**Name: ANURAG KUMAR JAVA ASSIGNMENT Emp id: INN912  
  
  
OOPS CONCEPT  
  
JAVA INHERITANCE-**Java inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (called a subclass or derived class) to inherit properties and behaviors (fields and methods) from an existing class (called a superclass or base class).  
  
Inheritance enables a new class to reuse the code from an existing class. The subclass inherits all non-private members (fields and methods) from the superclass.  
  
The **extends** keyword is used to define a subclass.  
  
class Superclass {  
 // Fields and methods  
 }   
class Subclass **extends** Superclass {   
// Additional fields and methods   
}

**Types of Inheritance-**  
  
 **Single Inheritance**: A class can inherit from only one superclass.  
 **Multilevel Inheritance**: A class can inherit from a superclass, and then another class can inherit from that subclass, forming a chain.  
 **Hierarchical Inheritance**: Multiple classes can inherit from the same superclass.  
  
**Benefits of Inheritance are-**  
  
**Code Reusability**: Allows sharing of common code among classes, reducing redundancy.  
**Polymorphism**: A subclass can override methods of the superclass, allowing for dynamic method dispatch.  
**Extensibility**: Makes it easier to extend existing functionality by creating new subclasses.

**POLYMORPHISM-**

Polymorphism is a key concept in object-oriented programming (OOP) which means ‘many forms’ that allows objects to be treated as instances of their parent class rather than their actual class. This enables one interface to be used for a general class of actions, making code more flexible and reusable. In Java, polymorphism primarily occurs in two forms: compile-time (method overloading) and runtime (method overriding).  
  
**METHOD OVERLOADING-**

Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type or number of parameters). The method to be called is determined at compile time based on the method signature.  
  
class Printer {

void print(String s) {

System.out.println(s);

}

void print(int i) {

System.out.println(i);

}

}

public class Main {

public static void main(String[] args) {

Printer printer = new Printer();

printer.print("Hello, World!"); // Calls print(String s)

printer.print(100); // Calls print(int i)

}

}

**METHOD OVERRIDING-**

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method call is resolved at runtime, allowing for dynamic method invocation.class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal;

myAnimal = new Dog();

myAnimal.sound(); // Output: Dog barks

myAnimal = new Cat();

myAnimal.sound(); // Output: Cat meows

}

}

**ABSTRACTION**Abstraction in Java is a key concept in object-oriented programming (OOP) that focuses on hiding the complex implementation details and showing only the essential features of an object. This is achieved using abstract classes and interfaces, allowing programmers to work with high-level concepts and simplify complex systems.  
  
An abstract class in Java cannot be instantiated and is meant to be sub classed. It can have both abstract methods (methods without a body) and concrete methods (methods with an implementation).

**Abstract Method**: A method declared without an implementation. Subclasses must provide implementations for these methods.

**Concrete Method**: A method with an implementation in the abstract class, which can be used by subclasses.  
  
abstract class Animal {

// Abstract method

abstract void makeSound();

// Concrete method

void breathe() {

System.out.println("This animal breathes.");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.makeSound(); // Output: The dog barks.

myDog.breathe(); // Output: This animal breathes.

}

}

### ENCAPSULATION- Encapsulation is a fundamental concept in object-oriented programming (OOP) that involves bundling the data (fields) and methods (functions) that operate on the data into a single unit, known as a class. It also restricts direct access to some of the object's components, which can help prevent unintended interference and misuse of the data. In Java, encapsulation is implemented using access modifiers to control the visibility of class members. Key Points of Encapsulation

1. **Data Hiding**:
   * Encapsulation helps to hide the internal state of an object from the outside world. This is done by declaring the class variables as private.
   * The internal representation of an object is hidden from view outside the object’s definition.
2. **Access Modifiers**:
   * **Private**: The member is accessible only within the same class.
   * **Public**: The member is accessible from any other class.
   * **Protected**: The member is accessible within the same package and subclasses.
   * **Default (no modifier)**: The member is accessible within the same package.
3. **Getter and Setter Methods**:
   * Provide public methods to access and update the value of private fields. These are known as getter and setter methods.
   * This allows for controlled access to the fields, adding a layer of validation or logic if necessary.

public class Person {

// Private fields

private String name;

private int age;

// Public getter for name

public String getName() {

return name;

}

// Public setter for name

public void setName(String name) {

this.name = name;

}

// Public getter for age

public int getAge() {

return age;

}

// Public setter for age

public void setAge(int age) {

if (age > 0) { // Adding a validation check

this.age = age;

}

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person();

person.setName("John");

person.setAge(25);

System.out.println("Name: " + person.getName()); // Output: Name: John

System.out.println("Age: " + person.getAge()); // Output: Age: 25

}  
}  
  
  
**RULES FOR JAVA METHOD OVERRIDING**

1. **Overriding and Access Modifiers**

The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass. Doing so will generate a compile-time error. 

// Overriding and Access-Modifiers

class Parent {

// private methods are not overridden

private void m1()

{

System.out.println("From parent m1()");

}

protected void m2()

{

System.out.println("From parent m2()");

}

}

class Child extends Parent {

// new m1() method

// unique to Child class

private void m1()

{

System.out.println("From child m1()");

}

// overriding method

// with more accessibility

@Override public void m2()

{

System.out.println("From child m2()");

}

}

// Driver class

class Main {

public static void main(String[] args)

{

Parent obj1 = new Parent();

obj1.m2();

Parent obj2 = new Child();

obj2.m2();

}

}

1. **Final methods cannot be overridden**

If we don’t want a method to be overridden, we declare it as final.

// final methods cannot be overridden

class Parent {

// Can't be overridden

final void show() {}

}

class Child extends Parent {

// This would produce error

void show() {}

}

1. **Static methods cannot be overridden (Method Overriding Vs Method Hiding):**

When you define a static method with the same signature as a static method in the base class, it is known as method hiding.

class Parent {

// Static method in base class

// which will be hidden in subclass

static void m1()

{

System.out.println("From parent "

+ "static m1()");

}

// Non-static method which will

// be overridden in derived class

void m2()

{

System.out.println("From parent "+ "non - static(instance) m2() ");

}

}

class Child extends Parent {

// This method hides m1() in Parent

static void m1()

{

System.out.println("From child static m1()");

}

// This method overrides m2() in Parent

@Override public void m2()

{

System.out.println("From child "+ "non - static(instance) m2() ");

}

}

class Main {

public static void main(String[] args)

{

Parent obj1 = new Child();

// As per overriding rules this

// should call to class Child static

// overridden method. Since static

// method can not be overridden, it

// calls Parent's m1()

obj1.m1();

// Here overriding works

// and Child's m2() is called

obj1.m2();

}

}

1. **Private methods cannot be overridden**

Private methods cannot be overridden as they are bonded during compile time. Therefore, we can’t even override private methods in a subclass.  
  
class SuperClass {

private void privateMethod()

{  
System.out.println("This is a private method in SuperClass");

}

public void publicMethod()

{

System.out.println("This is a public method in SuperClass");

privateMethod();

}

}

class SubClass extends SuperClass {

// This is a new method with the same name as the

// private method in SuperClass

private void privateMethod()

{

System.out.println("This is a private method in SubClass");

}

// This method overrides the public method in SuperClass

public void publicMethod()

{

System.out.println("This is a public method in SubClass");

privateMethod(); // calls the private method in

// SubClass, not SuperClass

}

}

public class Test {

public static void main(String[] args)

{

SuperClass obj1 = new SuperClass();

obj1.publicMethod(); // calls the public method in

// SuperClass

SubClass obj2 = new SubClass();

obj2.publicMethod(); // calls the overridden public

// method in SubClass

}

}  
  
  
**5. The overriding method must have the same return type (or subtype)**

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**.  
  
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!";

}

}

public class Test {

public static void main(String[] args)

{

SuperClass obj1 = new SuperClass();

obj1.method();

SubClass obj2 = new SubClass();

obj2.method();

}

}  
  
**6. Invoking overridden method from sub-class**

We can call the parent class method in the overriding method using the super keyword. 

// Base Class

class Parent {

void show() { System.out.println("Parent's show()"); }

}

// Inherited class

class Child extends Parent {

// This method overrides show() of Parent

@Override void show()

{

super.show();

System.out.println("Child's show()");

}

}

class Main {

public static void main(String[] args)

{

Parent obj = new Child();

obj.show();

}

}  
  
**CONSTRUCTORS**

A Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

class Geeks {

// Constructor

Geeks()

{

super();

System.out.println("Constructor Called");

}  
 public static void main(String[] args)

{

Geeks geek = new Geeks();

}

### } Key Points about Constructors

1. **Name**:
   * The name of the constructor must match the name of the class.
2. **No Return Type**:
   * Constructors do not have a return type.
3. **Types of Constructors**:
   * **Default Constructor**: If no constructor is explicitly defined, Java provides a default constructor with no parameters.
   * **Parameterized Constructor**: Constructors that take arguments to initialize an object with specific values.
4. **Overloading**:
   * Constructors can be overloaded, meaning a class can have more than one constructor with different parameter lists.
5. **Usage of this**:
   * The keyword this is used within a constructor to refer to the current object.
   * this() can be used to call another constructor in the same class.
6. **Constructor Chaining**:
   * When a constructor calls another constructor in the same class (using this()) or in the superclass (using super()).

**TYPES OF CONSTRUCTOR IN JAVA**

### **Default Constructor**

A default constructor is a no-argument constructor that is automatically provided by Java if no other constructors are explicitly defined in the class. It initializes the object with default values (zero, null, etc.).

#### It has no parameters.If no constructors are defined in a class, the Java compiler automatically creates a default constructor.

class Car {

String model;

int year;

// Default constructor

public Car() {

model = "Unknown";

year = 0;

}

void display() {

System.out.println("Model: " + model + ", Year: " + year);

}

public static void main(String[] args) {

Car car = new Car();

car.display(); // Output: Model: Unknown, Year: 0

}

}

### **2. Parameterized Constructor**

A parameterized constructor is defined by the programmer and can accept one or more parameters to initialize the object with specific values.

It takes arguments to set initial values for the object's attributes. Multiple parameterized constructors can be defined, enabling constructor overloading.

class Car {

String model;

int year;

// Parameterized constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

void display() {

System.out.println("Model: " + model + ", Year: " + year);

}

public static void main(String[] args) {

Car car = new Car("Tesla", 2022);

car.display(); // Output: Model: Tesla, Year: 2022

}

}

### **Constructor Overloading**

Constructor overloading means having more than one constructor in the same class with different parameter lists. It allows creating objects in different ways.

class Car {

String model;

int year;

// Default constructor

public Car() {

this.model = "Unknown";

this.year = 0;

}

// Parameterized constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

// Another parameterized constructor

public Car(String model) {

this.model = model;

this.year = 2023; // Default year

}

void display() {

System.out.println("Model: " + model + ", Year: " + year);

}

public static void main(String[] args) {

Car car1 = new Car();

car1.display(); // Output: Model: Unknown, Year: 0

Car car2 = new Car("Tesla", 2022);

car2.display(); // Output: Model: Tesla, Year: 2022

Car car3 = new Car("Ford");

car3.display(); // Output: Model: Ford, Year: 2023

}

}

**INTERFACE IN JAVA**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.



interface printable{

void print();

}    
class A6 implements printable{

public void print(){System.out.println("Hello");  
}

public static void main(String args[]){

A6 obj = new A6();

obj.print();

 }

}

**EXCEPTION HANDLING**  
  
**HIERARCHY OF EXCEPTIONS IN JAVA**

The hierarchy of Exceptions in the Java programming language begins with the Throw able class – which comes from the Object class and is its direct subclass while The Exception class presents all This Throw able class further branches into two subclasses – Error and Exception.

### **TYPES OF EXPECTION**

1. **Checked Exceptions**:
   * Checked at compile-time.
   * Must be either caught or declared in the method signature using throws.
   * Examples: IOException, SQLException.
2. **Unchecked Exceptions**:
   * Checked at runtime.
   * Subclasses of RuntimeException.
   * Examples: NullPointerException, ArrayIndexOutOfBoundsException.
3. **Errors**:
   * Represent serious problems not typically handled by applications.
   * Examples: OutOfMemoryError, StackOverflowError.

**TRY**Try keyword is used to define a block in which exceptions might occur.

try {  
 // code that may throw an exception

}

**CATCH**Used to handle exceptions that occur in the associated try block.

try {

// code that may throw an exception

} catch (ExceptionType e) {

// handle the exception

}  
  
**FINALLY**

A block that is always executed after the try and catch blocks, regardless of whether an exception was thrown or not. It is typically used for cleanup code.  
  
try {

// code that may throw an exception

} catch (ExceptionType e) {

// handle the exception

} finally {

// cleanup code

}  
  
**THROW**Throw is used to explicitly throw an exception.

if (someCondition) {

throw new IllegalArgumentException("Invalid argument");

}  
  
**THROWS**

**Throws is used in a method signature to declare the method that can throw exception.**

public void someMethod() throws IOException {

// method code

### } **MULTIPLE CATCH BLOCKS**

Used to handle different types of exceptions that might be thrown in a try block.  
  
try {  
// code that may throw multiple exceptions

} catch (IOException e) {

// handle IOException

} catch (SQLException e) {

// handle SQLException

}

### **EXCEPTION HANDLING WITH METHOD OVERRIDING**

When overriding a method in a subclass, the overridden method in the subclass: It cannot throw broader checked exceptions than those declared in the superclass method. It can throw fewer or no exceptions.

Superclass:

class Superclass {

public void display() throws IOException {

// method code

}

}

Subclass:

class Subclass extends Superclass {

@Override

public void display() throws FileNotFoundException {

// method code

}

}  
  
  
  
  
**COLLECTIONS**

### **HIERARCHY OF COLLECTION FRAMEWORK**

The Java Collection Framework provides a unified architecture for representing and manipulating collections. It includes interfaces and classes that extend these interfaces.

#### Key Interfaces:

* Collection
  + Set
    - HashSet
    - LinkedHashSet
    - TreeSet
  + List
    - ArrayList
    - Vector
    - LinkedList
  + Queue
    - PriorityQueue
* Map
  + HashMap
  + LinkedHashMap
  + TreeMap
  + ConcurrentHashMap

### **Collection Interface**

The Collection interface is the root interface in the collection hierarchy. It represents a group of objects, known as elements.

#### Methods:

* add(E e)
* remove(Object o)
* size()
* clear()
* isEmpty()
* contains(Object o)
* iterator()

### **Iterator Interface**

The Iterator interface provides methods to iterate over any collection. It allows the caller to remove elements during the iteration.

#### Methods:

* boolean hasNext()
* E next()
* void remove()

### **Set Interface**

A Set is a collection that does not allow duplicate elements.

#### Implementations:

* HashSet: Unordered, allows null.
* LinkedHashSet: Ordered by insertion, allows null.
* TreeSet: Ordered by natural ordering or a comparator, does not allow null.

### **List Interface**

A List is an ordered collection that allows duplicate elements.

#### Implementations:

* ArrayList: Resizable array implementation, allows null.
* Vector: Synchronized resizable array, allows null.
* LinkedList: Doubly linked list implementation, allows null.

### **Queue Interface**

A Queue is a collection used to hold multiple elements prior to processing. Typically orders elements in a FIFO (first-in-first-out) manner.

#### Implementation:

* PriorityQueue: Orders elements according to their natural ordering or by a comparator, allows null.

### **Map Interface**

A Map is an object that maps keys to values. A map cannot contain duplicate keys, and each key can map to at most one value.

#### Implementations:

* HashMap: Unordered, allows null keys and values.
* LinkedHashMap: Ordered by insertion, allows null.
* TreeMap: Ordered by natural ordering or a comparator, does not allow null keys.
* ConcurrentHashMap: Thread-safe, does not allow null keys or values.

### **Comparator Interface**

The Comparator interface is used to order the objects of a user-defined class.

#### Methods:

* compare(T o1, T o2)
* equals(Object obj)

### **Comparable Interface**

The Comparable interface is used to order the objects of the implementing class.

#### Method:

* int compareTo(T o)

### **Classes**

#### ArrayList

* A resizable array implementation of the List interface. Provides fast random access and is not synchronized.

#### Vector

* Similar to ArrayList, but synchronized. Suitable for thread-safe operations.

#### LinkedList

* Implements both List and Deque interfaces. Provides doubly linked list implementation.

#### PriorityQueue

* Implements the Queue interface and provides a priority queue implementation based on a priority heap.

#### HashSet

* Implements the Set interface and backed by a hash table. Provides constant-time performance for basic operations.

#### LinkedHashSet

* Extends HashSet and maintains a linked list of entries in the set, preserving insertion order.

#### TreeSet

* Implements the NavigableSet interface, backed by a TreeMap. Provides log(n) time cost for basic operations.

#### HashMap

* Implements the Map interface, provides constant-time performance for basic operations.

#### ConcurrentHashMap

* A thread-safe implementation of the Map interface. Provides better concurrency than Hashtable.

**MULTITHREADING**

### Multithreading is a Java feature that allows concurrent execution of two or more parts of a program to maximize CPU utilization. **Lifecycle of a Thread**

1. **New**: A thread is in the new state if you create an instance of the Thread class but before the invocation of the start() method.
2. **Runnable**: After invoking the start() method but before the thread starts running, the thread is in the runnable state. The thread may be running or ready to run.
3. **Running**: The thread is in the running state when the run() method is executing.
4. **Blocked/Waiting**: The thread is in this state when it is waiting for a resource or for another thread to perform a particular action.
5. **Terminated**: The thread is in this state when it has completed execution or has been terminated.

### **Thread Priority in Multithreading**

### Java threads have priority, which helps the thread scheduler determine the order in which threads are scheduled. Thread priority is an integer value between MIN\_PRIORITY (1) and MAX\_PRIORITY (10). The default priority is NORM\_PRIORITY (5). **Runnable Interface in Java**

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The Runnable interface has a single method, run(), which contains the code that constitutes the new thread's task.

public class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread is running");

}

}

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start();

}

### } **start() Function in Multithreading**

The start() method causes the thread to begin execution; the Java Virtual Machine calls the run() method of this thread.

public void start()

### **Thread.sleep() Method in Java**

The Thread.sleep() method pauses the execution of the current thread for a specified period.

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

### **Thread.run() in Java**

The run() method is the entry point for the thread. It is executed by the Thread object after the start() method is called.

public void run()

### **Deadlock in Java**

Deadlock is a situation where two or more threads are blocked forever, waiting for each other. This happens when two threads have a circular dependency on a pair of synchronized objects.

### **Synchronization in Java**

Synchronization is used to control the access of multiple threads to shared resources. Java provides synchronized blocks and methods to achieve synchronization.

#### Method Level Lock

A method can be synchronized to make it thread-safe by adding the synchronized keyword.

public synchronized void syncMethod() {

// synchronized method code

}

#### Block Level Lock

A synchronized block can be used to synchronize a block of code within a method.

public void someMethod() {

synchronized (this) {

// synchronized block code

}

}  
  
**Executor Framework in Java**

The Executor framework provides a high-level API for managing threads in Java. It decouples the task submission from the task execution and allows for managing a pool of threads.

ExecutorService executor = Executors.newFixedThreadPool(10);

executor.submit(new MyRunnable());

executor.shutdown();

### **Callable Interface in Java**

The Callable interface is similar to Runnable, but it can return a result and throw a checked exception. It has a single method, call().

public class MyCallable implements Callable<String> {

public String call() throws Exception {

return "Hello from Callable";

}

}

public class Main {

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> future = executor.submit(new MyCallable());

System.out.println(future.get()); // Output: Hello from Callable

executor.shutdown();

}

}