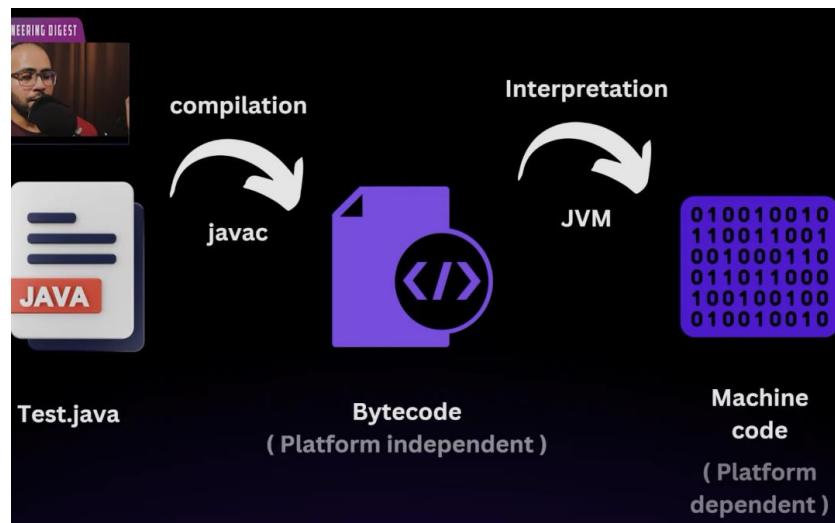
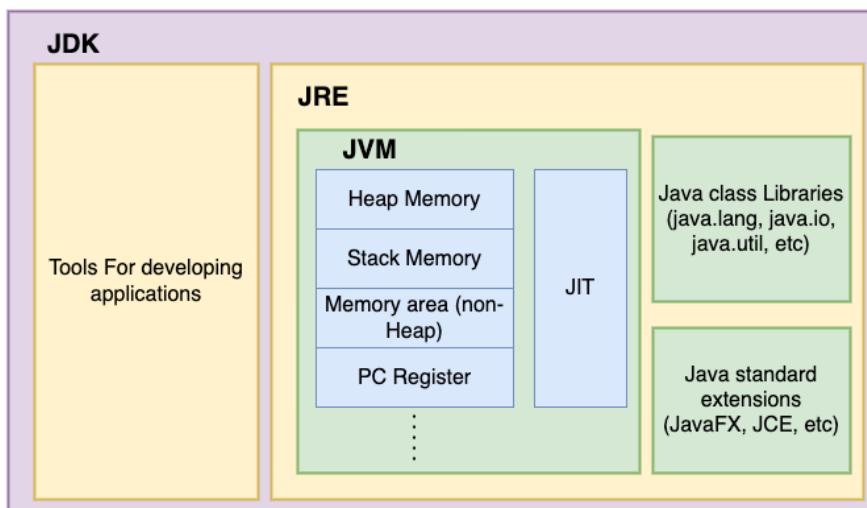
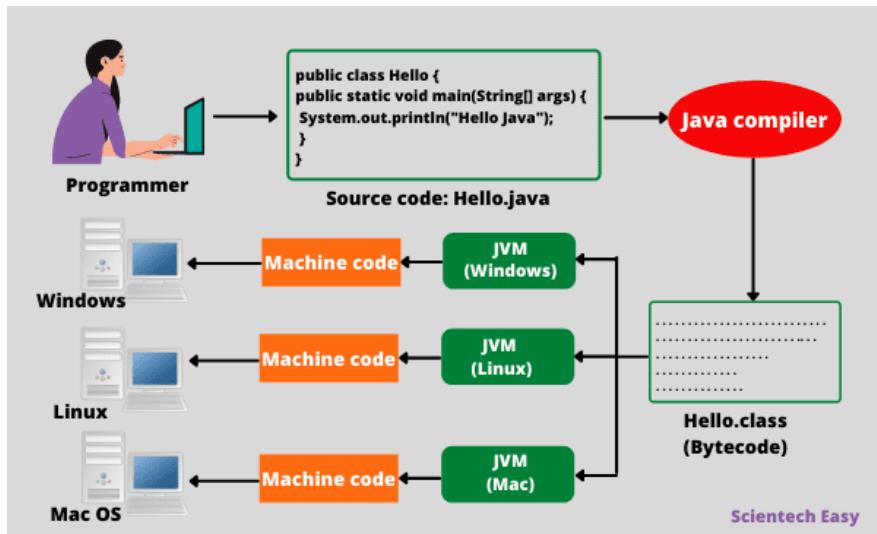


JDK – It includes everything needed to **develop and run** Java programs. (JRE + compiler (javac) + debugger + development tools)

JRE – It provides a runtime environment for executing java code (JVM + standard libraries + other resources)

JVM – A virtual machine that runs Java bytecode. It is Platform-dependent (each OS has its own implementation). It converts the bytecode into machine code that the specific operating system and hardware can understand.





Javac test.java => produce bytecode in test.class file

Java test => jvm converts bytecode to machine code and run it.

JVM also uses JIT compiler to improve performance. Frequently executed code is **compiled to machine code by the JIT compiler** and cached, so subsequent executions are much faster.

```
class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

This is the main method. In Java, the main method is the entry point of a program. It has a specific signature:

public: Access modifier indicating that the method can be accessed from outside the class.

static: Indicates that the method belongs to the class rather than an instance of the class.

void: Specifies that the method does not return any value.

main: The name of the method.

String[] args: The method accepts an array of strings as parameters. This is where command-line arguments can be passed to your program.

Main method is public so that JVM call access this method using `ClassName.main()` and start executing our program. JVM can call main method without creating object.

```
public static void main(String[] strings){
    System.out.println("Hello world!");
    System.out.println(strings[0]);
    System.out.println(strings[1]);
}
```

```
C:\Users\Vipul\Documents\Engineering Digest\Java>javac Test.java  
C:\Users\Vipul\Documents\Engineering Digest\Java>java Test apple mango  
Hello world!  
apple  
mango
```

```
System.out.println(Integer.MIN_VALUE);  
System.out.println(Integer.MAX_VALUE);
```

Access min and max values like this -

```
int a = 10; // 4 bytes  
long b = a; // 8 bytes  
float c = a; // 4 bytes  
  
float f = 1.7f;  
int g = f;
```

Java does not automatically convert because there is loss of decimal part, we need to do explicitly type casting.

Note :- unsigned right shift (>>) it fills leftmost bits with 0s

Normally when do right shift (>>) it fills leftmost bits with 1s for negative integers.

System.out.println("Arjun") → out is static member of System class

Println() → it appends new line after printing

Print() → print without appending new line

Printf() → System.out.printf("Name: %s, Age: %d", "Arjun", 25); // Output: Name: Arjun, Age: 25

Strings

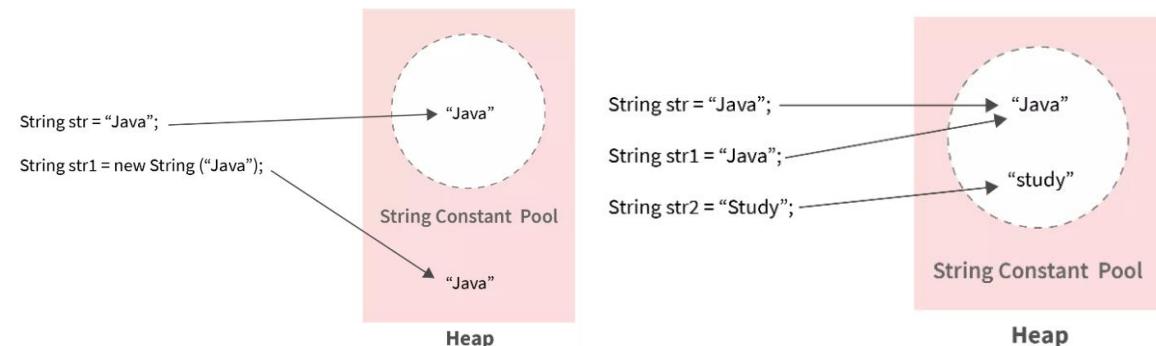
```
int a=10; // stored in stack memory  
Student student = new Student(); // student is reference variable  
                                // memory is allocated in heap  
student.name = "Arjun";  
student.address = "dda flats";  
student.rollno = 16;
```

When we first time write any string literal, then JVM puts that in string pool which is portion in heap memory. And whenever we write that string literal again, it will point to the same memory location in string pool.

Reference variables are stored in stack memory which points to memory location in heap memory. If a reference variable is an instance variable (i.e., belongs to an object), it is stored in the **heap memory**, as part of the object.

For Primitives (like int, char, boolean) == compares **actual values**.

For Objects (like String, Integer, custom classes) == compares **memory addresses** (i.e., whether both variables point to the **same object** in memory).



```
String x = "Ram";  
String a = new String(x);  
String b = new String( original: "Ram" );  
String c = "Ram";  
String d = "Ram";  
System.out.println(a == b); // false  
System.out.println(c == d); // true
```

```
String name = "Ram Sharma";  
int length = name.length(); // 10  
char c = name.charAt(4); // S  
System.out.println("Ram Sharma".equals(name)); // true  
System.out.println("ram sharma".equalsIgnoreCase(name)); // true
```

.equals() in String compares by content.

```
String str1 = "remote";
String str2 = "ramote";
int i = str2.compareTo(str1); // compares lexicographically
System.out.println(i); // -4
System.out.println("AA".compareToIgnoreCase(str: "aa")); // 0

String name = "Arjun Pahadia";
String substring = name.substring(beginIndex: 6); // go till end
String substring2 = name.substring(6, 10); // last index is exclusive
System.out.println(substring); // Pahadia
System.out.println(substring2); // Paha

// these methods create new string, original remains as it is
System.out.println("ARJUN".toLowerCase()); // arjun
System.out.println("arjun".toUpperCase()); // ARJUN
System.out.println("    arjun    ".trim()); // arjun
```

```
String name = "Amar Panchal";
String newName = name.replace(target: "Panchal", replacement: "Sharma");
System.out.println(newName); // Amar Sharma
System.out.println(name.replace(oldChar: 'a', newChar: 'o').toUpperCase()); // AMOR PONCHOL

System.out.println(name.contains("Pan")); // true
System.out.println(name.startsWith("Am")); // true
System.out.println(name.endsWith("al")); // true
System.out.println("").isEmpty()); // true
System.out.println("    ".isBlank()); // true
System.out.println(name.indexOf('a')); // 2, returns index of first occurrence
System.out.println(name.lastIndexOf(ch: 'a')) // 10
System.out.println(name.indexOf("Pan")); // 5
System.out.println(name.indexOf(str: "a", fromIndex: 5)); // 6
```

```

int i=10;
String s = String.valueOf(i);
System.out.println(i); // 10
System.out.println(Integer.parseInt(s: "123")); // 123

String formattedStr = String.format("My name is %s and I am %d yrs old", "Arjun", 23);
System.out.println(formattedStr); // My name is Arjun and I am 23 yrs old

String str = "apple,banana,grape";
String[] fruits = str.split(regex:","); // [apple, banana, grape]

String[] words = {"Java", "is", "awesome"};
String result = String.join(delimiter: " ", words);
System.out.println(result); // Java is awesome

String str1 = "hello";
char[] chars = str1.toCharArray(); // [h, e, l, l, o]
byte[] bytes = str1.getBytes(); // array of Ascii codes of chars

```

Note :-

- A String is immutable, meaning once created, its value cannot be changed. Any modification results in the creation of a new String object. It is thread-safe.
- A StringBuilder is mutable, meaning you can modify its content without creating a new object. It is not thread-safe.
- StringBuffer is mutable, allowing modifications without creating a new object. It is thread-safe, meaning all methods are synchronized, making it slower but safe to use in multi-threaded environments.

```

String s1 = "Hello";
s1 = s1 + ", World"; // new string is created in string pool

StringBuilder sb = new StringBuilder("Hello");
sb.append(", World"); // Modifies the same object

StringBuffer sbf = new StringBuffer("Hello");
sbf.append(", World"); // Modifies the same object

```

```

StringBuilder sb = new StringBuilder();
sb.append("Hello"); // Append text
sb.insert(offset: 1, str: "Java"); // Insert a string
sb.replace(start: 1, end: 3, str: "World"); // Replace part of string
sb.delete(1, 4); // Delete range
sb.reverse(); // Reverse string
sb.charAt(0); // Get character at position
sb.length(); // Get length
sb.substring(1, 4); // Get substring

```

Internally, `StringBuilder` uses character array of 16 size which doubles its size when it becomes full and all characters are copied.

Feature	<code>String</code>	<code>StringBuilder</code>	<code>StringBuffer</code>
Mutability	Immutable	Mutable	Mutable
Thread Safety	Yes	No	Yes
Performance	Slow (due to immutability)	Fast (no synchronization)	Slower (due to synchronization)
Storage	String pool (for literals)	Heap	Heap
Use Case	Small, fixed texts	Single-threaded apps	Multi-threaded apps

```

String str = "hello";
String reversed = new StringBuilder(str).reverse().toString();
System.out.println(reversed); // "olleh"

```

Reverse a String –

Changing any particular index character –

```

String a = "arjun";

// Convert to char array
char[] chars = a.toCharArray();
// Change the 3rd character (index 2)
chars[2] = 'z';
String modified = new String(chars);

```

Arrays

```
int array[] = {1,2,3,4,5};  
int[] arr = new int[10]; // initial all values 0  
// arr is reference variable which points starting address in heap  
  
arr[3] = 100;  
System.out.println(arr.length); // 10  
for(int i:arr){  
    System.out.println(i);  
}  
System.out.println(Integer.MAX_VALUE); // 2147483647  
  
// 2D array  
int[][] arr2 = new int[3][4];  
int[][] matrix = {  
    {1,2,3},  
    {4,5,6},  
    {7,8,9}  
};  
  
char c[][] = new char[3][];  
c[0] = new char[4];  
c[1] = new char[3];  
c[2] = new char[7];
```

Arrays in Java are always stored on the **heap**.

The **reference** to the array (like `arr`) is stored on the **stack** if it's a local variable.

OOPS

It is a programming paradigm in which we deals with classes and objects.

Class is a blueprint for objects. Ex – Car has properties like brand, model, color, max speed and has behaviours like accelerate, brake, etc. These properties and behaviours are defined in a blueprint.

We can create multiple objects of Car from Car class.

4 Pillars of OOPS

- **Encapsulation** – Clubing of data and methods into a single unit called class. It focuses on restricting access to data and methods using access modifiers. We achieves it by:
 1. Making fields (variables) private
 2. Providing public getter and setter methods to access and modify the private fields.
- **Inheritance** – It allows child class to inherit some properties and behaviours from parent class. It helps in code reusability.

```
class Animal{
    String name;
    String age;
}
class Cat extends Animal{
    String breed;
}
Ex - |
```

Types of inheritance :-

1. Single Inheritance - a **child class** inherits from only one **parent class**.
2. Multilevel inheritance – class A inherits from class B and class B inherits from class C, forming a chain.
3. Hierarchical inheritance - **multiple child classes inherit from a single parent class**.
4. Multiple Inheritance – Java does not support it to avoid diamond problem. However, Java supports multiple inheritance through interfaces.

Diamond Problem in Concept:

css

```
graph TD; A[A] --- B[B]; A --- C[C]; B --- D[D]
```

- Class A has a method foo().
- Class B and C both override foo().
- Class D inherits from both B and C.
- Now: Which foo() should D use? From B or C? → This is the diamond problem.

Order of constructor called –

The screenshot shows a Java code editor with the following code:

```
class Grandparent{
    Grandparent(){
        System.out.println("Grandparent constructor called");
    }
}
class Parent extends Grandparent{
    Parent(){
        System.out.println("Parent constructor called");
    }
}
class Child extends Parent{
    Child(){
        System.out.println("Child constructor called");
    }
}
public class Main {
    public static void main(String[] args){
        Child child = new Child();
    }
}
```

Below the code editor is a terminal window titled "Main" showing the execution of the code and its output:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/j
Grandparent constructor called
Parent constructor called
Child constructor called
```

If you define a parameterized constructor, the default (no-argument) constructor is not automatically provided by the compiler.

Note :- this keyword is a reference to the current object — the object on which the current method or constructor is being invoked.

Super keyword refers the immediate parent of child class

When a child class constructor is invoked, the compiler **automatically inserts a call to the parent class's no-argument constructor (super())** at the very beginning of the child constructor. We can also do this explicitly but it should be done at beginning.

If the parent class does **not** have a no-argument constructor and instead has a parameterized constructor, the compiler **cannot insert super() automatically**. In such cases, you **must explicitly call the appropriate parent class constructor** using super(arguments) and pass the required arguments.

The parent class data must be initialized before the child class data is initialized.

```

class Parent {
    int data;
    Parent(int n){
        this.data = n;
        System.out.println("Parent constructor called");
    }
    public void parentMethod(){
        System.out.println("Parent method called");
    }
}
class Child extends Parent{
    Child(){
        super(n: 10);
        System.out.println("Child constructor called");
    }
    public void childMethod(){
        System.out.println("child method called");
        super.parentMethod();
    }
}
public class Main {
    public static void main(String[] args){
        Child child = new Child();
        child.childMethod();
    }
}

```

'this' in Parent constructor refers to the child class object as the control flow reached there after we created child class object. No object of parent was created.

O/P →

```

Parent constructor called
Child constructor called
child method called
Parent method called

```

- **Polymorphism** – literally means "many forms". In programming, it means the ability of code to behave differently in different context.

Types –

1. **Compile time** – for a set of code we'll decide at compile time what operation to perform and its behaviour. It can be achieved by method overloading (decide at compile time which method to call based on the parameter)

Method overloading – same name but different parameter list

```

public int sum(int a, int b){
    return a+b;
}
public int sum(int a, int b, int c){
    return a+b+c;
}
public float sum(float a, float b){
    return a+b;
}

```

Ex-

Note :- we differentiate methods based on signature (name + parameter list) not return type.

```

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

```

Error -

2. **Runtime** – we decide at runtime what operation to perform and the behaviour of code. It can be achieved by method overriding.

Ex – Dog & Cat class overrides the makeSound() method of parent class Animal.

Reference variable of parent class can hold the memory address of child class object.

```

class Animal{
    public void makeSound(){
        System.out.println("Some sound");
    }
}
class Cat extends Animal{
    public void makeSound(){
        System.out.println("Meow!");
    }
}
class Dog extends Animal{
    public void makeSound(){
        System.out.println("Woof!");
    }
}
public class Main {
    public static void main(String[] args){
        Animal dog = new Dog();
        dog.makeSound(); // Woof!
    }
}

```

JVM will decide on runtime whose makeSound() method to call.

It is also called dynamic method dispatch or Upcasting.

```

class Parent {
}
class Child extends Parent{
}
public class Main {
    public static void main(String[] args){
        Parent parent = new Child(); // upcasting
        Child child = (Child) parent; // downcasting
    }
}

```

```

class Parent {
}
class Child extends Parent{
    public void sayBye(){
        System.out.println("Bye");
    }
}
public class Main {
    public static void main(String[] args){
        Parent parent = new Child();
        parent.sayBye(); // we only call parent class methods with reference of parent class
    }
}

```

- **Abstraction** – hiding implementation details from end user. Ex- we press the TV remote button to increase/decrease volume but we don't care how remote works internally. We just care about the fact that on pressing the volume should increase/decrease.

How is Abstraction Achieved in Java?

Java achieves abstraction in two main ways:

1. Using Abstract Classes

2. Using Interfaces

Abstract Classes

- Declared using abstract keyword
- Can have both **abstract methods** (no body) and **concrete methods** (with body)
- We cannot make object of abstract class. A class which extends an abstract class should either override the abstract methods of parent class or make the child class also abstract.

Abstract classes should not have public constructors. Constructors of abstract classes can only be called in constructors of their subclasses. So there is no point in making them public. The **protected** modifier should be enough.

Abstract methods of an abstract class should not be private as they won't be accessible outside class. We'll get ERROR

If u don't mention access modifier (package private) then its child classes in other packages cannot override it as that won't be accessible there. So making it protected can help. (it will be accessible within same package and inside child classes). Now child classes can override it and write its own implementation.

```
public abstract class Animal {  
    public abstract void sayHello();  
  
    public void sleep(){  
        System.out.println("zzz.....");  
    }  
  
    public static void main(String[] args) {  
        Animal animal1 = new Dog(); // allowed  
        Animal animal2 = new Animal(); // not allowed becoz there is no definition for sayHello()  
    }  
}  
  
class Dog extends Animal{  
  
    @Override  
    public void sayHello() {  
        System.out.println("Woof!");  
    }  
}
```

🔍 Abstract Class vs Interface (Quick Comparison)		
Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods	Only abstract (and default/static)
Fields	Can have variables	Only <code>public static final</code> constants
Multiple inheritance	✗ Only one class	✓ Multiple interfaces
Constructors	✓ Yes	✗ No

Access Modifier – keywords which describe the visibility of class, method and instance variables.

Private constructor – useful when u create a Utils class having all static methods and singleton class.

```

public class Utils {
    no usages
    private Utils(){
        [REDACTED]
    }

    no usages
    public static void sayHello(){
        System.out.println("Hello");
    }
    1 usage
    public static void sayBye(){
        System.out.println("Bye...");
    }
}

```



```

public class School {

    3 usages
    private static School instance;
    1 usage
    private School(){
        [REDACTED]
    }
    1 usage
    public static School getInstance(){
        if(instance == null){
            instance = new School();
        }
        return instance;
    }
}

```

On class we can only have public or default access modifier.

Protected: The access level is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Context\Access Modifier	private	default (no modifier)	protected	public
Same Class	Yes	Yes	Yes	Yes 
Same Package	No	Yes	Yes	Yes
Subclass (same package)	No	Yes	Yes	Yes
Subclass (different package)	No	No	Yes	Yes
Different Package (non-subclass)	No	No	No	Yes

Static Keyword

The static keyword in Java is used for memory management primarily.

It can be applied to variables, methods, blocks, and nested classes.

The main concept behind static is that it belongs to the class rather than instances of the class.

Ex –

```
public class Student {  
  
    2 usages  
    public static int count = 0;  
  
    5 usages  
    public Student() {  
        count++;  
    }  
  
    Student student1 = new Student();  
    Student student2 = new Student();  
    Student student3 = new Student();  
    Student student4 = new Student();  
    Student student5 = new Student();  
    System.out.println(Student.count);
```

O/P → 5

static method **cannot directly access non-static (instance) variables or methods.**

this and super cannot be used in static context.

Note :- static block is executed **once when the class is loaded into memory by the JVM before any object is created or any static method is called.**

Static block is used for performing static initialization, for 1 time setup task.

```
▶ public class Main {  
    public static int count;  
    static {  
        count = 22;  
        System.out.println("Hello from static block");  
    }  
    public static int getCount(){  
        return count;  
    }  
  
▶     public static void main(String[] args){  
        System.out.println(Main.getCount());  
    }  
  
Main x  
| :  
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens jav  
Hello from static block  
22
```

Final Keyword

Used with class fields to make them constants.

U can only assign value to a final variable at the time of declaration or in static block or inside constructor if field is not static.

```
public static final double PI;  
static {  
    PI = 3.14;  
}  
  
or  
  
public static final double PI = 3.14; // initialise at declaration  
static {  
    PI = 3.14; // not allowed here  
}  
  
or  
  
public class Main {  
    public final double PI ;  
    public Main() {  
        PI = 3.14;  
    }  
}
```

Final keyword can be used with methods if we want that no child class should be allowed to override the method –

```
class Car{  
    public final void airBags(){  
        System.out.println("4 airBags");  
    }  
}  
class EV extends Car{  
    @Override  
    public void airBags(){ // error  
        System.out.println("2 airBags");  
    }  
}
```

We can also make a class final if we don't want any other class to extend our class –

```
final class PremiumCar{  
    public final void airBags(){  
        System.out.println("4 airBags");  
    }  
}  
class EV extends PremiumCar{ // error  
}
```

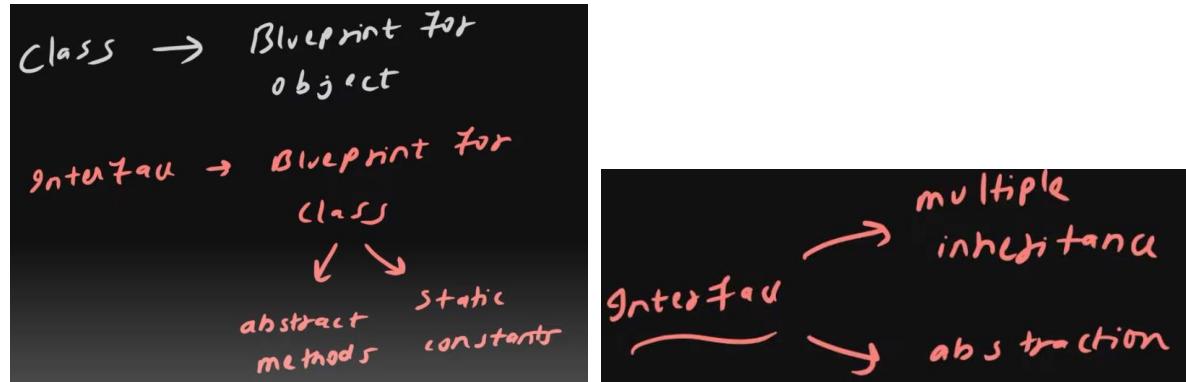
Using final on constructor makes no sense because we use final to prevent overriding and constructor is already not inherited. There is not concept of overriding constructors.

```

class PremiumCar{
    public final PremiumCar(){ // error
    }
}

```

Interface



Interfaces does not have concrete methods. Classes that implement the interface must provide the implementation for all its abstract methods, ensuring full abstraction.

```

public interface Animal {
    public static final int MAX_AGE = 100;
    // public, final, static are redundant

    public abstract void eat();
    // modifier public & abstract are redundant

    void sleep();
}

```

Class implementing Animal should either provide implemenation of eat() and sleep() or that class should be declared abstract –

```

public class Dog implements Animal{
    @Override
    public void eat() {
        System.out.println("dog eat");
    }
    @Override
    public void sleep() {
        System.out.println("dog sleep");
    }
}

Or

public abstract class Dog implements Animal{
}

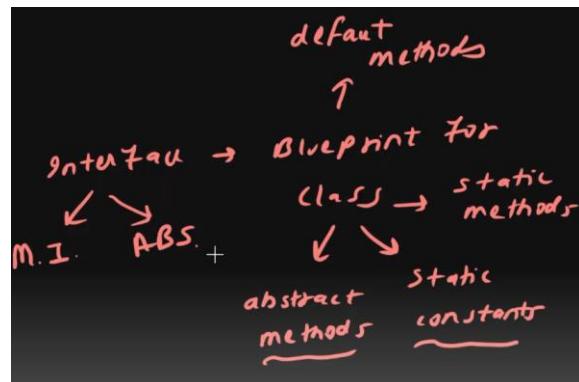
```

```

public static void main(String[] args){
    Dog dog = new Dog();
    dog.eat();
    System.out.println(Dog.MAX_AGE); // 100
    System.out.println(Animal.MAX_AGE); // 100
}

```

After Java 8, interfaces can have static and default methods –



```

public interface Animal {
    public static final int MAX_AGE = 100;
    public abstract void eat();
    void sleep();

    public static void info(){
        System.out.println("This is an Animal interface");
    }

    default void run(int n){ // implementation class will inherit default methods
        // they don't need to override it but can override if they wish

        System.out.println("Running...");
        this.eat(); // here 'this' will point the dog who called run() method
        System.out.println(n);
    }
}

public static void main(String[] args) {
    Animal.info(); // This is an Animal interface

    Dog.info(); // Error :- can only be invoked by Interface
    Animal.run(); // not allowed because it is default not static

    Dog dog = new Dog();
    dog.run(n: 10); // Running...
                    // dog eat
                    // 10
}
}

```

Multiple Inheritance in interfaces –

```
interface Camera{
    void click();
}

interface MusicPlayer{
    void play();
}

class SmartPhone implements Camera, MusicPlayer{
    @Override
    public void click() {
        System.out.println("photo clicked");
    }
    @Override
    public void play() {
        System.out.println("Playing music");
    }
}
```

Difference between abstract class & Interfaces –

- Abstract class has instance variables and to initialize them it has constructor. But we don't have these in interface
- 1 class only extend only 1 abstract class whereas 1 class can implement multiple interfaces.

Inner Classes

Types of Inner Classes

- **Member Inner Class**
- **Static Nested Class**
- **Local Inner Class**
- **Anonymous Inner Class**

Member Inner Class – Inner will behave as member of outer class. It is associated with instance of outer class. Inside inner class methods we can access fields of outer class.

Engine is acting as a property for Car –

```
public class Car {  
    private String model;  
    private boolean isEngineOn;  
    public Car(String model){  
        this.model = model;  
        this.isEngineOn = false;  
    }  
  
    class Engine{  
        void start(){  
            if(!isEngineOn){  
                isEngineOn = true;  
                System.out.println(model + " engine started.");  
            }else{  
                System.out.println(model + " engine is already on.");  
            }  
        }  
  
        void stop(){  
            if(isEngineOn){  
                isEngineOn = false;  
                System.out.println(model + " engine stopped.");  
            }else{  
                System.out.println(model + " engine is already off.");  
            }  
        }  
    }  
}
```

```
Car car = new Car(model: "Tata Safari");  
Car.Engine engine = car.new Engine();  
engine.start();
```

Without using it, our code increases as we need to create separate Engine class having Car as its instance variable –

```
public class Engine {  
    I  
    9 usages  
    private Car car;  
  
    no usages  
    public Engine(Car car){  
        this.car = car;  
    }  
  
    no usages  
    public void start() {  
        if (!car.isEngineOn()) {  
            car.setEngineOn(true);  
            System.out.println(car.getModel() + " engine started.");  
        } else {  
            System.out.println(car.getModel() + " engine is already on.");  
        }  
    }  
}
```

```

Car car = new Car(model: "Fronx");
Engine engine = new Engine(car);
engine.start();
engine.stop();

```

Static Nested class - It belongs to the outer class rather than its instance.

```

public class Computer {
    private String brand;
    private String model;
    private OperatingSystem os;
    public Computer(String brand, String model, String osName) {
        this.brand = brand;
        this.model = model;
        this.os = new OperatingSystem(osName);
    }
    public OperatingSystem getOs() { return os; }

    static class USB{ // it is the part of Class not associated with any instance
        private String type;
        public USB(String type) {
            this.type = type;
        }
        public void displayInfo(){
            System.out.println("USB type: " + type);
        }
    }

    class OperatingSystem{
        private String osName;
        public OperatingSystem(String osName) {
            this.osName = osName;
        }
        public void displayInfo(){
            System.out.println("OS name: " + osName);
        }
    }
}

public static void main(String[] args){
    Computer computer = new Computer(brand: "HP", model: "ABC", osName: "XYZ");
    computer.getOs().displayInfo();

    Computer.USB usb = new Computer.USB(type: "TYPE-C");
    Computer.USB usb2 = new Computer.USB(type: "TYPE-C");
}

```

Anonymous Inner class – useful when we have to implement an interface without creating its separate implementation class or we have to extend a class without creating its separate child class.

Anonymous because it does not have any name.

Used for creating an object for 1 time use.

```
interface Payment{
    void pay(double amount);
}

class ShoppingCart{
    private double totalAmount;
    public ShoppingCart(double totalAmount) {
        this.totalAmount = totalAmount;
    }
    public void processPayment(Payment paymentMethod){
        paymentMethod.pay(totalAmount);
    }
}

public class Main {
    public static void main(String[] args){
        ShoppingCart shoppingCart = new ShoppingCart(totalAmount: 150);

        // Creating an object "on the fly" using an anonymous inner class
        shoppingCart.processPayment(new Payment() { // we could create a class implementing
                                                    // Payment interface and then pass its instance
            @Override
            public void pay(double amount) {
                System.out.println("Paying : " + amount + " using PayPal");
            }
        });
    }
}
```

```
class Greeting{
    public void sayHello(){
        System.out.println("Hello from Greeting!");
    }
}

public class Main {
    public static void main(String[] args){
        // Creating an anonymous inner class extending Greeting
        Greeting customGreeting = new Greeting(){
            @Override
            public void sayHello(){
                System.out.println("Hello from custom greeting");
            }

            public void sayBye(){
                System.out.println("Bye from custom greeting");
            }
        };
        customGreeting.sayHello(); // Hello from custom greeting
        customGreeting.sayBye(); // ERROR: cannot call new methods of Inner class
                               // because reference variable is of type Greeting
    }
}
```

Local Inner Class

Used to encapsulate the logic. Ex - validation logic is tightly coupled with reservation process, so by placing the validation logic inside reserveRoom() we are setting the context that validate() function will only be used inside reserveRoom() method. So place it here only why to keep it outside.

Local Inner class can access all the instance variables of object.

```
class Hotel{
    private String name;
    private int totalRooms;
    private int reservedRooms;
    public Hotel(String name, int totalRooms, int reservedRooms) {
        this.name = name;
        this.totalRooms = totalRooms;
        this.reservedRooms = reservedRooms;
    }
    public void reserveRoom(String guestName, int noOfRooms){
        class ReservationValidator{
            boolean validate(){
                if(guestName==null || guestName.isBlank() || noOfRooms<0 || reservedRooms + noOfRooms > totalRooms){
                    return false;
                }
                return true;
            }
        }
        ReservationValidator validator = new ReservationValidator();
        if(validator.validate()){
            reservedRooms += noOfRooms;
            System.out.println("reservation success");
        }else{
            System.out.println("reservation failed");
        }
    }
}
public class Main {
    public static void main(String[] args){
        Hotel hotel = new Hotel(name: "Sunshine Hotel", totalRooms: 10, reservedRooms: 5);
        hotel.reserveRoom(guestName: "Akshit", noOfRooms: 3); // reservation success
        hotel.reserveRoom(guestName: "Aakash", noOfRooms: 5); // reservation failed
    }
}
```

Note :- Java is not a pure object oriented language because it still has primitive data types like int, float, double, etc. Java's **Collections** (like List, Map, Set) only work with **objects**, not primitive types (int, char, boolean, etc.).

Wrapper Classes – used to wrap primitive data types into objects.

```

int a = 1; // a is primitive variable, data stored in stack memory
Integer b = 2; // b is reference variable of Integer class, data stored in heap
Integer c = Integer.valueOf(3); // behind the scene this is happening
                                // we are doing explicit boxing which is not needed
Integer n = 4; // Java does Autoboxing
int i = n.intValue(); // Unboxing, this is also done by Java automatically
                      // no need to do explicitly
int j = n;

Integer x = null; // allowed
int y = null; // ERROR

Boolean isAdult = true;
Character ch = 'a';
Float f = 1.2f;
Double d = 1.2;
Long l = 100L;

```

```

public class Test {
    public static void main(String[] args) {
        Student x = new Student();
        x.id = 1;
        fun(x);
        System.out.println(x.id);
    }

    1 usage
    private static void fun(Student a) {
        Student student = new Student();
        student.id = 2;
        a = student;
    }
}

```

O/P → 1

```

public static void main(String[] args){
    Integer a = 1;
    fun(a);
    System.out.println(a);
}

private static void fun(Integer b) {
    b = 3; // b will start pointing to 3
}

```

O/P → 1

In Java, the wrapper classes are immutable, meaning its values cannot be changed once they are set. However, when you perform operations on them, a new instance is created to hold the result.

```

Integer a = 10;
Integer b = a;

a = a + 1; // autoboxes to a new Integer object

System.out.println(a); // 11
System.out.println(b); // 10

```

```

Integer a = 1000, b = 1000, c=1, d=1;
System.out.println(a==b); // false
System.out.println(c==d); // true
// this is because inside Integer.java there is an inner private class, IntegerCache.java
// that caches all Integer objects between -128 and 127.
// If the value is in the range -128 to 127, it returns same instance from the cache.

// comparing values
System.out.println(a.equals(b)); // true

```

Enums – used for storing fixed set of constants.

```

public class Day {
    public static final String SUNDAY = "SUNDAY";
    public static final String MONDAY = "MONDAY";
    public static final String TUESDAY = "TUESDAY";
    public static final String WEDNESDAY = "WEDNESDAY";
    public static final String THURSDAY = "THURSDAY";
    public static final String FRIDAY = "FRIDAY";
    public static final String SATURDAY = "SATURDAY";
}

public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

```

SUNDAY is a Day, MONDAY is a Day, They are final instances of Day class

On compile time, Day enum becomes a final class which extends java.lang.Enum<Day> class –

```

public final class Day extends java.lang.Enum<Day> {
    public static final Day SUNDAY = new Day("SUNDAY", 0);
    public static final Day MONDAY = new Day("MONDAY", 1);
    public static final Day TUESDAY = new Day("TUESDAY", 2);
    public static final Day WEDNESDAY = new Day("WEDNESDAY", 3);
    public static final Day THURSDAY = new Day("THURSDAY", 4);
    public static final Day FRIDAY = new Day("FRIDAY", 5);
    public static final Day SATURDAY = new Day("SATURDAY", 6);

    private static final Day[] VALUES = { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };

    private Day(String name, int ordinal) {
        super(name, ordinal);
    }

    public static Day[] values() {
        return VALUES.clone();
    }

    public static Day valueOf(String name) {
        for (Day day : VALUES) {
            if (day.name().equals(name)) {
                return day;
            }
        }
        throw new IllegalArgumentException("No enum constant " + name);
    }
}

```

Note :- In enum, the first thing you write should be enum constants. After that we can add fields and methods.

```
enum Day{
    SUNDAY(lower: "sunday"),
    MONDAY(lower: "monday"),
    TUESDAY(lower: "tuesday"),
    WEDNESDAY(lower: "wednesday"),
    THURSDAY(lower: "thursday"),
    FRIDAY(lower: "friday"),
    SATURDAY(lower: "saturday");

    private String lower;
    private Day(String lower){
        System.out.println("Our custom constructor called"); // this will be printed 7 times
        // by enum constants which are basically objects of Day class are instantiated with parameter
        // and parameterized constructor is called

        this.lower = lower;
    }

    public String getLower() {
        return lower;
    }

    public void display(){
        System.out.println("Today is " + this.name());
    }
}
```

```
public class Main {
    public static enum Months{ // 'static' is redundant for inner enums
        JAN, FEB, MARCH, APRIL, MAY, JUNE, JULY, AUG, SEPT, OCT, NOV, DEC
    }

    public static void main(String[] args) {
        System.out.println(Months.MAY); // MAY

        Day monday = Day.MONDAY;
        int ordinal = monday.ordinal();
        System.out.println(ordinal); // 1
        System.out.println(monday.name()); // MONDAY
        System.out.println(monday.toString()); // MONDAY

        Day enumDay = Day.valueOf(name: "MONDAY"); // converting string into ENUM
        // if not possible then error

        System.out.println(enumDay); // MONDAY
        enumDay.display(); // Today is MONDAY
        System.out.println(enumDay.getLower()); // monday
    }
}
```

```

public static void main(String[] args) {
    Day[] values = Day.values();
    for(Day i: values){
        System.out.println(i); // it will print all
    }

    Day d = Day.MONDAY;
    switch (d){
        case MONDAY:{
            System.out.println("Today is monday");
            break;
        }
        case TUESDAY:{
            System.out.println("Today is tuesday");
            break;
        }
        default:{
            System.out.println("Weekend!! ");
        }
    }

    // new switch case syntax after jdk12 return value
    String res = switch (d){
        case MONDAY -> "M";
        case TUESDAY -> "T";
        default -> "Weekend";
    };
}

```

Exception Handling

Types of errors

1. Syntax error
2. Logical error
3. Runtime error

Program will crash during run time errors.

The Exception Handling is a way to handle the runtime errors so that the normal flow of the application can be maintained.

Exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

```
public static int divide(int a, int b) {  
    try{  
        return a / b;  
    }catch (ArithmaticException| e){  
        System.out.println(e);  
        return -1;  
    }  
}
```

sout() uses toString() method to print object,

```
public class Student extends Object{  
    2 usages  
    Class 'Student' explicitly extends 'java.lang.Object'  
    Remove redundant 'extends Object' Alt+Shift+Enter
```

Every class extends Object class.

```
java.lang.Object  
└─ java.lang.Throwable  
    ├─ java.lang.Error  
    |   └─ java.lang.VirtualMachineError  
    |       ├─ java.lang.OutOfMemoryError  
    |       ├─ java.lang.StackOverflowError  
    |   └─ java.lang.LinkageError  
    |       ├─ java.lang.ClassFormatError  
    |       ├─ java.lang.NoClassDefFoundError  
└─ java.lang.Exception  
    ├─ java.lang.RuntimeException  
    |   ├─ java.lang.ArithmaticException  
    |   ├─ java.lang.NullPointerException  
    |   ├─ java.lang.IndexOutOfBoundsException  
    |   ├─ java.lang.IllegalArgumentException  
    |   └─ java.io.IOException  
    |       ├─ java.io.FileNotFoundException  
    |       ├─ java.io.EOFException  
    |   └─ java.sql.SQLException  
    └─ java.lang.reflect.InvocationTargetException  
        ├─ java.net.SocketException  
        └─ java.net.UnknownHostException
```

```
public static int divide(int a, int b) {  
    try {  
        return a / b;  
    } catch (RuntimeException e) {  
        System.out.println(e);  
        return -1;  
    }  
}
```

Reference variable of type RuntimeException can

point to the object of ArithmeticException.

```
try {  
    Student student = null;  
    student.setId(123);  
    System.out.println(student.getId());  
    return a / b;  
} catch (Exception e) {  
    System.out.println(e);  
    return -1;  
} catch (NullPointerException e){  
    System.out.println("Null pointer exception :(");  
    return -1;  
} catch (ArithmaticException e){  
    System.out.println("Arithmatic exception :(");  
    return -1;  
}
```

Error :- exception has already been caught

```
try {  
    Student student = null;  
    student.setId(123);  
    System.out.println(student.getId());  
    return a / b;  
}catch (NullPointerException e){  
    System.out.println("Null pointer exception :(");  
    return -1;  
}catch (ArithmaticException e){  
    System.out.println("Arithmatic exception :(");  
    return -1;  
} catch (Exception e) {  
    System.out.println(e);  
    return -1;  
}
```

Correct way of catching exception -> first catch specific exceptions and at the last catch generic exception.

```
try {  
    return a/b;  
}catch (ArithmaticException | NullPointerException e){  
    System.out.println("Some common message");  
}
```

Catch multiple exceptions in single catch()

```

try {
    return a/b;
} catch (ArithmaticException | RuntimeException e){
    System.out.println("Some common message");
}

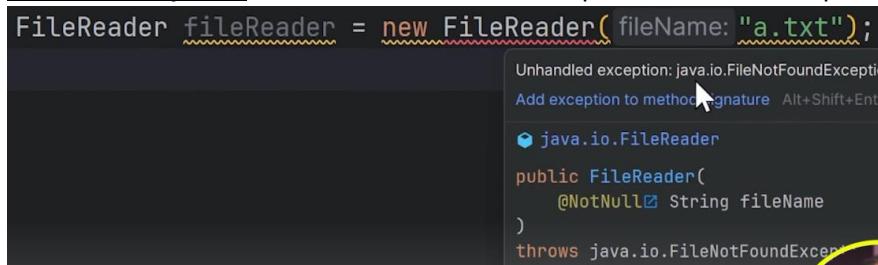
```

Error :- Types in multi-catch must be disjoint:

'java.lang.ArithmaticException' is a subclass of 'java.lang.RuntimeException'

Unchecked Exceptions – These exceptions are not being checked at compile time. Ex- ArrayIndexOutOfBoundsException, NullPointerException, all subclasses of RuntimeException. **Do not need to be caught or declared.**

Checked Exceptions – which are checked at compile time. Ex- IOException, SQLException



It will say to handle the exception – surround with try/catch or add exception to method signature using throws.

```

try{
    FileReader fileReader = new FileReader(fileName: "a.txt");
} catch (IOException e){
}

```

Throws – It is used to declare that this method can throw that exception so caller of method need to handle the exception. Caller should either use try-catch or throws the exception or else JVM will handle it by terminating the program.

```

public static void method2() throws FileNotFoundException{
    method1();
}

public static void method1() throws FileNotFoundException{
    FileReader fileReader = new FileReader(fileName: "a.txt");
}

```

Throw – used to forcefully throw some exception

```

public static void method1() throws FileNotFoundException {
    try {
        FileReader fileReader = new FileReader(fileName: "a.txt");
    } catch (FileNotFoundException e) {
        System.out.println("FILE NOT FOUND");
        throw new FileNotFoundException("oops");
    }
}

```

```
public static void method1() throws FileNotFoundException {
    throw new FileNotFoundException();
}
```

Here we have to

use **throws** keyword also because `FileNotFoundException` is checked exception so we need to make the caller aware about it so that caller will handle it.

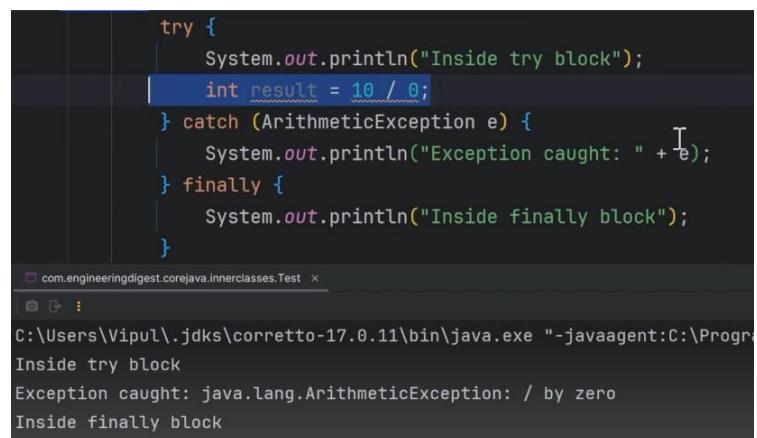
But if we **throw** unchecked exception then using **throws** is not necessary –

```
public static void method1(){
    throw new ArithmeticException();
}
```

When throwing **Exception** we need to use **throws** because it is parent class for checked exceptions -

```
public static void method1() throws Exception{
    throw new Exception();
}
```

Finally –



```
try {
    System.out.println("Inside try block");
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Exception caught: " + e);
} finally {
    System.out.println("Inside finally block");
}
```

com.engineeringdigest.corejava.innerclasses.Test

C:\Users\Vipul\.jdks\corretto-17.0.11\bin\java.exe "-javaagent:C:\Program

Inside try block
Exception caught: java.lang.ArithmetricException: / by zero
Inside finally block

Purpose of `finally`

- It is used to **clean up resources**: close files, release connections, etc.
- It **always runs after the `try` and `catch` blocks** — even if:
 - An exception is thrown
 - No exception is thrown
 - A `return` is used in `try` or `catch`
 - A `break / continue` is used in a loop

```
try {
    return a / b;
} catch (Exception e) {
    return -1;
} finally {
    System.out.println("Bye");
}
```

Object of BufferedReader must be closed, we can do it in finally block –

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName: "example.txt"));
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("IOException caught: " + e.getMessage());
} finally {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException e) {
        System.out.println("Error closing reader: " + e.getMessage());
    }
}
```



Try-with-resource - no need of writing finally for closing resources

```
try (BufferedReader reader = new BufferedReader(new FileReader(fileName: "example.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("IOException caught: " + e.getMessage());
}
```

Object created within () will be closed automatically if that object class implements AutoCloseable interface.

Custom Exception Class

```
class BankAccount{
    private double balance;
    public BankAccount(double amount){
        this.balance = amount;
    }
    public void withdraw(double amount) throws InsufficientBalanceException {
        if(amount > balance){
            throw new InsufficientBalanceException(amount);
        }
        balance -= amount;
    }
}
class InsufficientBalanceException extends Exception{
    private double amount;
    InsufficientBalanceException(double amount){
        super("insufficient exception");
        this.amount = amount;
    }
    public double getAmount() {
        return amount;
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount bankAccount = new BankAccount(amount: 10);
        try {
            bankAccount.withdraw(amount: 11);
        } catch (InsufficientBalanceException e) {
            System.out.println(e);
            System.out.println(e.getAmount());
        }
    }
}
```

Rules when overriding method -

✓ Correct Rule (for checked exceptions):

When overriding a method, the child class cannot throw broader (or new) checked exceptions than the parent class method.

It can throw:

- “No exception”
- “Same exception”
- “Or a subclass (narrower exception) of the parent’s declared exception.”

```
class Parent {  
    void method() throws IOException { }  
}  
  
class Child extends Parent {  
    @Override  
    void method() throws FileNotFoundException { } // ✓ OK (subclass of IOException)  
}
```

✗ But this is illegal:

java

Copy Edit

```
class Child extends Parent {  
    @Override  
    void method() throws Exception { } // ✗ Exception is broader than IOException  
}
```

⚠ What about unchecked exceptions?

- Unchecked exceptions (`RuntimeException` and its subclasses) can be thrown freely — no restriction during overriding.

java

Copy Edit

```
class Parent {  
    void method() { }  
}  
  
class Child extends Parent {  
    @Override  
    void method() throws NullPointerException { } // ✓ OK  
}
```

3. Access Modifier Rules

Rule	Explanation
✓ Can widen visibility	E.g., <code>protected</code> → <code>public</code>
✗ Cannot reduce visibility	E.g., <code>public</code> → <code>protected</code> is illegal

Generics

Ex- ArrayList stores array of Object class instances -

```
ArrayList list = new ArrayList();
list.add("Hello");
list.add(123);
list.add(3.14);

String str = (String) list.get(0);
String integer = (String) list.get(1);
```

ClassCastException at runtime

No Type safety
Manual casting
No compile time checking

Issues without generics -

Generic types allow you to define a class, interface, or method with placeholders (type parameters) for the data types they will work with.

It helps avoid duplication of code and giving compile time error making it type safe.

```
class Box<T>{
    private T value;
    public T getValue() { return value; }
    public void setValue(T value) { this.value = value; }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> box = new Box<>(); // Box is now type safe
        box.setValue(1);
        String i = box.getValue(); // compile time error
        // Inconvertible types; cannot cast 'java.lang.Integer' to 'java.lang.String'
    }
}
```

```

class Pair<K, V>{
    private K key;
    private V value;
    public Pair(K key, V value){
        this.key = key;
        this.value = value;
    }
    public K getKey(){
        return key;
    }
    public V getValue(){
        return value;
    }
}
public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("age", 30);
    }
}

```

Generic Interface –

```

interface Container<T>{
    void add(T item);
    T get();
}

class StringContainer implements Container<String>{
    private String item;
    @Override
    public void add(String item) { this.item = item; }
    @Override
    public String get() { return item; }
}

class GenericContainer<T> implements Container<T>{
    private T item;
    @Override
    public void add(T item) { this.item = item; }
    @Override
    public T get() { return item; }
}

```

Bounded type parameters –

```
class Box<T extends Number>{
    private T value;
    public T getValue(){
        return value;
    }
    public void setValue(T value){
        this.value = value;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> box = new Box<>();
        Box<String> stringBox = new Box<>(); // Error becoz String class doesn't extend Number
    }
}
```

T must be **Number or any subclass of Number**.

Multiple bounds –

```
interface Printable{
    void print();
}

class MyNumber extends Number implements Printable{
    private final int value;
    public MyNumber(int value) {
        this.value = value;
    }
    @Override
    public void print() { System.out.println("MyNumber: " + value); }
    @Override
    public int intValue() { return value; }
    @Override
    public long longValue() { return value; }
    @Override
    public float floatValue() { return value; }
    @Override
    public double doubleValue() { return value; }
}
```

```

class Box<T extends Number & Printable>{ // first should be class then interface
                                         // if bound is on interface then also use 'extends Printable'
    private T item;
    public Box(T item){
        this.item = item;
    }
    public void display() { item.print(); }
    public T getItem() { return item; }
}
public class Main {
    public static void main(String[] args) {
        MyNumber myNumber = new MyNumber(value: 12);
        Box<MyNumber> box = new Box<>(myNumber);
        box.display();
    }
}

```

Generics in Enum → enums are already type safe

```

enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class Main {
    public static void main(String[] args) {
        Day day = Day.MONDAY;
        Day day1 = "MONDAY"; // compilation error
    }
}

```

```

enum Operation{
    ADD, SUBTRACT, MULTIPLY, DIVIDE;
    public <T extends Number> double apply(T a, T b){
        switch (this){
            case ADD: return a.doubleValue() + b.doubleValue();
            case SUBTRACT: return a.doubleValue() - b.doubleValue();
            case MULTIPLY: return a.doubleValue() * b.doubleValue();
            case DIVIDE: return a.doubleValue() / b.doubleValue();
            default: throw new AssertionError(detailMessage: "Unknown operation: " + this);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        double res1 = Operation.ADD.apply(a: 10, b: 15);
        System.out.println(res1); // 25.0
    }
}

```

Generics Constructors – a non generic class can have generic constructor

```
class Box{
    public <T extends Number> Box(T value){
    }

    public static void main(String[] args) {
        Box box = new Box(value: 10);
    }
}
```

Generic Methods –

```
public static void main(String[] args) {
    Integer[] intArray = {1,2,3,4};
    String[] stringArray = {"Hello", "World"};
    printArray(intArray);
    printArray(stringArray);
}

public static <T> void printArray(T[] array){
    for(T element : array){
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Method overloading –

```
public static void main(String[] args) {
    display(element: 10); // Integer display: 10
    display(element: 10.12); // Generic display: 10.12
}

public static <T> void display(T element){
    System.out.println("Generic display: " + element);
}

public static void display(Integer element){
    System.out.println("Integer display: " + element);
}
```

Wildcard –

In Java Generics, wildcards (?) → are a special kind of type argument that can be used in method arguments or class definitions to represent an unknown type. They allow for more flexible and dynamic code by letting the type be specified later or be more loosely defined.

If the method is read only and does not return a value or perform any write operation, then mentioning T is not necessary and we can use ? wildcard just for simplicity.

When we return something we'll use T to maintain type safety.

```
public void printArrayList(ArrayList<?> list){  
    for(Object o : list){  
        System.out.println(o);  
    }  
}  
  
public <T> T getFirst(ArrayList<T> list){  
    return list.get(0);  
}
```

We'll get error if doing write operation with wildcard –

```
public static void main(String[] args) {  
    ArrayList<?> list = new ArrayList<String>();  
    list.add("arjun"); // ERROR  
}  
  
public void copy(ArrayList<?> source, ArrayList<?> dest){  
    for(Object item : source){  
        dest.add(item); // ERROR  
}
```

Boundary with wildcard –

```
public static void main(String[] args) {  
    System.out.println(sum(Arrays.asList(1,2.2,22))); // 25.2  
    List<Number> list = new ArrayList<>();  
    list.add(10); list.add(20);  
    System.out.println(sum(list)); // 30  
  
    printNumbers(Arrays.asList(1,2,3));  
    printNumbers(new ArrayList<String>()); // not allowed  
}  
  
public static double sum(List<? extends Number> numbers){ // upper bound  
    // any subclass of Number (such as Integer, Double, Float, etc), including Number itself  
    double sum=0;  
    for(Number o : numbers){  
        sum += o.doubleValue();  
    }  
    return sum;  
}  
  
public static void printNumbers(List<? super Integer> list){ // lower bound  
    // class that is a supertype of Integer (Object, Number, etc), including Integer itself  
    for(Object obj : list){  
        System.out.println(obj);  
    }  
}
```

```
List<? extends Number> list = Arrays.asList(1,2,3);  
list.add(10); // not allowed  
list.add(null); // allowed  
  
List<? super Number> list2 = Arrays.asList(1,2,3);  
list2.add(123); // allowed  
list2.add(null);
```

In super it will get added as Object, but in extends we don't know what exactly is the type that's why it doesn't allow to add.

Type Erasure –

```
class Box<T>{
    private T value;
    public T getValue() { return value; }
    public void setValue(T value) { this.value = value; }
}
public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setValue("Hello");
        String value = stringBox.getValue();
```

When u add value compiler will check whether u r adding String or not. But in bytecode, there are no Generics. Box<String>, the String gets erased. So after doing type checking compiler will erase information of generics.

Internally after compilation, it will look like –

```
class Box{
    private Object value;
    public Object getValue() { return value; }
    public void setValue(Object value) { this.value = value; }
}
public class Main {
    public static void main(String[] args) {
        Box stringBox = new Box();
        stringBox.setValue("Hello");
        String value = (String) stringBox.getValue();
    }
}
```

```
class NumberBox<T extends Number>{
    private T number;
    public void setNumber(T number){ this.number = number; }
    public T getNumber(){ return number; }
}
public class Main {
    public static void main(String[] args) {
        NumberBox<Integer> intBox = new NumberBox<>();
        intBox.setNumber(10);
        Integer value = intBox.getNumber();
    }
}
```

Ex2 –

After compilation -

```
class NumberBox{  
    private Number number;  
    public void setNumber(Number number){ this.number = number; }  
    public Number getNumber(){ return number; }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        NumberBox intBox = new NumberBox();  
        intBox.setNumber(10);  
        Integer value = (Integer) intBox.getNumber();  
    }  
}
```

Generics Exception -

It becomes little bit complicated because Exceptions are thrown at runtime and on runtime there is no generic information due to type erasure but JVM needs exact type to create exception object.

```
class GenericException<T> extends Exception{  
}  
}
```

Compilation Error

Java does not support generic exceptions due to type erasure. Type erasure means that generic type information is removed at runtime. Since exceptions are closely tied to runtime operations (like catching them in try-catch blocks), having generic exceptions wouldn't work as expected. For example, if you had an exception like MyGenericException<T>, you wouldn't be able to catch it with a specific type parameter because that type information would be erased at runtime.

Workaround -

```
class MyException extends Exception{  
    public <T> MyException(T value) {  
        super("Exception related to value :" +  
              value.toString() + " of type: " +  
              value.getClass().getName());  
    }  
}
```

```

public class Main {
    public static void main(String[] args) {
        try{
            throw new MyException("string");
        }catch (MyException e){
            System.out.println(e.getMessage());
        }

        try{
            throw new MyException(123);
        }catch (MyException e){
            System.out.println(e.getMessage());
        }
    }
}

```

O/P →

```

Exception related to value :string of type: java.lang.String
Exception related to value :123 of type: java.lang.Integer

```

Passing variable arguments –

<pre> public static void main(String[] args){ System.out.println(sum(...a: 1,2,3,4,5)); //15 } public static int sum(int... a){ int sum=0; for(int i:a) sum+=i; return sum; } </pre>	<pre> public static void main(String[] args){ System.out.println(sum(new int[]{1,2,3,4,5})); //15 } public static int sum(int[] a){ int sum=0; for(int i:a) sum+=i; return sum; } </pre>
---	---

```

System.out.println(Math.max(4,90));
System.out.println(Math.min(634, 6));
System.out.println(Math.abs(-39));
System.out.println(Math.ceil(1.24)); // 2.0
System.out.println(Math.floor(1.24)); // 1.0
System.out.println(Math.round(1.52)); // 2
System.out.println(Math.sqrt(25)); // 5.0
System.out.println(Math.pow(12, 2)); // 144.0
System.out.println((int)(Math.random()*11)); // random no between 1 to 10

```

Note :- In a **switch case statement** in Java, if you do not use the break; statement at the end of a case, the program will continue executing the body of all subsequent cases until it encounters a break statement or reaches the end of the switch block.

A **constructor** is a special method in a class that is used to **initialize objects** of that class. It has the following unique properties:

- Its **name is the same as the class name.**
- It **does not have a return type**, not even void.

Types of constructor –

- Default
- Parameterized

We can also overload constructors.

Once you define a **parameterized constructor** in a class, the **default constructor** (provided by Java automatically) becomes no longer available.

Optional – needed to avoid null checks everytime

```
public static void main(String[] args) {  
    String name = getName(id: 2);  
    System.out.println(name.toUpperCase());  
}  
  
private static String getName(int id){  
    // get from db  
    return null;  
}
```

O/P → NullPointerException

```
public static void main(String[] args) {  
    Optional<String> name = getName(id: 2);  
    if(name.isPresent()){  
        System.out.println(name.get()); // Ram  
    }  
  
    name.ifPresent(x -> System.out.println(x)); // Ram  
    name.ifPresent(System.out::println); // Ram  
}  
  
private static Optional<String> getName(int id){  
    String name = "Ram";  
    return Optional.of(name); // still can throw NPE if name = null  
}
```

isPresent() checks if value is not null

ifPresent() takes a consumer and perform the task if value is not null.

```
private static Optional<String> getName(int id){  
    String name = null;  
    return Optional.ofNullable(name); // No NPE  
}
```

Better to use - }

ofNullable ➔ Returns an Optional describing the given value, if non-null, otherwise returns an empty Optional.

```
public static void main(String[] args) {  
    Optional<String> name = getName(id: 2);  
    if(name.isEmpty()){  
        System.out.println("empty box"); // empty box  
    }  
}  
private static Optional<String> getName(int id){  
    return Optional.empty();  
}  
  
public static void main(String[] args) {  
    Optional<String> name = getName(id: 2);  
    String nameToBeUsed = name.isPresent() ? name.get() : "NA";  
    nameToBeUsed = name.orElse(other: "NA"); // returns the value if present, otherwise returns default  
    System.out.println(nameToBeUsed); // NA  
}  
private static Optional<String> getName(int id){  
    return Optional.empty();  
}  
  
Optional<String> name = getName(id: 2);  
String nameToBeUsed = name.orElseGet(()->"NA");  
// If a value is present, returns the value,  
// otherwise returns the result produced by the supplying function  
  
String nameToBeUsed = name.orElseThrow(()->new NoSuchElementException());  
// If a value is present, returns the value,  
// otherwise throws an exception produced by the exception supplying function  
  
public static void main(String[] args) {  
    Optional<String> optional = getName(id: 2);  
    Optional<Integer> optional1 = optional.map(x -> x.length());  
    // apply mapping function if value present, otherwise return empty optional  
  
    optional1.ifPresent(System.out::println); // 3  
}  
private static Optional<String> getName(int id){  
    return Optional.ofNullable(value: "ram");  
}
```

SOLID:-

Single responsibility principle – suppose in book manager class we create methods to addbook, removebook, printbook and savebooksToFile. This violates the principle

Book manager class should have – addBook, removebook

Book printer class – printbook;

Filemanager class – savebookstofile

Open close principle – do not modify the existing code rather than extend the code.

Ex- class discount has method calculate() which return hardcoded 10% discount.

Open close -create Discount method and 2 implementation class regulardiscount and premium discount class.

Regular discount class give 10% ,

Premium discount give 15%

Liskov substitution principle – subclasses should be replaceable with their parent class without affecting the application's correctness.

Ex – class bird has fly method

Penguin class extends birdclass which overrides fly method and throw exception
UnsupportedOperationException("penguin can't fly")

Correct – create flyable interface, create birdclass, create flyingbird class which extends birdclass and implement flyable interface.

Penguin can now extend birdclass

Interface segregation principle – classes should not be forced to implement interfaces it does not use.

Ex – interface worker has 2 methods – work() and eat()

Class humanworker implement worker – implement both methods

Class robot worker implement worker – implement work but throw exception for eat()

Correct – make 2 separate interfaces – workable and eatable

Human worker implement both , robot worker implement only 1

Dependency inversion principle – high level modules(business logic) should not depend on low level modules(sub task), both should depend on abstractions.

Ex- class CreditCardPayment has pay(), class PaymentService uses object of CreditCardPayment and inside processPayment calls its pay method.

Problem – in future adding new payment type is difficult as it is tightly coupled with creditcardpayment

Correct – interface Payment has pay method. Both CreditCardPayment and PayPalPayment class implement it and override pay method. Now Payment can be used in PaymentService class.