**OOPM NOTES - JAVA**

What is Functional Programming?

It is a programming paradigm where:

* Functions are mathematical functions: same input → same output.
* Uses immutability (data isn’t modified, new data is returned): functions don’t modify global state, they just return new values.
* Functions can be passed as arguments, returned from other functions (first-class citizens).
* Flow: "Transform data by passing it through functions."
* Example: Python

What is Procedural Programming?

It is a programming paradigm where:

* It is based on the concept of procedure call.
* Think of it as step-by-step instructions grouped into procedures (functions).
* Functions can change global or local state (variables).
* Side effects are common (printing, updating values, modifying data).
* Flow: "Do step 1, then call function A to do step 2, then update X, then loop again."
* Example: C/C++

What is Object Oriented Programming?

* Object-Oriented Programming (OOP) is a programming paradigm where we structure code around objects rather than functions and logic.
* Each object has state (data/attributes) and behavior (functions/methods)
* The core idea of OOPs is to bind data and the functions that operate on it, preventing unauthorized access from other parts of the code.
* The four key principles are encapsulation, abstraction, inheritance, and polymorphism.
* Follows the DRY (Don't Repeat Yourself) Principle, ensuring that common logic is written once (e.g., in parent classes or utility methods) and reused throughout the application.
* The goal is to make software modular, reusable, easier to maintain and mimic real world systems.
* For example, in a banking system, you might have a BankAccount class with attributes like balance and methods like deposit or withdraw. Different types of accounts (savings, current) can inherit from it and override behaviors as needed.
* Example: Java, Python, C++

Difference between C, C++, Java, python?

Points:

Programming Paradigm - Procedural, procedural+OOP, OOP, OOP+functional

Code Translation - Compiled to machine code, compiled to machine code, compiled to bytecode->jvm->interpreted, interpreted+bytecode compiled

Preprocessor directives – yes, yes, no, no

Header files – yes, yes, no, no

Memory Allocation - malloc calloc free, new, new+garbagecollection, new+garbagecollection

Exception handling - no, try-catch, try-catch-finally, try-except-finally

Multithreading- no no yes yes

Inheritance - no yes yes yes

What is Java?

Java is a programming language and a platform. It is a high level, robust, object-oriented and secure programming language. Platform: Any hardware or software environment in which a program runs, is known as a platform.

Features of Java?

Simple (easy to learn, simple syntax)

Object Oriented (• Object, Class • Inheritance • Polymorphism • Abstraction • Encapsulation)

Secure & Robust (No explicit pointers, runs inside virtual machine, exception handling, robust memory management)

Platform independent

Interpreted

Portable

High Performance (faster than other interpreted languages because Java bytecode is "close" to native code. slower than a compiled language because java is an interpreted language)

Multithreaded (deal with many tasks at once by defining multiple threads)

Architecture neutral (no implementation-based features, size of primitive types is fixed)

Distributed (facilitates users to create distributed applications)

Dynamic (supports dynamic loading of classes. Classes are loaded only when they are needed)

What is an Object?

An object is an instance of a class. It means that class is a template/blueprint from which objects are created. Object is both a physical as well as logical entity. They have three characteristics:

State: represents data of an object (variables)

Behaviour: Represents functionality of the object (methods)

Identity: Object identity is typically a unique id which is not shown to the user but used internally by the JVM.

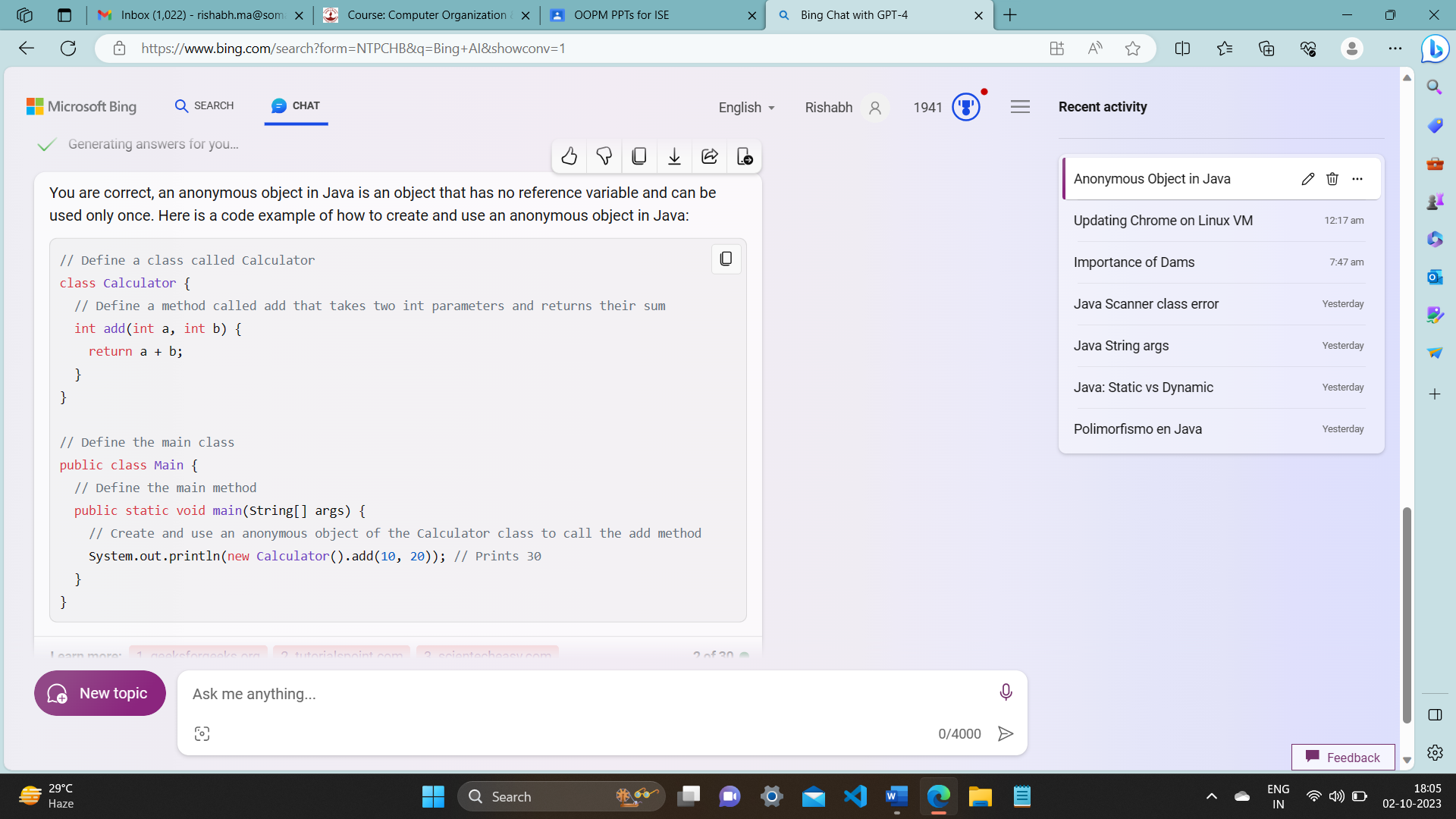
Objects are created using:

new keyword - new keyword is used to allocate memory at runtime.

newInstance() method

clone() method

What is an anonymous object?



What is a Class?

A class defines a group of objects having common properties. It is a template/blueprint from which objects are created. It is a logical entity. Classes can contain variables, methods, constructors, and interface.

//Static Members

Belong to the class, not to any specific object.

Shared by all instances of the class.

Can be accessed without creating an object.

Access using class name

Declared using the static keyword.

Only static members can be accessed directly in a static context (like main).

Static methods cannot access non-static variables directly.

STATIC METHODS CANNOT BE OVERRIDDEN

class Demo {

static int count = 0; //static variable

//static method

static void showCount() {

System.out.println("Count: " + count);

}

}

public class Main {

public static void main(String[] args) {

// Accessing static variable and method without object

Demo.count = 5;

Demo.showCount();

}

}

//Non-Static(Instance) Members

Belong to specific objects, not the class.

Each object has its own copy of non-static variables.

Must create an object to access them.

Each object maintains its own state.

Non-static methods can access static variables but static methods cannot access non-static variables without an object.

class Demo {

int number; // non-static variable

void setNumber(int n) { // non-static method

number = n;

}

void showNumber() {

System.out.println("Number: " + number);

}

}

public class Main {

public static void main(String[] args) {

Demo obj1 = new Demo();

Demo obj2 = new Demo();

obj1.setNumber(10);

obj2.setNumber(20);

obj1.showNumber(); // Number: 10

obj2.showNumber(); // Number: 20

}

}

//example of static non-static memory

class Counter {

static int count = 0; // static variable

int id; // non-static variable

Counter() {

count++; // shared

id = count; // unique per object

}

//static method

static void printCount() {

System.out.println("Total: " + count);

}

//instance method

void printId() {

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

}

}

public class Test {

public static void main(String[] args) {

Counter c1 = new Counter();

Counter c2 = new Counter();

c1.printId(); // ID: 1

c2.printId(); // ID: 2

Counter.printCount(); // Total: 2

}

}

The 4 PILLARS of OOP

**Inheritance:**

Inheritance is a mechanism in which a child class can acquire all the properties and behaviour of the parent class. The class that is inherited from is called the "parent" or "superclass," and the class that inherits is called the "child" or "subclass." It is useful because it helps code reusability and thus prevents the need of rewriting code. You can create new classes based on existing classes so we can reuse the methods and fields of the parent class while defining new methods and fields in the child class. Inheritance is used for method overriding.

Inheritance creates a hierarchical relationship between classes. It represents IS-A relationship or parent-child relationship (dog IS-A animal)

class Employee {

    static int salary = 10000;

}

public class Programmer extends Employee {

    static int bonus = 5000;

    public static void main(String[] args)

    {

        System.out.println("Salary is " + salary);

        System.out.println("Bonus is " + bonus);

    }

}

**Polymorphism:**

Polymorphism is a concept which allows **a single method, or operator to behave differently in different contexts.**

There are two types of polymorphism in java:

Compile time polymorphism (static binding) – resolved at compile time.

It is also called method overloading. Multiple methods with the same name but different parameter lists are defined within the same class. The parameter lists may differ by number of arguments or types of arguments or both. The compiler decides which version of the method to call based on the method signature (name + parameters).

class Calculator

{

int add (int a, int b)

return a + b;

double add (double a, double b, double c)

return a + b + c;

}

Runtime polymorphism (dynamic binding) – resolved at runtime

It is achieved with the help of method overriding. In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass. In runtime polymorphism, the decision about which method to call is made at runtime based on the actual type of the object. The reference type is written as the parentclass and the objecttype is written as the childclass.

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

@override

void sound() {

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

}

void fetch() {

System.out.println("Dog fetches ball");

}

}

public class Main {

public static void main(String[] args) {

Animal a1 = new Dog(); //runtime polymorphism

a1.sound(); //Bark

a1.fetch(); //error

}

}

Dynamic Method Dispatch:

The process by which the correct method is called at runtime is known as dynamic method dispatch. It allows a subclass to provide a specific implementation of a method while still adhering to the method signature in the superclass.

Animal myPet = new Dog();

myPet.makeSound();

// Calls the makeSound method of Dog at runtime

Here, the reference variable **myPet** is of type **Animal**, but it refers to an instance of **Dog**. The method **makeSound** invoked is determined at runtime based on the actual type of the object (runtime polymorphism).

abstract class Bike{

abstract void run();

}

class Honda4 extends Bike {

void run() {

System.out.println("running safely");

}

public static void main(String args[]){

Bike obj = new Honda4();

obj.run();

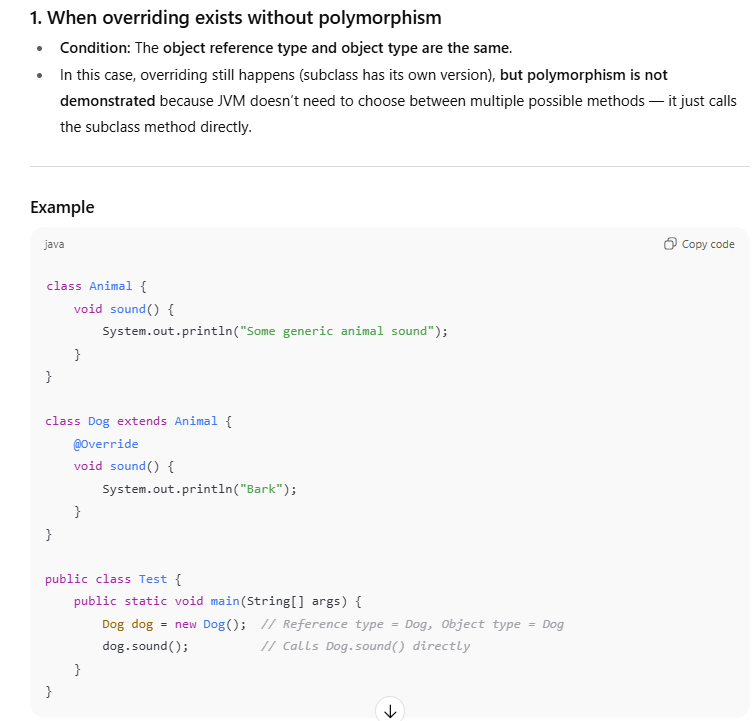
}

}

Here's an analogy: Think of **Bike** as a generic term for any kind of bike, and **Honda4** as a specific brand and model of bike. When you say **Bike obj = new Honda4();**, you're saying, "I have a bike, and specifically, it's a Honda4." This allows you to focus on the common features shared by all bikes, as defined in the **Bike** class, while still being able to use the specific features of a **Honda4** when needed. This concept is a key part of object-oriented programming, promoting flexibility, maintainability, and code reuse.

Method overriding (without polymorphism)

Method overriding in java is when you declare a method in the subclass which already exists in the superclass. This is done so that child class can provide its own implementation to a method that was already defined by the superclass.



Why we write @override ?

It is not necessary but recommended as it tells the compiler that the method is meant to be overriding, makes code readable and also catches mistakes - if u spell the method wrong it wont make a new function but will tell u that u have spelt it wrong.

**Abstraction:**

Abstraction is a way of hiding the complex implementation details from the user and showing only the essential functionality. Example: sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery. Abstraction allows you to focus on what the object does instead of how it does it.

Why Use It?

To reduce complexity

To enforce a contract (what must be done, not how)

To allow flexibility in implementation (different classes can implement the logic differently)

How to achieve in Java:

* **Interfaces**
* **Abstract classes**

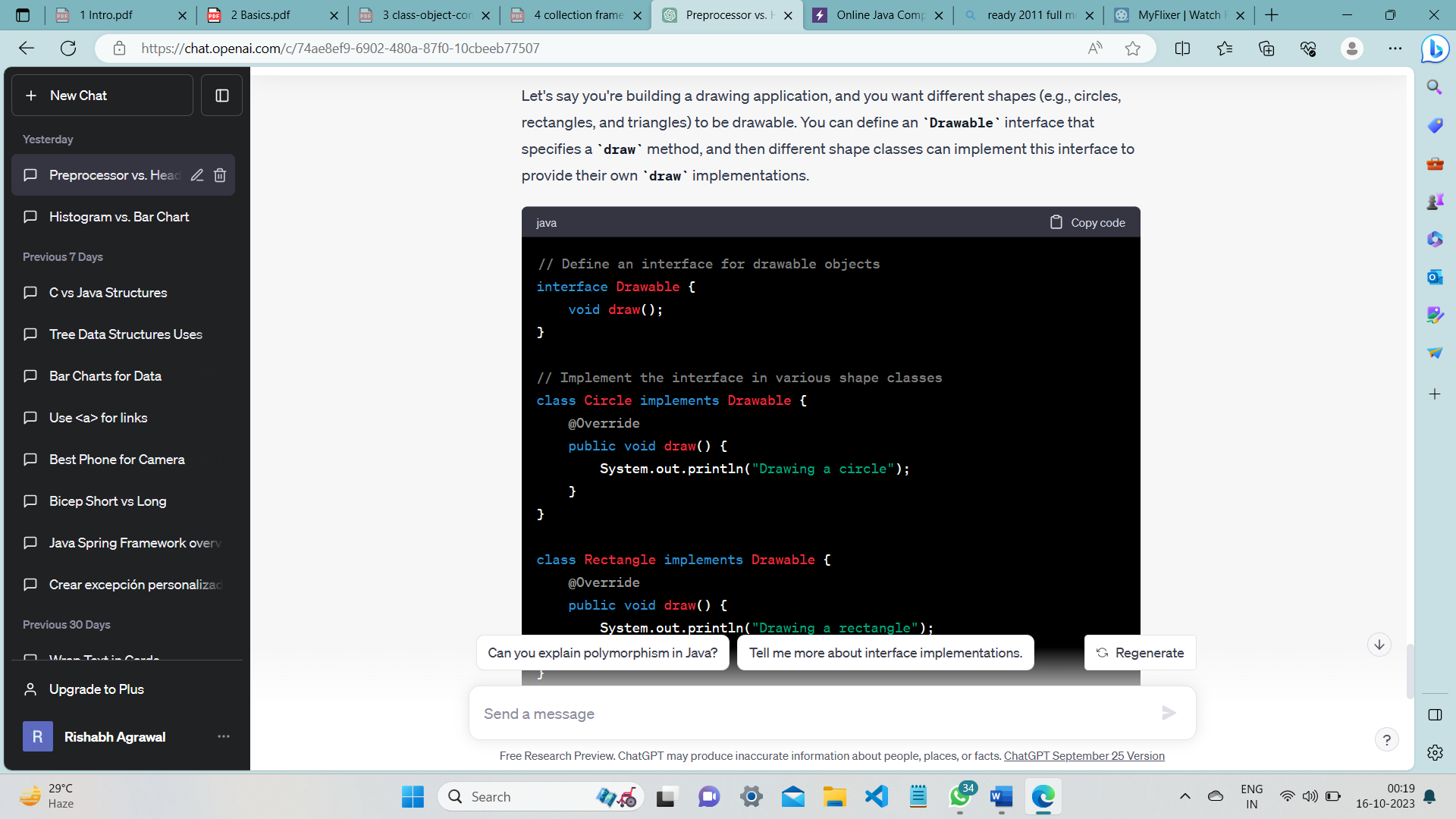
//Java follows camelCase syntax for naming the class(noun) and interface(adjective)

I**nterface**

An interface is a blueprint of a class that defines a set of abstract methods (methods without a body) that any class implementing the interface must provide concrete implementations for. It is similar to an abstract class but ALL methods in an interface must be abstract. An interface is declared using interface keyword and can be implemented by other classes using the implement keyword. They are used to achieve 100% abstraction. A class can extend only 1 class but it can implement multiple interfaces. You cannot create objects directly from an interface. Objects are created from the subclasses that implement the interface. Interfaces end with ‘able’. Example flyable

NOTE:

* Whenever a class implements an interface, they must provide definitions for all the methods in that interface. If the subclass does not provide an implementation for any of the methods declared in the interface, it must be declared as an abstract class itself.
* If an abstract class implements an interface, it need not implement all methods defined in the interface. However, each concrete subclass must implement the methods defined in the interface.
* All methods are abstract and public by default.
* All variables in an interface are are public, static(belonging to class), and final(constants) by default.
* Interfaces cannot contain instance variables.
* Interface cannot have constructers.



Why do we use interface?

* They are used to achieve 100% abstraction.
* They can be used to achieve effects of multiple inheritance which is not allowed in Java.
* Interfaces define a common set of methods that multiple classes can adhere to.

Can one interface inherit another interface?

Yes, one interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

**Abstract class**

A method that has been declared but not defined is an abstract method.

Eg. public abstract void draw(int size);

Any class that has an abstract method is known as an abstract class. An abstract class must be defined with the keyword abstract.

Eg. abstract class MyClass {...}

Absract class can have both abstract and non abstract methods

AN ABSTRACT CLASS CANNOT BE INSTANTIATED EVEN IF IT HAS ALL CONCRETE METHODS (You cannot create an object of an abstract class).

You can extend an abstract class. If the extended subclass defines all the abstract methods present in the superclass then it can be instantiated otherwise the subclass must be abstract too.

Abstract class can have constructors. When a subclass is instantiated, the constructor of the parent abstract class is executed.(through super) Constructors are called from the top of the inheritance chain (the most superclass) down to the bottom (the most subclass).

You can have abstract class without any abstract methods (this is useful to prevent instantiation of that class)

Abstract class can have static (belong to class) and non static (belong to object) methods and variables.

ABSTRACT METHODS CANNOT BE STATIC AS ABSTRACT METHODS NEED TO BE OVERRIDEN AND STATIC METHODS CANNOT BE OVERRIDEN.

EXAMPLE: Abstract class

abstract class Employee {

    abstract void displayTitle();

}

class Programmer extends Employee {

    void displayTitle() {

        System.out.println("I am a programmer");

    }

    public static void main(String[] args)

    {

        // Since we have defined all the abstract methods of the superclass, the subclass can be instantiated

        Programmer p = new Programmer();

        p.displayTitle();

    }

}

EXAMPLE: Abstract class constructor called through child class

abstract class BankAccount {

String accountNumber;

// Constructor of abstract class

BankAccount(String accountNumber) {

this.accountNumber = accountNumber;

System.out.println("BankAccount constructor called");

}

// Abstract method (must be implemented by subclasses)

abstract void prntAccountType();

}

class SavingsAccount extends BankAccount {

double interestRate;

SavingsAccount(String accountNumber, double interestRate) {

super(accountNumber); // ✅ calls abstract class constructor

this.interestRate = interestRate;

System.out.println("SavingsAccount constructor called");

}

@Override

void prntAccountType() {

System.out.println("This is a Savings Account with interest rate " + interestRate + "%");

}

}

public class Main {

public static void main(String[] args) {

SavingsAccount sa = new SavingsAccount("SA123", 4.5);

sa.pritAaccountType();

}

}

//OUTPUT

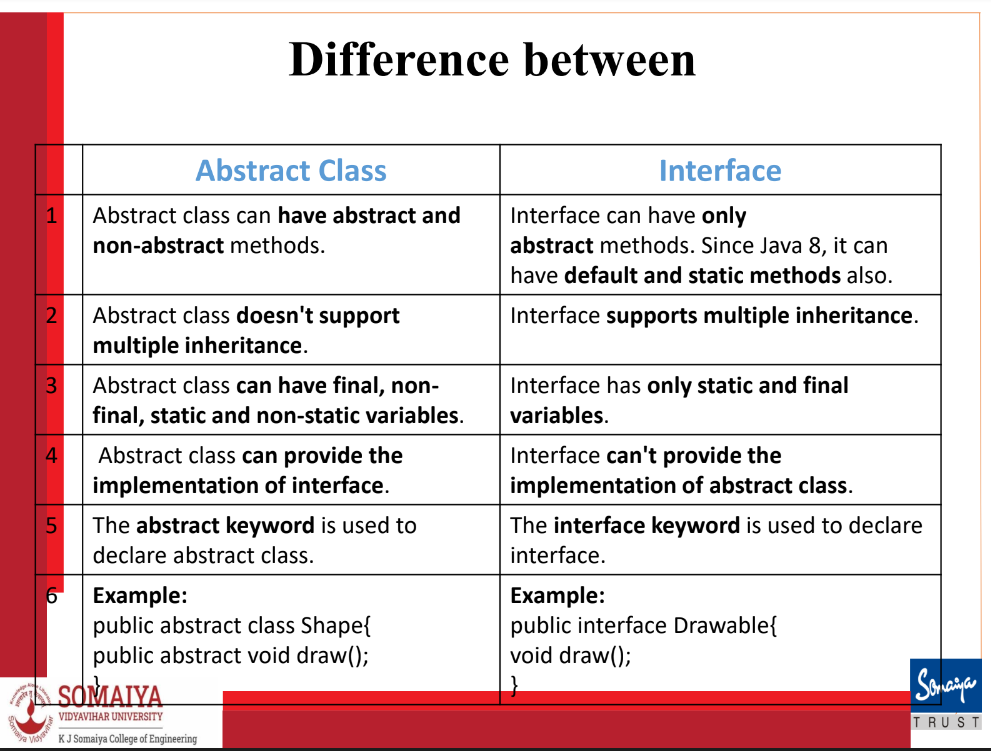
BankAccount constructor called

SavingsAccount constructor called

This is a Savings Account with interest rate 4.5%

Why static methods cannot be overridden ?

1. Static methods belong to the class, not the object
   * Overriding is an object-level concept → it requires runtime polymorphism (method resolved based on the actual object).
   * But static methods are class-level → they don’t depend on any object.
   * Hence, they cannot participate in overriding.
2. Static methods are resolved at compile time
   * For overriding, the JVM must wait until runtime to decide which method to call.
   * For static methods, the compiler already decides which method to call based on the reference type → so no runtime polymorphism happens.
3. What happens instead? → Method Hiding
   * If a subclass declares a static method with the same signature as in the parent class, it hides the parent method.
   * The difference: which method is called depends on the reference type, not the object type.



**Encapsulation**:

Encapsulation also called data hiding in Java is a process of wrapping data and methods together in a single unit and controlling access to it via access specifiers. (you can make it read only / write only, add validation, etc)

Example: A BankAccount class hides the balance (private variable) and uses getter/setter methods to access or modify it safely.

//getters and setters

public class User {

private String name; // private field

// Getter (read access)

public String getName() {

return name;

}

// Setter (write access)

public void setName(String name)

{

// Optional: validation

if (name != null && !name.isEmpty())

{

this.name = name;

}

}

}

//usage

User u = new User();

u.setName("Alice");

System.out.println(u.getName());

How is java platform independent?

Java is often referred to as platform-independent because of its "Write Once, Run Anywhere" (WORA) capability.

There is a 2-step code translation process. Java is both compiled and interpreted

Source code is compiled to machine independent byte code by javac compiler. It can be executed on any platform (Windows, Linux, Mac etc)

Byte code is interpreted as machine code by java interpreter i.e., JVM

What is multithreading?

Multithreading in Java refers to the capability of a single program to execute multiple threads concurrently. A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. An advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Why use Multithreading?

1. Better CPU utilization – Multiple threads can run on multiple cores.
2. Responsiveness – GUI applications remain responsive while tasks run in background threads.
3. Parallel execution – Tasks like file I/O, network calls, computations can run concurrently.

How is java robust? (stable and strong)

It has strong automatic memory management via garbage collection

It has no explicit pointers.

Java Programs run inside a virtual machine.

There is an exception handling mechanism in java.

How is java architecture neutral?  
Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

How is java high performance?

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to machine code. Java is an interpreted language that is why it is a little slower than compiled languages (e.g. C++).

Java is a dynamic language. Explain

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. Java supports dynamic compilation.

What are the ways to read input in Java?

* Command Line Interpreter – String[] args

Use args.length – without bracket

Use args[i]

* Scanner Class

Located in java.util package

Scanner sc = new Scanner(System.in);

int num = sc.nextInt();

double d = sc.nextDouble();

String str = next(); //reads upto a whitespace

String str = sc.nextLine(); //reads upto newline char

char a= sc.next().charAt(0); //to get char

sc.close();

* Bufferedreader Class

Located in java.io package.

**Static (Class) Variable**

* A static variable, also called a class variable, is declared with the keyword static inside a class but outside any method.
* It is shared by all instances of the class, meaning there is only one copy of the variable for the entire class.
* It can be accessed without creating an object, using the class name, which makes it memory-efficient.
* Example usage : If one object increases the variable by 1 and another object also increases it by 1, the final value reflects both increments (e.g., 2).
* Static variables are initialized when the class is loaded.

**Instance Variable**

* An instance variable is declared inside a class but outside any method, without the static keyword.
* Each object of the class has its own copy of the variable.
* It can have different values for different objects of the same class.
* Example usage: If one object increments the variable, the change is reflected only in that object; other objects’ variables remain unchanged.
* Instance variables are initialized when an object of the class is created and can be accessed using the object name.

What is a method?

A method refers to a function associated with a class. Since all functions in java are associated with a class, method is synonymous too function. It is collection of statements grouped together to perform a specific operation. Modifier is optional.

Syntax:

Modifier returnType nameOfMethod (Parameter List)

{

//body

}

What is a static method in java?

A static method belongs to the class rather than an object of the class. It is declared with the keyword static. It can be called without creating an object of the class. It is called using the classname directly. Static methods can access and modify static data members but they cannot modify non static data members or call a non-static method. You cannot use the keywords “this” and “super”.

class Calculator {

// static method

static int add(int a, int b) {

return a + b;

}

}

public class Test {

public static void main(String[] args) {

// Call static method without creating object

int sum = Calculator.add(10, 20);

System.out.println("Sum: " + sum);

}

}

What is static block?

Static block s used to initialize the static data member. It is executed before main method at the time of class loading. So this is one of the way to execute a program without main() method.

class A2

{

static{System.out.println("static block is invoked")};

}

public static void main(String args[])

{

System.out.println("Hello main");

}

OUTPUT:

static block is invoked

Hello main

**Operators in java**

Unary Operator +, -, ++, --

Arithmetic Operator +, -, /, \*, %

Relational Operator >, >=, <, <=, ==, !=

Logical Operator &&, ||, !

Assignment Operator =, +=, -=, \*=, /=, %=

Shift Operator **<<** (left shift), **>>** (right shift)

Bitwise Operator &, |(OR), ^(XOR)

Ternary Operator ?:

What are access specifiers/modifiers in java?

The access modifiers in Java specifies the accessibility of a variable, method, constructor, interface or class. We can change the access level by applying the access modifier on it.

**Public**: Class, Method, Constructor or Interface which are public can be accessed from any other class in the java universe i.e., both within package and outside package. However if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses

public int score;

**Protected**: The access level of methods, variables and constructors declared protected is all classes within the package and subclasses outside the package. Cannot be applied to class and interface. Methods and fields in an interface cannot be declared protected. Protected access gives the subclass a chance to use the method or variable, while preventing a nonrelated class from trying to use it.

protected int score;

**Default (Package-pri)**: Default access modifier is applied when we do not explicitly declare an access modifier. The variable or method which has default access level can be accessed from any class within the same package.

int score;

**Private**: Methods, Variables and Constructors which are private can only be accessed from within the class itself. Classes and interfaces cannot be private. This modifier is the most restrictive access level. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world. It provides true encapsulation

private int score;

Within Class Within Package Outside package(only in subclass) Outside Package

public ✅ Yes ✅ Yes ✅ Yes ✅ Yes

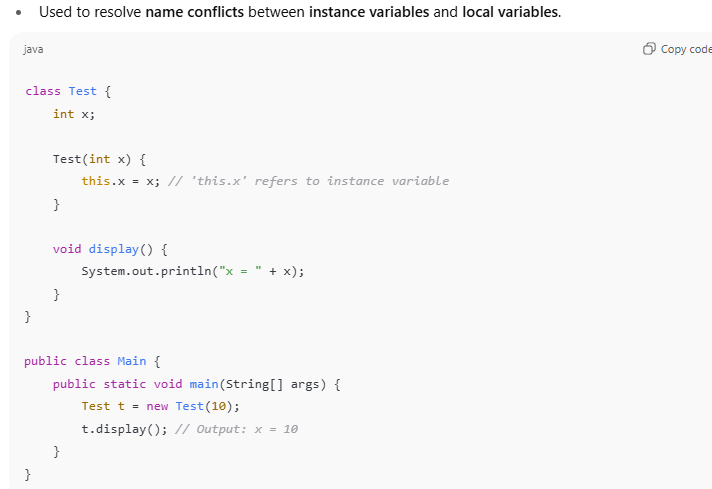
protected ✅ Yes ✅ Yes ✅ Yes ❌ No

default-no keyword ✅ Yes ✅ Yes ❌ No ❌ No

private ✅ Yes ❌ No ❌ No ❌ No

What is ‘this’ keyword in java?

The ‘this’ keyword in Java is a reference variable that refers to the current instance of a class. Using this you can refer the members of a class such as constructors, variables and methods. It can be used for various purposes, especially when there is a name conflict between the local variables and the instance variables



1. this keyword can be used to refer current class instance

2. this keyword can be used to invoke current class constructor.

3. this keyword can be used to call another method of current class (implicitly)

4. this can be passed to pass current object as an argument in the method call.

5. this keyword can also be used to return the current class object.

What is a constructor?

* A constructor initializes an object when it is created.
* It has the same name as the class name and has no return type.
* A constructor is used to give initial values to the instance variables defined by the class and perform any other startup procedures required.
* All classes have constructors, whether you define one or not as java automatically provides a default constructor that initializes all the variables to 0, null, etc

What are the types of constructors in java?

There are two types of constructors:

**Default Constructor:**

Default constructor provides the default values to the object like 0, NULL etc. If no constructor is defined for a class, default constructor is used to initialize the object.

**Parameterised Constructor:**

It is a constructor that we define.It has parameters. It is used to provide different values to different objects.

\*There is no copy constructor provided in java. But, we can copy the values of one object to another :

* By manually using a constructor
* By assigning the values of one object into another
* By clone() method of Object class

How to initialize using method?

class Rectangle{

int length;

int width;

void insert(int l,int w){

length=l;

width=w;

}

public static void main(String args[]){

Rectangle r1=new Rectangle();

r2=new Rectangle();

r1.insert(11,5);

r2.insert(3,15);

How to initialize using constructor?

class Rectangle {

int length;

int width;

Rectangle(int l, int w) {

// Assign the parameters to the instance variables

length = l;

width = w;

}

public static void main(String args[]) {

// Create two objects of the Rectangle class using the constructor

Rectangle r1 = new Rectangle(11, 5);

Rectangle r2 = new Rectangle(3, 15);

What is constructor overloading?

Constructor overloading in Java is a technique that allows a class to have more than one constructor with different parameters. The compiler distinguishes between the overloaded constructors by looking at the number, type and order of the parameters.

Here is an example of a class called Rectangle that has three overloaded constructors:

// Define a class called Rectangle

class Rectangle {

// Declare two instance variables: length and width

int length;

int width;

// Define a constructor that takes no parameters and sets the length and width to zero

Rectangle() {

length = 0;

width = 0;

}

// Define a constructor that takes one parameter and sets the length and width to that value

Rectangle(int side) {

length = side;

width = side;

}

// Define a constructor that takes two parameters and sets the length and width to those values

Rectangle(int length, int width) {

this.length = length;

this.width = width;

}

}

// Create an object of the Rectangle class using the default constructor

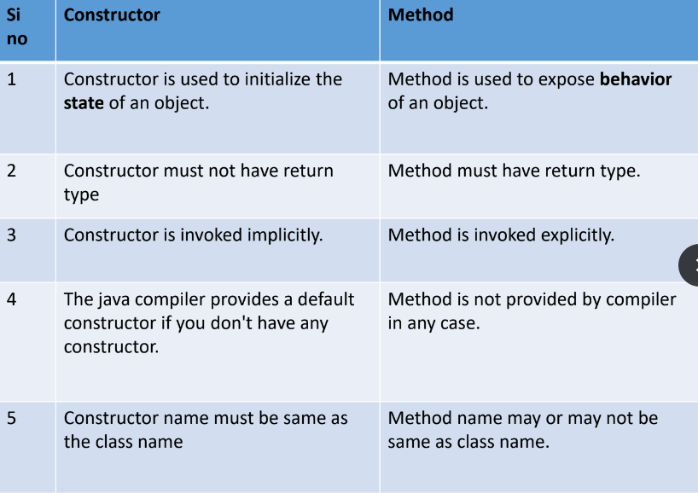
Rectangle r1 = new Rectangle();

// Create an object of the Rectangle class using the constructor with one parameter

Rectangle r2 = new Rectangle(5);

// Create an object of the Rectangle class using the constructor with two parameters

Rectangle r3 = new Rectangle(3, 4);



What is a destructor?

A destructor is a special member function of a class that is executed whenever object is deleted. It has the same name as the class name and has the tilde (~) as a prefix. It has no arguments or return types. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc. Java does not have destructors. It uses garbage collection.

What are arrays in Java?

Arrays are objects. They are a group of elements that have contiguous memory location. Java array is an object that contains elements of homogenous data type. We can store only fixed number of elements in a java array. They are fast since they are low level.

There are two types of array.

* Single Dimensional Array

int[] numbers = {1, 2, 3, 4, 5};

int[] numbers = new int[5];

* Multidimensional Array

int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[][] matrix = new int[3][3];

What is Jagged Array?

It is a new feature supported by Java, where column size varies, i.e. each row may have varying length.

10 20 30

11 22 22 33 44

77 88

int twoD[][] = new int[3][];

twoD[0] = new int[3];

twoD[1] = new int[5];

twoD[2] = new int[2];

What is array of objects?

You can create an array of objects to hold instances of a class. Like regular arrays, arrays of objects have a fixed size once created. It allows direct access to elements based on index.

class Groceries{

String name;

double price;

int date;

Groceries(String name, double price, int date){

this.name=name;

this.price=price;

this.date=date;

}

}

class Main{

public static void main(String args[]){

Scanner sc=new Scanner(System.in);

System.out.println("enter number of groceries");

int n=sc.nextInt();

Groceries arr[]=new Groceries[n];

for(int i=0;i<n;i++){

System.out.println("enter name");

String name=sc.next();

System.out.println("enter price");

double price =sc.nextDouble();

System.out.println("enter date");

int date=sc.nextInt();

arr[i]=new Groceries(name, price,date);

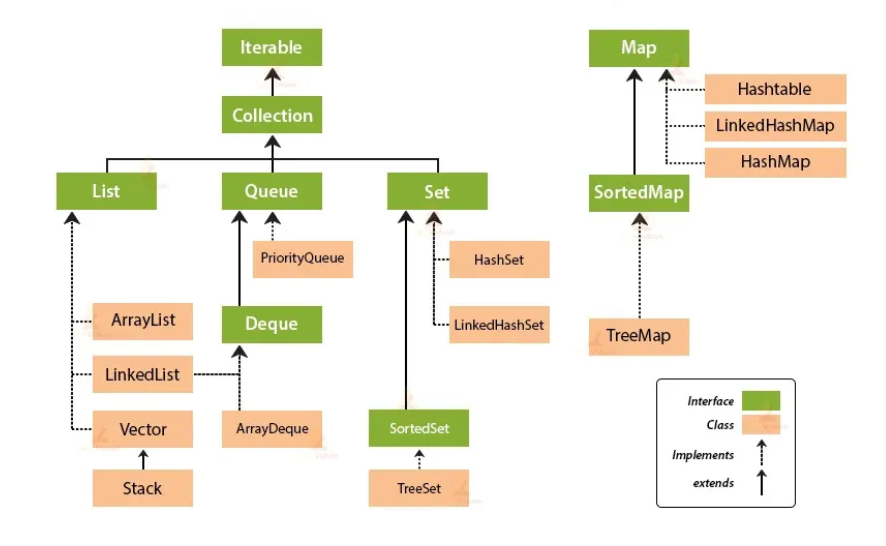
}

}

}

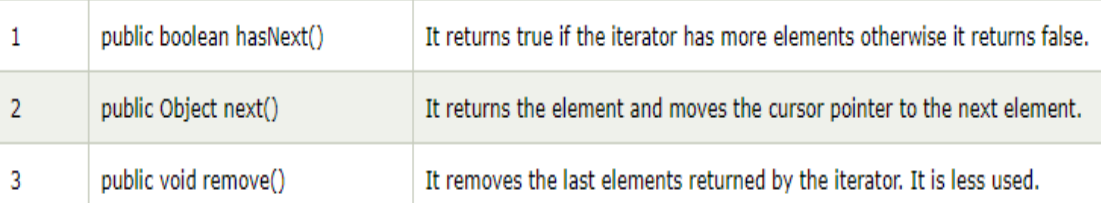
What is the Java Collection Framework?

The Java Collections Framework is a set of classes, interfaces and algorithms in Java that provide a standardized architecture to store, organize, and manipulate groups of objects. It is part of the java.util package. It helps to achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion. It provides a unified and consistent interface for working with different types of collections, making it easier for developers to work with data structures without having to worry about the underlying implementation details and memory management



What is iterator interface?

Iterator interface provides the facility of iterating the elements in a forward direction only.



What is collection interface?

Root Interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. Collection interface builds the foundation on which the collection framework depends.

Methods unique to collection interface

1. **boolean add(E element)**: Adds the specified element to the collection.
2. **boolean addAll(Collection<? extends E> c)**: Adds all elements from the specified collection to the collection.
3. **boolean remove(Object o)**: Removes the first occurrence of the specified element from the collection.
4. **boolean removeAll(Collection<?> c)**: Removes all elements from the collection that are contained in the specified collection.
5. **boolean retainAll(Collection<?> c)**: Retains only the elements in the collection that are contained in the specified collection.
6. **void clear()**: Removes all elements from the collection.
7. **boolean contains(Object element)**: Returns **true** if the collection contains the specified element.
8. **boolean containsAll(Collection<?> c)**: Returns **true** if the collection contains all elements in the specified collection.
9. **int size()**: Returns the number of elements in the collection.
10. **boolean isEmpty()**: Returns **true** if the collection contains no elements.
11. **Object[]name toArray()**: Returns an array containing all of the elements in the collection.
12. **boolean equals(Object o)**: Compares the specified object with this collection for equality.
13. **Iterator<E> iterator()**: Returns an iterator over the elements in the collection.

What is List Interface?

It extends the collection interface.

It is an ordered collection and can have duplicate values.

Common implementing classes include ArrayList, LinkedList, and Vector.

To instantiate the List interface, we must use :

List <data-type> list1= new ArrayList();

List <data-type> list2 = new LinkedList();

List <data-type> list3 = new Vector();

List <data-type> list4 = new Stack();

Methods unique to list interface

1. **void add(int index, E element)**: Inserts the specified element at the specified position in the list.
2. **boolean addAll(int index, Collection<? extends E> c)**: Inserts all elements in the specified collection into the list at the specified position.
3. **E remove(int index)**: Removes the element at the specified position in the list.
4. **E get(int index)**: Returns the element at the specified position in the list.
5. **E set(int index, E element)**: Replaces the element at the specified position in the list with the specified element.
6. **int indexOf(Object o)**: Returns the index of the first occurrence of the specified element in the list.
7. **int lastIndexOf(Object o)**: Returns the index of the last occurrence of the specified element in the list.
8. **List<E>name subList(int fromIndex, int toIndex)**: Returns a view of the portion of the list between the specified **fromIndex** (inclusive) and **toIndex** (exclusive).
9. **ListIterator<E> listIterator()**: Returns a list iterator over the elements in the list.
10. **ListIterator<E> listIterator(int index)**: Returns a list iterator over the elements in the list, starting at the specified position.

//In list interface we can print entire list directly.

List list1 = new List();

System.out.println(list1).

Output- [a,b,c,d]

Which is correct from below 2 ?

LinkedList <data-type> list1 = new LinkedList();

List <data-type> list2 = new LinkedList();

Both of the declarations you provided are correct, but they are used in slightly different contexts:

1. Using the LinkedList Type Explicitly:

In this declaration, you are explicitly stating that list2 is of type LinkedList. While this is syntactically correct and might be required if you need to use specific methods or properties of LinkedList, it's generally recommended to use the more flexible list interface as the type.

1. Using the List Interface:

In this declaration, you are using the List interface as the type. This is often preferred because it allows you to switch the underlying implementation (e.g., from LinkedList to ArrayList) without changing the rest of your code. This is known as coding to the interface, and it promotes flexibility and easier maintenance.

list2 = new ArrayList<>(); // Change the implementation to ArrayList

So, in general, the second declaration is more flexible and adheres to good coding practices. It's often recommended to use interfaces when declaring variables to allow for easier changes in the future. The underlying implementation (LinkedList in this case) can be swapped without affecting the rest of the code as long as you only use methods declared in the List interface.

What is ArrayList ?

It implements the List interface so we can use all the methods of List interface here.

Java ArrayList class uses a dynamic array for storing the elements (resizable)

We can add or remove elements anytime, much more flexible than the traditional array.

Elements can be accessed directly by using the index.

ArrayList can have the duplicate elements.

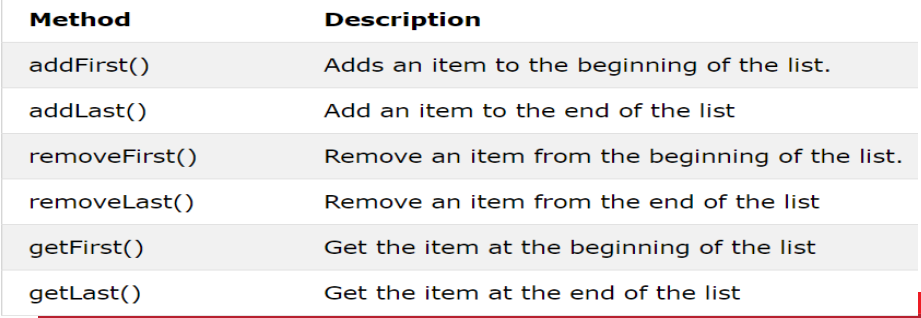
It is found in the java.util package.

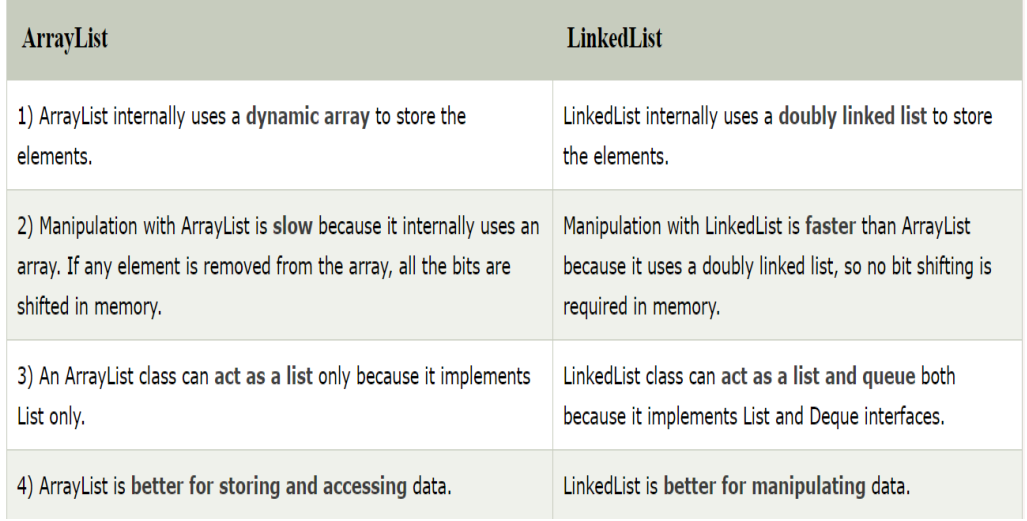
ArrayList increments by 50% of the current array size if the number of elements exceeds its capacity. (Initial capacity is 10)

What is LinkedList?

* Implements the List and Deque interfaces.
* It is a Doubly-linked list.
* Suitable for frequent insertions and removals.

Methods unique to LinkedList



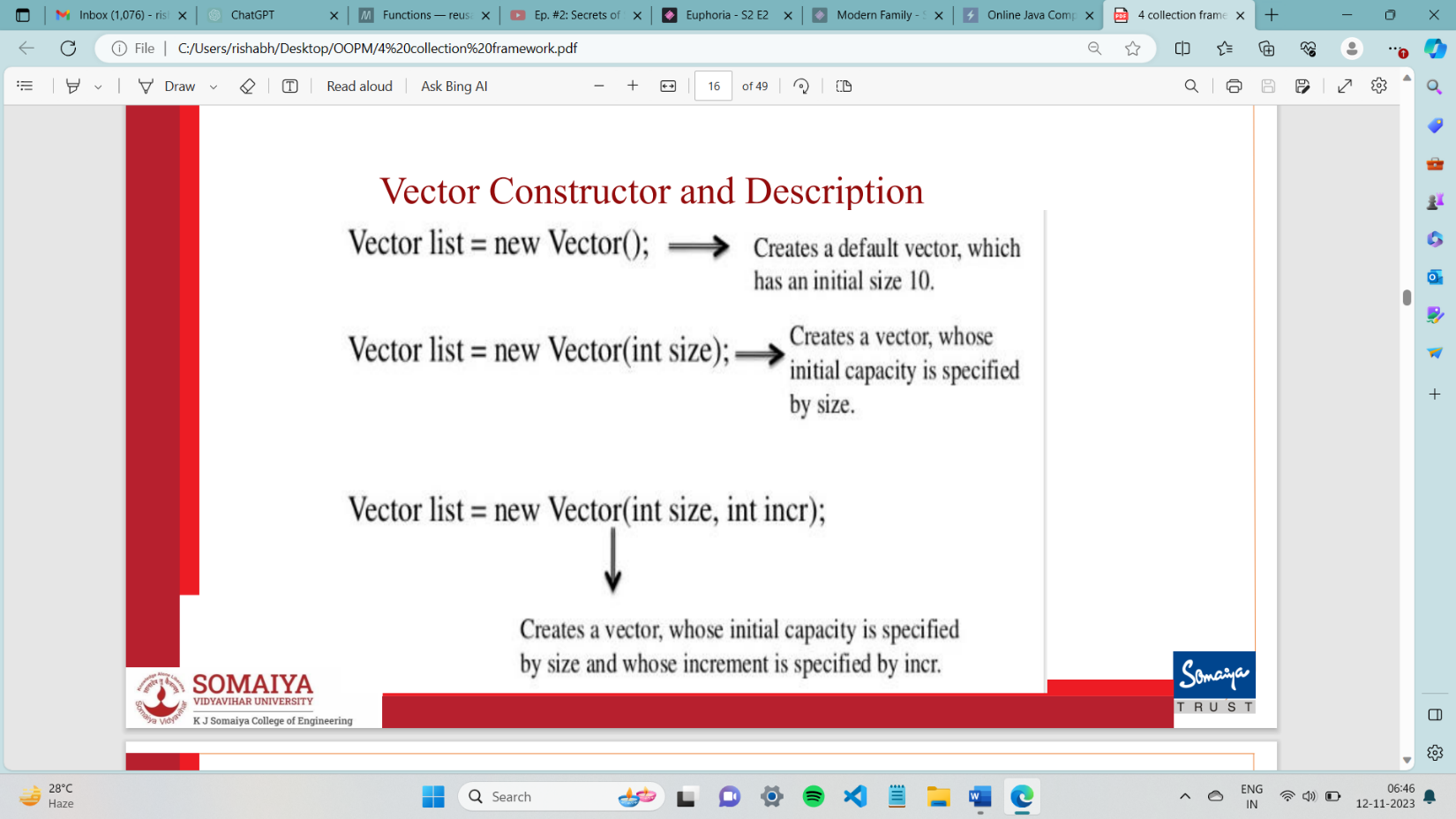


What are vectors ?

Vectors are a dynamic array of objects which can store heterogenous objects. They are useful when you don’t know the exact size in advance or you just need one that can change sizes over the lifetime of a program as they can grow and shrink as necessary. We cannot store primitive data types in vector they must be converted to objects, as vector can only store objects. Vectors are synchronized. Its defined in java.util package. To traverse elements of a vector class we use Enumeration interface. It is ordered and can duplicates.

• Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement arguments.

• The default initial capacity of vectors are 10 and the default increment is 0. (it doubles in capacity when capacity is reached). If u set capacity increment to >0 then it increases by that number.



Methods unique to vector class

1. void addElement(Object Element) :Adds the element to the end of this vector, increasing its size by one.
2. void removeElementAt(int index) : Removes the element at the specified position in this vector. Shifts any subsequent elements to the left (subtracts one from their indices).
3. void removeAllElements() : Removes all components from this vector and sets its size to zero
4. E elementAt(int index) : Returns the element at the specified index in the vector.
5. void insertElementAt(Object element, int index) : Inserts the specified element at the specified position in the vector, shifting the current elements at that position and beyond to the right.
6. void setElementAt(Object element, int index) : Sets the element at the specified index of this vector to be the specified element.
7. E firstElement(): Returns the first element in the vector.
8. object lastElement() : Returns the last element in the vector.
9. int capacity() : Returns the current capacity of the StringBuilder or StringBuffer instance. The capacity is the maximum number of characters the instance can hold without reallocating its internal buffer.
10. void setSize(int newSize) : This method is used to set the size of the vector. If the current size is greater than the specified size, the vector is truncated. If the current size is less than the specified size, new **null** elements are added at the end of the vector.
11. void trimToSize() : Reduces the capacity of this Vector to be its current size.
12. void copyInto(Object[] Array) : used to copy the components of the vector into the specified array. The method copies elements from the beginning of the vector to beginning the specified array.it replaces any pre-existing elements if they come in the way. If the specified array is smaller than the size of the vector, only the elements that fit into the array are copied.

Differences between a Vector and an Array

* A vector is a dynamic array, whose size can be increased, where as an array size cannot be changed.
* A vector is a class where as an array is not.
* Vectors can store heterogenous objects, where as an array can store only homogeneous values.
* Vector is part of collection framework
* Vector is synchronized.

Advantages of Arrays:

* Arrays supports efficient random access to the members.
* They are more appropriate for storing fixed number of elements

Disadvantages of Arrays:

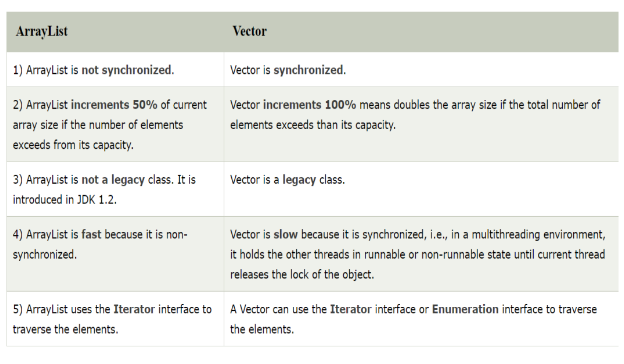
* Dynamic creation of arrays is not possible
* Multiple data types cannot be stored

Advantages of Vector:

* Size of the vector can be changed
* Heterogenous objects can be stored

Disadvantages of Vector:

* A vector is an object, memory consumption is more.
* They are more appropriate for storing uncertain number of elements



When to used which list type?

* Use arrays when direct access by index is important and you do not need to insert or delete elements frequently and size is fix
* Use Array List for a dynamic array without the need for synchronization.
* Use LinkedList when frequent insertions or deletions are required and you do not need to access elements by their index frequently.
* Use Vector if you need a thread-safe dynamic array.

What is Stack?

* Stack extends the vector class.
* A Stack is a collection that follows Last-In-First-Out (LIFO) order.

Stack<Integer> stack = new Stack<>();

stack.push(10);

stack.push(20);

System.out.println(stack.pop()); // 20

System.out.println(stack.peek()); // 10

What is the queue interface ?

* Extends the Collection interface.
* Represents a collection designed for holding elements prior to processing.
* Common implementing classes include LinkedList, ArrayDeque and PriorityQueue.
* Key Methods: offer(), poll(), peek().

What is priorityQueue?

* A queue where elements are processed according to priority rather than insertion order.
* Default Ordering: ascending(minheap)

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(30);

pq.add(10);

pq.add(20);

System.out.println(pq.poll()); // 10 (smallest element first)

What is the Deque Interface?

* Stands for “Double Ended Queue.”
* Extends the Queue interface.
* you can add and remove elements from both ends.
* Common implementing classes include ArrayDeque and LinkedList.
* Key Methods: addFirst(), addLast(), removeFirst(), removeLast().

What is ArrayDeque?

* Resizable array implementation of Deque.
* Advantages: Faster than Stack and LinkedList for stack/queue operations.

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.addFirst(10);

deque.addLast(20);

System.out.println(deque.removeFirst()); //10

What is the Set Interface?

* Represents a unordered collection of unique elements.
* Common Implementations: HashSet, LinkedHashSet, TreeSet.

What is HashSet?

* Implements the Set interface.
* Uses a hash table to store elements.
* Does not guarantee the order of elements.

HashSet<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Apple"); // duplicate ignored

What is LinkedHashSet?

* Hash table + linked list **maintains insertion order**.

What is the SortedSet Interface?

* Extends Set to maintain elements in sorted order.

What is TreeSet:

* Implements the SortedSet interface.
* Uses a Red-Black tree to store elements.
* Elements are sorted in ascending or according to a specified comparator.

What is the map interface?

* Not a sub interface of Collection.
* Represents a collection of key-value pairs.
* Common implementing classes include HashMap, TreeMap, HashTable and LinkedHashMap.

What is hashTable?

* Legacy synchronized hash table implementation of Map.
* Null: Does not allow null key/value.

What is HashMap?

* Implements the Map interface.
* Uses a hash table to store key-value pairs.
* Does not guarantee the order of key-value pairs.
* Allows one null key and multiple null values.

What is LinkedHashMap?

* HashMap + linked list maintains insertion order.

What is the SortedMap interface?

* Extends Map to maintain keys in sorted order.

What is TreeMap:

* Implements the SortedMap interface.
* Uses a Red-Black tree to store key-value pairs.
* Keys are sorted in ascending or according to a specified comparator.

Points to differentiate between data structures:

1. Internal data structure used
2. Size
3. Homo hetero
4. Primitive or objects
5. Synchronization
6. Speed
7. Memory
8. Legacy
9. Interface
10. Increase by %
11. Storing vs manipulation
12. Access by index

**GENERAL SYNTAX EXPLANATION**

**void func(int index, Collection<? extends E> c)**:

1. **Collection<?>**: This represents a collection of elements of an unknown type. The wildcard **?** is a wildcard character that denotes an unknown type. In this context, it means that the **Collection** can contain elements of any type.
2. **extends E**: This is a bounded wildcard. It specifies that the unknown type (**?**) must be a subtype of (or the same as) the type **E**. In other words, it allows the collection **c** to contain elements of type **E** or any subtype of **E**.

What are wrapper classes?

Java is an object-oriented language and can view everything as an object. A Wrapper class in Java is a class whose object wraps or contains primitive data types. They are used to convert primitive data types to objects. It is useful because classes in the collection framework only allow objects as data type and not primitive data types, so Wrapper classes help us overcome this problem.

EXAMPLE  
int k = 100; //The int data type k is converted into an object, it1 using Integer class.

Autoboxing /wrapping

Integer it1 = new Integer(k);

Unboxing/Unwrapping

int m = it1.intValue();

boolean- Boolean

char- Character

int- Integer

long- Long

float- Float

double- Double

What is autoboxing and unboxing?

The process of automatically converting a primitive data type into the object of its corresponding wrapper class is called autoboxing.

The process of automatically converting a Wrapper Class Object into its corresponding primitive data type is called unboxing.

What is a String class?

String class provides many operations for string manipulations like utility, comparisons, conversions, etc.

In java.lang package

String objects are read-only, i.e. immutable.

String str1 = "Hello";

String str2 = new String (“Hello”);

What is a StringBuffer class?

All the operations of String class can be performed (Refer above).

Unlike the String class, StringBuffer class is mutable (changeable).

StringBuffer class is used in operations where the string has to be modified.

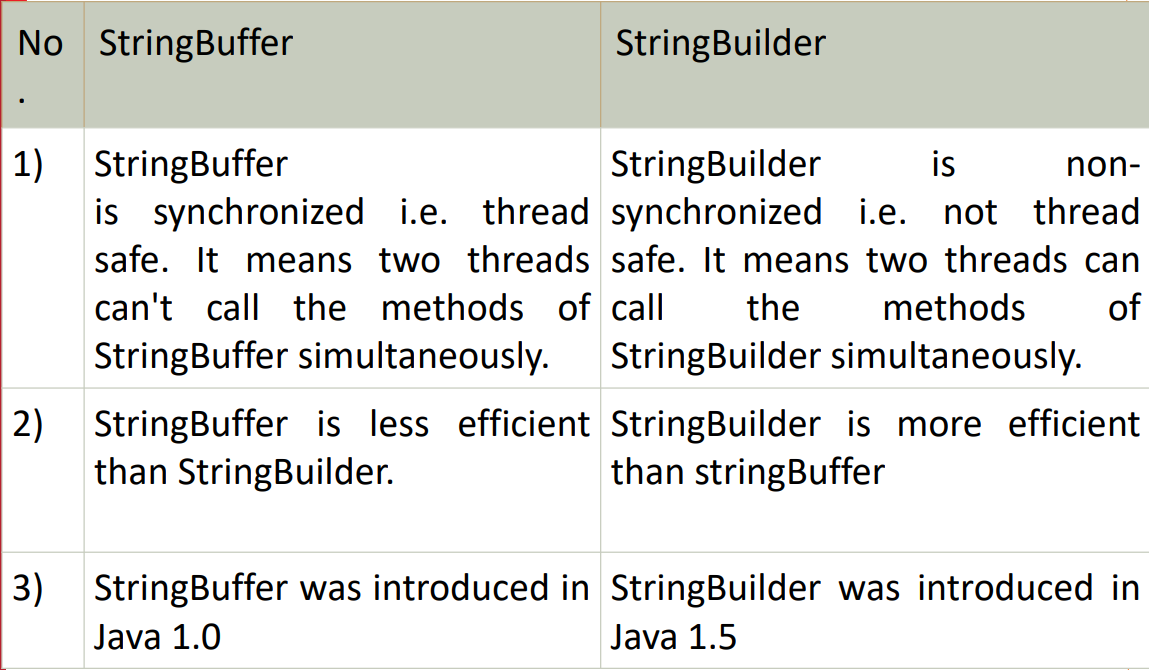
StringBuffer is synchronized i.e. thread safe.

What is a StringBuilder class?

The StringBuilder in Java represents a mutable sequence of characters and provides an alternative to String Class, as it creates a mutable sequence of characters.

The function of StringBuilder is very much similar to the StringBuffer class, However the StringBuilder class differs from the StringBuffer class on the basis of synchronization. It is non - synchronized.

* Use **String** when you need immutable string(string is not expected to change frequently.)
* Use **StringBuffer** when you need a mutable string and thread safety is a concern.
* Use **StringBuilder** when you need a mutable string and where thread safety is not critical.



STRING CLASS METHODS

int length() :Returns the length of the string.

charAt(int index) :Returns the character at the specified index

int compareTo( String anotherString) : Compares the Strings.

int compareToIgnoreCase( String anotherString) :Compare the Strings.

String replace(char oldChar, char newChar) : Returns a new string with all instances of the oldChar replaced with newChar.

trim() : Trims leading and trailing white spaces.

String toLowerCase(): converts to lowercase

String toUpperCase(): converts to uppercase

boolean equals(String anotherString): Returns true if Strings are equal

bolean equalsIngnoreCase(String s1): Returns true if Strings are equal

String concat(String S1): Returns concatenated string

String substring(int beginIndex, int endIndex): Returns substring starting from beginIndex character to endIndex character(not including endIndex character)

String substring(int beginIndex): Returns substring from beginIndex character to end

Char[] toCharArray(): Convert this string to character array.

STRINGBUFFER CLASS METHODS

int length(): Returns the length of the buffer

synchronized void setCharAt(int index, char ch): Replaces the character at the specified position

s1.setLength(int n): Truncates or extends the buffer. If n<s1.length(), s1 is truncated else zeroes are added to s1

StringBuffer append(String str): Appends the string to this string buffer.

StringBuffer append(datatype x): Appends the string representation of the argument to this string buffer.

STRINGBUILDER CLASS METHODS

• StringBuilder append(X x): This method appends the string representation of the X type argument to the sequence.

• int capacity(): This method returns the current capacity.

• char charAt(int index): This method returns the char value in this sequence at the specified index.

• StringBuilder delete(int start, int end): This method removes the characters in a substring of this sequence.

• void ensureCapacity(int minimumCapacity): This method ensures that the capacity is at least equal to the specified minimum.

• void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin): This method characters are copied from this sequence into the destination character array dst.

• int indexOf(): This method returns the index within this string of the first occurrence of the specified substring.

• StringBuilder insert(int offset, boolean b): This method inserts the string representation of the booalternatelean argument into this sequence.

• StringBuilder insert(): This method inserts the string representation of the char argument into this sequence.

• int lastIndexOf(): This method returns the index within this string of the last occurrence of the specified substring.

• int length(): This method returns the length (character count).

• StringBuilder replace(int start, int end, String str): This method replaces the characters in a substring of this sequence with characters in the specified String.

• StringBuilder reverse(): This method causes this character sequence to be replaced by the reverse of the sequence.

• String substring(): This method returns a new String that contains a subsequence of characters currently contained in this character sequence.

• String toString(): This method returns a string representing the data in this sequence

What is toString() Method?

• toString() method is a special method that can be defined in any class.

• This method should return a String argument.

• When an object is used in print statement or when one tries concatenation operation with Strings, this method gets invoked automatically.

Example:

class Circle {

double x, y, r;

public Circle(double centreX, double centreY, double radius) {

x = centreX;

y = centreY;

r = radius;

}

public String toString() {

String s = "I am a Circle with centre [" + x + "," + y + "] and radius [" + r + "]";

return s;

}

}

public class Main {

public static void main(String[] args) {

Circle c = new Circle(10, 20, 30);

System.out.println(c);

// I am a Circle with centre [10.0,20.0] and radius [30.0]

}

}

//When comparing strings using the **==** operator, it checks for reference equality, while the **equals()** method checks for content equality.

String s1 = "Amit";

String s2 = "Amit";

String s3 = new String("abcd");

String s4 = new String("abcd");

System.out.println(s1.equals(s2));

System.out.println((s1==s2));

System.out.println(s3.equals(s4));

System.out.println((s3==s4));

Output : true true true false

String s1 = "Amit"; String s2 = "Amit";

In this part, two string literals "Amit" are created. In Java, string literals are interned, which means that the compiler maintains a pool of unique string literals, and if the same literal is encountered again, the same reference is reused. Therefore, **s1** and **s2** refer to the same string literal in the string pool.

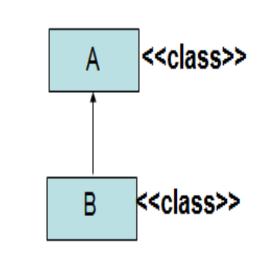
String s3 = new String("abcd"); String s4 = new String("abcd");

Here, two new string objects are explicitly created using the new keyword. Even though the content of s3 and s4 is the same, they refer to different objects in memory because each new String("abcd") creates a new string object.

**Types of Inheritance in Java**

1. Single Inheritance:

Type of inheritance in which one class extends only one other class is called Single Inheritance.



2. Multilevel Inheritance:

In Multilevel Inheritance one class inherits from a derived class which itself inherits from another class. Therefore, a derived class becomes a superclass for the new class. This forms a chain of inheritance.

Example Code:

class Human {

    static String name = "Rahil";

}

class Employee extends Human{

    static int salary = 10000;

}

public class Programmer extends Employee {

    static int bonus = 5000;

    public static void main(String[] args)

    {

        System.out.println("Name is " + name);

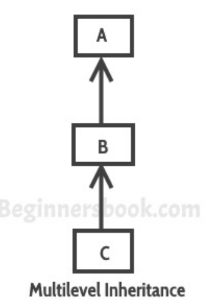
        System.out.println("Salary is " + salary);

        System.out.println("Bonus is " + bonus);

    }

}

Diagram:



1. Hierarchical inheritance:

In hierarchical inheritance, multiple subclasses inherit from a single superclass.

Example Code:

class Employee{

    static int salary = 10000;

}

class Manager extends Employee {

    static int bonus = 10000;

}

class CEO extends Employee {

    static int bonus = 100000;

}

public class Programmer extends Employee {

    static int bonus = 5000;

    public static void main(String[] args)

    {

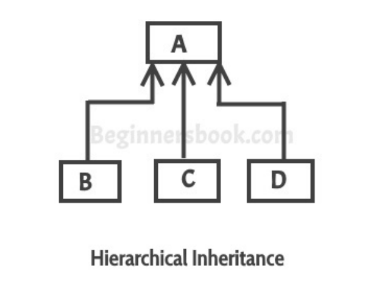
        System.out.println("Salary is " + salary);

        System.out.println("Bonus is " + bonus);

    }

}

Diagram:



1. Multiple Inheritance

In multiple inheritance one subclass tries to inherit more than one superclass.

MULTIPLE INHERITANCE IS NOT SUPPORTED IN JAVA.

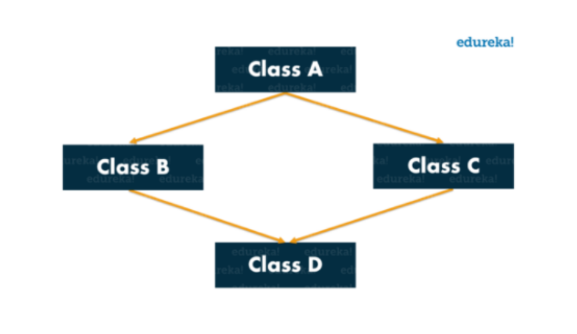
Reasons:

Diamond Problem: One of the main issues with multiple inheritance is the diamond problem, which occurs when a class inherits from two classes that have a common ancestor. It creates ambiguity in the method resolution, as it's unclear which version of a method to call from the common ancestor. This can lead to unexpected behavior and code complexity.

Ambiguity: When the 2 superclasses which the child class is inheriting from has the same field/method it becomes unclear which one should be inherited.

HOWEVER, we can achieve multiple inheritance in Java with the help of interfaces.

Diagram for multiple inheritance:



1. Hybrid Inheritance:

Hybrid inheritance is a combination of more than one types of inheritance.

Example:

Combination of multilevel and hierarchical inheritance.

What is **super** keyword in Java?

Super keyword in java is used to refer to the fields or methods of the immediate superclass of the current class. Super should always be the first line of a method/constructor.

Uses:

* To refer to the superclass’ method or field when it is overridden in the child class. When both parent class and child class have fields of the same name we can use the keyword super to access the parent class’ field.
* We can use the super keyword to call the parent class’ constructor.

Code Example1: access parent class variable

class Employee {

    int salary = 10000;

}

public class Programmer extends Employee {

    int salary = 20000;

    void printSalary(){

        System.out.println("Superclass Salary is " + super.salary);

        System.out.println("Subclass salary is " + salary);

    }

    public static void main(String[] args)

    {

        Programmer p = new Programmer();

        p.printSalary();

    }

}

Code Example2: calling parent class method

class A {

void display() {

System.out.println("Display method in class A");

}

}

class B extends A {

void display() {

System.out.println("Display method in class B");

}

// Method in class B that calls the display method from class A

void callDisplayFromA() {

super.display();

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of class B

B b = new B();

// Calling the display method from class B

b.display(); // This will call the display method in class B

// Calling the method in class B that calls display from class A

b.callDisplayFromA(); // This will call the display method in class A using super

}

}

//Code Example 3: calling parent class constructor

Super keyword can also be used to call a parent class constructor. It is very useful in the case where the parent class has parameterized constructors and you want to perform setup procedures of parent class. Even if you dont add super() in the subclass constructor it is called by default.

class A {

A() {

System.out.println("Parent constructor");

}

}

class B extends A {

B() {

super(); // Calls parent constructor

System.out.println("Child constructor");

}

}

public class Main {

public static void main(String[] args) {

B obj = new B();

}

}

OUTPUT

Parent constructor

Child constructor

//Code example 4: You can also use super to call a parameterised constructor.

class Parent {

    int parentValue;

    Parent(int value) {

        this.parentValue = value;

    }

}

class Child extends Parent {

    int childValue;

    Child(int parentValue, int childValue) {

        super(parentValue);

        this.childValue = childValue;

    }

}

What is a class diagram?

A class diagram is a type of diagram that represents the structure and relationships of classes or objects in a system. It identifies the classes and objects, the relationships between them, the attributes of objects and the links and the associations between objects.

A class diagram has boxes which each contain the following 3 sections:

1. Class Name: the topmost section which contains the name of the class
2. Attributes/Fields: The middle section represents the attributes or fields of the objects.
3. Methods: The lowest section contains the methods of the class.

Use of Class Diagrams

• Identify the classes and objects

• Identify relationship among classes and objects

• Identify attributes and operations of objects and links

• Identify associations between objects

• Organize and simplify objects classes using inheritance, generalization and specialization

Different approaches to identifying classes

* Noun phase approach (nouns are classes and verbs are methods)
* Common class patterns approach
* Use case driven, sequence/collaboration approach
* Classes responsibilities and collaborators approach

Representation of access specifiers:

+ public

# protected

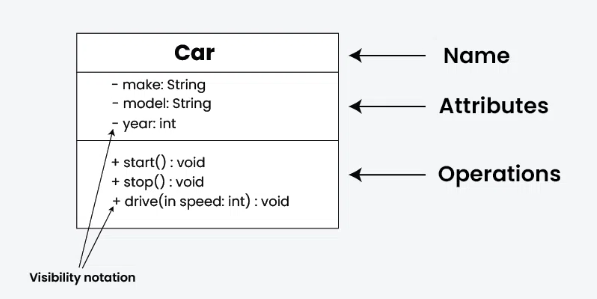
~ package (default)

- private

underline static attributes and methods

write <<interface>> on top of interfaces' names

use italics for an abstract class name



Explain the types of relations between classes in a class diagram

**Generalisation**

Generalization represents an "is-a" relationship between classes. It is used to model inheritance, where one class is a more general version (superclass or parent) and another class is a more specific version (subclass or child). The subclass inherits attributes and behaviors from the superclass. Their hierarchies are drawn top down with arrows pointing from child to parent. If parent is a class, then arrow is solid line with black arrow, if parent is an interface, then it is dashed line and white arrow.

Example: Dog is a generalization of Animal.

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

// Generalization: Dog IS-A Animal

class Dog extends Animal {

void bark() {

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

}

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.eat(); // from Animal

d.bark(); // from Dog

}

}

**Association**

Association represents a "has-a" relationship between classes. It signifies that one class is related to another class, usually through the use of attributes or methods. Associations can be one-to-one, one-to-many, or many-to-many.

They are of two types. Both represent a "has-a" relationship but differ in terms of the strength of the relationship and the lifecycle of the associated objects.

* **Aggregation:** Aggregation is a weak form of association, where one class is associated with another class, but the associated class can exist independently of the source class. It represents a weak "whole-part" relationship, where the part (associated class) can exist independently of the whole (source class). The parent class holds a reference to the child object but does not own its lifecycle. Aggregation is often represented by a hollow diamond-headed line.

Example: A Team HAS-A Player, but Player can exist without the Team.

class Player {

String name;

Player(String name) {

this.name = name;

}

}

class Team {

String teamName;

Player player;

Team(String teamName, Player player) {

this.teamName = teamName;

this.player = player;

}

void show() {

System.out.println(player.name + " plays for " + teamName);

}

}

public class Main {

public static void main(String[] args) {

Player p = new Player("Rishabh"); //created outside class

Team t = new Team("Warriors", p);

t.show();

}

}

* **Composition:** Composition is a stronger form of association, where the associated class is part of the source class, and it cannot exist independently. It represents a strong "whole-part" relationship, where the part (associated class) is a vital component of the whole (source class). The parent owns the lifecycle of the child object. If the whole is destroyed, the parts are destroyed too. Composition is often represented by a solid diamond-headed line.

Example: A House HAS-A Room, but Room cannot exist without House.

class Room {

String name;

Room(String name) {

this.name = name;

}

}

class House {

String address;

Room room;

House(String address) {

this.address = address;

this.room = new Room("Living Room"); //created inside class

}

void show() {

System.out.println("House at " + address + " has " + room.name);

}

}

public class Main {

public static void main(String[] args) {

House house = new House("123 Street");

house.show();

}

}

What is an exception and what is exception handling in Java?

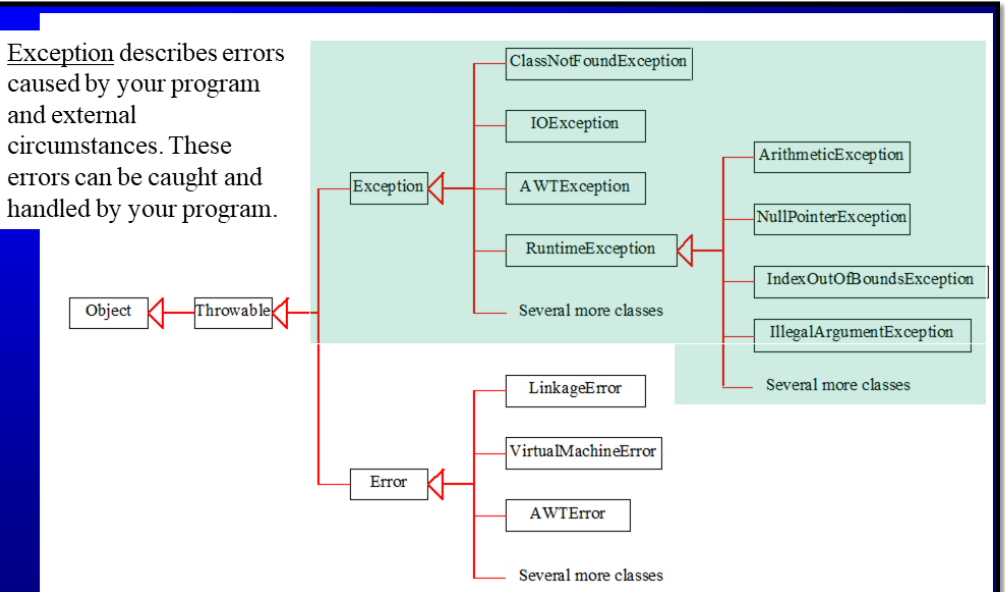
An exception is an unwanted event that occurs which interrupts the normal flow of the program. When an exception occurs the execution of the program gets terminated and we get a system generated error message. In java these exceptions can be handled. Exception handling means providing a meaningful message to the user regarding the error that occurred instead of the system defined message which is hard to understand.

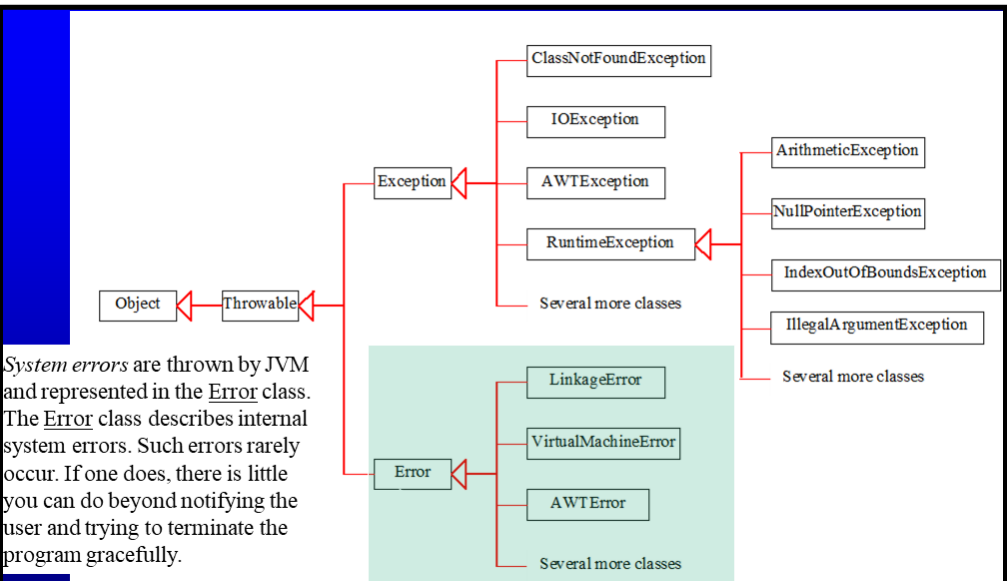
Advantage of exception handling ?

The flow of the program does not break when an exception occurs. Instead of the program terminating we can handle the exception gracefully and allow the remaining lines of code to execute. We can print a customized message for better understanding of user.

Is Exception a subclass or a superclass or both?

The class exception is a subclass of the class Throwable and is a superclass of various exceptions like Runtime exception, ClassNotFoundException, IOException and others.





What are the types of exceptions in Java?

Checked exceptions:

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error Eg. ClassNotFoundException, IOException

Unchecked Exceptions

These exceptions are not checked at compile time so compiler does not check whether they have been handled or not, so it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. Runtime Exceptions and their subclasses are known as Unchecked Exceptions. Eg. ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException

What are some common examples of exceptions occurring in java?

**ArithmeticException**: trying to divide by 0

**NullPointerException**: thrown when program attempts to use an object reference that has the null value • Invoking a method from a null object. • Accessing or modifying a null object’s field. • Taking the length of null, as if it were an array. • Accessing or modifying the slots of null object, as if it were an array.

String s=null;

System.out.println(s.length());

**NumberFormatException**: a runtime exception that occurs when you try to convert a string to a numeric type (like int, double, etc.), but the string does not have the appropriate format for the conversion. Causes- string might be null or empty, there might be leading or trailing space, string might be alphanumeric.

Integer.parseInt(null)

**ArrayIndexOutOfBoundsException**: occurs whenever we are trying to access any item of an array at an index which is not present in the array. In other words, the index may be negative or exceed the size of an array

What are try and catch blocks in Java?

A try block contains the set of statements where an exception could occur. A Try block cannot exist by itself, it must always be followed by one or more catch blocks or a finally block or both.

A catch block is where you handle the exceptions that are generated in the try block. It cannot exist alone it must exist with a try block. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.

SYNTAX: catch (Exceptiontype Object)

//Example:

public class Main {

    public static void main(String[] args)

    {

        try {

            int x = 50/0;

        }

        catch (ArithmeticException e)

        {

            System.out.println("Exception: " + e);

        }

    }

}

//Example of using multiple catch blocks with one try block.

public class Main {

    public static void main(String[] args)

    {

        try {

            int x = 50/0;

        }

        catch (ArithmeticException e)

        {

            System.out.println("Exception: " + e);

        }

        catch (Exception e) //generic exception that catches all exceptions

        {

            System.out.println("Base Exception: " + e);

        }

    }

}

//You can have multiple catch blocks with specific exceptions first and general ones later.

What is finally block in Java?

It Contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing db connections, closing files, cleanup etc.

A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

Finally block is optional, however if you place a finally block then it will always run after the execution of try block.

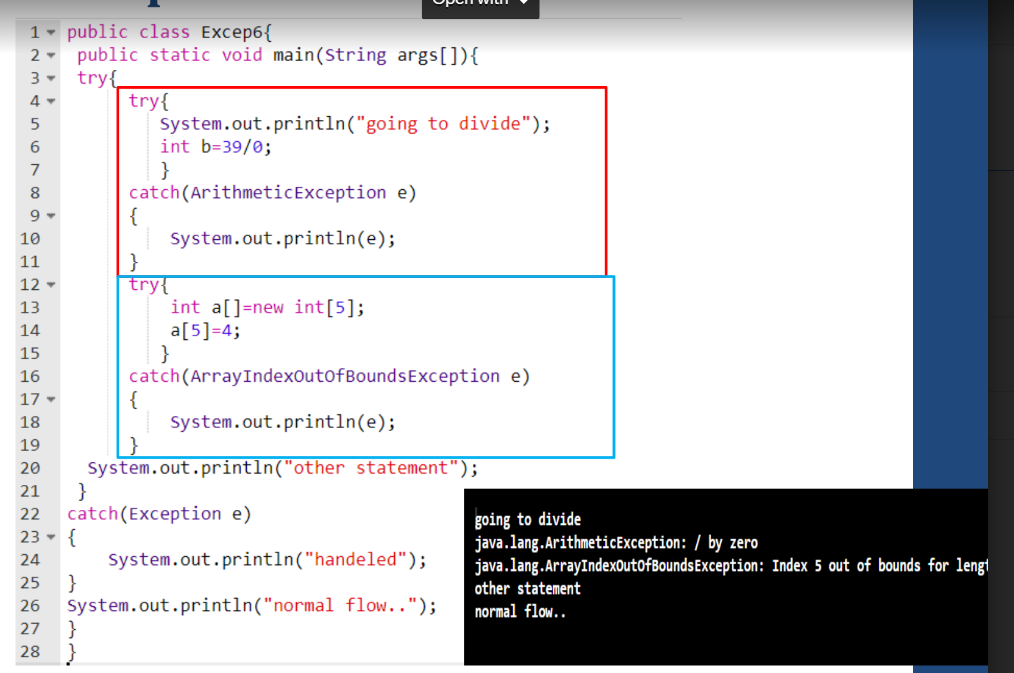
In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

The statements present in the finally block execute even if the try block contains control transfer statements like return, break or continue.

Finally is executed first and then compiler error is shown incase we are not handling exception. (See slides 38 39)

What are nested try catch blocks and can you have a try block without catch block inside another try block?

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found then that catch block executes. If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don’t handle exception. The checking moves outwards from child to parent.



What is throw keyword in java?

Throw keyword is used for throwing exceptions in already existing Exception classes We can define our own set of conditions or rules and throw an exception explicitly in a method using throw keyword.

• Syntax of throw keyword:

throw new exceptionname("error message");

//using throw with unchecked exception:

public class Main {

    static int div(int a, int b)

    {

        if(b > a)

        {

            throw new ArithmeticException("Divisor is greater than dividend");

        }

        return a/b;

    }

    public static void main(String[] args)

    {

        int x = div(10, 20);

    }

}

What is the throws keyword in java?

the throws keyword is used in the method signature to indicate that a particular method might throw one or more specified exceptions during its execution. When a method is declared with throws, it means that the method is not handling the exceptions itself, and it is leaving the responsibility of handling those exceptions to the caller or the calling method. Throws is useful for code readability in larger codebases to warn that a method might throw an exception.

//throw with checked exception requires throws

class Example {

static void myMethod() throws CustomException {

// method body

if (/\* some condition \*/) {

throw new CustomException("An error occurred");

}

}

public static void main(String[] args) {

try {

myMethod();

} catch (CustomException e) {

// Handle the exception

System.out.println("Caught CustomException: " + e.getMessage());

}

}

}

What are user-defined custom exceptions in Java?

In Java, we can create our own exception class that is derived class of the Exception class. Creating our own Exception is known as custom exception or user-defined exception .Using the custom exception, we can have your own exception and message.

Following are few of the reasons to use custom exceptions:

• To catch and provide specific treatment to a subset of existing Java exceptions.

• Business logic exceptions: These are the exceptions related to business logic and workflow

In order to create custom exception, we need to extend Exception class that belongs to java.lang package. We need to passed a string to the constructor of superclass i.e. Exception class

Example:

class InvalidDivisionException extends Exception {

    InvalidDivisionException(String s)

    {

        super(s);

    }

}

public class Main {

    static int div(int a, int b) throws InvalidDivisionException

    {

        if(b > a)

        {

            throw new InvalidDivisionException("Divisor is greater than dividend");

        }

        return a/b;

    }

    public static void main(String[] args)

    {

        try {

            int x = div(10, 20);

        }

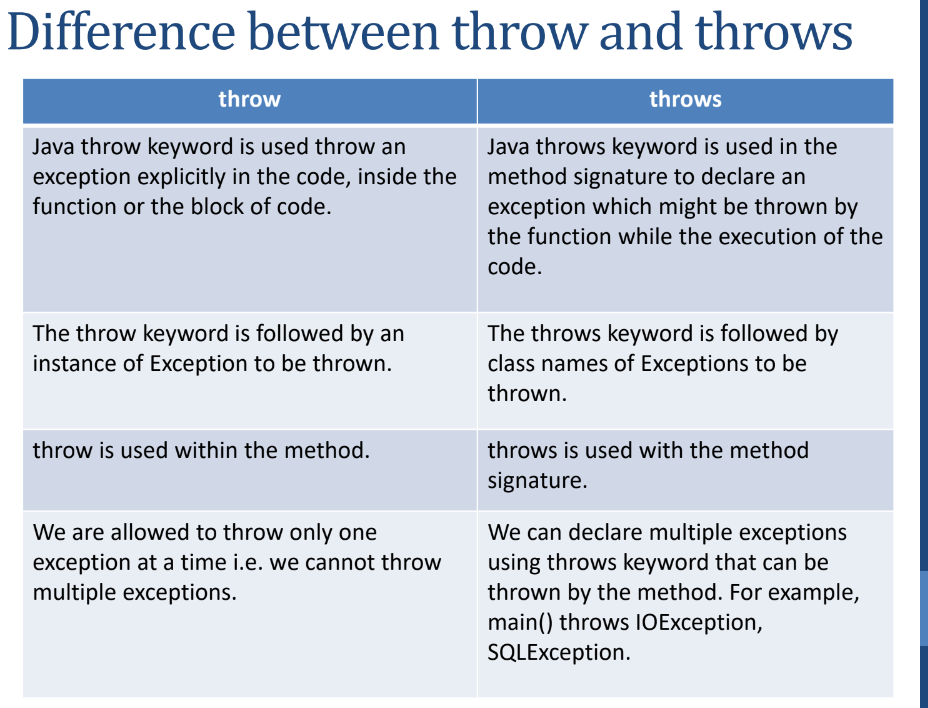
        catch (InvalidDivisionException e) {

            System.out.println("Galat hai bhai " + e);

        }

    }

}



What is the finalise() method?

In java the finalise method is equivalent to the c++ concept of destructor. When the job of an object is over, or to say, the object is no more used in the program, the object is known as garbage. The process of removing the object from a running program is known as garbage collection. In Java, memory is managed automatically by the garbage collector, and objects are automatically deallocated when they are no longer reachable. The finalize () method is a method of the Object class, and it gets called by the garbage collector before an object is reclaimed. finalize () method can be best utilized by the programmer to close the I/O streams.  
class Demo {

@Override

protected void finalize() throws Throwable {

System.out.println("finalize() called before object is destroyed");

}

}

public class FinalizeDemo {

public static void main(String[] args) {

Demo obj = new Demo();

obj = null; // eligible for GC

System.gc(); // suggest JVM to run GC

System.out.println("Main method ends");

}

}

What is a **package**?

A package can be defined as a grouping of related classes, interfaces and enumerations providing organisation, access protection and namespace management and achieving data encapsulation

What are the advantages of packages?

**Grouping**: Can be used to categorise the classes and interfaces and make locating similar classes easier.

**Access Protection:** Packages provide a way to control access to classes and other members. Members of a package can have different access modifiers (e.g., public, private, protected). This allows you to control which classes are accessible from outside the package and which are not.

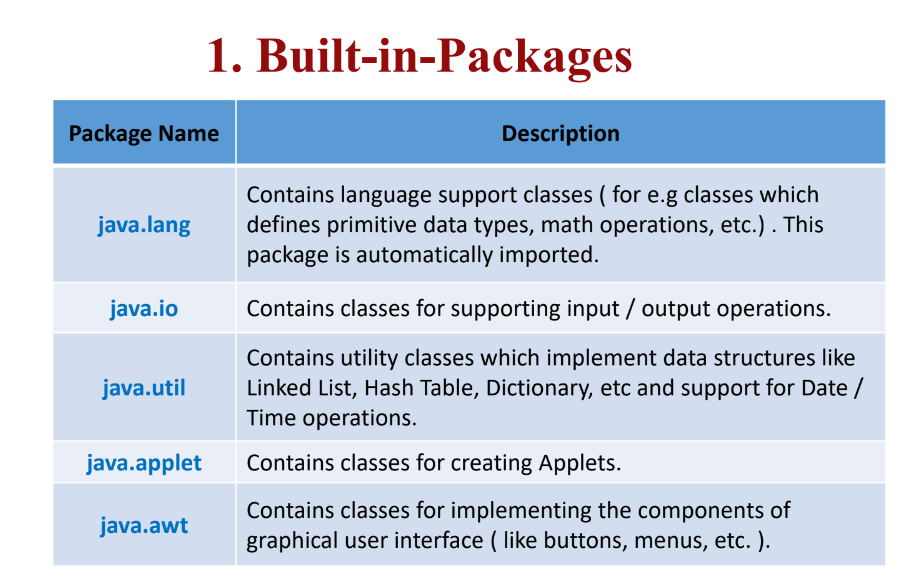
**Namespace Management:** Packages help in managing the namespace by avoiding naming conflicts. Two classes with the same name can coexist if they belong to different packages.

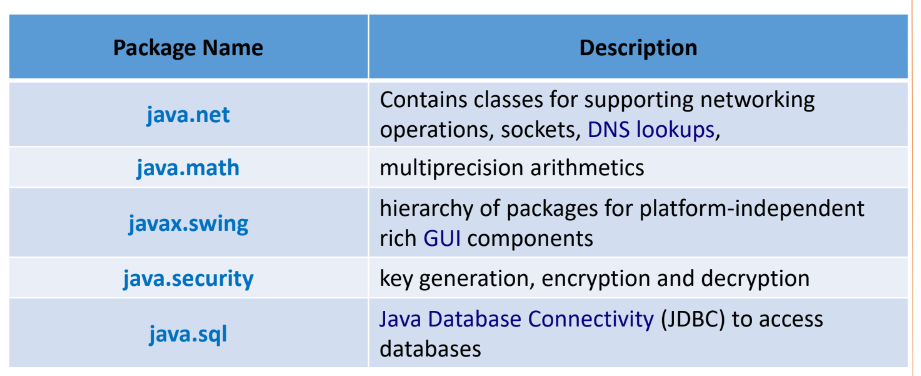
**Data encapsulatio:** While a package may consist of many classes, it's common to expose only a subset of them externally. This means that some classes are designed for internal use within the package and are not meant to be accessed directly by other packages or classes. This is a form of information hiding and helps in encapsulating the implementation details.

What are the categories of packages?

* Built in packages- these are standard packages which come inbuilt as a part of java runtime environment.
* User-defined packages: These are custom packages defined by the user to bundle together related classes and interfaces.

Give some examples of built in packages in Java.





What are user defined packages in Java and how to create one?

A user defined package in java is a custom package defined by the programmer to bundle together related classes and packages. To create a package, have a package statement at the top of the file.

package [packagename]; //declare this in every file u want to include in package

import packagename.classname; //to use classes from a package

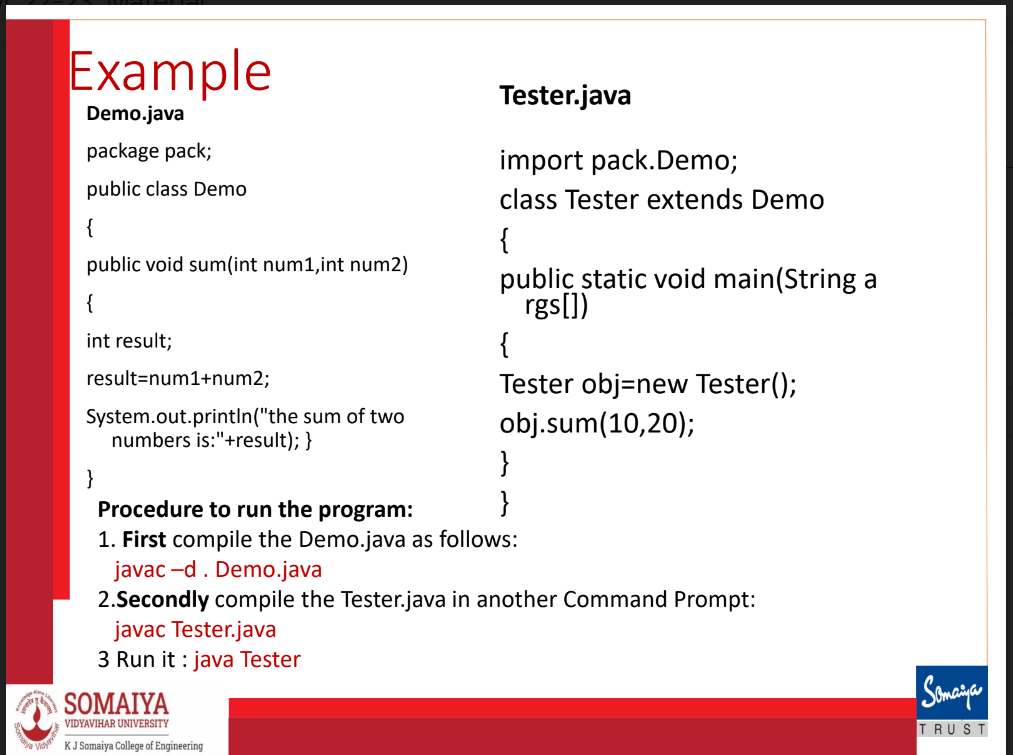
Eg.

package engineering;

public class abc {

// Rest of code

}



What is a subpackage?

Package inside the package is called the subpackage. It should be created to categorize the package further. The packages that come lower in the naming hierarchy are called "subpackage"

• The standard of defining subpackage is package.subpackage

**Break with label** statement is used to terminate the execution of a labelled statement, allowing you to break out of nested loops or switch statements.

label:

for (int i = 0; i < 5; i++){

for (int j = 0; j < 5; j++){

if (some Condition){

break label; // This will break out of the outer loop labelled as label

}  
 }

}

In this example, label is a user-defined label that is associated with the outer loop. When the break label; statement is encountered, the control flow will exit both the inner and outer loops. In many cases, alternative approaches, such as using Boolean flags or restructuring your code, can lead to more maintainable solutions. Labelled breaks are more commonly used in situations where breaking out of nested structures is necessary.

**Object References:**

Unlike languages like C or C++, Java abstracts away the concept of explicit pointers. Instead, Java relies on references to objects. A reference is like a “safe pointer”: it refers to an object, but you cannot do pointer arithmetic or access raw memory. The memory is managed automatically by the JVM and garbage collector. Developers work with objects through references, and they don't have direct access to memory addresses.

In Java, when you create an object using the new keyword, you get a reference to that object. This reference is a variable that points to the memory location where the object is stored.

Why References Instead of Pointers?

Safety: Pointers in languages like C and C++ can be prone to errors such as accessing invalid memory locations, leading to bugs and security vulnerabilities. Java's use of references adds a layer of safety by abstracting memory management.

No Memory Leaks: Java's automatic garbage collection eliminates the need for manual memory management. Developers don't have to worry about deallocating memory, reducing the likelihood of memory leaks.

Simplified Syntax: References in Java are used without the need for pointer arithmetic or explicit memory manipulation. This simplifies the language syntax and makes Java code more readable.

Portability: Java is designed to be platform-independent. Using references instead of pointers allows Java programs to run on any platform with a Java Virtual Machine (JVM) without concerns about low-level memory details.

Encapsulation: The use of references helps encapsulate the internal details of an object. Developers interact with objects through references, and the underlying memory management is abstracted away.

**Garbage Collection:**

In Java, the garbage collector (GC) is a part of the Java Virtual Machine (JVM) responsible for automatic memory management. Its primary function is to reclaim memory occupied by objects that are no longer reachable or in use by the program. The garbage collector allows Java developers to avoid manual memory management tasks, such as explicit memory deallocation, reducing the risk of memory leaks and improving overall program robustness.

class Person {

String name;

Person(String n) { name = n; }

}

Person p1 = new Person("Alice");

p1 = new Person("Bob"); // The "Alice" object is now unreachable

How the garbage collector works in Java?

Automatic Memory Management: In Java, developers use the new keyword to dynamically allocate memory for objects. The garbage collector automatically identifies and reclaims memory that is no longer referenced by the program.

Reachability: Objects that are reachable from the root of the object graph are considered live, and their memory is not eligible for garbage collection. The root objects include local variables, static variables, and active threads.

Mark-and-Sweep Algorithm: The garbage collector typically uses a mark-and-sweep algorithm to identify and reclaim memory. It involves two main phases:

Mark Phase: Traverses the object graph from the roots, marking reachable objects.

Sweep Phase: Identifies and frees memory occupied by unmarked (unreachable) objects.

//FINAL keyword in JAVA

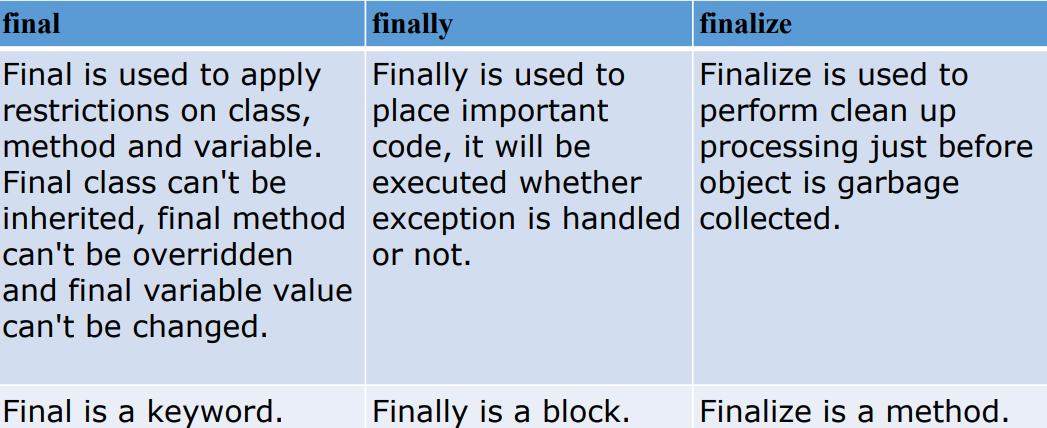
final variables cannot have their value changed.

final method cannot be overridden.

final class cannot be extended.

class CANNOT be both final and abstract since abstract class needs to be overridden

What is the difference between final, finally and finalise?



A file can have many classes.

At max one public class is allowed in a single file and its name must match filename.

The filename must match the public class name because the compiler and JVM use the filename to locate the public class and generate the corresponding .class file.

Only one public class is allowed per file to avoid ambiguity — otherwise, the compiler wouldn’t know which class the filename refers to. Non-public classes can exist alongside the public class, but the file is named after the public class.

point of entry for java code is the public static void main method. it can be in any class.

Multiple classes can have main methods. Each class can be run independently using the java ClassName command.

Compilation process

The javac compiler is used to compile Java source code into bytecode. It translates the human-readable Java source code into a platform-independent intermediate representation known as Java bytecode. The bytecode is stored in files with a .class extension. The produced bytecode is not native machine code but is instead a set of instructions for the Java Virtual Machine (JVM). This bytecode can be executed on any system with a compatible JVM.

javac filename.java

creates classname1.class classname2.class.....etc

Execution

java classname1

If it has main method it will run

IMPORT in Java

Purpose: Allows you to use classes from other packages without fully qualifying their names.

import java.io.\* means importing all classes from java.io package

import java.util.ArrayList;

ArrayList<String> list = new ArrayList<>();

Key points:

You can only access class members via an object (or static access if static).

Importing a package does not include all classes in your compiled .class file — it only tells the compiler where to find the symbols.

Using \* (star) imports multiple classes can slightly increase compilation time but does not affect runtime performance or file size.

Best practice: explicitly import only the classes you need.

Java Virtual Machine (JVM):

The JVM interprets or compiles the bytecode into native machine code at runtime. This step ensures that Java applications are platform-independent. The JVM is responsible for executing Java bytecode and providing a runtime environment for Java applications.

Java Runtime Environment (JRE)

The JRE includes the JVM along with libraries and other runtime components necessary for executing Java applications. When running a Java application, the JRE is needed to provide the necessary runtime support.

Java Development Kit (JDK)

JDK is a comprehensive software development kit. Developers use the JDK for both compiling Java source code and developing Java applications. The JDK includes the JRE, development tools, and additional libraries.

In summary, the javac compiler is responsible for translating Java source code into platform-independent bytecode. This bytecode can then be executed by the Java Virtual Machine, making Java applications versatile and capable of running on various platforms without modification. The JDK, including the javac compiler, is a complete development kit for creating and running Java applications.

**Multithreading**

REFER TO PPT FOR NOTES

INTERVIEW QUESTION:

class Base {

void print() {

System.out.println("Base");

}

}

class Derived extends Base {

void print() {

System.out.println("Derived");

}

void display() {

print(); // ①

super.print(); // ②

}

}

public class Test {

public static void main(String[] args) {

Base b = new Derived();

b.print(); // ③

b.display(); // ④

}

}

What gets printed?

A. Base

B. Derived

C. Derived Derived

D. Derived Base

Answer: B. Derived

Only line ③ runs. Since the object is Derived, b.print() runs Derived.print() (runtime polymorphism).

Line ④ gives compile error — as the object we have made is base b = new derived i.e the left side reference type is base and display() isn't in Base.

if it was derived b = new derived it would have worked