What is oop? Discuss the various oops principles in java?

OOP stands for Object-Oriented Programming. It is a programming paradigm that is based on the concept of “objects”, which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). Java is an object-oriented programming language that follows the OOPs concepts. The four main principles of OOPs are:

1. **Abstraction**: Abstraction is the process of hiding the implementation details and showing only the functionality to the user. In Java, we can achieve abstraction using abstract classes and interfaces.
2. **Encapsulation**: Encapsulation is the process of binding data members (variables) and member functions (methods) together into a single unit called a class. It helps in data hiding and prevents unauthorized access to data.
3. **Inheritance**: Inheritance is the process by which one class acquires the properties (methods and fields) of another class. It helps in code reusability.
4. **Polymorphism**: Polymorphism means “many forms”. It is the ability of an object to take on many forms (behaviors). In Java, we can achieve polymorphism through method overloading and method overriding.

**Difference Between Method Overloading and Method Overriding in Java**

**[M](https://www.geeksforgeeks.org/difference-between-method-overloading-and-method-overriding-in-java/?ref=gcse" \l "article-meta-div)**

[MKS075](https://www.geeksforgeeks.org/difference-between-method-overloading-and-method-overriding-in-java/?ref=gcse" \l "article-meta-div)

 [Read](javascript:void(0))

 [Discuss](javascript:void(0))

The differences between Method Overloading and Method Overriding in Java are as follows:

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is a compile-time polymorphism. | Method overriding is a run-time polymorphism. |
| Method overloading helps to increase the readability of the program. | Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |
| It occurs within the class. | It is performed in two classes with inheritance relationships. |
| Method overloading may or may not require inheritance. | Method overriding always needs inheritance. |
| In method overloading, methods must have the same name and different signatures. | In method overriding, methods must have the same name and same signature. |
| In method overloading, the return type can or can not be the same, but we just have to change the parameter. | In method overriding, the return type must be the same or co-variant. |
| Static binding is being used for overloaded methods. | Dynamic binding is being used for overriding methods. |
| Poor Performance due to compile time polymorphism. | It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime. |
| Private and final methods can be overloaded. | Private and final methods can’t be overridden. |
| The argument list should be different while doing method overloading. | The argument list should be the same in method overriding. |

**Method Overloading in Java**

[Method Overloading](https://www.geeksforgeeks.org/different-ways-method-overloading-java/)is a **Compile time polymorphism**. In method overloading, more than one method shares the same method name with a different signature in the class. In method overloading, the return type can or can not be the same, but we have to change the parameter because, in java, we can not achieve method overloading by changing only the return type of the method.

**Example of Method Overloading:**

* Java

|  |
| --- |
| // Java Program to Implement  // Method Overloading  import java.io.\*;    class MethodOverloadingEx {        static int add(int a, int b) { return a + b; }        static int add(int a, int b, int c)      {          return a + b + c;      }        // Main Function      public static void main(String args[])      {          System.out.println("add() with 2 parameters");          // Calling function with 2 parameters          System.out.println(add(4, 6));            System.out.println("add() with 3 parameters");          // Calling function with 3 Parameters          System.out.println(add(4, 6, 7));      }  } |

**Output**

add() with 2 parameters

10

add() with 3 parameters

17

**Method Overriding in Java**

[Method Overriding](https://www.geeksforgeeks.org/overriding-in-java/) is a **Run time polymorphism**. In method overriding, the derived class provides the specific implementation of the method that is already provided by the base class or parent class. In method overriding, the return type must be the same or co-variant (return type may vary in the same direction as the derived class).

**Example of Method Overriding:**

* Java

|  |
| --- |
| // Java Program to implement  // Method Overriding  import java.io.\*;    // Base Class  class Animal {      void eat()      {          System.out.println("eat() method of base class");          System.out.println("eating.");      }  }    // Inherited Class  class Dog extends Animal {      void eat()      {          System.out.println("eat() method of derived class");          System.out.println("Dog is eating.");      }  }    class MethodOverridingEx {      // Main Function      public static void main(String args[])      {          Dog d1 = new Dog();          Animal a1 = new Animal();            d1.eat();          a1.eat();            Animal animal = new Dog();          // eat() method of animal class is overridden by          // base class eat()          animal.eat();      }  } |

**Output**

eat() method of derived class

Dog is eating.

eat() method of base class

eating.

eat() method of derived class

Dog is eating.

**Output explanation:** Here, we can see that a method eat() has overridden in the derived class name **Dog** that is already provided by the base class name **Animal**. When we create the instance of class Dog and call the eat() method, we see that only derived class eat() method run instead of base class method eat(), and When we create the instance of class Animal and call the eat() method, we see that only base class eat() method run instead of derived class method eat().

2a. Write a program in java to display student id and name using default and parameterized constructors?

class Student {

int id;

String name;

Student() {

id = 0;

name = "";

}

Student(int id, String name) {

this.id = id;

this.name = name;

}

void display() {

System.out.println("Student ID: " + id);

System.out.println("Student Name: " + name);

}

public static void main(String args[]) {

Student s1 = new Student();

s1.display();

Student s2 = new Student(1, "John");

s2.display();

}

}

Copy

This program defines a Student class with two data members id and name. It has two constructors, one is the default constructor which initializes the data members to 0 and empty string respectively. The other is a parameterized constructor which takes two arguments id and name and initializes the data members with these values. The display() method is used to display the values of id and name. In the main method, we create two objects of the Student class, one using the default constructor and the other using the parameterized constructor. We then call the display() method on both objects to display their values.

2b. what is inheritance and explain the terms parent class and child class ? State the advantages of inheritance in java?

Inheritance is a mechanism in Java that allows one class to inherit the properties and methods of another class. The class that is inherited from is called the parent class or superclass, and the class that inherits is called the child class or subclass. Inheritance is an important part of object-oriented programming because it allows for code reuse and makes it easier to create new classes based on existing ones.

The child class can access all the public and protected members of the parent class, but not its private members. The child class can also add new members and methods to itself. There are several types of inheritance in Java, including single inheritance, multilevel inheritance, hierarchical inheritance, and hybrid inheritance.

The advantages of inheritance in Java include minimizing duplicate code, making code more flexible to change, allowing for method overriding, and enabling data hiding. However, there are also some disadvantages to inheritance, such as decreased execution speed and a lack of independence between the base and inherited classes.

I hope this helps! Let me know if you have any other questions.

3a.discussthe difference between checked and unchecked exceptions in java?

In Java, there are two types of exceptions: checked exceptions and unchecked exceptions. The main difference between them is that the compiler checks checked exceptions at compile time, while it does not check unchecked exceptions.

Checked exceptions are the exceptions that are checked by the compiler at compile time. They are the exceptions that are expected to be handled by the programmer. If a method throws a checked exception, then the calling method must either handle the exception or declare it in its throws clause. Examples of checked exceptions include IOException, SQLException, and ClassNotFoundException.

Unchecked exceptions, on the other hand, are not checked by the compiler at compile time. They are the exceptions that occur at runtime and are not expected to be handled by the programmer. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and ClassCastException.

The advantage of using checked exceptions is that they force the programmer to handle the exception or declare it in its throws clause. This makes the code more robust and less prone to errors. The disadvantage is that it can make the code more complex and harder to read.

The advantage of using unchecked exceptions is that they make the code simpler and easier to read. The disadvantage is that they can lead to unexpected errors at runtime if they are not handled properly.

I hope this helps! Let me know if you have any other questions.

3b. What is Deadlock in java? What are the precautions to avoid a deadlock state in java?

Deadlock in Java is a situation where two or more threads are blocked forever, waiting for each other to release the resources they need to proceed. Deadlocks can occur in a multithreaded environment when two or more threads are waiting for the same resource, and each thread is holding a resource that the other thread needs. This creates a circular dependency, and neither thread can proceed.

To avoid deadlocks in Java, you can take the following precautions:

* Avoid unnecessary locks: Use locks only for those members on which it is required. Unnecessary use of locks leads to a deadlock situation.
* Avoid nested locks: Avoid giving a lock on another resource if you already hold one.
* Use synchronized blocks: Use synchronized blocks instead of synchronized methods to avoid deadlocks.
* Use thread-safe collections: Use thread-safe collections instead of non-thread-safe collections to avoid deadlocks.
* Use atomic operations: Use atomic operations instead of synchronized blocks to avoid deadlocks.
* Use the Thread.join() method: Use the Thread.join() method to guarantee that one thread finishes before starting another.

4a. with regards to eventhandling, describe the different layout managers in java.

Java has several layout managers that are used to arrange components in a particular manner. The Java LayoutManagers facilitate us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are several classes that represent the layout managers, including:

* java.awt.BorderLayout: This layout manager is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region: public static final int NORTH, public static final int SOUTH, public static final int EAST, public static final int WEST, and public static final int CENTER.
* java.awt.FlowLayout: This layout manager is used to arrange the components in a line, one after another. It is useful when we want to arrange the components horizontally or vertically.
* java.awt.GridLayout: This layout manager is used to arrange the components in a grid of cells. It is useful when we want to arrange the components in a tabular form.
* java.awt.CardLayout: This layout manager is used to arrange the components in a stack of cards, where only one card is visible at a time.
* java.awt.GridBagLayout: This layout manager is used to arrange the components in a grid of cells, where each cell can have different sizes. It is useful when we want to arrange the components in a complex form[1](https://www.javatpoint.com/java-layout-manager).

Event handling in Java is the procedure that controls an event and performs appropriate action if it occurs. The code or set of instructions used to implement it is known as the Event handler. It consists of two major components: The event source and The event listener. The java.awt.event package provides many event classes and Listener interfaces for event handling. Some of the most commonly used classes include ActionEvent, ActionListener, MouseEvent, MouseMotionListener, MouseWheelEvent, MouseWheelListener, KeyEvent, KeyListener, ItemEvent, ItemListener, TextEvent, TextListener, AdjustmentEvent, AdjustmentListener, WindowEvent, WindowListener, ComponentEvent, ComponentListener, ContainerEvent and ContainerListener. Java has several layout managers that are used to arrange components in a particular manner. The Java LayoutManagers facilitate us to control the positioning and size of the components in GUI forms. There are several classes that represent the layout managers, including:

* java.awt.BorderLayout: This layout manager is used to arrange the components in five regions: north, south, east, west, and center.
* java.awt.FlowLayout: This layout manager is used to arrange the components in a line, one after another.
* java.awt.GridLayout: This layout manager is used to arrange the components in a grid of cells.
* java.awt.CardLayout: This layout manager is used to arrange the components in a stack of cards, where only one card is visible at a time.
* java.awt.GridBagLayout: This layout manager is used to arrange the components in a grid of cells, where each cell can have different sizes. It is useful when we want to arrange the components in a complex form[1](https://www.javatpoint.com/event-handling-in-java). How else can I help?

4b.With the help of suitable examples explain the different kind of operators in java.

Sure, I can help you with that. Java has many types of operators, which are used to perform different operations on variables and values. Here are some examples of the different types of operators in Java:

* **Arithmetic Operators**: These operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, and modulus. For example, +, -, \*, /, and % are arithmetic operators.
* **Relational Operators**: These operators are used to compare two values and return a boolean value. For example, <, >, <=, >=, ==, and != are relational operators.
* **Logical Operators**: These operators are used to perform logical operations on boolean values. For example, &&, ||, and ! are logical operators.
* **Bitwise Operators**: These operators are used to perform bitwise operations on integer values. For example, &, |, ^, ~, <<, and >> are bitwise operators.
* **Assignment Operators**: These operators are used to assign values to variables. For example, =, +=, -=, /=, %=, &=, |=, and ^= are assignment operators.
* **Unary Operators**: These operators require only one operand. They can be either prefix or postfix. For example, ++ and -- are unary operators.
* **Ternary Operator**: This operator is used to assign one of two values to a variable depending on a condition. It is also known as the conditional operator. For example, (condition) ? value1 : value2 is a ternary operator.

I hope this helps! Let me know if you have any other questions.

5a. With the help of suitable examples discuss in detail the java AWT buttons and AWT labels.

Java AWT (Abstract Window Toolkit) provides a set of classes for creating graphical user interfaces (GUIs) in Java. AWT buttons and labels are two of the most commonly used components in Java GUI programming.

**AWT Buttons**: AWT buttons are used to create clickable buttons in a Java GUI. The Button class is used to create a labeled button that has platform-independent implementation. Here is an example of how to create a button in Java:

import java.awt.\*;

public class ButtonExample {

public static void main(String[] args) {

Frame f = new Frame("Button Example");

Button b = new Button("Click Here");

b.setBounds(50, 100, 80, 30);

f.add(b);

f.setSize(400, 400);

f.setLayout(null);

f.setVisible(true);

}

}

Copy

This code creates a button labeled “Click Here” and adds it to a frame. The setBounds() method sets the position and size of the button on the frame.

**AWT Labels**: AWT labels are used to display text in a Java GUI. The Label class is used to create a component for placing text in a container. It is used to display a single line of read-only text. Here is an example of how to create a label in Java:

import java.awt.\*;

public class LabelExample {

public static void main(String[] args) {

Frame f = new Frame("Label Example");

Label l = new Label("Hello World");

l.setBounds(50, 100, 100, 30);

f.add(l);

f.setSize(400, 400);

f.setLayout(null);

f.setVisible(true);

}

}

Copy

This code creates a label with the text “Hello World” and adds it to a frame. The setBounds() method sets the position and size of the label on the frame.

I hope this helps! Let me know if you have any other questions.

5b. Describe in details the String Constructor in java.

Java String Constructors allows us to create a string object from different type of arguments. We can create string object using double quotes. For example, String s1 = "Java String";. However, sometimes we want to create a string object from different sources such as byte array, character array, StringBuffer, and StringBuilder. The String class constructors are provided to create a string object from these arguments.

Here is a list of some of the Java String Constructors:

* String(): creates an empty string. It’s mostly useless because String is immutable.
* String(String original): creates a string object from another string. Since String is immutable, it’s of no use.
* String(byte[] bytes): constructs a new string from the byte array using system default encoding.
* String(byte[] bytes, String charsetName): uses the specified character encoding. If the encoding is not supported, UnsupportedEncodingException is thrown.
* String(byte[] bytes, Charset charset): a better way to specify an encoding for constructing the string object.
* String(byte[] bytes, int offset, int length): The offset specifies the index of the first byte to decode. The length specifies the number of bytes to decode. This constructor throws IndexOutOfBoundsException if offset is negative, length is negative, or offset is greater than bytes.length - length.
* String(byte[] bytes, int offset, int length, Charset charset): It’s similar to the above constructor except that we have to specify the encoding to use.
* String(byte[] bytes, int offset, int length, String charsetName): Similar to above except that the character set encoding name is passed as a string. This will throw UnsupportedEncodingException if the encoding is not supported.
* String(char[] value): creates the string object from the character array.
* String(char[] value, int offset, int count): The offset specifies the index of the first character. The length specifies the number of characters to use. This constructor throws IndexOutOfBoundsException if offset is negative, length is negative, or offset is greater than value.length - length.
* String(int[] codePoints, int offset, int count): creates a string from the input Unicode code points array. It throws IllegalArgumentException if any of the code points are invalid. This constructor throws IndexOutOfBoundsException if the offset is negative, the length is negative, or offset is greater than codePoints.length - length.
* String(StringBuffer buffer): creates a new string from the contents of the string buffer. This constructor internally calls StringBuffer toString () method.
* String(StringBuilder buffer): creates a new string from the contents of the string builder.

I hope this helps! Let me know if you have any other questions.

PART C

1 . with the help of suitable examples discuss the various access modifiers in java.

In Java, access modifiers are used to control the visibility and accessibility of classes, methods, and variables. There are four types of access modifiers in Java:

* **Private**: The private access modifier is the most restrictive. It is used to restrict access to a class’s members within the same class. For example:

public class MyClass {

private int x;

private void myMethod() {

// code here

}

}

Copy

In this example, the x variable and myMethod() method can only be accessed within the MyClass class.

* **Default**: The default access modifier is used when no access modifier is specified. It is used to restrict access to a class’s members within the same package. For example:

class MyClass {

int x;

void myMethod() {

// code here

}

}

Copy

In this example, the x variable and myMethod() method can only be accessed within the same package.

* **Protected**: The protected access modifier allows access to a class’s members within the same package and subclasses in other packages. For example:

public class MyClass {

protected int x;

protected void myMethod() {

// code here

}

}

Copy

In this example, the x variable and myMethod() method can be accessed within the same package and subclasses in other packages.

* **Public**: The public access modifier allows access to a class’s members from anywhere. For example:

public class MyClass {

public int x;

public void myMethod() {

// code here

}

}

Copy

In this example, the x variable and myMethod() method can be accessed from anywhere.

I hope this helps! Let me know if you have any other questions.

# Access Modifiers in Java

 [Read](javascript:void(0))

 [Discuss](javascript:void(0))

 [Courses](javascript:void(0))

 [Practice](javascript:void(0))

 [Video](javascript:void(0))

in Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element. Let us learn about Java Access Modifiers, their types, and the uses of access modifiers in this article.

## Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

### ****1. Default Access Modifier****

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

**Program 1:**

* Java

|  |
| --- |
| // Java program to illustrate default modifier  package p1;    // Class Geek is having Default access modifier  class Geek  {      void display()      {          System.out.println("Hello World!");      }  } |

**Program 2:**

* Java

|  |
| --- |
| // Java program to illustrate error while  // using class from different package with  // default modifier  package p2;  import p1.\*;    // This class is having default access modifier  class GeekNew  {      public static void main(String args[])      {          // Accessing class Geek from package p1          Geek obj = new Geek();            obj.display();      }  } |

**Output:**

Compile time error

### ****2. Private Access Modifier****

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

* Any other **class of**the **same package will not be able to access** these members.
* Top-level classes or interfaces can not be declared as private because
  + private means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

* Java

|  |
| --- |
| // Java program to illustrate error while  // using class from different package with  // private modifier  package p1;    class A  {  private void display()      {          System.out.println("GeeksforGeeks");      }  }    class B  {  public static void main(String args[])      {          A obj = new A();          // Trying to access private method          // of another class          obj.display();      }  } |

**Output:**

error: display() has private access in A

obj.display();

### ****3. Protected Access Modifier****

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

**Program 1:**

* Java

|  |
| --- |
| // Java program to illustrate  // protected modifier  package p1;    // Class A  public class A  {  protected void display()      {          System.out.println("GeeksforGeeks");      }  } |

**Program 2:**

* Java

|  |
| --- |
| // Java program to illustrate  // protected modifier  package p2;  import p1.\*; // importing all classes in package p1    // Class B is subclass of A  class B extends A  {  public static void main(String args[])  {      B obj = new B();      obj.display();  }    } |

**Output:**

GeeksforGeeks

### ****Public Access modifier****

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

**Program 1:**

* Java

|  |
| --- |
| // Java program to illustrate  // public modifier  package p1;  public class A  {  public void display()      {          System.out.println("GeeksforGeeks");      }  } |

**Program 2:**

* Java

|  |
| --- |
| package p2;  import p1.\*;  class B {      public static void main(String args[])      {          A obj = new A();          obj.display();      }  } |

**Output:**

GeeksforGeeks

**Important Points:**

* If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
* Avoid public fields except for constants.

## Algorithm to use access modifier in Java

#### Here’s a basic algorithm for using access modifiers in Java:

* ***Define a class:****Create a class that represents the object you want to manage.*
* ***Define instance variables:****Within the class, define instance variables that represent the data you want to manage.*
* *Specify an access modifier: For each instance variable, specify an access modifier that determines the visibility of the variable. The three main access modifiers in Java are private, protected, and public.*
* ***Use private for variables that should only be accessible within the class:****If you want to prevent access to a variable from outside the class, use the private access modifier. This is the most restrictive access modifier and provides the greatest level of encapsulation.*
* ***Use protected for variables that should be accessible within the class and its subclasses****: If you want to allow access to a variable from within the class and its subclasses, use the protected access modifier. This is less restrictive than private and provides some level of inheritance.*
* ***Use public for variables that should be accessible from anywhere****: If you want to allow access to a variable from anywhere, use the public access modifier. This is the least restrictive access modifier and provides the least amount of encapsulation.*
* ***Use accessor and mutator methods to manage access to the variables:****In order to access and modify the variables, use accessor (getter) and mutator (setter) methods, even if the variables have a public access modifier. This provides a level of abstraction and makes your code more maintainable and testable.*
* ***In this exampl****e, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.*

## FAQs in Access Modifiers

### 1. What are access modifiers in Java?

Access modifiers in Java are the keywords that are used for controlling the use of the methods, constructors, fields, and methods in a class.

### ****2. What is void in Java?****

Void in Java is used to specify no return value with the method.

### ****3. What are the 12 modifiers in Java?****

12 Modifiers in Java are public, private, protected, default, final, synchronized, abstract, native, strictfp, transient, and volatile.

**2. Differentiate between multilevel inheritance and Hierarical Inheritance in java. Support your answer with suitable code examples.**

In Java, inheritance is a mechanism by which one class is allowed to inherit the features (fields and methods) of another class. There are different types of inheritance in Java, including multilevel inheritance and hierarchical inheritance.

**Multilevel Inheritance**: In multilevel inheritance, a class is derived from a class which is also derived from another class. This creates a parent-child relationship between the classes. For example:

class A {

void methodA() {

System.out.println("This is method A");

}

}

class B extends A {

void methodB() {

System.out.println("This is method B");

}

}

class C extends B {

void methodC() {

System.out.println("This is method C");

}

}

public class Main {

public static void main(String[] args) {

C c = new C();

c.methodA();

c.methodB();

c.methodC();

}

}

Copy

In this example, class C inherits from class B, which in turn inherits from class A. This creates a multilevel inheritance hierarchy. The methodA(), methodB(), and methodC() methods can be called on an object of class C.

**Hierarchical Inheritance**: In hierarchical inheritance, many subclasses inherit from one single class. This creates a parent-child relationship between the classes. For example:

class A {

void methodA() {

System.out.println("This is method A");

}

}

class B extends A {

void methodB() {

System.out.println("This is method B");

}

}

class C extends A {

void methodC() {

System.out.println("This is method C");

}

}

public class Main {

public static void main(String[] args) {

B b = new B();

C c = new C();

b.methodA();

b.methodB();

c.methodA();

c.methodC();

}

}

Copy

In this example, both classes B and C inherit from class A. This creates a hierarchical inheritance hierarchy. The methodA() method can be called on objects of both classes B and C.

I hope this helps! Let me know if you have any other questions.