Java Exam Preparation

Last edited time	@January 16, 2024 11:31 PM
Created	@January 3, 2024 11:25 PM
Tags	OOP Year 2 Term 2
Contributors	Shagor

Resources : Slides, ChatGPT, TutorialsPoint

- Java
  - 1991, at Sun Microsystems
  - James Gosling, Patrick Chris, Frank and Mike Sheridan
  - portable, platform independent language, cross-platform
  - Java with C and C++
    - From C → Syntax
    - From C+ → OOP
  - Java's impacts
    - simplified internet programming
    - innovated a new type of networked program applet
      - a special kind of transmitted over the internet and automatically executed inside a Java-compatible web browser
    - portability and security
  - Bytecode
    - highly optimized set of instructions designed to be executed by what is called the Java Virtual Machine a part of JRE
    - it is not executable code, it must be executed by a JBM
    - JVM was designed as an interpreter for bytecode
    - HotSpot JVM

        - includes a just-in-time compiler, JIT is a part of the JVM
        - selected portions of bytecode are compiled into executable code
    - OOP
        - three traits : encapsulation, polymorphism and inheritance

- Literals
    - fixed values that are represented in their human readable form
    - Hexadecimal : 0x, 0X
    - Octal : 0

- Type Conversion
    - Type Conversion in Assignment
        - the value of the right side is automatically converted into the type of the left side
        - implicit type conversion
        - not all type conversations are implicitly allowed, ex : boolean and int
        - automatic type conversion will take place if
            - compatible
            - the destination type is larger than the source type
    - Casting Incompatible types
        - A cast is an instruction to the compiler to convert one type to another
        - an explicit type conversion

```
(target-type) expression
```

- Taking input from keyboard

```
....
import java.io.*;
throws java.io.IOException{
    char ch;
    ch = (char) System.in.read();
}
..
```

- Array

```
// Syntac
type[] array = new type[size];
type array[] = new type[size];
// example
int[] sample = new int[10];
int[] arr = {1,2,3,4,5,6,7,8,9}; // array intializers

//multidimensional array / 2D aray
int [][] table = new int[row][column];
/*
 iruegular array : specify only the memory of first dime
nstion and
second dimension manually
*/
int [][] table = new int[5][];
table[0] = new int[6];
...
```

  - **sample** holds a reference
  - Assigning array references

```
int [] arr = new int[10];
int [] arr2 = new int[10];
...
```

```
arr = arr2
..
Now, if we changes in arr or arr2, both will be affec
ted.
becasue, by asigning we are simply changing  what obj
ect that
 variable refers too. we are not making a copy.
```

- length

```
array.length => output the size of array
//example
int[][] arr = new int[5][6];
row -> arr.length
col -> arr[0].length
```

- For-each style for loop
  - sequential fashion, form start to finish
  - syntax streamlined
  - it prevents boundary errors

```
int[] arr = {1,2,3}
for(int x:arr) print x with a space;
ouput : 1 2 3
we can't change the value of arr using "x"
```

- Strings
  - In java, string are **objects**

```
String str = new String("Hello");
String str2 = "Hello";
String str4 = str + str2; // concatenation

// Oparting
str2.charAt(index)
str2.compareTo(str) : 0 = equal, < 0 = less than str,
```

```
                                        > 0 = greater than str

str2.indexOf(str) -> first index of str, -1 otherwise
str2.lastIndexOf(str) -> last index of str, -1 othersie
```

- Arrays of string

```
String[] str = new String[10];
for(int i=0; i<str.length; i++){
    str[i] = "hello";
}
str[5] = "hello 5";
for(string s:str) print s;
```

- Strings are immutable
  - the character sequence that makes up the string cannot be altered

  Substring methos

  ```
  String substr = str.substring(start index, end index);
  ```

- **Inference with Local Variable**
  - it would not be necessary specify an explicit type for an initialized variable because it could be inferred from the type of its initialized
  - Conditions
    - use “var” keyword, reserved keyword, you cant use it as identifier
    - must be initialized
    - in array, don’t need to use []
    - var cant be use with an array initializer

    ```
    var d = 10.6;
    var arr = new int[100];
    var str = "Hello";
    var newClass = new newClass(10);
    ```

```
//wrong
var b; // not initialized
var b[] == new int a[1000]; // [] brackers used
```

- OOP
  - new operator
    - the new operator dynamically allocates memory for an object and returns a **reference** to it
    - impact of reference

```
Vehicle a = new Vehicle();
Vehicle b = a;
a.value = 100;
print a.value and b.value => 100 100
```

  - Constructors
    - **Java automatically provides a default constructor**
      - non-initialized values are zero, null and false
    - when own constructor is defined, the default constructor is no longer used
  - **this** keyword
    - When a method is called, “this” is automatically passed an implicit argument that is a reference to the invoking object
  - Access modifiers : 3
    - public
      - accessed by any other code in program
    - private

        - accessed only within the class or other members of its class
        - cannot be access from outside of the class
    - protected

- **How to achieve call by reference in java?**

Ans. To perform call by reference in Java, we can use non-primitive data types such as object, class, string, array, and interface. With the help of an example, we have previously shown how we may use these non-primitive data types to achieve call by reference in Java.

- **Why is there no call by reference in Java?**

Java does not support call by reference because we need to supply the address and addresses are kept in pointers. Java does not support pointers since they violate security.

Java is always Pass by value. A few ways to achieve call-by-reference

- Making a public member in class

```
class myClass{
  public int a;
  public void update(myClass ob){
    ob.a  = 100;
  }
};
class Main{
  public static void main(String []args){
    myClass ob=new myClass();
    ob.a = 5;
    System.out.println(ob.a);
    ob.update(ob);
    System.out.println(ob.a);
  }
}
```

- Return a value from a function and update it

```
class Main{
    public static int update(int x){
        x++;
        return x;
    }
  public static void main(String []args){
        int i = 5;
        System.out.println(i);
        i = update(i);
        System.out.println(i);
  }
}
```

- Creating array/single element array

```
class Main{
    public static void update(int []x){
      x[0]++;
    }
  public static void main(String []args){
        int [] i = {1};
        System.out.println(i[0]);
        update(i);
        System.out.println(i[0]);
  }
}
```

Polymorphism

- Method overloading
  - two or more methods within same class can share the same name as long as their parameter declarations are different
  - it is not sufficient for two methods to differ only in their return types.
  - Overloading Constructor
    - reason : allow one object to initialize another

-

Static

- when a member is declared **static** it can be accessed before any objects of its class are created and without reference to any object
- both method and variable can be static
- no object needs to be created
- when an object is declared, no copy of a static variable is made
- **static method restrictions**
  - directly call only other static methods and variables in their class
  - do not have **this** reference
- static blocks
  - executed when the class is first loaded
  - even before constructor
  - executed before the class can be used for any other purpose

```
class  my {
my(){
    print constructor
}
static{
    print static
}
}
var my = new my();
output :
static
constructor
```

- **Nested class**
  - class within a class

- nested class is a member of enclosing class
- a nested class is not known outside of its block
- a class can be nested within a method, then it will be unknown outside of the method

Inheritance

- Superclass : a class that is inherited
- Subclass: the class that does the inheriting
- “extends” keyword
- java does not support the inheritance of multiple superclasses into a single subclass
- subclass cant access the only private properties of superlass
- **accessor method** is used to share private properties
  -

```
class {
private int b=5;
int accessor() return b;
}
```

- **constructor**
  - if superclass and subclass both have constructors
    - super is used
      - **super(parameter-list)**, must be the first statement inside a subclass constructor
      - **super.variable** to access superclass variable or methods

```
class A{
int i;
}
class B extends A{
    B(int a, int b){
```

```
            super.i = a; // access the i of A
            i = b; // access the i of B
        }
    }
```

- Multilevel hierarchy
    - Superclass → Subclass → Subclass ….

- Dynamic Method Dispatch
    - runtime polymorphism
    - an overridden method is resolved at run time rather than compile time.

- Package
    - package are groups of related classes
- the Java compiler automatically imports two entire packages for each source file: (1) the `java.lang` package and (2) the current package (the package for the current file).

- Interface
    - An interface is syntactically similar to an abstract class, in that you can specify one or more methods that have no body.
    - specifies what must be done, but not how to do it
    - one class can implement any number of interfaces

```
access interface intface{
....
};

class myClass implements intface{
```

```
....
}
```

- when no access modifier is included then default access result
- in interface, methods are **implicitly public**
- variables area **public, final and static**
- methods that implement an interface must be public
- a class doesn't implement all methods of the interface → must be declared as abstract, so no objects cant be created of that class and it will work like a abstract superclass
- interface reference can be created → similar to a superclass reference to access a subclass object
- interface can be extended by using "extends"
- interface default method
  - we can give default function definition if the method is not implemented by the class
- static interface method
- private interface method
  - **Code re-usability**
  - encapsulation

- **Exception handling**
  - an exception is an error occurs at run time
  - exception handling streamlines error handling by allowing your program to define a block of code called an exception handler, that is executed automatically when an error occurs.
  - In java all exceptions are represented by classes : T**hrowable**
    - it has two subclass
      - Exception : errors from program activity

        - Error : occur in JVM, not in own's code
    - **exception handling managed via five keywords**
        - try
            - problem statement that needs to be monitored
            - if exception occurs, it is thrown
        - catch
            - catch this exception
        - throw
            - to manually throw an exception
        - throws
            - an exception that is thrown out of a method must be specified as such by a throws clause
        - finally
            - code must be executed upon exiting from a try block
    - uncaught exception
        - if program does not catch an exception, then it will be caught by the JVM, it will display error message and terminate execution
    - some exceptions
        - ArithmeticException exc → divided by 0
        - ArrayIndexOutOfBoundsException exc
        - Throwable exc → some exception occurred (superclass of all exceptions)
    - catching subclass exception
        - a catch clause for a superclass will also match of its subclasses
        - the superclass of all exception is "Throwable"
        - to catch both subclass and superclass type, first catch should be subclasses and then superclass
    - Rethrow

        - an exception caught by one catch statement can be rethrown so that it can be caught by an outer catch
        - allow multiple handlers access
    - **Throws**
        - if a method generates an exception that it does not handle, it must declare that exception in a throws clause
        - throws keyword is used to declare an exception as well as pass the caller

Problem

- **What is bytecode ? Explain Boolean data type with example?**

A:

- highly optimized set of instructions designed to be executed by what is called the Java Virtual Machine a part of JRE.
- It is an intermediate code between source code and machine code. Enables portability and platform independence for interpreted languages.
- Bytecode is an intermediate representation of a program that is generated by the Java compiler. Instead of generating machine-specific code, Java compilers translate Java source code into bytecode

Boolean Data Type: Represents a binary value, typically denoted as true or false. 1 byte.

```
public class BooleanExample {
    public static void main(String[] args) {
        boolean isJavaFun = true;
        System.out.println("Is Java fun? " + isJavaFun);
        if (isJavaFun) {
            System.out.println("Yes, Java is fun!");
        } else {
            System.out.println("No, Java is not fun.");
        }
```

```
    }
}
```

- **Explain the For-Each version of the for loop with example.**

A: A concise way to iterate over all elements in an array or a collection. It simplifies the syntax and make it easier to traverse because it doesn't requires indexing.
Syntax:

```
for (element_type element : array_or_collection) {
    // Code to be executed for each element
}
/*Here, element_type is the data type of the elements in th
e array or
collection, and array_or_collection  is the array or collec
tion you
want to iterate over.*/
```

```
public class ForEachExample {
    public static void main(String[] args) {
        // Declare an array of integers
        int[] numbers = {1, 2, 3, 4, 5};

        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

- Is Java pure object oriented ? Java is platform independent but JVM is not, why ?

A: Java is **not** considered a pure object-oriented programming language.

- The main reason is it supports primitive type values. For an object-oriented programming language, data should be represented in the form of objects. As

Java uses primitive data types, it is not considered a pure object-oriented programming language.

- It supports the use of the static keyword.

While Java itself is platform-independent, the JVM is platform-dependent. This is because the JVM is specific to each operating system and hardware architecture. When you write Java code, you compile it into bytecode, and this bytecode can be run on any system with the appropriate JVM for that system.

The reason JVM is not platform-independent is that it needs to be implemented differently for each operating system and hardware combination to interact with the underlying system resources.

- How to create a two dimensional array in which the sizes of the second dimension are unequal? Explain with a programming example.

```
public class UnequalSize2DArrayExample {
    public static void main(String[] args) {
        int[][] unevenArray = new int[3][]; // Array with 3
"rows"

        unevenArray[0] = new int[3];  // First row with 3 e
lements
        unevenArray[1] = new int[4];   // Second row with 4
elements
        unevenArray[2] = new int[2];   // Third row with 2
elements

    }
}
```

In this example, `unevenArray` is a two-dimensional array where each "row" is an array with a different size. The first row has 3 elements, the second row has 4 elements, and the third row has 2 elements. The program then prints the elements of the two-dimensional array using nested loops.

This flexibility allows you to create a jagged array, which is a two-dimensional array with varying row sizes. It's worth noting that each "row" is a separate array object,

and they can have different lengths.

- What is the purpose of keyword **new**? Explain what happens when this keyword is used in a program.

A: In Java, the `new` keyword is used to dynamically allocate memory for an object at runtime. It is an essential part of the process of object creation.

  - Memory Allocation: allocates memory spaces on the heap
  - Object initialization: called the constructor
  - Return reference: returns a reference to the newly created object

- Find error in the following fragment of a program.

```
final class A{
// ...
}
class B extends A{
// ...
}
```

In Java, when a class is declared with the `final` keyword, it means that the class cannot be subclassed or extended by other classes. The `final` keyword, when applied to a class, indicates that the class is complete and cannot be further modified in terms of inheritance.

- Why **trim()** method is used? Illustrate with a programming example.

A: The `trim()` method in Java is used to remove leading and trailing whitespace from a string. Whitespace includes spaces, tabs, and line breaks.

```
public class TrimExample {
    public static void main(String[] args) {
        String originalString = "   Hello, Trim!   ";
        String trimmedString = originalString.trim();

        // Displaying the results
```

```
        System.out.println("Original String: \"" + originalString + "\"");
        System.out.println("Trimmed String: \"" + trimmedString + "\"");
    }
}
/*Output
Original String: "   Hello, Trim!   "
Trimmed String: "Hello, Trim!"
*/
```

- **Define bytecode. Why does Java use Unicode?**

A: Bytecode

- highly optimized set of instructions designed to be executed by what is called the Java Virtual Machine a part of JRE.
- It is an intermediate code between source code and machine code. Enables portability and platform independence for interpreted languages.
- Bytecode is an intermediate representation of a program that is generated by the Java compiler. Instead of generating machine-specific code, Java compilers translate Java source code into bytecode

Java uses Unicode for character representation t**o ensure consistent handling of characters across different platforms and languages**. Unicode is a standardized character encoding that assigns a **unique numeric value** to each character, regardless of the platform, program, or language.

- **What are the three main principles of object-oriented programming? Explain.**
    - **Encapsulations:** Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on the data into a single unit known as a class. Encapsulation helps in organizing code, hiding implementation details, and promoting modularity.

- **Inheritance:** Inheritance is a mechanism that allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (base class or superclass).
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables **a single interface to represent different types or forms**. There are two types of polymorphism: compile-time (method overloading) and runtime (method overriding).

- **You know there are four integer types: int, short, long, and byte. However, char can also be categorized as an integer type in Java. Why?**

In Java, the `char` type is often categorized as an integer type because it represents **16-bit Unicode** characters. While `int`, `short`, `long`, and `byte` are integral data types used to store integer values, `char` is specifically designed to store Unicode characters, which are essentially numeric representations of characters from various writing systems.

- **"Java allows variables to be initialized dynamically"-do you agree with the statement? Justify your answer with an example.**

Yes.

Initialization is the process of providing value to a variable at declaration time. Dynamic initialization of object refers to initializing the objects at run time i.e. the initial value of an object is to be provided during run time. **Dynamic initialization can be achieved using constructors and passing parameters** values to the constructors.

```
public class Car {
    String make;
    String model;
    int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

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

```
        Car myCar = new Car("Toyota", "Camry", 2022);
        myCar.displayInfo();
    }
}
```

- **What is type conversion and casting? Explain with proper examples.**
  - **Type Conversion/Implicit Type Conversion:** Occurs **automatically** when a value of one data type is assigned to a variable of another **compatible data type.** Java performs implicit type conversion when the conversion **does not result in a loss of precision** or information.

```
int intValue = 42;
long longValue = intValue; // Implicit conversion from int to long
```

  - **Explicit Type Conversion (Casting):** Requires **manual** intervention and is done using casting operators. Explicit casting is needed when **converting from a larger data type to a smaller one,** or when precision loss is possible.

```
double doubleValue = 3.14;
int intValue = (int) doubleValue; // Explicit casting from double to int
```

- **Differentiate between a class and an object. What do you mean by this keyword? Explain with an example.**

Feature	Class	Object
Definition	A blueprint or template for creating objects.	An instance of a class; a concrete realization based on the class blueprint.
Nature	Abstract, representing a type.	Concrete, representing an instance.
Properties	Defines attributes and behaviors but does not have specific values for them.	Has specific values for attributes and can perform actions defined by behaviors.

Feature	Class	Object
Creation	Used to create objects.	Created based on a class.
Instantiation	A class itself cannot be instantiated; it needs to be instantiated to create objects.	Represents a specific instance created from a class.
Usage	Used to structure and organize code.	Used to model and represent real-world entities.
Example	If `Car` is a class, it might define attributes like `make`, `model`, and `year`, and behaviors like `startEngine`.	If `myCar` is an object of the `Car` class, it has specific values for `make`, `model`, and `year` and can perform actions like starting the engine.
Memory	Class definition does not occupy memory at runtime.	Each object created from a class consumes memory space to store its state.
Keyword	No keyword associated directly; the class keyword is used in the class definition.	No specific keyword; the instance is created using the `new` keyword followed by the class constructor.

this keyword

- • The `this` keyword in Java is a reference variable that refers to the current object.
- • It is used to differentiate between instance variables and parameters with the same name within a method or constructor.
- • `this` is often used to access instance variables or invoke methods of the current object.

```
public class Car {
    String make;
    String model;
    int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
```

```
    public void displayInfo() {
        // Use of 'this' to access instance variables
        System.out.println("Car Make: " + this.make);
        System.out.println("Car Model: " + this.model);
        System.out.println("Manufacturing Year: " + this.ye
ar);
    }

    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry", 2022);

        myCar.displayInfo();
    }
}
```

In this example:

- The `Car` class is defined with instance variables (`make`, `model`, `year`), a parameterized constructor, and a method (`displayInfo`).
- Inside the constructor and the `displayInfo` method, the `this` keyword is used to reference the current object's instance variables.
- In the `main` method, an object of the `Car` class (`myCar`) is created, and its information is displayed using the `displayInfo` method.

- **What is method overloading? Explain with an example program.**

Method overloading is a feature in Java that allows a class to have **multiple methods with the same name but different parameters.** The parameters can differ in terms of the number, type, or order. Overloaded methods provide flexibility and improve code readability.

```
public class MathOperations {

    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
```

```
        return a + b + c;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        MathOperations mathOps = new MathOperations();

        int result1 = mathOps.add(5, 10);
        int result2 = mathOps.add(3, 7, 12);
        double result3 = mathOps.add(2.5, 3.5);

        System.out.println("Result 1: " + result1);
        System.out.println("Result 2: " + result2);
        System.out.println("Result 3: " + result3);
    }
}
```

- **Discuss about the method overloading and constructor overloading with an example.**

Constructor overloading is a concept similar to method overloading, but it involves having multiple constructors in a class with different parameter lists.

```
public class OverloadingExample {

    // Constructor Overloading
    public OverloadingExample() {
        System.out.println("Default Constructor");
    }

    public OverloadingExample(int value) {
        System.out.println("Parameterized Constructor with
one int parameter: " + value);
    }
```

```
    public OverloadingExample(double value1, double value2)
{
        System.out.println("Parameterized Constructor with
two double parameters: " + value1 + ", " + value2);
    }

    // Method Overloading
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String str1, String str2) {
        return str1 + str2;
    }

    public static void main(String[] args) {
        // Constructor Overloading
        OverloadingExample defaultConstructor = new Overloa
dingExample();
        OverloadingExample intConstructor = new Overloading
Example(42);
        OverloadingExample doubleConstructor = new Overload
ingExample(3.14, 2.7);

        System.out.println("-------------------------");

        // Method Overloading
        OverloadingExample example = new OverloadingExample
();

        int sumInt = example.add(5, 10);
        double sumDouble = example.add(3.5, 2.5);
        String concatenatedString = example.add("Hello, ",
"World!");
```

```
        // Displaying results
        System.out.println("Sum (int): " + sumInt);
        System.out.println("Sum (double): " + sumDouble);
        System.out.println("Concatenated String: " + concat
enatedString);
    }
}
```

- **Write a java program to implement visibility controls such as public, private, protected access modes. Assume suitable data, if any.**

```
class VisibilityExample {
    public String publicMessage = "This is a public messag
e.";
    private String privateMessage = "This is a private mess
age.";
    protected String protectedMessage = "This is a protecte
d message.";
    String defaultMessage = "This is a default message.";

    public void displayPublicMessage() {
        System.out.println("Public Message: " + publicMessa
ge);
    }

    private void displayPrivateMessage() {
        System.out.println("Private Message: " + privateMes
sage);
    }

    protected void displayProtectedMessage() {
        System.out.println("Protected Message: " + protecte
dMessage);
    }

    void displayDefaultMessage() {
```

```
        System.out.println("Default Message: " + defaultMes
sage);
    }
}

class Subclass extends VisibilityExample {
    void displayProtectedFromSubclass() {
        System.out.println("Protected Message from Subclas
s: " + protectedMessage);
    }

    void displayProtectedMethodFromSubclass() {
        displayProtectedMessage();
    }
}

public class VisibilityControlsExample {
    public static void main(String[] args) {
        VisibilityExample example = new VisibilityExample
();

        System.out.println("Public Message: " + example.pub
licMessage);
        example.displayPublicMessage();

        System.out.println("-------------------------");

        Subclass subclass = new Subclass();
        subclass.displayProtectedFromSubclass();
        subclass.displayProtectedMethodFromSubclass();

        System.out.println("-------------------------");

        // Uncommenting the lines below will result in comp
ilation errors
        // because private and default members are not acce
ssible from outside the class and package.
```

```
        // System.out.println("Private Message: " + exampl
e.privateMessage);
        // example.displayPrivateMessage();

        // System.out.println("Default Message: " + exampl
e.defaultMessage);
        // example.displayDefaultMessage();
    }
}
```

- **What is an abstract class? "A class that contains at least one abstract method must, itself, be declared abstract"- Is it true or False? Answer accordingly.**

An abstract class in Java is a class that cannot be **instantiated on its own and may contain both abstract methods (methods without a body) and concrete methods.** • To declare an abstract class, the `abstract` keyword is used.

True. If a class contains at least one abstract method, it must be declared as abstract itself.

```
abstract class Shape {
    // Abstract method (no body)
    public abstract double calculateArea();

    // Concrete method with implementation
    public void display() {
        System.out.println("This is a shape.");
    }
}
```

- **"A superclass reference can refer to a subclass object."- Explain why this is important as it relates to method overriding.**

This is important, because it’s a way of achieving runtime polymorphism or dynamic method dispatch.

When a method is called on an object through a reference variable, the actual method that gets executed is determined at runtime based on the type of the object, not the type of the reference variable. It enables the execution of the overridden method in the subclass, even if the reference variable is of the superclass type.

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark! Bark!");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound();
    }
}
```

- **What is inheritance? Explain the benefits of inheritance with an example**

Inheritance is a mechanism that allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (base class or superclass).

Benefits:

- **Code Reusability**
- **Extensibility**
- **Polymorphism**
- **Structural Hierarchy**

```
// Base class (Superclass)
class Vehicle {
```

```
    String brand;
    int year;

    // Constructor
    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    // Method
    void start() {
        System.out.println("The vehicle is starting.");
    }
}

// Subclass 1
class Car extends Vehicle {
    int numberOfDoors;

    // Constructor
    public Car(String brand, int year, int numberOfDoors) {
        super(brand, year); // Call to superclass construct
or
        this.numberOfDoors = numberOfDoors;
    }

    // Method overriding

    void start() {
        System.out.println("The car engine is starting.");
    }

    void drive() {
        System.out.println("The car is in motion.");
    }
}

// Subclass 2
```

```java
class Motorcycle extends Vehicle {
    boolean hasSideCar;

    // Constructor
    public Motorcycle(String brand, int year, boolean hasSi
deCar) {
        super(brand, year); // Call to superclass construct
or
        this.hasSideCar = hasSideCar;
    }

    // Method overriding

    void start() {
        System.out.println("The motorcycle engine is starti
ng.");
    }

    void ride() {
        System.out.println("The motorcycle is in motion.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        // Creating instances of subclasses
        Car myCar = new Car("Toyota", 2022, 4);
        Motorcycle myMotorcycle = new Motorcycle("Harley-Da
vidson", 2021, false);

        // Accessing inherited members
        System.out.println("Car Brand: " + myCar.brand);
        System.out.println("Motorcycle Year: " + myMotorcyc
le.year);

        // Calling overridden methods
        myCar.start();          // Calls overridden method
in Car class
```

```
        myMotorcycle.start();   // Calls overridden method
in Motorcycle class

        // Calling subclass-specific methods
        myCar.drive();
        myMotorcycle.ride();
    }
}
```

- **Explain the usage of abstract classes and methods? With an example program**

Abstract classes and methods in Java are used to define a common structure for a group of related classes. Abstract classes cannot be instantiated on their own; they are meant to be extended by subclasses. Abstract methods declared in an abstract class are meant to be implemented by the concrete (non-abstract) subclasses. Abstract classes provide a way to achieve abstraction and define a blueprint for classes that share common characteristics.

```
abstract class Shape {
    abstract double calculateArea();

    void display() {
        System.out.println("This is a shape.");
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }


    double calculateArea() {
```

```
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double calculateArea() {
        return length * width;
    }
}

public class AbstractClassExample {
    public static void main(String[] args) {
        Circle myCircle = new Circle(5.0);
        Rectangle myRectangle = new Rectangle(4.0, 6.0);

        myCircle.display();
        System.out.println("Area of Circle: " + myCircle.ca
lculateArea());

        System.out.println("-------------------------");

        myRectangle.display();
        System.out.println("Area of Rectangle: " + myRectan
gle.calculateArea());
    }
}
```

- **Write a java program to implement multilevel inheritance with 4 levels of hierarchy.**

```java
class A {
    void displayA() {
        System.out.println("Class A");
    }
}

class B extends A {
    void displayB() {
        System.out.println("Class B");
    }
}

class C extends B {
    void displayC() {
        System.out.println("Class C");
    }
}

class D extends C {
    void displayD() {
        System.out.println("Class D");
    }
}

public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        D objD = new D();

        // Accessing methods from different levels of hiera
rchy
        objD.displayA(); // From class A
        objD.displayB(); // From class B
        objD.displayC(); // From class C
        objD.displayD(); // From class D
    }
}
```

- **What is garbage collection in Java? Explain finalize method in Java.**

Garbage collection is a process in Java that automatically **reclaims memory** occupied by objects that are **no longer in use or reachable by the program**. Java employs an automatic garbage collector to manage memory and deallocate objects that are no longer referenced. This helps in **avoiding memory leaks and makes memory management** more convenient for developers.

The finalize() method is a method defined in the Object class in Java, which is the base class for all other classes. It is invoked by the garbage collector before an object is garbage collected, providing an opportunity to perform cleanup operations.

- The primary purpose of the finalize() method is to perform any necessary cleanup before an object is destroyed.
- It is recommended to use try-finally or try-catch-resources constructs for resource cleanup.

```
import java.lang.*;

 class prep {

    protected void finalize() throws Throwable
    {
        try {

            System.out.println("inside prep finalize()");
        }
        catch (Throwable e) {

            throw e;
        }
        finally {

            System.out.println("Calling finalize method"
                               + " of the Object class");

            // Calling finalize() of Object class
            super.finalize();
```

```
        }
    }

    // Driver code
    public static void main(String[] args) throws Throwable
    {

        // Creating demo's object
        prep d = new prep();

        // Calling finalize of demo
        d.finalize();
    }
}
```

- **Compare and Contrast differences between interfaces vs abstract classes.**

Feature	Interfaces	Abstract Classes
Inheritance	Supports multiple inheritance. A class can implement multiple interfaces.	Supports single inheritance. A class can extend only one abstract class.
Constructor	Cannot have constructors.	Can have constructors, and they are used during object creation.
Fields	Can have only static and final fields.	Can have instance variables, static variables, and final variables.
Access Modifiers	All methods are implicitly public and abstract. Fields are implicitly public, static, and final.	Can have different access modifiers for methods and fields.
Method Type	Methods are implicitly abstract and public. Can also have default and static methods starting from Java 8.	Methods can be abstract or concrete. They may have any access modifier (public, private, protected).
State	Cannot contain state (fields) before Java 8 (Java 8 onwards, static and final fields are allowed).	Can contain state (fields).

Feature	Interfaces	Abstract Classes
Constructor Invocation	No constructor chaining because there are no constructors in interfaces.	Supports constructor chaining through the constructor of the superclass.
Instance Creation	Cannot be instantiated.	Cannot be instantiated; used as a blueprint for subclasses.
Use Cases	Used to achieve multiple inheritance and to define contracts for classes.	Used when a common base implementation is needed and to provide a partial or complete implementation.
Dependency	Promotes a loosely coupled design as a class can implement multiple interfaces without worrying about the implementation details.	Tighter coupling, as a class can extend only one abstract class and is bound to its implementation.
Versioning	Easier to add new methods without affecting existing classes that implement the interface.	May lead to issues with existing subclasses if new methods are added, especially in backward compatibility scenarios.
Abstract Methods	All methods are implicitly abstract.	May have abstract and concrete methods.
Default Methods	Introduced in Java 8. Allows adding new methods to interfaces without breaking existing implementations.	Not applicable; abstract classes do not have default methods.

- **What is an exception? Explain how an exception can be handled in Java? And also list the benefits of Exception Handling.**

An exception in Java is an event that disrupts the normal flow of the program's instructions during execution. It is typically caused by errors or unexpected conditions that occur at runtime. Exceptions can be of different types, such as runtime exceptions, checked exceptions, and errors.

```
try {
    // Code that may cause an exception
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
```

```
} finally {
    // Code that will be executed regardless of whether an ex
}
```

- The `try` block contains the code that may throw an exception.
- The `catch` blocks handle specific types of exceptions that may occur in the `try` block.
- The `finally` block contains code that will be executed regardless of whether an exception occurred or not. It is optional.

Benefits:

- **Error Localization :** Exception handling helps in localizing and handling errors where they occur.
- **Program Robustness:** enhances program robustness by preventing abnormal program termination
- **Separation of Concerns:** promotes separation of concerns by allowing the code that may cause exceptions
- **Graceful degradation :** enables graceful degradation by providing a mechanism to respond to unexpected situations

- **Distinguish between exception and error.**

Feature	Exception	Error
Type	Checked or Unchecked Exception	Unchecked Exception (Runtime Exception)
Cause	Caused by external factors or user code	Generally caused by e**xternal factors beyond the control of the application or indicate serious issues in the JVM or the system**
Handling	Intended to be caught and handled by the program	Typically not caught or handled by the program; often indicates irrecoverable situations
Examples	`IOException`, `SQLException`, `NullPointerException`	`OutOfMemoryError`, `StackOverflowError`, `NoClassDefFoundError`

- **Define an exception called "NotEqualException" that is thrown when a float value is not equal to 3.14. Write a java program that uses the above user defined exception.**

```
// User-defined exception class
class NotEqualException extends Exception {
    public NotEqualException(String message) {
        super(message);
    }
}

// Main program
public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            float floatValue = 3.0f; // Change this value t
o see the exception

            // Check if the float value is equal to 3.14
            if (floatValue != 3.14f) {
                throw new NotEqualException("Float value is
not equal to 3.14");
            } else {
                System.out.println("Float value is equal to
3.14");
            }
        } catch (NotEqualException e) {
            // Handle the custom exception
    System.err.println("Caught NotEqualException: " + e.get
Message());
        }
    }
}
```

- **Java is guaranteed to be "Write Once, Run Anywhere." Explain.**

"Write Once, Run Anywhere" (WORA) is a key principle of Java that highlights its platform independence. This principle is achieved through the use of the Java Virtual Machine (JVM) and bytecode.

Java source code is compiled into an intermediate form called bytecode. This bytecode is not specific to any particular hardware or operating system. The compiled bytecode is executed by the Java Virtual Machine (JVM). The JVM is a software-based machine that interprets and executes Java bytecode.

- **Briefly explain the use of try, catch, throw, throws and finally with an example.**

1. `try` **block:**
    - The `try` block contains the code that might throw an exception.
2. `catch` **block:**
    - The `catch` block follows the `try` block and handles the exception if it occurs. Multiple `catch` blocks can be used to handle different types of exceptions.
3. `throw` **statement:**
    - The `throw` statement is used to explicitly throw an exception. It is typically used within the `try` block.
4. `throws` **clause:**
    - The `throws` clause is used in a method signature to declare the exceptions that the method might throw. It informs the caller about the potential exceptions.
5. `finally` **block:**
    - The `finally` block contains code that will be executed whether an exception occurs or not. It is optional but commonly used for cleanup operations.

    .

```
try {
    // Code that may cause an exception
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
```

```
} finally {
    // Code that will be executed regardless of whether
an exception occurred or not
}
```

- throw vs throws

Feature	`throw`	`throws`
Purpose	Used to explicitly **throw an exception** in code.	Used in **method signatures to declare exceptions** that the method might throw.
Usage	Used inside the `try` block to throw an exception.	Used in method declarations followed by the exception types separated by commas.
Example	`throw new IOException("Invalid input");`	`void exampleMethod() throws IOException { ... }`
Applicability	Used for throwing exceptions programmatically.	Used for declaring exceptions that a method may throw, allowing the caller to handle them or propagate them further.
Location	Inside a method or a block of code.	In the method signature before the method body.
Exception Type	Can throw any subclass of `Throwable`.	Specifies the specific exceptions that a method might throw.
Number of Types	Can throw one exception at a time.	Can declare multiple exceptions using a comma-separated list.
Handling	Requires a corresponding `catch` block or the method must declare the exception using `throws`.	Alerts the calling code about potential exceptions and allows it to handle them.

- **What is portability? How does Java solve portability problem?**

Portability refers to the ability of software to run on different platforms or systems without modification. In the context of programming languages and applications,

a portable program can be written once and then executed on various platforms with minimal or no modifications.

How Java Solves the Portability Problem:

1. **Bytecode:**
    - Java source code is compiled into an intermediate form called bytecode. Bytecode is not **platform-specific** and can be executed on any device or system that has a Java Virtual Machine (JVM) implemented for that platform.
2. **Java Virtual Machine (JVM):**
    - The JVM is a software-based machine that **interprets and executes Java bytecode**. Each platform or operating system has its own JVM implementation, making it responsible for handling platform-specific details.

- **Write the names of bitwise operators?**

The bitwise operators in Java are:

1. **AND (`&`):**
2. **OR (`|`):**
3. **XOR (`^`):**
4. **NOT (`~`):**
5. **Left Shift (`<<`):**
6. **Right Shift (`>>`):**
7. **Unsigned Right Shift (`>>>`):**

- **Why *static* members are declared in a class? What are the restrictions when methods are declared as static?**

why:

- **Memory efficiency :** Static members are allocated memory only once, regardless of the number of instances created.
- **Global Access**

static method restrictions

- directly call only other static methods and variables in their class
- do not have **this** reference
- Cannot Use `super` in a Static Context:

- **Why *super* keyword is used? Explain with a programming example.**

The `super` keyword in Java is used to refer to the immediate parent class's members (variables or methods, constructors). It is often used to differentiate between a subclass's members and those of its superclass with the same name.

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        // Calling the superclass's makeSound using super
        super.makeSound();

        // Adding subclass-specific behavior
        System.out.println("Woof! Woof!");
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

In this example:

- The `Animal` class has a `makeSound` method.

- The `Dog` class extends `Animal` and overrides the `makeSound` method.
- Inside `Dog`'s `makeSound` method, `super.makeSound()` is used to explicitly call the `makeSound` method of the superclass (`Animal`), followed by additional behavior specific to the `Dog` class.

- **What is the difference between method overriding and overloading? Explain.**

Feature	Method Overriding	Method Overloading
Definition	Occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.	Involves having multiple methods in the same class with the same name but different parameter lists (number, type, or order).
Signature	The overridden method in the subclass must have the same method signature (name, return type, and parameters) as the one in the superclass.	The overloaded methods must have different parameter lists. The return type or access modifiers may remain the same or differ.
Inheritance	Related to inheritance and occurs in a superclass-subclass relationship.	Not dependent on inheritance; can occur within the same class or between a superclass and its subclass.
Occurrence	Happens when a subclass wants to provide a specialized implementation for a method defined in its superclass.	Involves having multiple methods in the same class with the same name but different parameter lists.
Polymorphism	Contributes to runtime polymorphism. The version of the method in the subclass replaces the one in the superclass during runtime.	Does not involve polymorphism. The appropriate method to be executed is determined at compile-time based on the method signature.
Example	`java class Animal { void makeSound() { System.out.println("Some generic sound"); } } class Dog extends Animal { void makeSound() { System.out.println("Woof! Woof!"); } }`	`java class Calculator { int add(int a, int b) { return a + b; } double add(double a, double b) { return a + b; } }`

- **What is *dynamic method dispatch*? Explain dynamic method dispatch with a programming example.**

Dynamic Method Dispatch is a mechanism in object-oriented programming languages, including Java, where **the method that gets executed is determined at runtime based on the actual type of the object.** It enables polymorphism and allows a superclass reference variable to refer to a subclass object and invoke overridden methods.

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow!");
    }
}

public class DynamicMethodDispatchExample {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Dog(); // Dog object assigned to Ani
mal reference
        myAnimal.makeSound(); // Dynamic method dispatch, c
alls Dog's makeSound

        myAnimal = new Cat(); // Cat object assigned to Ani
mal reference
```

```
        myAnimal.makeSound(); // Dynamic method dispatch, c
alls Cat's makeSound
    }
}
```

In this example:

- We have a superclass `Animal` with a method `makeSound`.
- There are two subclasses, `Dog` and `Cat`, both of which override the `makeSound` method.
- In the `main` method, we create an `Animal` reference variable (`myAnimal`) and assign it successively to objects of type `Dog` and `Cat`.
- The `makeSound` method is called on the `myAnimal` reference. At runtime, the JVM determines the actual type of the object that `myAnimal` is referring to and calls the overridden method in that specific subclass (`Dog` or `Cat`).

- Given the following hierarchy:

```
class Alpha {...
class Beta extends Alpha {....
Class Gamma extends Beta {....
```

In what order do the constructors for these classes complete their execution when a Gamma object is instantiated?

[Keynote: Superclass → subclass of this superclass- > subclass of the …..]

1. **Topmost Superclass (Alpha):**
    - The constructor of the topmost superclass (`Alpha`) is executed first.
2. **Immediate Superclass (Beta):**
    - The constructor of the immediate superclass (`Beta`) is executed next.
3. **Derived Class (Gamma):**
    - Finally, the constructor of the derived class (`Gamma`) is executed.

Therefore, when a `Gamma` object is instantiated, the order of constructor execution is as follows:

Output:
Alpha's Constructor
Beta's Constructor

Gamma's constructor

- Find error in the following fragment of a program. What type of error will occur?

```
class A{
public static void main(String args[]){
int x=99;
{ int x=9;}.
}
}
```

A variable in a **method** can be defined only for **once**. While the "x" is declared twice. So, it will generate an error.

Attempting to declare the variable `x` again within the inner block will result in a **compilation error.**

- **int a=32; a=a>>2; After applying this, What will be the value of a?**

[Right shift : x >> y ⇒ dividing x by 2^y

Left shift : x << y ⇒ multiplying x by 2^y]

$\frac{32}{2^2} = 8$

- Find out the error of the following code. Explain, why?

```
 class A{
final void xy() { System.out.println("xy"); }
}
class B extends A{
void xy() { System.out.println("xy"); } }
```

In Java, when a method in a superclass is declared as `final`, it means that the method cannot be overridden by any subclass. However, in the code, it is attempting to override the `final` method `xy` in the subclass `B`. This is not allowed, and it will result in a **compilation error.**

- **Explain the two ways that the members of a package can be used by other packages**
  - **Importing Individual Members Explicitly**

    In this approach, specific members (classes, interfaces, or other items) of a package are imported explicitly into the code of another package using the `import` statement. This allows developers to selectively import only the components they need, avoiding naming conflicts.

```
package com.example.packageB;

import com.example.packageA.MyClassA;

public class MyClassB {
    public static void main(String[] args) {
        // Using MyClassA from packageA in packageB
        MyClassA objA = new MyClassA();
    }
}
```

  - **Importing the Whole Package**

    Alternatively, developers can import all the members of a package into another package using the `import` statement followed by the package name with an asterisk (`*`). This approach imports all the classes and interfaces of the specified package.

```
package com.example.packageB;

import com.example.packageA.*;

public class MyClassB {
```

```
    public static void main(String[] args) {
        // Using MyClassA from packageA in packageB
        MyClassA objA = new MyClassA();
    }
}
```

- **What standard Java package is automatically imported into a program?**

`java.lang` package is automatically imported into every Java program

This means that classes and interfaces from the `java.lang` package can be used in a Java program without the need for an explicit `import` statement.

The `java.lang` package is a fundamental package in Java and includes essential classes and interfaces that are commonly used, such as `Object`, `String`, and basic data types like `int` and `boolean`. Since it is automatically imported, you can directly use these classes and data types without explicitly importing them.

- **Explain the difference between protected and default access.**

Feature	`protected` Access Modifier	Default (Package-Private) Access Modifier
Scope	Accessible within the same package and subclasses, even if they are in different packages.	Accessible only within the same package.
Outside Package	Accessible in subclasses outside the package.	Not accessible in subclasses outside the package.
Example Code	`java public class A { protected int x; }`	`java class B { int y; }`
Example Usage	`java public class C extends A { void method() { System.out.println(x); } }`	`java class D { void method() { B obj = new B(); System.out.println(obj.y); } }`
Inheritance	Useful when you want to expose a member to subclasses, regardless of the package.	Useful when you want to limit access to members within the same package.

- **How many classes can implement an interface? How many interfaces can a class implement?**

A class in Java can implement **multiple** interfaces, and an interface can be implemented by **multiple** classes.

- **Can interfaces be extended? Explain with a programing example.**

Yes, an interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface **can extend any number of interfaces**. The interface declaration includes a **comma-separated list** of all the interfaces that it extends.

```
interface A {
   void funcA();
}
interface B extends A {
   void funcB();
}
class C implements B {
   public void funcA() {
      System.out.println("This is funcA");
   }
   public void funcB() {
      System.out.println("This is funcB");
   }
}
public class Demo {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
   }
}
```

- **Is it possible to define a static method in an interface?**

Yes, starting from Java 8, it is possible to define a **static method** in an interface.

- **Can an interface have a private method?**

Beginning with Java 9, you can have **private** methods in interfaces.

Since **private** methods are only accessible within the interface in which it has been defined, you can take advantage of such methods to write sensitive code which you would not want to be accessed by any class or interface

- **Differentiate between instance variable and class variable**

Feature	Instance Variable	Class Variable (Static Variable)
Declaration	Declared inside a class but outside any method.	Declared with the `static` keyword.
Memory Allocation	Each instance of the class has its own copy.	Shared among all instances of the class.
Access Modifiers	Can have different access modifiers.	Usually declared as `private` or `public`.
Initialization	Initialized when an object is created.	Initialized when the class is loaded.
Usage	Pertains to the specific instance of the class.	Shared among all instances of the class.
Keyword	No specific keyword used.	Declared using the `static` keyword.
Accessed Using	Accessed using an object of the class.	Accessed using the class name (ClassName.variable).

- **Mention the differences between constructor and destructor. How can you use multiple constructors in a class?**

Feature	Constructor	Destructor (Not present in Java)
Purpose	Initializes an object when created.	Not applicable; Java relies on garbage collection.
Name	Same as the class name.	Not applicable; Java doesn't have destructors.
Invocation	Automatically invoked when an object is created.	Not applicable; No direct equivalent in Java.

Feature	Constructor	Destructor (Not present in Java)
Usage	Used for initialization tasks.	Not present in Java; memory cleanup is handled by garbage collection.
Multiple Constructors	Can have multiple constructors with different parameter lists (overloading).	Achieved using constructor overloading.

Multiple constructors in a class can be used through constructor overloading, where you define more than one constructor with different parameter lists.

```
public class MyClass {
    private int value;

    // Default constructor
    public MyClass() {
        value = 0;
    }

    // Parameterized constructor
    public MyClass(int initialValue) {
        value = initialValue;
    }

    // Another parameterized constructor
    public MyClass(String stringValue) {
        // Convert the string to an integer and set the val
ue
        value = Integer.parseInt(stringValue);
    }

    // Getter method
    public int getValue() {
        return value;
    }

    public static void main(String[] args) {
        // Using different constructors
        MyClass obj1 = new MyClass();          // Default c
onstructor
```

```
        MyClass obj2 = new MyClass(42);         // Parameter
ized constructor with an int
        MyClass obj3 = new MyClass("123");      // Parameter
ized constructor with a string

        // Accessing values
        System.out.println("Value of obj1: " + obj1.getValu
e());
        System.out.println("Value of obj2: " + obj2.getValu
e());
        System.out.println("Value of obj3: " + obj3.getValu
e());
    }
}
```

- **Demonstrate partial implementation of an interface with proper example.**

In Java, a class implementing an interface can choose to provide a partial implementation by declaring the class as abstract and implementing only some of the interface methods. Other methods can be left unimplemented, to be implemented by concrete subclasses

```
// Interface with multiple methods
interface MyInterface {
    void method1();  // Unimplemented method
    void method2();  // Unimplemented method
    void method3();  // Unimplemented method
}

// Abstract class implementing the interface partially
abstract class MyAbstractClass implements MyInterface {
    public void method1() {
        System.out.println("Implemented method1 in the abst
ract class");
    }

    // method2 is not implemented here
```

```
    // method3 is not implemented here
}

// Concrete subclass providing the remaining implementation
s
class MyConcreteClass extends MyAbstractClass {

    public void method2() {
        System.out.println("Implemented method2 in the conc
rete class");
    }


    public void method3() {
        System.out.println("Implemented method3 in the conc
rete class");
    }
}

public class InterfacePartialImplementationExample {
    public static void main(String[] args) {
        MyConcreteClass myObject = new MyConcreteClass();

        // Calling the implemented methods
        myObject.method1();
        myObject.method2();
        myObject.method3();
    }
}
```

- **What is Exception? There are three categories of exceptions in Java. Explain each of them with proper example.**

An exception in Java is an event that disrupts the normal flow of the program's instructions during execution.

- **Checked Exceptions (Compile-time Exceptions):** These exceptions are checked at compile-time. The compiler ensures that these exceptions are either handled using `try-catch` blocks or declared in the method's `throws` clause.

```
public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            Scanner scanner = new Scanner(file);
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.ge
tMessage());
        }
    }
}
```

- **Unchecked Exceptions (Runtime Exceptions):** These exceptions are not checked at compile-time and are subclassed from `RuntimeException`. They usually indicate programming bugs or logical errors.

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        System.out.println(numbers[4]);  // ArrayIn
dexOutOfBoundsException
    }
}
```

- **Error:** Errors are serious, often unrecoverable problems that occur at runtime. They are typically outside the control of the application and should not be caught or handled by the program.

```
public class ErrorExample {
    public static void main(String[] args) {
        try {
            // .....
```

```
        } catch (OutOfMemoryError e) {

            System.out.println("Out of memory erro
r: " + e.getMessage());
        }
    }
}
```

- **Discuss about the restrictions of a static method. Demonstrate static variables, methods and blocks.**

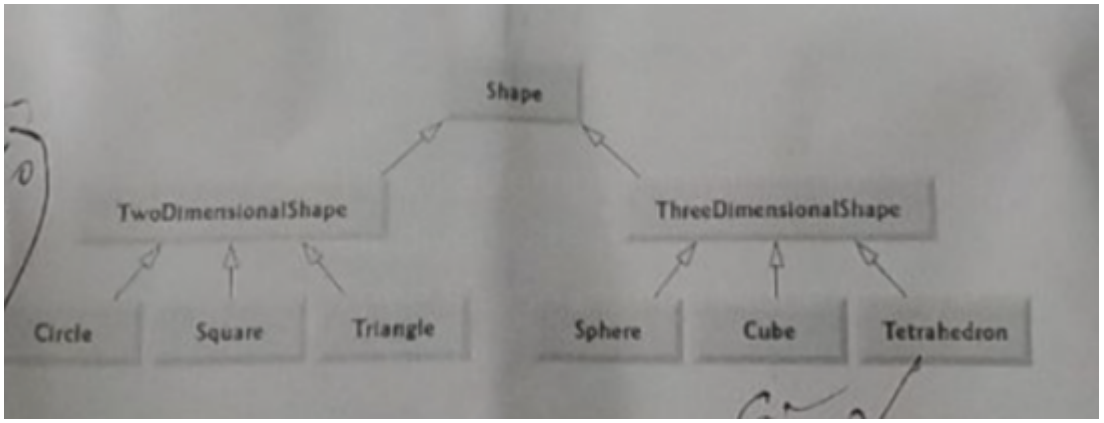
Restrictions of a Static Method:

1. **No Access to Instance Members:**
    - A static method cannot directly access or refer to instance variables or instance methods, as it operates at the class level rather than on a specific instance.
2. **Cannot Use `this` Keyword:**
    - The `this` keyword is not applicable within a static method since there is no specific instance associated with it.
3. **No Direct Access to Non-Static Members:**
    - A static method cannot directly access non-static (instance) members of a class. It can only access other static members directly.

```
public class StaticExample {
    // Static variable (class variable)
    private static int staticVariable = 0;

    // Instance variable
    private int instanceVariable;

    // Static block - executed when the class is lo
aded
    static {
        System.out.println("Static block execute
d");
```

```
        // Can initialize static variables here
        staticVariable = 10;
    }

    // Static method
    public static void staticMethod() {
        System.out.println("Static method called");
        // Can only access static members directly
        System.out.println("Static Variable: " + st
aticVariable);
        // Cannot access instanceVariable directly
from a static method
        // System.out.println("Instance Variable: "
+ instanceVariable); // Compilation error
    }

    // Instance method
    public void instanceMethod() {
        System.out.println("Instance method calle
d");
        // Can access both static and instance memb
ers
        System.out.println("Static Variable: " + st
aticVariable);
        System.out.println("Instance Variable: " +
instanceVariable);
    }

    public static void main(String[] args) {
        // Calling static method without creating a
n instance
        staticMethod();

        // Creating an instance of the class
        StaticExample obj = new StaticExample();
        // Calling instance method
        obj.instanceMethod();
```

```
    }
}
```




```
class shape{
    public double w, h;
    shape(double w){
        this.w =w ;
    }
    shape(double w, double h){
        this.w = w; this.h = h;
    }
}

abstract class TwoDShape extends shape{
    TwoDShape(double w, double h){
        super(w, h);
    }
    TwoDShape(double w){
        super(w);
    }
    abstract double getArea();
};

class Square extends TwoDShape{
    Square(double w, double h){
```

```
        super(w, h);
    }
    double getArea(){
        return w*h;
    }
}

class Circle extends TwoDShape{
    Circle(double w){
        super(w);
    }
    double getArea(){
        return 3.14*w*w;
    }
}
class Triangle extends TwoDShape{
    Triangle(double w, double h){
        super(w, h);
    }
    double getArea(){
        return 0.5*w*h;
    }
}

abstract class ThreeDShape extends shape{
    ThreeDShape(double w){
        super(w);
    }
    ThreeDShape(double w, double h){
       super(w, h);
    }
    abstract double getArea();
    abstract double getVolume();
}

class Sphere extends ThreeDShape {
    Sphere(double w){
        super(w);
```

```
    }
    double getArea(){
        return 4*3.14*w*w;
    }
    double getVolume(){
        return (4/3)*3.14*w*w*w;
    }
}

class Cube extends ThreeDShape{
    Cube(double w){
        super(w);
    }
    double getArea(){
        return 6*w*w;
    }
    double getVolume(){
        return w*w*w;
    }
}

class Tetrahedron extends ThreeDShape{
    Tetrahedron(double w){
        super(w);
    }
    double getArea(){
        return Math.sqrt(3)*w*w;
    }
    double getVolume(){
        return (Math.sqrt(2)/12)*w*w*w;
    }
};


class HelloWorld {
    public static void main(String[] args) {
       Triangle tr = new Triangle(5, 6);
       System.out.println(tr.getArea());
```

```
    }
}
```

- **Differences between runtime and compile-time polymorphism**

Sr. No.	Key	Compile-time polymorphism	Runtime polymorphism
1	Basic	Compile time polymorphism means binding is occuring at compile time	R un time polymorphism where at run time we came to know which method is going to invoke
2	Static/DynamicBinding	It can be achieved through static binding	It can be achieved through dynamic binding
4.	Inheritance	Inheritance is not involved	Inheritance is involved
5	Example	Method overloading is an example of compile time polymorphism	Method overriding is an example of runtime polymorphism

Example of Compile-time Polymorphism

```
public class Main {
   public static void main(String args[]) {
      CompileTimePloymorphismExample obj = new CompileTimeP
loymorphismExample();
      obj.display();
      obj.display("Polymorphism");
   }
}
class CompileTimePloymorphismExample {
   void display() {
      System.out.println("In Display without parameter");
   }
   void display(String value) {
```

```
      System.out.println("In Display with parameter" + valu
e);
   }
}
```

Example of Runtime Polymorphism

```
public class Main {
   public static void main(String args[]) {
      RunTimePolymorphismParentClassExample obj = new RunTi
mePolymorphismSubClassExample();
      obj.display();
   }
}

class RunTimePolymorphismParentClassExample {
   public void display() {
      System.out.println("Overridden Method");
   }
}

public class RunTimePolymorphismSubClassExample extends Run
TimePolymorphismParentExample {

   public void display() {
      System.out.println("Overriding Method");
   }
}
```