JAVA OOPS

**Inheritance**

OOP is a programming paradigm based on the concept of "objects," which can contain data and code. Java is a strongly object-oriented 2 language. ● Inheritance: Allows a class (subclass/child class) to inherit properties and behaviors (method) from another class (superclass/parent class). This promotes code reusability.

Ex, by class

1. granparent
2. parent
3. child

**Super();**

* In Java, super() is used **inside a constructor** to call the **constructor of the parent class** (superclass).
* **To initialize the parent class variables or logic**.
* It helps in **code reusability** by invoking the already written constructor logic of the superclass.
* It must be the **first statement** in the constructor.

**Key Notes:**

* If you don’t explicitly write super(), Java automatically inserts it **if the parent has a no-arg constructor**.
* If the superclass **only has a parameterized constructor**, then super(arguments) must be used explicitly.

**package** OOPS;

**public** **class** grandparent {

**public** grandparent() //this is constructor

{

System.***out***.println(" i am in grandparent constructor");

}

**public** **void** farmhous()

{

System.***out***.println("parents have farmhous method");

}

**public** **void** village()

{

System.***out***.println("parents have village method");

}

}

**public** **class** ParentClass **extends** grandparent {

**public** ParentClass(){ //this is constructor

**super**(); // super call should be first 🡪 it calls grandparent class constructor

System.***out***.println("we are in parent class constructor");

}

**public** **void** tv() {

System.***out***.println("parents have TV method");

}

**public** **void** fridge() {

System.***out***.println("parents have fridge method");

}

}

**package** OOPS;

**public** **class** ChildClass **extends** ParentClass {

// java support multi-level inheritance not multiple inheritance

// Class can extends only one class

**public** ChildClass() //this is constructor --> as soon as object is created it calls for constructor

{

**super**(); // super call should be first 🡪 it calls parent class constructor

System.***out***.println("i am in child class constructor");

}

**public** **void** education() //Return Type🡪 void -returns> nothing , int -returns> integer value

{

System.***out***.println("chlid have education method ");

}

**public** **static** **void** main(String args[]) {

System.***out***.println("i am in child class main method ");

ChildClass ob1 = **new** ChildClass(); // creating object of child calss

ob1.education();

ob1.tv();

ob1.fridge();

ob1.farmhous();

ob1.village();

// parent level object can access only parent methods and methods which parents have inherited from grndparentclass.

ParentClass p1 = **new** ParentClass(); // left side is refernce/ assignment, right side new object creation

p1.tv();

p1.fridge();

p1.village();

p1.farmhous();

**This is not allowed**

ChildClass ch = new ParentClass(); //we are creating object of parentclass, and assiging it to child class

**This is allowed**

// upcasting

**ParentClass b = new ChildClass();** //we are creating object of ChildClass and assiging it to parent

class reference

// object can access methtods which ever class reference is created.

Meaning it can access parent methods and methods which parent class extended , in this ex, parentclass extended grandparent

b.tv();

b.fridge();

b.farmhous();

b.village();

}

}

Overloading

**package** OOPS;

**public** **class** area {

// overloading -> methods have same name but different parameters.

// overloading is compiled time polymorphism

// we can achieve in same class

**public** **void** area(**int** l) {

System.***out***.println("area of circle --> " + 3.14 \* l \* l);

}

**public** **void** area(**int** l, **int** b) {

System.***out***.println("area of trainagle -->" + 0.5 \* l \* b);

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

area a = **new** area();

a.area(4);

a.area(5, 6);

}}

**1. Definition:**

**Method overloading** means **defining multiple methods with the same name** in the same class, but with **different parameter lists**.

**✅ 2. Compile-time Polymorphism:**

* Overloading is an example of **compile-time polymorphism (static binding)**.
* The decision of which method to call is made during **compilation**.

**✅ 3. Rules for Overloading:**

To overload a method, you must change **at least one** of the following:

* Number of parameters
* Type of parameters
* Sequence of parameters

❌ You **cannot overload** just by changing the **return type**.

**✅ 4. Same Method Name, Different Signature:**

Each overloaded method has a **different signature** (method name + parameter types).

**✅ 5. Constructors Can Also Be Overloaded:**

Like methods, **constructors** can also be overloaded in Java.

**✅ 6. Inheritance is Not Required:**

Overloading can happen **within the same class**, no need for inheritance.

### ✅ ****7. Improves Readability****:

Using the **same name** for logically similar operations improves **code readability and maintainability**.

Overriding

**Method overriding** means providing a **specific implementation** of a method in a subclass that is **already defined in its superclass**.

**package** OOPS;

**public** **class** parentoverriding {

**public** **void** area(**int** l) {

System.***out***.println("area of circle --> " + 3.14 \* l \* l);

}

}

**. Only Inheritance-based:**

* Overriding requires **inheritance** (i.e., child class and parent class relationship).

**package** OOPS;

**public** **class** childoverriding **extends** parentoverriding {

// we need 2 different class to achieve overriding .

// method name should be same as well as parameters should be same.

// overridden method

@Override

**public** **void** area(**int** l) {

System.***out***.println("area of circle --> " + 3.14 \* l \* l);}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

childoverriding o1 = **new** childoverriding(); // creating object of child class

o1.area(5);

creating object of child class and assigning to parent class reference .In this case it will call overridden method in child class .

**parentoverriding o2 = new childoverriding();**

**o2.area(6);**

// to call parent class method we need to create object of parent class only

parentoverriding o3 = **new** parentoverriding();

o3.area(5);

}

}

**2. Run-time Polymorphism:**

* Overriding is an example of **run-time polymorphism** (dynamic method dispatch).
* The method to be executed is determined **at runtime** based on the object type.

**✅ 3. Access Modifier Rule:**

* The overriding method **cannot have a more restrictive access modifier** than the overridden method.
  + For example, if the parent method is public, the child method **must also be** public.

**✅ 4. Cannot Override:**

* **Constructors**
* **Private methods** (they are not inherited)
* **Final methods** (cannot be changed)
* **Static methods** (they are hidden, not overridden)

Casting

**package** OOPS;

**public** **class** casting {

**public** **static** **void** main(String args[]) {

**int** n = 16;

// Widening (Implicit) Up-Casting: Automatic conversion from a smaller type to

// larger type. No data loss

**double** d = n;

// Narrowing (Explicit)/down-Casting: Manual conversion from a larger type to a

// smaller type. Requires a cast operator ()

// and can lead to data loss if the larger value doesn't t in the smaller type

**int** a = (**int**) d;

}

}

// byte🡪 short 🡪 int 🡪 long 🡪 float 🡪 double this is sequence from smaller to larger

● Upcasting: Converting a subclass type to a superclass type. This is implicitly done by the JVM and is always safe.

It's a form of polymorphism.

class Animal

{ }

class Dog extends Animal

{

}

// Upcasting

Animal myAnimal = new Dog(); // Dog object is treated as an Animal object //we are creating object of ChildClass and assiging it to parent class reference

Abstaract Class

* Abstraction: The concept of hiding the complex implementation details and showing only the essential features of an object.
* This can be achieved using abstract classes and interfaces.
* Abstract Class: A class that cannot be instantiated directly and may contain abstract methods (methods without an implementation).
* 0Subclasses must implement these abstract methods.

**package** OOPS;

**abstract** **class** demoabstract { //parentclass

**abstract** **void** area();

**public** **void** rectangle() {

System.***out***.println("in rectangle class");

}

}

**package** OOPS;

**public** **class** abstractTest **extends** demoabstract {

@Override

**public** **void** area() {

// **TODO** Auto-generated method stub

System.***out***.println("in overridden method from abstract");

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

abstractTest ab = **new** abstractTest();

ab.area();

ab.rectangle();

**This is not allowed**

// demoabstract ab1 = new demoabstract(); // cannot create object of abstract class

demoabstract ab2 = **new** abstractTest(); // creating object of child class and assigning to parent class

ab2.area();

ab2.rectangle();

}}

### ****Key Features:****

| **Feature** | **Description** |
| --- | --- |
| 🔸 Can't Instantiate | You **cannot create objects** of abstract classes. |
| 🔸 Abstract Methods | Can contain **0 or more abstract methods**. |
| 🔸 Concrete Methods | Can also contain **normal methods** with implementation. |
| 🔸 Constructors | Can have **constructors**, but they are called via subclass. |
| 🔸 Fields | Can contain variables, including final/static fields. |

### ****Why Use Abstract Class?****

* To provide a **common template or structure** for multiple subclasses.
* To enforce **common behavior** that must be implemented by child classes.
* To avoid **code duplication** and promote **code reusability**

Interface

### ****What is an Interface?****

An **interface** in Java is a **blueprint** of a class. It is used to achieve **abstraction** and **multiple inheritance**.  
It contains **abstract methods (by default)** and can also have **default, static, and private methods** (from Java 8 onwards).

**Key Features of Interface:**

| **Feature** | **Description** |
| --- | --- |
| 🔸 Only Method Signatures | All methods are **abstract by default** (until Java 7). |
| 🔸 Cannot Instantiate | You **cannot create objects** of an interface. |
| 🔸 Implements Keyword | A class **implements** an interface. |
| 🔸 Multiple Interfaces | A class can **implement multiple interfaces**. |
| 🔸 Fields | All fields are public static final by default (constants). |

**✅ 4. From Java 8+:**

* **Default methods**: Method with body using default keyword.
* **Static methods**: Can be called using InterfaceName.method().
* **Private methods**: Can be used inside default/static methods.

**package** OOPS;

**interface** interfacedemo {

**public** **void** triangle();

**public** **void** circle();

/\*

\* From Java 9+: Default methods: Method with body using default keyword.

\*

\* Static methods: Can be called using InterfaceName.method().

\*

\* Private methods: Can be used inside default/static methods.

\*/

**static** **void** area() {

**int** a = 50;

System.***out***.println("i am in interface class with default method");

}

**default** **void** arearectangle() {

**int** a = 50;

System.***out***.println("i am in interface class with default method");

}

**private** **void** areasquare() {

**int** a = 50;

System.***out***.println("i am in interface class with default method");

}

}

**package** OOPS;

**interface** interfacedemo2 {

**public** **void** year();

**static** **void** days() {

**int** a = 50;

System.***out***.println("i am in interface class with default method");

}

**default** **void** months() {

**int** a = 50;

System.***out***.println("i am in interface class with default method"); } }

**package** OOPS;

**public** **class** interfaceTest **implements** interfacedemo, interfacedemo2 {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

interfaceTest in = **new** interfaceTest(); // creating object of child class

in.arearectangle(); //**by using object we can call default methods only**

in.circle(); //this is overridden method in child class

in.triangle(); //this is overridden method in child class

in.year(); //this is overridden method in child class

interfacedemo.*area*(); //this is to call static method . syntax -> interfacename.methodname

interfacedemo2.*days*(); //this is to call static method . syntax -> interfacename.methodname

in.months();

}

@Override

**public** **void** triangle() {

// **TODO** Auto-generated method stub

System.***out***.println("this is area of triangle");

}

@Override

**public** **void** circle() {

// **TODO** Auto-generated method stub

System.***out***.println("this is area of cirlce");

}

@Override

**public** **void** year() {

// **TODO** Auto-generated method stub

System.***out***.println("this is from interface demo 2");

}

}

### ****When to Use Interface:****

* When you need **multiple inheritance**.
* When you want to **enforce contracts** across unrelated classes.
* When building **plugin-like** systems or **APIs**.

### ****Difference Between Abstract Class and Interface**** (Short) :

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| Methods | Can have abstract + normal | Only abstract (Java 7), default/static (Java 8+) |
| Inheritance | Supports single inheritance | Supports multiple inheritance |
| Constructors | Yes | No |
| Access Modifiers | Can use any access modifier | Methods are public by default |

### Key Points:

* private methods in interfaces **cannot be accessed outside the interface**.
* They are used to **refactor code** used in multiple default/static methods.
* They **improve code readability and reduce duplication**.