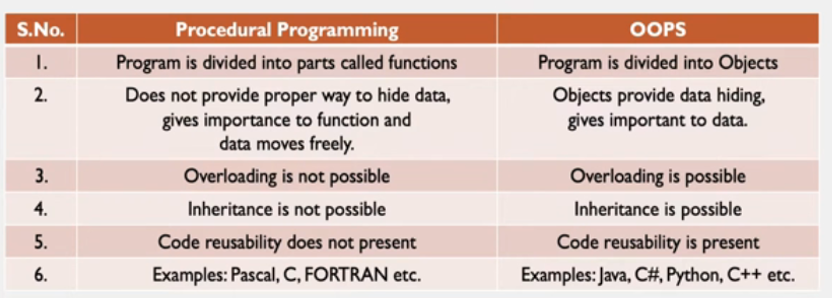
# OOPs🡪

**Object-Oriented Programming** is a programming approach that organizes code into objects and classes and makes it more structured and easier to manage. A class is a blueprint that defines properties and behaviors, while an object is an instance of a class representing real-world entities. Like car, ATM, School.



# Java Class

A [Class](https://www.geeksforgeeks.org/classes-objects-java)is a**user-defined blueprin**t or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. It is not a real-world entity and does not occupy memory.

In general, class declarations can include these components in order:

* + **Modifiers**, **Class name**, **Body** (Data member, Method, Constructor, Nested Class, Interface)

# Java Object

An [Object](https://www.geeksforgeeks.org/object-class-in-java/)is a basic unit of Object-Oriented Programming that represents real-life entities. An object mainly consists of:

* **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
* **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
* **Identity**: It is a unique name given to an object that enables it to interact with other objects.

****

### Ways to Create an Object of a Class

#### Using New Keyword

*Test t = new Test();*

#### Using Reflection

#### Using Serialization/deserialization

#### Using Clone Method

# 4 Pillars of Java OOPs Concepts

## 1. Abstraction

**Abstraction in Java** is the process of hiding the implementation details and only showing the essential details or features to the user.

**Real-life Example:**

* Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.
* **The television remote control**is the best **example**of **abstraction**. It simplifies the interaction with a TV by hiding all the complex technology. We don’t need to understand how the tv internally works, we just need to press the button to change the channel or adjust the volume.

**Note:** In Java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java) 100% and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java).

**Example:**

// Working of Abstraction in Java  
abstract class Geeks {  
 abstract void turnOn();  
 abstract void turnOff();  
}  
  
// Concrete class implementing the abstract methods  
class TVRemote extends Geeks {  
 @Override  
 void turnOn() {  
 System.*out*.println("TV is turned ON.");  
 }  
  
 @Override  
 void turnOff() {  
 System.*out*.println("TV is turned OFF.");  
 }  
}  
  
// Main class to demonstrate abstraction  
public class Main {  
 public static void main(String[] args) {  
 Geeks remote = new TVRemote();  
 remote.turnOn();  
 remote.turnOff();  
 }  
}

Advantage –

* Abstraction makes complex systems easier to understand by hiding the implementation details.
* Abstraction keeps different part of the system separated.
* Abstraction maintains code more efficiently.
* Abstraction increases the security by only showing the necessary details to the user.

## 2. Encapsulation/data hiding

It is defined as the **wrapping up of data(variable) and method(operate on the data) into a single unit.** Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

**Example:**

// Java program demonstrating Encapsulation  
class Programmer {  
 private String name;  
  
 // Getter and Setter for name  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
}  
  
public class Geeks {  
 public static void main(String[] args) {  
 Programmer p = new Programmer();  
 p.setName("Geek");  
 System.*out*.println("Name=> " + p.getName());  
 }  
}

Advantages - **data protection, security, and flexibilit**y, reusability, data hiding(Users don’t see the implementation of the class. The user only knows that we are passing the values to a setter method and variables are getting initialized with that value.)

### obj.name vs obj.getName() in Java 🡪

| **Feature** | **obj.name** | **obj.getName()** |
| --- | --- | --- |
| **Type** | Direct access to a field (variable) | Access via a method (getter) |
| **Encapsulation** | Violates it (if the field is public) | Supports it (field is private) |
| **Control** | No control – anyone can read/write | Full control – you can add logic |
| **Flexibility** | Rigid – can’t change without affecting callers | Flexible – you can change logic later |

**✅ Why getName() is better:**

1. **Encapsulation**: Keeps internal data hidden.
2. **Control**: You can add validation, formatting, logging later.

public String getName() {  
 return name.toUpperCase(); // or log access  
}

1. **Read-only**: You can expose getName() without a setName() to make the value read-only.
2. **Compatibility**: Changing internal logic later won't break the code that uses getName().

## 3. Inheritance

It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Advantage : reusability, method overriding, abstraction

Note : constructor could not be inherited.

### Example:

// Java program to illustrate the  
// concept of inheritance  
  
// base class  
class Bicycle {  
 // the Bicycle class has two fields  
 public int gear;  
 public int speed;  
  
 // the Bicycle class has one constructor  
 public Bicycle(int gear, int speed)  
 {  
 this.gear = gear;  
 this.speed = speed;  
 }  
  
 // the Bicycle class has three methods  
 public void applyBrake(int decrement)  
 {  
 speed -= decrement;  
 }  
  
 public void speedUp(int increment)  
 {  
 speed += increment;  
 }  
  
 // toString() method to print info of Bicycle  
 public String toString()  
 {  
 return ("No of gears are " + gear + "\n"  
 + "speed of bicycle is " + speed);  
 }  
}  
  
// derived class  
class MountainBike extends Bicycle {  
  
 // the MountainBike subclass adds one more field  
 public int seatHeight;  
  
 // the MountainBike subclass has one constructor  
 public MountainBike(int gear, int speed,  
 int startHeight)  
 {  
 // invoking base-class(Bicycle) constructor  
 super(gear, speed);  
 seatHeight = startHeight;  
 }  
  
 // the MountainBike subclass adds one more method  
 public void setHeight(int newValue)  
 {  
 seatHeight = newValue;  
 }  
  
 // overriding toString() method  
 // of Bicycle to print more info  
 @Override public String toString()  
 {  
 return (super.toString() + "\nseat height is "  
 + seatHeight);  
 }  
}  
  
// driver class  
public class Test {  
 public static void main(String args[])  
 {  
  
 MountainBike mb = new MountainBike(3, 100, 25);  
 System.*out*.println(mb.toString());  
 }  
}

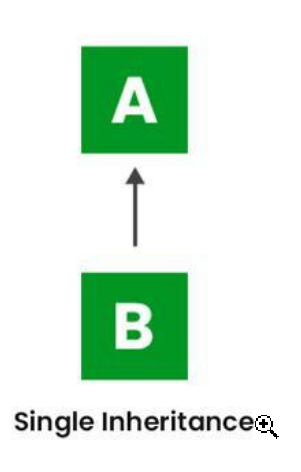
**Note:**During inheritance only the object of the subclass is created, not the superclass. For more, refer to [Java Object Creation of Inherited Class](https://www.geeksforgeeks.org/gfact-52-java-object-creation-of-inherited-classes).

### Types of Inheritance in Java

Below are the different types of inheritance which are supported by Java.

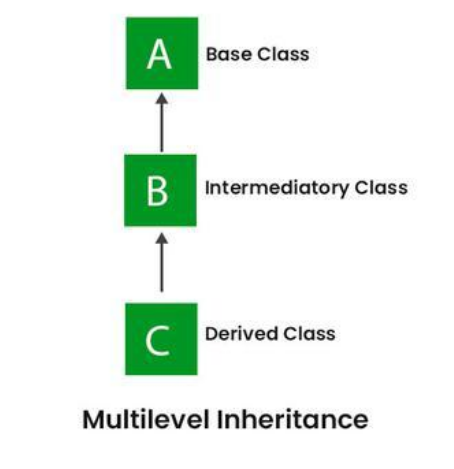
#### Single Inheritance 🡪

a sub-class is derived from only one super class.



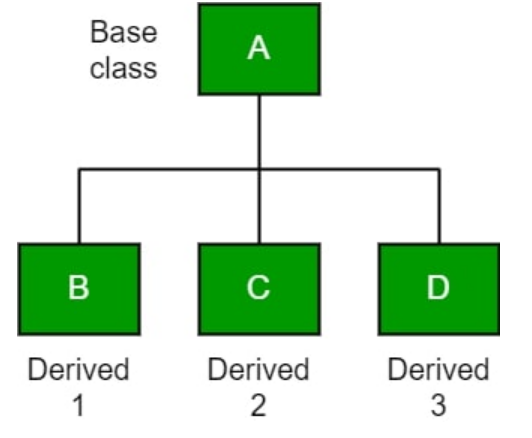
#### Multilevel Inheritance🡪

a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.



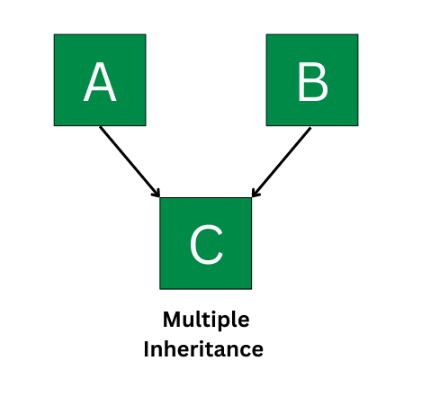
#### Hierarchical Inheritance🡪

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.



#### Multiple Inheritance🡪

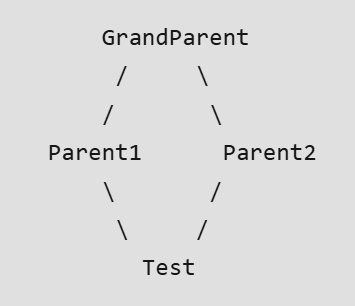
In [Multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance), one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces.



###### Example :

// Interface 1 that defines coding behavior  
interface Coder {  
 void writeCode();  
}  
  
// Interface 2 that defines testing behavior  
interface Tester {  
 void testCode();  
}  
  
// Class implementing both interfaces   
class DevOpsEngineer implements Coder, Tester {  
 @Override  
 public void writeCode() {  
 System.*out*.println("DevOps Engineer writes automation scripts.");  
 }  
  
 @Override  
 public void testCode() {  
 System.*out*.println("DevOps Engineer tests deployment pipelines.");  
 }  
  
 // Additional method specific to DevOpsEngineer  
 public void deploy() {  
 System.*out*.println("DevOps Engineer deploys code to cloud.");  
 }  
}  
  
// Driver class  
public class Main {  
 public static void main(String[] args) {  
 DevOpsEngineer devOps = new DevOpsEngineer();  
  
 devOps.writeCode();  
 devOps.testCode();  
 devOps.deploy();  
 }  
}

###### Diamond problem 🡪 can be resolved using default method of interface.



1.

// Interface 1  
interface PI1 {  
 default void show(){  
 System.*out*.println("Default PI1");  
 }  
}  
  
// Interface 2  
interface PI2 {  
 default void show(){  
 System.*out*.println("Default PI2");  
 }  
}  
  
  
class TestClass implements PI1, PI2 {  
  
 // Overriding default show method  
 @Override  
 public void show(){  
 PI1.super.show();  
 PI2.super.show();  
 }  
  
 // Declared new Method  
 public void showOfPI1() {  
 PI1.super.show();  
 }  
  
 // Declared new Method  
 public void showOfPI2() {  
 PI2.super.show();  
 }  
  
 // main Method  
 public static void main(String args[]) {  
  
 // Instance of Class  
 TestClass d = new TestClass();  
  
 // Using show Method  
 d.show();  
  
 // Executing the Methods  
 System.*out*.println("Now Executing showOfPI1()" +  
 " showOfPI2()");  
 d.showOfPI1();  
 d.showOfPI2();  
 }  
}

2.

// Java program to demonstrate How Diamond Problem  
// Is Handled in case of Default Methods  
  
// Interface 1  
interface GPI {  
  
 // Default method  
 default void show()  
 {  
  
 // Print statement  
 System.*out*.println("Default GPI");  
 }  
}  
  
// Interface 2  
// Extending the above interface  
interface PI1 extends GPI {  
}  
  
// Interface 3  
// Extending the above interface  
interface PI2 extends GPI {  
}  
  
// Main class  
// Implementation class code  
class TestClass implements PI1, PI2 {  
  
 // Main driver method  
 public static void main(String args[])  
 {  
  
 // Creating object of this class  
 // in main() method  
 TestClass d = new TestClass();  
  
 // Now calling the function defined in interface 1  
 // from whom Interface 2and 3 are deriving  
 d.show();  
 }  
}

#### Hybrid Inheritance🡪

Combination of above inheritance.

## 4. Polymorphism

Poly means many and morphism means form. [polymorphism.](https://www.geeksforgeeks.org/polymorphism-in-java/) Means a same method behave deifferently in different situations.

Real-world example –

A person – father/son/husband

Water – ice/liquid/gas

Advantage – code reusability, dynamic behavior, abstraction, flexibility

#### Example:

// Base class Person  
class Person {  
  
 // Method that displays the   
 // role of a person  
 void role() {  
 System.*out*.println("I am a person.");  
 }  
}  
  
// Derived class Father that   
// overrides the role method  
class Father extends Person {  
  
 // Overridden method to show   
 // the role of a father  
 @Override  
 void role() {  
 System.*out*.println("I am a father.");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
  
 // Creating a reference of type Person   
 // but initializing it with Father class object  
 Person p = new Father();  
  
 // Calling the role method. It calls the   
 // overridden version in Father class  
 p.role();  
 }  
}

### Types of Polymorphism

#### Compile Time Polymorphism/static/ early binding

Compile-time polymorphism is a polymorphism that is resolved during the compilation process. Can be achieved by method overloading.

private, final and static methods and variables uses static binding and bonded by compiler.

#### Runtime Polymorphism/ Dynamic Method Dispatch

It is a process in which a function call to the overridden method is resolved at Runtime. Can be achieved by inheritance and method overriding.

* **Runtime Polymorphism with Object Members**

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

class A {  
 void m1() {  
 System.*out*.println("Inside A's m1 method");  
 }  
}  
  
class B extends A {  
 // overriding m1()  
 void m1() {  
 System.*out*.println("Inside B's m1 method");  
 }  
}  
  
class C extends A {  
 // overriding m1()  
 void m1() {  
 System.*out*.println("Inside C's m1 method");  
 }  
}  
  
class Dispatch {  
 public static void main(String args[]) {  
 A a = new A();  
 B b = new B();  
 C c = new C();  
  
 // obtain a reference of type A  
 A ref;  
 ref = a;  
  
 // calling A's version of m1()  
 ref.m1();  
  
 // now ref refers to a B object  
 ref = b;  
  
 // calling B's version of m1()  
 ref.m1();  
  
 // now ref refers to a C object  
 ref = c;  
  
 // calling C's version of m1()  
 ref.m1();  
 }  
}  
/\*output --  
Inside A's m1 method  
Inside B's m1 method  
Inside C's m1 method  
\*/

* **Runtime Polymorphism with Data Members**

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

class A {  
 int x = 10;  
}  
  
class B extends A {  
 int x = 20;  
}  
  
public class Test  
{  
 public static void main(String args[])  
 {  
 A a = new B(); // object of type B  
 // Data member of class A will be accessed  
 System.*out*.println(a.x);//10  
 }  
}

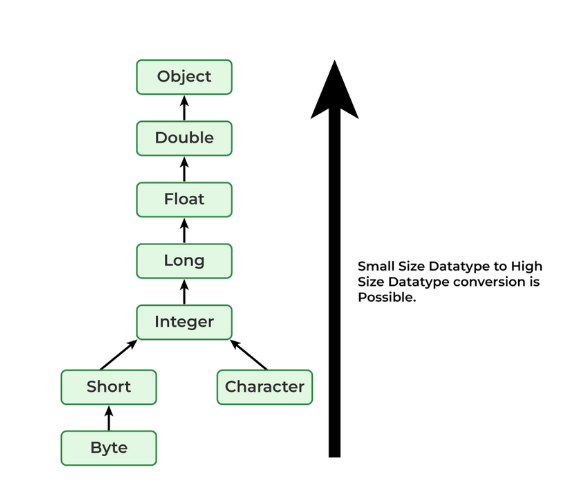
# Method Overloading

**Method Overloading** allows us to define multiple methods with the same name but different parameters within a class. This difference can be in the number of parameters, the types of parameters, or the order of those parameters.

#### What if the Exact Prototype Does Not Match With Arguments?

 Priority-wise, the compiler takes these steps:

* [Type Conversion](https://www.geeksforgeeks.org/type-conversion-java-examples/) but to a higher type(in terms of range) in the same family.
* Type conversion to the next higher family (suppose if there is no long data type available for an int data type, then it will search for the float data type).



Example:

class Demo {  
  
 public void show(int x) {  
 System.*out*.println("In int" + x);  
 }  
 public void show(String s) {  
 System.*out*.println("In String" + s);  
 }  
 public void show(byte b) {  
 System.*out*.println("In byte" + b);  
 }  
}  
  
class UseDemo {  
 public static void main(String[] args) {  
 byte a = 25;  
 Demo obj = new Demo();  
  
 // it will go to byte argument  
 obj.show(a);  
  
 // String  
 obj.show("hello");  
  
 // Int  
 obj.show(250);  
  
 // Since char is not available, so the datatype  
 // higher than char in terms of range is int.  
 obj.show('A');  
  
 // String  
 obj.show("A");  
  
 // since float datatype is not available and so it's higher  
 // datatype, so at this step their will be an error.  
 obj.show(7.5);  
 }  
}

**🧠 Java Overloading Resolution Rules (simplified):**

1. Exact match (fixed-arity) >
2. Widening (e.g., int → long) >
3. Boxing (e.g., int → Integer) >
4. Varargs (least priority)

# Method Overriding

**Overriding in Java**occurs when a**subclass or child class implements a method that is already defined in the superclass or base class**.The subclass method must match the parent class method’s **name, parameters, and return type.**

Static methods cannot be overridden.

#### Rules for Java Method Overriding

**1. Access Modifiers in overriding**

A subclass can make an overridden method more accessible, e.g., upgrade protected to public, but not less e.g., downgrade public to private. Trying to hide a method this way causes compiler errors.

**2. Final/static/private Methods Cannot Be Overriden**

**3. Method Must Have the Same Return Type (or subtype)**

From Java 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child’s return type should be a sub-type of the parent’s return type. This phenomenon is known as the [covariant return type](https://www.geeksforgeeks.org/covariant-return-types-java/).

class SuperClass {  
 public Object method() {  
 System.*out*.println(  
 "This is the method in SuperClass");  
 return new Object();  
 }  
}  
  
class SubClass extends SuperClass {  
 public String method() {  
 System.*out*.println(  
 "This is the method in SubClass");  
 return "Hello, World!";  
 // having the Covariant return type  
 }  
}  
  
public class Geeks {  
 public static void main(String[] args) {  
 SuperClass obj1 = new SuperClass();  
 obj1.method();  
  
 SubClass obj2 = new SubClass();  
 obj2.method();  
 }  
}

**4. Invoking Parent’s Overridden Method Using super**

We can call the parent class method in the overriding method using the [super keyword](https://www.geeksforgeeks.org/super-keyword/).

Note: Use super.methodName() to call the parent’s version.

#### Overriding and Constructor

We cannot override the constructor as the parent and child class can never have a constructor with the same name (the constructor name must always be the same as the Class name).

#### Exception-Handling in Overriding

Below are two rules to note when overriding methods related to exception handling.

**Rule 1:** If the **super-class** overridden method**does not**throw an exception, the subclass overriding method can only throw the [unchecked exception](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/), throwing a checked exception will lead to a compile-time error.

**Example:**Below is an example of a Java program**when the parent class method does not declare the exception.**

class Parent {  
 void m1() { System.*out*.println("From parent m1()"); }  
  
 void m2() { System.*out*.println("From parent m2()"); }  
}  
  
class Child extends Parent {  
 @Override  
 // no issue while throwing unchecked exception  
 void m1() throws ArithmeticException  
 {  
 System.*out*.println("From child m1()");  
 }  
  
 @Override  
 // compile-time error  
 // issue while throwing checked exception  
 void m2() throws Exception  
 {  
 System.*out*.println("From child m2");  
 }  
}

**Rule 2:** If the superclass overridden method does throw an exception, the subclass overriding method can only throw the same, subclass exception. Throwing parent exceptions in the [Exception hierarchy](https://www.geeksforgeeks.org/exceptions-in-java/) will lead to compile time error. Also, there is no issue if the subclass overridden method does not throw any exception.

# Association

relationship between two different objects. It is a relationship where all objects have their own life cycle and there is no owner. Let's take an example of teacher and student. Multiple students and associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own life cycle. Both can create and delete independently.

#### Aggregation

when objects can survive individually means end of one object will not end other object. Aggregation is a specialized form of association where all objects have their own lifecycle but there is ownership and child object cannot belongs to another parent object. let's take an example of department and teacher. A single teacher cannot belongs to multiple departments, but if we delete the department teacher object will not destroy. We can think about has a relationship.

#### Composition

end of one object will end another object. Composition is again a specialized form of aggregation and we can call this as a direct relationship(Has-a). it is a strong type of aggregation. The child does not have their life cycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between house and rooms. House can contain multiple rooms there is no dependent life of room and any room cannot belongs to two different house if we delete the house room will automatically delete.

# Interview Questions 🡪

1. **Can we achieve method overloading by changing the return type?**

➡️ Overloading does not depend on the return type of the method, two methods cannot be overloaded by just changing the return type.

1. **What is ambiguity in method overloading?**

➡️ When the compiler cannot decide which overloaded method to call due to similar signatures.

void show(int a, float b) {}  
void show(float a, int b) {}  
// show(10, 20) causes ambiguity

1. **Difference between Overriding vs Overloading**

| **Features** | **Method Overriding** | **Method Overloading** |
| --- | --- | --- |
| **Definition** | Method overriding is about the same signature in subclass. | Method overloading is about same method name, different parameters. |
| **Polymorphism** | It is also known as Runtime polymorphism | It is also known as Compiletime polymorphism |
| **Inheritance** | Requires inheritance. | Can be in the same class or subclass |
| **Return Type** | Return type must be same | Return type can be different |
| **Exceptions** | Must follow overriding rules. | No restrictions. |

1. **What happens if we overload a method with primitive and wrapper types?**

➡️ Java uses a process called **method resolution** during **compile time**, and it follows a specific order:

**Exact match → Widening → Autoboxing → Varargs**

public class Test {  
  
 void show(int x) {  
 System.*out*.println("Primitive int: " + x);  
 }  
  
 void show(Integer x) {  
 System.*out*.println("Wrapper Integer: " + x);  
 }  
  
 public static void main(String[] args) {  
 Test t = new Test();  
 t.show(10); // Line A  
 t.show(new Integer(20)); // Line B  
 }  
} Primitive int: 10

Wrapper Integer: 20

**❗What if we only have show(Integer x)?**

// void show(int x) is removed  
t.show(10); // Compiler autoboxes int to Integer

✅ This will still work due to **autoboxing** (Java automatically wraps int to Integer).

**❗What if both methods are present and you pass null?**

t.show(null); // Compile-time error: reference is ambiguous

Because null is valid for both Integer and int, but:

* int can't accept null
* Integer can

But since both are overloaded versions, the compiler gets confused and throws an **ambiguous method call error**.

1. **Can a constructor be overridden? Why?**

➡️ No

1. **Explain the diamond problem. How does Java avoid it?**
2. **What is the difference between IS-A and HAS-A relationships in OOP?**

➡️ **IS-A**: Inheritance (Dog IS-A Animal).

**HAS-A**: Composition (Car HAS-A Engine).

1. **What is the order of initialization in Java when a class is extended?**
   1. Static block of superclass
   2. Static block of subclass
   3. Instance block of superclass
   4. Constructor of superclass
   5. Instance block of subclass
   6. Constructor of subclass
2. **What will happen if a subclass constructor calls an overridden method from its superclass constructor?**

➡️ *Can lead to unexpected behavior because the subclass constructor hasn't completed yet.*

1. **What if a subclass has the same method name as the parent class but different parameter types – is it overriding or overloading?**

➡️ *It’s overloading. Overriding requires the same signature.*

1. **Can you override a method that throws a checked exception with a method that doesn’t throw any exception?**

➡️ *Yes, that’s allowed. You can reduce or eliminate exceptions.*

1. **Can a constructor be overridden? What about overloaded?**

* **Overridden?** ❌ No – constructors are not inherited.
* **Overloaded?** ✅ Yes – same class, different parameter lists.

1. **What happens when you overload a method with a varargs parameter and a fixed parameter?**

void print(int... x)

void print(int x)

➡️ Calling print(10) invokes the fixed parameter method – exact match has higher priority.

1. **What happens if you try to override a final method?**

➡️ Compile-time error: Final methods cannot be overridden.

1. **How does method resolution happen with inheritance, overloading, and autoboxing combined?**

class A {

void show(Number n) { System.out.println(\"A:Number\"); }

}

class B extends A {

void show(Integer i) { System.out.println(\"B:Integer\"); }

}

new B().show(10); // Output?

➡️ Output: B:Integer, because int 10 is autoboxed to Integer, and the most specific method in B is chosen.

1. **What if both superclass and subclass define a field with the same name? Which one is accessed?**

➡️ Fields are **not polymorphic**. Access depends on the reference type, not the object type.

class A { int x = 10; }

class B extends A { int x = 20; }

A obj = new B();

System.out.println(obj.x); // Output: 10

1. **What happens if you override a method and make it more restrictive (e.g., from public to protected)?**

➡️ Compile-time error: Cannot reduce the visibility of the inherited method.

1. **Can abstract classes have constructors? Why?**

➡️ Yes. Abstract classes can have constructors to initialize fields and perform setup logic. They are called when subclass objects are created.

1. **How does Java handle multiple interfaces with the same default method?**

➡️ You must **override** the method in your class and resolve the conflict.

interface A { default void show() { System.out.println(\"A\"); } }

interface B { default void show() { System.out.println(\"B\"); } }

class C implements A, B {

public void show() {

A.super.show(); // resolve explicitly

}}

1. **Why is Java not 100% Object-Oriented?**

➡️ Because it supports **primitive types** (int, char, boolean, etc.) that are **not objects**.

**Static methods** (like Math.sqrt()) can be called without creating an object.

# Code snippet🡪

class A {  
 void show() {  
 System.*out*.println("A");  
 }  
}  
  
class B1 extends A {  
 void show() {  
 System.*out*.println("B1");  
 }  
}  
  
class D extends B1 {  
 void show() {  
 System.*out*.println("D");  
 }  
}  
  
public class Main1 {  
 public static void main(String[] args) {  
 A obj = new D();  
 obj.show();  
 }  
}

#### Answer

“D”

class Test {  
 void print(Number n) {  
 System.*out*.println("Number");  
 }  
  
 void print(Integer n) {  
 System.*out*.println("Integer");  
 }  
}  
  
public class Main2 {  
 public static void main(String[] args) {  
 Test t = new Test();  
 Number n = new Integer(5);  
 t.print(n);  
 }  
}

#### Answer

“Number” as it is overloading.

class Parent {  
 static void show() {  
 System.*out*.println("Parent");  
 }  
}  
  
class Child extends Parent {  
 static void show() {  
 System.*out*.println("Child");  
 }  
}  
  
public class Main3 {  
 public static void main(String[] args) {  
 Parent p = new Child();  
 p.*show*();  
 }  
}

#### Answer

“Parent.

class A {}  
class B1 extends A {}  
class D extends B1 {}  
class B2 extends A {}  
class G extends B2 {}  
  
public class Main4 {  
 public static void check(A obj) {  
 if (obj instanceof B1) {  
 System.*out*.println("Allowed branch: " + obj.getClass().getSimpleName());  
 } else {  
 System.*out*.println("Not allowed");  
 }  
 }  
  
 public static void main(String[] args) {  
 *check*(new D());  
 *check*(new G());  
 }  
}

#### Answer

“Allowed branch: D

Not allowed”

class TypeTest {  
 void show(int x) {  
 System.*out*.println("int");  
 }  
  
 void show(double x) {  
 System.*out*.println("double");  
 }  
  
 public static void main(String[] args) {  
 TypeTest t = new TypeTest();  
 t.show(10.5f); // What gets printed?  
 }  
}

#### Answer

“double”

class Base {  
 Base() {  
 System.*out*.println("Base Constructor");  
 }  
}  
  
class Derived extends Base {  
 Derived() {  
 System.*out*.println("Derived Constructor");  
 }  
}  
  
public class Main6 {  
 public static void main(String[] args) {  
 new Derived(); // What’s the output?  
 }  
}

#### Answer

“Base Constructor Derived Constructor”

class Parent {  
 void show() throws Exception {  
 System.*out*.println("Parent");  
 }  
}  
  
class Child extends Parent {  
 @Override  
 void show() { // Compile error or not?  
 System.*out*.println("Child");  
 }  
}

#### Answer

“No compile error”

class Animal {  
 private void speak() {  
 System.*out*.println("Animal");  
 }  
}  
  
class Dog extends Animal {  
 private void speak() {  
 System.*out*.println("Dog");  
 }  
}  
  
public class Main8 {  
 public static void main(String[] args) {  
 Animal a = new Dog();  
 // a.speak(); // Uncomment this — what happens?  
 }  
}

#### Answer

“compile error”

* public class Main9 {  
   public static void printList(List<?> list) {  
   for (Object obj : list) {  
   System.*out*.print(obj + " ");  
   }  
   }  
    
   public static void main(String[] args) {  
   List<String> list = List.of("A", "B", "C");  
   *printList*(list); // What’s the output?  
   }  
  }

#### Answer

“A B C”

### 10.

class Parent {  
 Parent() {  
 System.*out*.println("Parent Constructor");  
 show();  
 }  
  
 void show() {  
 System.*out*.println("Parent show");  
 }  
}  
  
class Child extends Parent {  
 int x = 10;  
  
 Child() {  
 System.*out*.println("Child Constructor");  
 }  
  
 @Override  
 void show() {  
 System.*out*.println("Child show: " + x);  
 }  
}  
  
public class Test {  
 public static void main(String[] args) {  
 Parent p = new Child();  
 }  
}

#### Answer

Parent Constructor

Child show: 0

Child Constructor

**If a method is overridden**, and you're calling it from a constructor (even the superclass constructor), Java will **resolve it at runtime** based on the actual object **being created**, **not** the class where the constructor lives.

### 11.

class OverloadNullTest {  
 void test(Object o) {  
 System.*out*.println("Object version");  
 }  
  
 void test(String s) {  
 System.*out*.println("String version");  
 }  
  
 public static void main(String[] args) {  
 OverloadNullTest obj = new OverloadNullTest();  
 obj.test(null);  
 }  
}

#### Answer

String version

* null is a valid value for **any reference type** (Object, String, etc.)
* But since String is **more specific than Object**, the compiler chooses test(String s)

If we add

void test(Integer i) {  
 System.*out*.println("Integer version");  
}

**❌ Compilation error**

❗ Now Java doesn’t know whether to pick test(String) or test(Integer) — both are equally specific.  
Hence: **“reference to test is ambiguous”** — **compile-time error**

Also string and integer classes are peer classes both classes inherit Object class.

### 12.

class VarargsTest {  
 void show(int a, int b) {  
 System.*out*.println("Regular method");  
 }  
  
 void show(int... nums) {  
 System.*out*.println("Varargs method");  
 }  
  
 public static void main(String[] args) {  
 VarargsTest vt = new VarargsTest();  
 vt.show(10, 20); // Line A  
 vt.show(10); // Line B  
 }  
}

#### Answer

Regular Method, Varargs Method

**Line A: vt.show(10, 20);**

* Matches **exactly** with show(int a, int b) → this is an **exact method match**.
* Java always prefers a fixed-arity method over a varargs one **if an exact match exists**.

**🔹 Line B: vt.show(10);**

* No matching method with exactly one int → varargs method show(int... nums) is used.
* Varargs can accept 1 or more parameters.

🧠 Rule to Remember:

**Java always prefers the most specific fixed-parameter method over varargs**, if one exists.

### 13.

class VarargsTest {  
 void show(int a) {  
 System.*out*.println("One int");  
 }  
  
 void show(int... nums) {  
 System.*out*.println("Varargs");  
 }  
  
 public static void main(String[] args) {  
 VarargsTest vt = new VarargsTest();  
 vt.show(10); // <-- which one?  
 }  
}

#### Answer

One int

### 14.

class VarargsNullTest {  
 void print(String s) {  
 System.*out*.println("String");  
 }  
  
 void print(String... s) {  
 System.*out*.println("Varargs");  
 }  
  
 public static void main(String[] args) {  
 VarargsNullTest obj = new VarargsNullTest();  
 obj.print(null); // Line A  
 obj.print((String[]) null); // Line B  
 }  
}

#### Answer

String

Varargs

Line 1 -  Both methods (print(String) and print(String...)) can accept null.

 But print(String) is **more specific** than print(String...).

 So the compiler picks print(String).

Line 2 -  Here you're **explicitly passing a String[]** (even though it's null).

 So Java directly matches the **String... version**, because String... is just **syntactic sugar** for String[].

 The compiler picks print(String...) **even though it's null**.

### 15.

class Parent {  
 Parent() {  
 System.*out*.println("Parent no-arg constructor");  
 }  
  
 Parent(int x) {  
 System.*out*.println("Parent one-arg constructor: " + x);  
 }  
}  
  
class Child extends Parent {  
 Child() {  
 this(10);  
 System.*out*.println("Child no-arg constructor");  
 }  
  
 Child(int x) {  
 super(x);  
 System.*out*.println("Child one-arg constructor: " + x);  
 }  
}  
  
public class Test {  
 public static void main(String[] args) {  
 Child c = new Child();  
 }  
}

#### Answer

Parent one-arg constructor: 10

Child one-arg constructor: 10

Child no-arg constructor

**❌ What does NOT get called?**

* **Parent() (no-arg constructor)** is **never called**, because Child(int x) explicitly uses super(x) to call the one-arg constructor.

### 16.

class Animal {  
 void makeSound() {  
 System.*out*.println("Some generic animal sound");  
 }  
}  
  
class Dog extends Animal {  
 void bark() {  
 System.*out*.println("Dog barks");  
 }  
}  
  
public class Test {  
 public static void main(String[] args) {  
 Animal a = new Animal(); // Not a Dog  
 Dog d = (Dog) a; // Line X  
 d.bark(); // Line Y  
 }  
}

#### Answer

Runtime Error

Dog d = (Dog ) a; ...is **syntactically valid** — the compiler thinks you know what you’re doing.

**❌ What happens at runtime?**

At runtime:

* a is **not** actually a Dog. It's just a plain Animal.
* You’re trying to **downcast** a Parent type object that was **never a Child**.

🧠 Rule to Remember:

✅ **Allowed downcasts** (compile-time):

Parent p = new Child();

Child c = (Child) p; // Safe

❌ **Dangerous downcasts** (runtime):

Parent p = new Parent();

Child c = (Child) p; // ClassCastException ❌