**1.1. What are the four major pillars of OOP?**

The major pillars on which OOP relies are [Encapsulation](https://howtodoinjava.com/java/oops/encapsulation-in-java-and-its-relation-with-abstraction/), [Inheritance](https://howtodoinjava.com/java/oops/java-inheritance/), [Polymorphism](https://howtodoinjava.com/java/oops/what-is-polymorphism-in-java/), and Abstraction.

* **Encapsulation** provides security to our application; it consists of *private* variable declarations (data hiding) and accessor methods (getters/setters) to access the variables.
* **Inheritance** reduces code redundancy by providing code reusability.
* **Polymorphism** provides flexibility by allowing to the declaration of methods with the same name for different purposes.
* **Abstraction** provides security by hiding the internal implementation of a class and only exposing the details necessary in the context.

**1.2. Is Java a pure object-oriented programming language?**

No, Java is not a pure object-oriented programming language because:

* Java does not provide multiple inheritance support for classes.
* Java does not provide support for operator overloading.
* We use primitive variables like *byte*, *short*, *char*, *int*, *float*, *long*, and *double* these are not objects.

Because of the above reasons, we can’t say Java is 100% pure object-oriented.

**1.3. What are the tightly encapsulated and loosely encapsulated classes?**

A tightly encapsulated class does not allow public access to any of its data members and only allows accessors and mutator methods to modify them. In general, we can say a class is tightly encapsulated if and only if every variable inside the class is *private* scoped.

Note that an immutable class is always tightly encapsulated, but not every tightly encapsulation class is immutable.

**public** **class** Student {

**private** **int** rollNo;

**private** String studentName;

**private** String collageName;

*//Getters and Setters*

}

If at least one variable inside the class is not declared *private*, then that class is known as a loosely encapsulated class.

**public** **class** Student {

**private** **int** rollNo;

**private** String studentName;

String collageName;

*//Getters and Setters*

}

**1.4. What are Coupling and Cohesion?**

**Coupling** is the degree of dependency between the components.

**Tight coupling** generally happens when a class knows too much about the internal working of the dependent classes. Many times, we cannot change one component without changing the other.

In the following example, we use the *ReportService* class that prepares the data and writes a PDF report using the *PdfReportWriter* instance. Suppose we want to add the capability to write excel reports as well. Can we add the capability without changing the *ReportService*? No.

**public** **class** ReportService {

**public** Object[][] createReportData() {...}

**private** writeReport(Object[][] data) {

**new** PdfReportWriter().writeData(data);

}

}

In a **loosely coupled** system, components depend on each other to the least extent practically possible. This **allows us to change individual components without affecting other parts of the software**.

In the following code, we solved the previous problem using low coupling. Now the *ReportService* class only expects an *IReportWriter* implementation, and that class would be responsible for generating the report. We can inject as many supported report types, without touching any of the other classes.

**public** **interface** IReportWriter {

**public** **void** writeData(Object[][] data);

}

**public** **class** PdfReportWriter **implements** IReportWriter {

**public** **void** writeData(Object[][] data) {...}

}

**public** **class** ExcelReportWriter **implements** IReportWriter {

**public** **void** writeData(Object[][] data) {...}

}

**public** **class** ReportWriter **implements** IReportWriter {

IReportWriter writer;

**public** Object[][] createReportData() {...}

**private** writeReport(Object[][] data) {

writer.writeData(data);

}

}

**Cohesion** is creating components with clear well-defined responsibilities. If we maintain only one component for all functionalities, then it’s known as **low cohesion** and has several disadvantages.

**public** **class** AppManager {

**public** **void** login(){...}

**public** **void** register(){...}

**public** **void** forgotPassword(){...}

**public** **void** addToCart(){...}

**public** **void** checkout(){...}

}

Maintaining high cohesion by creating different components for different functionality is always a good programming practice.

**public** **class** LoginManager {

**public** **void** login(){...}

}

**public** **class** RegistrationManager {

**public** **void** register(){...}

}

...

...

The above program follows low cohesion as every functionality is declared inside only one component. We can achieve low cohesion by following the [single responsibility principle](https://howtodoinjava.com/design-patterns/single-responsibility-principle/).

Note: It is highly recommended to follow loose coupling and high cohesion always.

**1.5. What are autoboxing and unboxing? How does it work on arguments to overloaded methods?**

Since Java 1.5, the compiler automatically converts the primitive data types into corresponding [wrapper class types](https://howtodoinjava.com/java/basics/java-wrapper-classes/), known as **autoboxing**. The reverse operation of this (wrapper class types to primitive automatically by the compiler) is known as **unboxing**.

**int** i = 10;

Integer j = i; *//Autoboxing*

**int** k = j; *//Unboxing*

In case of overloaded methods if the compiler is not able to find a method with exact parameters then the compiler will take **method resolution on the following priority**:

* first, the compiler will apply **widening** e.g. convert *int* to *long*.
* if no match is found, then the compiler will apply **autoboxing/unboxing** to find a matching method signature e.g. convert *int* to *Integer*.
* if no match is found, then the compiler will search for the method signature with a **variable-length argument** e.g. find method with *int…* argument type.
* finally, if no match is found, the compiler will give the error.

**int** i = 10;

doSomething(i);

**public** **static** **void** doSomething(**long** i {...} *//It will be matched first, if found.*

**public** **static** **void** doSomething(Integer i) {...} *//It will be matched, if no match after widening.*

**public** **static** **void** doSomething(**int**... i) {...} *//It will be matched, if no match after autoboxing.*

Please note that the **compiler will not do both, widening and autoboxing, together to find the match**.

**public** **class** Test {

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

**int** i = 10;

m1(i); *//Test.java:4: error: incompatible types: int cannot be converted to Long*

}

**public** **static** **void** m1(Long i){...}

}

**1.6. What are early binding and late binding (method overriding)?**

**Early binding** is also known as method overloading. In method overloading, we can declare two methods with the same name but with different parameters (i.e different method signatures).

In method overloading, the method resolution (to execute which method) is done by the compiler based on the reference type of the object. Hence this concept is also known as **compile-time polymorphism**.

**public** **class** Parent {

**public** **void** doSomething(**int** x) {

System.out.println("Parent Class int Method"); *//This method is invoked*

}

}

**public** **class** Child **extends** Parent {

**public** **void** doSomething(**float** x){

System.out.println("Child Class float Method");

}

}

Parent p = **new** Child();

p.doSomething(10);

**Late binding** is known as the method overriding. In method overriding, we can re-declare the *Parent* class’s method in the *Child* class with the same name and with exact same parameters (i.e same method signature).

In method overriding, **method resolution** (to execute which method) is done by the JVM based on the type of the object. Hence this concept is also known as **run-time polymorphism**.

**public** **class** Parent {

**public** **void** doSomething(**int** x) {

System.out.println("Parent Class int Method");

}

}

**public** **class** Child **extends** Parent {

**public** **void** doSomething(**int** x){

System.out.println("Child Class int Method");

}

}

Parent p = **new** Child();

p.doSomething(10); *//Child Class int Method*

Parent p = **new** Parent();

p.doSomething(10); *//Parent Class int Method*

See Also: [Method Overloading and Overriding](https://howtodoinjava.com/java/oops/method-overloading-overriding/)

**1.7. What are widening/upcasting and narrowing/downcasting?**

The **widening** is referred to automatically assigning the lower data type variable to the higher data type variable by the compiler. Since there is no data loss, the compiler does it automatically (hence it is also called **implicit** **typecasting**). Also, the promotion to the higher data type is referred to as **upcasting**.

**byte** b = 10;

**int** x = b; *//Upcasting a byte to int*

**The narrowing** is referred to manually and explicitly typecast the higher data type variable to the lower data type variable. Since data loss is involved, the compiler does NOT do it automatically (hence it is also called **explicit** **typecasting**). Also, the demotion to the lower data type is referred to as **downcasting**.

Remember that while downcasting most significant bits will be lost.

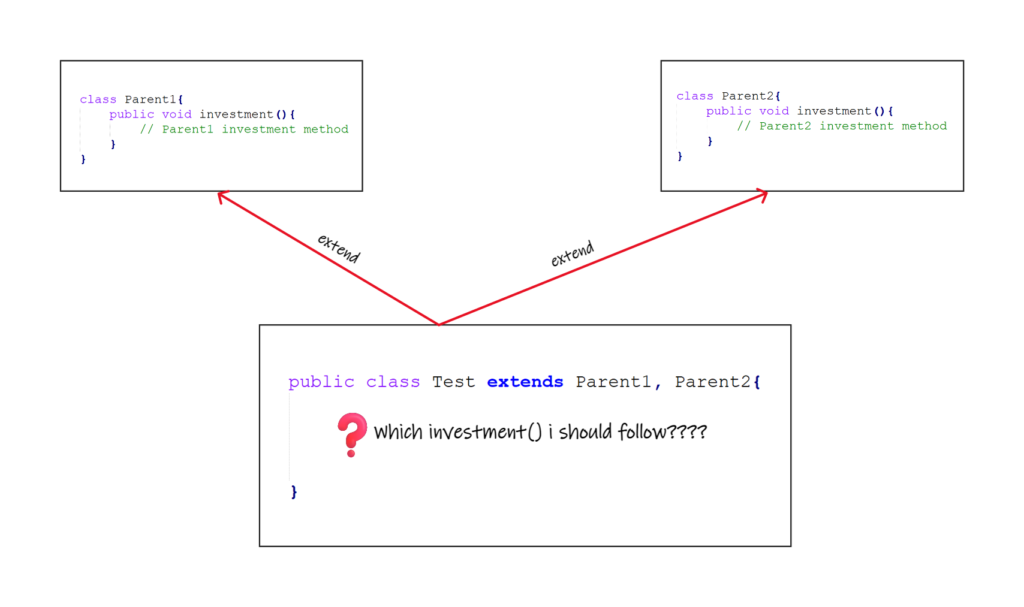
**double** d = 50.52354;

**int** y = (**int**) d;

**1.8. Why does Java not provide Multiple Inheritance?**

Having more than one parent class is known as [multiple inheritance](https://howtodoinjava.com/java/oops/multiple-inheritance-in-java/). Java won’t provide support for multiple inheritances as there may be a chance of raising **ambiguity problems**.

In the case of multiple inheritances, if two parent classes define the same methods, then extending compiler gets confused about which method to inherit. This problem is also known as the **diamond problem** or **diamond ambiguity**.



**1.9. What is RIWO (Read Indirectly Write Only) state?**

While **class loading**, the static control flow is executed which is responsible for identifying the *static* members, assigning values to *static* variables, executing the *static* blocks, and then executing the *static* main() method.

While the **creation of each object** **instance control flow** is executed which is responsible for identifying the instance members, assigning values to the instance variables, executing the instance blocks, and then executing the constructor.

The first and foremost step in both static control and instance control flow is to identify the *static* and instance members respectively when they are executed. **So, while JVM identifies the *static* or instance variables and initializes them with a default value, that variable stay in *RIWO* (Read Indirectly Write Only) state which says we can’t perform a direct read operation on that variable. We can only write some value to that variable.**

Attempting to make a read in such cases will give “**illegal forward reference**” error.

**public** **class** Test {

**static**{

*// System.out.println(x); CE: illegal forward reference*

x = 100; *// we can write*

}

**static** **int** x = 200;

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

System.out.println(x);

}

}









**2. Questions on the Scope of Variables and Methods**

**2.1. What are actual parameters and formal parameters?**

* The parameters defined at the time of method declaration as part of the method signature are known as **formal parameters**.
* The parameters that actually hold some value and are passed to the method at the time of the method invocation are known as **actual parameters**.

**public** **int** sum(**int** x, **int** y){...} *// Formal Parameters*

Integer a=10, b=20;

sum(a, b); *// Actual Parameters*

**2.2. Can we declare formal parameters as *final*?**

Yes. The formal parameters simply act as the local variable for methods in which they are declared and **final is the only modifier applicable to the local variables.**Note that we can’t re-assign value to any formal parameter declared as final.

**public** **static** **void** printMyName(**final** String name){

*// name = "Lokesh!"; //Compiler Error:final parameter name may not be assigned*

}

**2.3. Can we write *super()* and *this()* in one constructor?**

In Java, as a rule, *super()* or *this()* must be the first statement inside a constructor. That’s why it is never possible to keep *both super() and this()* in a single constructor. We can use either *super()* or *this()* inside one constructor but not both.

**public** **class** App{

App(){

**super**();

*//this(); //Compiler Error: Test.java:3: error: call to this must be first statement in constructor*

}

}

**2.4. What is the prototype of the default constructor generated by the compiler?**

If the programmer explicitly doesn’t define any constructor inside a class then compile is responsible provide the default constructor. and the prototype of the compiler-generated default constructor is:

**public** MyClass(){

**super**();

}

If the corresponding class is *public* then only the compiler will generate a *public* constructor otherwise it will generate *<default>* constructor only.

**2.5. Can we declare a *final* method inside an *abstract* class?**

Yes, we can create **non-abstract final methods** inside an *abstract* class. While extending this *abstract* class, we can’t override the *final* methods, but we can only use them inside the abstract class itself.

**abstract** **class** Car {

**public** **final** **void** createEngine() {}

**public** **abstract** **void** buildCar();

}

**class** MarutiCar **extends** Car {

**public** **void** buildCar() {

createEngine();

}

}

**2.6. Can we declare the abstract method inside a final class?**

No, we can’t declare an *abstract* method inside a *final* class because a final class cannot be extended, and having an *abstract* method makes it incomplete.

In Java, if there is even one *abstract* method inside the class then we must mark the class as *abstract*, and the***final-abstract* combination is illegal.**

**final** **class** MarutiCar { *// CE: MarutiCar is not abstract and does not override abstract method mileage()*

**public** **abstract** **int** mileage();

}

**2.7. What are the modifiers applicable for top-level classes?**

Modifiers that are applicable for top-level classes are:

* *public*
* *<default>*
* *abstract*
* *final*
* *strictfp*

**2.8. What are the modifiers applicable for local variables?**

The only applicable modifier for the local variable is ***final***.

**final** **double** tax;

**2.9. Can we override the *Parent* class’s *non-final* method as the *final* method in the *Child* class?**

Yes, we **can override the parent class non-final method as the *final* method** as it helps **to restrict the method overriding for the next-level child classes**.

But, we can’t override the parent class final method as non-final in the child class as **overriding a final method is impossible**.

**public** **class** Parent {

**public** **void** method(){...}

}

**class** ChildLevelOne **extends** Parent {

**public** **final** **void** method(){...} *//Allowed*

}

**class** ChildLevelTwo **extends** ChildLevelOne {

**public** **void** method(){...} *// CE: method() in ChildLevelTwo cannot override method() in ChildLevelOne*

}

**3. Questions on Method Overloading and Overriding**

**3.1. What is the difference between method hiding and method overriding?**

The**method overriding is applicable for *non-static* or instance methods only**. In the case of static methods,**redefining the static methods in the child class is known as method hiding**.

Generally, all rules of method overriding are applicable for method hiding. Except in method overriding, method resolution is done on runtime, whereas in method hiding, method resolution is always done by the compiler on compile time.

**class** Parent {

**public** **void** instanceMethod(){ ... }

**public** **static** **void** staticMethod(){ ... }

}

**public** **class** Child **extends** Parent {

**public** **void** instanceMethod(){ ... } *//Method overriding*

**public** **static** **void** staticMethod(){ ... } *//Method hiding*

}

**3.2. Can we overload the *main()* method?**

Yes, overloading the *main()* method is always possible but JVM will invoke only the *main()* method with *String[]* argument. For other overloaded methods, we have to invoke them explicitly just like a normal method call.

**public** **class** MainMethodOverloading {

**public** **static** **void** main(String[] args){ *//JVM will invoke this when program is run*

System.out.println("String[] argument method");

}

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

System.out.println("int[] argument method");

}

}

**3.3. Can we override the member variables?**

No, member variables cannot be overridden. We can only hide them in the child class. The resolution of variables of any class is always done by the compiler based on the reference type, not the runtime object type.

**class** Parent {

**int** x = 100;

**static** **int** y = 200;

}

**public** **class** Child **extends** Parent {

**int** x = 300;

**static** **int** y = 400;

}

Parent p = **new** Child();

System.out.println(p.x); *// 100*

System.out.println(p.y); *// 200*

Child c = **new** Child();

System.out.println(c.x); *// 300*

System.out.println(c.x); *// 400*

**3.4. Can we declare a method with the same name as the class name?**

Yes, it’s perfectly fine to have a method with the same name as the class name, but it is never recommended.

**public** **class** Application {

**public** **void** Application(){...} *// Allowed but not recommended*

}

**3.5. Can we override the parent class non-abstract method as abstract in the child class?**

Yes, we can override the parent class *non-abstract* method as an *abstract* method in the child class if it is not satisfied with the method implementation in the Parent class. In such cases, the next level child class is responsible for overriding that abstract method and providing a suitable implementation.

**public** **class** Parent {

**public** **void** method(){...}

}

**class** ChildLevelOne **extends** Parent {

**public** **abstract** **void** method();

}

**class** ChildLevelTwo **extends** ChildLevelOne {

**public** **void** method(){...}

}

**3.6. Can we override the parent class public method as protected in the child class?**

No, as per the method overriding rules, while overriding any method **we can not reduce the scope of the method**.

**public** **class** Parent {

**public** **void** method(){...}

}

**class** Child **extends** Parent {

**protected** **void** method(){...} *// CE: attempting to assign weaker access privileges; was public*

}

But, **we can increase the scope** of the overridden method.

**public** **class** Parent {

**void** method(){...}

}

**class** Child **extends** Parent {

**public** **void** method(){...} *// Allowed*

}

**4. Difference between**

**4.1. Difference between *IS-A* and *HAS-A* relationships?**

Through *inheritance*, we achieve the **IS-A** relationship that helps achieve code reusability and reduce code redundancy. *IS-A* relationship is implemented by using the “*extends*” keyword in Java.

**class** Television {...}

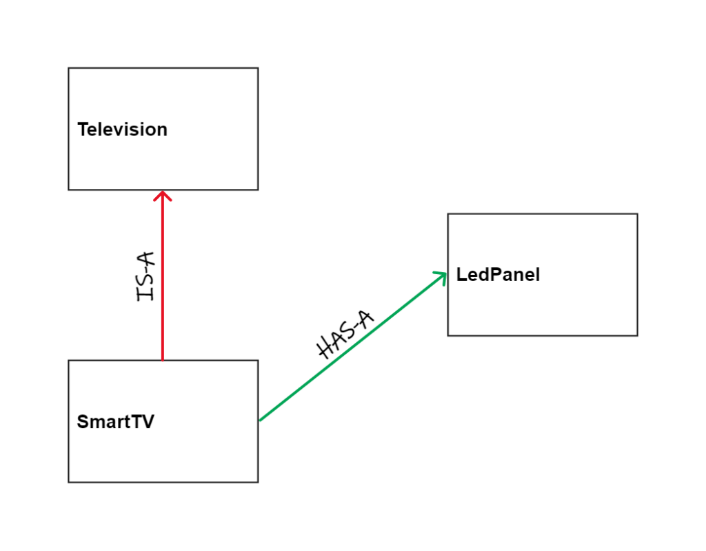
**class** SmartTV **extends** Television {...} *//SmartTV 'IS-A' type of Television.*

Through *association*, we achieve the ***HAS-A*** relationship that helps in reusing the functionality already defined by some other classes without extending those classes.

**class** SmartTV {

LedPanel led; *//SmartTV 'HAS-A' a LedPanel*

}



**4.2. Difference between “*==*” and “*equals()*“?**

The *“==”* is an equal operator meant for reference comparison. if two object references point to the same instance then only the *“==”* operator will return *true* otherwise it will return *false*.

String s1 = **new** String("Java");

String s2 = **new** String("Java");

String s3 = s1;

s1 == s2; *//false*

s1 == s3; *//true*

The *equals()* method is in the *Object* class. The default implementation returns true if and only if x and y refer to the same object (x == y has the value true). If we want to check the equality of the objects using custom logic, we need to override the *equals()* method in the custom class and provide the logic in it.

For example, *String* class overrides this method and checks the content of strings.

String s1 = **new** String("Java");

String s2 = **new** String("Java");

s1.equals(s2); *//true*

**4.3. Difference between *static* and instance variables?**

The ***static* variables are created at the time of class loading**. There is only one copy of the static variable is created for the entire class and shared by all instances of that class. Hence we can access the *static* variables of a class even without creating any instance.

**static** **int** x = 100; *// one copy shared among all objects*

System.out.println(ClassName.x); *// access using classname*

The **instance variables are created at runtime using the *‘new’* keyword** and belong to the instances of the class. We can access them using the instance reference.

**int** x = 100; *// individual copy for each object*

**5. Coding-Related Interview Questions**

**5.1. Is this valid method overloading?**

**public** **class** Test{

**public** **void** m1(**int** x){

System.out.println("void m1 method");

}

**public** **int** m1(**int** x){

System.out.println("int m1 method");

}

}

No, we can’t overload a method just by changing the method’s return type. To overload a method, the method name must be the same, and the parameter to that method must be different. Hence we’ll get compile time error in the above code.

Test.java:7: error: method m1(**int**) is already defined in **class** Test

**5.2. Is this valid method overriding?**

**class** Parent{

**private** **void** m1(){

System.out.println("Parent class m1");

}

}

**public** **class** Test **extends** Parent{

**protected** **void** m1(){

System.out.println("Child Class m1");

}

}

No, the overriding is not done on *private* methods as they are not available to the child class through inheritance. We will not get any compile time error in the above code but it’s not method overriding.

**class** Parent{

**public** **void** m1() **throws** InterruptedException{

System.out.println("Parent class m1");

}

}

**public** **class** Test **extends** Parent{

**public** **void** m1() **throws** Exception{

System.out.println("Child class m1");

}

}

No, the overriding method in the child class can’t throw the parent type of exception thrown by the parent class’s method. It must throw either the same exception or a child exception type.

Test.java:7: error: m1() in Test cannot override m1() in Parent

**5.3 Can we implement the following interfaces simultaneously in a class?**

**interface** A{

**public** **void** m1();

}

**interface** B{

**public** **int** m1();

}

No, we can’t implement these interfaces in one class because both interfaces have methods with the same signature, and compiler will not able to resolve which method to call when invoked from an implementing class.

**5.4. Why does the following code not generates *NullPointerException* at run time?**

**class** Name{

**static** **void** m1(){

System.out.println("Name class m1");

}

}

**public** **class** Test{

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

Name n = **null**;

n.m1();

}

}

JVM does not need an instance created to invoke the *static* members of a class. Static members are invoked using the class types. In the following example, *n.m1()* will be treated as *Name.m1()*, and thus no *NullPointerException*.