).

**Example:**

package oops.com;  
  
abstract class Vechile {  
 abstract void start();  
}  
class Car extends Vechile{  
 void start()  
 {  
 System.*out*.println("car start:");  
 }  
}  
class Scooter extends Vechile{  
 void start()  
 {  
 System.*out*.println("scooter start");  
 }  
  
 public static void main(String[] args) {  
 //Vechile v = new Vechile(); If we not create abstract class object  
 Car c = new Car();  
 c.start();  
 Scooter s = new Scooter();  
 s.start();  
 }  
}

 Output:

car start:

scooter start

* **Interfaces:**

A unique class type known as an interface

**Interface are the blueprint of class.** The interface tells the class what to do.

Interface in java is mainly used to achieve abstraction.

It support Multiple Inheritance.

It can be use to achieve loose coupling(if we change one class class that does not affect another class).

All method we create an interface using pubic and abstract keywod.

In interface we create default concrete method,static.private.interface

Interface method cannot have body.

Default and static method have body.

interface declare variable public ,static, final int a =10;

Ex.

default void display()

{

}

Run method is always use public keyword.

We use the implements keyword to implement an interface.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6)Abstract class cannot create object | same |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

If interface all method declare abstract keyword.

**Example:**

package oops.com;  
  
interface I1 {  
 void show();  
}  
class Cehcks implements I1  
{  
 public void show()  
 {  
 System.*out*.println("1");  
 }  
  
 public static void main(String[] args) {  
 Cehcks c= new Cehcks();  
 c.show();  
 }  
}

Output:

1

It support Multiple inheritance:

Ex:

package oops.com;  
  
interface I1 {  
 void show();  
}  
interface I2{  
 void display();  
}  
class Cehcks implements I1,I2  
{  
 public void show()  
 {  
 System.*out*.println("1");  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("2");  
 }  
  
 public static void main(String[] args) {  
 Cehcks c= new Cehcks();  
 c.show();  
 c.display();  
 }  
}

Output:

1

2

**Encapsulation**

Encapsulation in java is a mechanism of wrapping the data (variables)

and data(methods)together as a single unit.

Steps to achieve Encapsulation:



Declare the variable of a class as private.(it is hiding the data not accessible another class)

Provide public getter and setter methods to modify and view the variable value.

**i. Get and set methods:**

We use set method to set value of variable and get method to get the value of variable.

**Example:**

package oops.com;  
  
public class Student {  
 private int empid;  
  
 public void setEmpid(int empid) {  
 this.empid = empid;  
 }  
  
 public int getEmpid() {  
 return empid;  
 }  
}  
class school{  
 public static void main(String[] args) {  
 Student s = new Student();  
 s.setEmpid(102);  
 System.*out*.println(s.getEmpid());  
 }  
}

Output:

102

**Advantages of Encapsulating data:**

* We can hide our data more efficiently. Hence, after implementation, user will not have any idea about the inner working of the class. To the user, only setting and initializing values is visible.
* It makes our data reusable.
* Encapsulated data is easy to test.

**Disadvantages of Encapsulating data:**

* Size of the code increases exponentially.
* As the size of the code increases, we need to provide additional implementation for each method.
* We provide additional methods, this increases code execution.

ClassLoader Subsystem:

The ClassLoader is responsible for loading class files into the JVM.

It is a part of the JVM that reads the bytecode of classes and loads them into memory when needed.

The ClassLoader is important for dynamic loading of classes and for ensuring that classes are loaded only once during the execution of a program.

There are different types of class loaders in the JVM:

Bootstrap ClassLoader: This is the top-level class loader that loads core Java libraries (e.g., java.lang.\*, java.util.\*) from the rt.jar file.

Extension ClassLoader: It loads classes from the JDK's lib/ext directory or any other specified extension directories.

System ClassLoader: Also known as the application class loader, it loads classes from the classpath.