OOPJ (23CS305) MID-1 IMPORTANT QUESTIONS

UNIT-1

- Discuss the history of Java and explain its main features (Java Buzzwords).
- Explain operators in Java with examples (Program).
- Explain datatypes in java with examples (Program).
- Explain Variables and literals in Java with examples (Program).
- Explain the classes and objects in Java with examples (Program).
- Explain this keyword with Example Program.
- Explain the Constructor Overloading in java with Example (Program).
- Explain the Method Overloading in java with Example (Program).
- Explain Control Statements in java with Examples (Program).
- Explain inheritance with Example (Program).

Unit-1 Answers

History of Java & Java Buzzwords

- Developed by **James Gosling** at Sun Microsystems in **1991** (initially called *Oak*).
- Released in **1995** as *Java*.
- Later acquired by Oracle.

Main Features (Java Buzzwords):

1. Simple
2. Object-Oriented
3. Platform Independent (WORA: *Write Once, Run Anywhere*)
4. Secure
5. Robust
6. Multithreaded
7. Portable
8. Distributed
9. High Performance (JIT compiler)
10. Dynamic

Simple – Easy to learn and use; syntax similar to C/C++ but with no complex features like pointers, operator overloading, etc.

Object-Oriented – Everything in Java is treated as an object; supports OOP concepts (Inheritance, Encapsulation, Polymorphism, Abstraction).

Platform Independent (WORA) – Java programs are compiled into bytecode, which runs on the JVM (Java Virtual Machine) on any platform.
Secure – Provides no explicit pointer usage, has a built-in security manager, and uses bytecode verification to prevent malicious code.

Robust – Strong memory management, automatic garbage collection, and exception handling make programs reliable.

Multithreaded – Allows concurrent execution of multiple parts of a program (threads), useful in gaming, animations, and real-time apps.

Portable – Java code is independent of hardware/OS; bytecode can be carried and executed anywhere.

Distributed – Java supports networking and distributed computing with built-in APIs (e.g., RMI, Socket programming).

High Performance – Uses JIT (Just-In-Time) compiler to convert bytecode into native machine code for faster execution.

Dynamic – Supports dynamic memory allocation, runtime polymorphism, and can dynamically load classes/libraries as needed.

---

2. Operators in Java

Operators are used to perform operations on variables.
Types: Arithmetic, Relational, Logical, Assignment, Increment/Decrement, Bitwise, Ternary.

1. Arithmetic Operators are used for basic math_ **+ , - , * , / , %**
2. Relational Operators are compare values and return true/false.
== , != , > , < , >= , <=

---

3. Logical Operators are used with boolean values.
&& (AND), || (OR), ! (NOT)

---

4. Assignment Operators are Assign values.
= , += , -= , *= , /= , %=

---

5. Increment / Decrement Operators: ++ , --
Increase or decrease value by 1.

```
int a = 5;
System.out.println(++a); // pre-increment → 6
System.out.println(a++); // post-increment → 6 (then a=7)
System.out.println(--a); // pre-decrement → 6
System.out.println(a--); // post-decrement → 6 (then a=5)
```

---

6. Bitwise Operators: & , | , ^ , ~ , << , >>

Work at bit-level.

```
int a = 5, b = 3; // 5=0101 , 3=0011
System.out.println(a & b); // 1 (0001)
System.out.println(a | b); // 7 (0111)
System.out.println(a ^ b); // 6 (0110)
System.out.println(~a);    // -6 (2's complement)
System.out.println(a << 1); // 10 (left shift)
System.out.println(a >> 1); // 2 (right shift)
```

---

7. Ternary Operator- : ?

Short form of if-else.

```
condition ? value_if_true : value_if_false
int age = 18;
String result = (age >= 18) ? "Adult" : "Minor";
System.out.println(result); // Adult
```

Example:

```
public class OperatorExample {
   public static void main(String[] args) {
      int a = 10, b = 5;
      System.out.println("Arithmetic: " + (a + b));   // 15
      System.out.println("Relational: " + (a > b));   // true
      System.out.println("Logical: " + (a > 0 && b > 0)); // true
      System.out.println("Assignment: " + (a += 2));  // 12
      System.out.println("Ternary: " + ((a > b) ? "A is greater" : "B is greater"));
   }
}
```

---

3. Datatypes in Java

Two categories:

1. **Primitive**: byte, short, int, long, float, double, char, boolean.

2. **Non-primitive**: String, Arrays, Objects,Classs.

Primitive Data Types

Type	Size	Example Value	Usage
Byte	**1 byte**	**127**	**Useful for memory saving**
Short	**2 bytes**	**32000**	**Small integers**
Int	**4 bytes**	**1000**	**Default integer type**
Long	**8 bytes**	**100000L**	**Large integers**
Float	**4 bytes**	**3.14f**	**Decimal numbers (single precision)**
Double	**8 bytes**	**3.14159**	**Decimal numbers (double precision)**
Char	**2 bytes**	**'A'**	**Single character (Unicode)**
Boolean	**1 bit**	**true/false**	**Logical values**

Example:

```
public class DataType {
   public static void main(String[] args) {
      int num = 100;
      double price = 55.5;
      char grade = 'A';
      boolean status = true;
      String name = "Java";
       System.out.println("int: " + num);
      System.out.println("double: " + price);
      System.out.println("char: " + grade);
      System.out.println("boolean: " + status);
      System.out.println("String: " + name);
   }
}
```

4. Variables and Literals

- **Variable**: named memory location.
- **Literal**: fixed constant value.

Example:

```
public class VariableLiteralExample {
   public static void main(String[] args) {
      int age = 25;          // integer literal
      double pi = 3.14159;   // double literal
      char grade = 'A';      // char literal
      String name = "Java";  // string literal
      boolean flag = true;   // boolean literal

      System.out.println("Name: " + name + ", Age: " + age);
   }
}
```

---

5. Classes and Objects

Class: A **class** is a *blueprint* or *template* for creating objects. It defines **fields (variables)** and **methods (functions)**.

```
Class ClassName {

   // fields (variables)

   datatype variableName;

   // methods

   returnType methodName(parameters) {

      // method body

   }

}
```

Object: An **object** is an *instance* of a class. It represents a real-world entity and uses the fields/methods defined in the class.

Object Syntax: ClassName objectName = new ClassName();

Example:

```
class Student {
   int id;
   String name;
   void display() {
      System.out.println(id + " " + name);
   }
}
public class ClassObject {
   public static void main(String[] args) {
      Student s1 = new Student(); // object
      s1.id = 101;
      s1.name = "Ravi";
      s1.display();
   }
}
```

---

6. this Keyword

- The **`this` keyword** in Java is a reference variable that refers to **the current object of a class**.
- It is commonly used to **differentiate instance variables from local variables**, **call constructors**, or **pass the current object as a parameter**.

Uses of `this` Keyword

1. **Refer to current class instance variables**
2. **Call current class methods**
3. **Call current class constructor** (constructor chaining)
4. **Pass the current object as a parameter to another method or constructor**

Example:

```
class Employee {
   int id;
   String name;
   Employee(int id, String name) {
      this.id = id;     // using this to differentiate
      this.name = name;
   }
   void display()
```

```
{
      System.out.println(id + " " + name);
   }
}
public class ThisExample
{
   public static void main(String[] args) {
      Employee e1 = new Employee(1, "Raju");
      e1.display();
   }
}
```

---

7. Constructor Overloading

- **Constructor Overloading** is when a class has **more than one constructor** with the **same name** but **different parameters** (number, type, or order).
- Helps in **initializing objects in different ways**.
- Java determines which constructor to call based on the **arguments passed**.

Rules

1. All constructors have the **same name** (same as class name).
2. Must have **different parameter lists**.
3. Return type is **not used** for distinguishing constructors.

Syntax

```
class ClassName {
    ClassName() {
        // constructor with no parameters
    }

    ClassName(int a) {
        // constructor with one parameter
    }

    ClassName(int a, int b) {
        // constructor with two parameters
    }
}
```

Example:

```
class Person {
    String name;
    int age;
    Person() {
        name = "Unknown";
        age = 0;
    }
    Person(String n, int a) {
        name = n;
        age = a;
    }
    void display() {
        System.out.println(name + " " + age);
    }
}
public class ConstructorOverloading {
    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Person("Ravi", 22);
        p1.display();
        p2.display();
    }
}
```

---

8. Method Overloading

- **Method Overloading** occurs when a class has **more than one method with the same name** but **different parameters** (number, type, or order).
- It allows methods to perform **similar operations with different inputs**.
- Example: `add(int a, int b)` and `add(double a, double b)`

Rules

1. Methods must have the **same name**.
2. Parameter list **must be different** (number, type, or order).
3. Return type **alone cannot** distinguish methods.
4. Can occur **within the same class** or in **subclass (with inheritance)**.

Syntax

```
class ClassName {
    returnType methodName(int a)
{
/* code */
}
    returnType methodName(double a)
{
/* code */
}
    returnType methodName(int a, int b)
{
/* code */
}
}
```

Example:

```
class Calculator
{
   int add(int a, int b)
{
return a + b;
}
   double add(double a, double b)
{
return a + b;
}
}
public class MethodOverloading {
   public static void main(String[] args) {
      Calculator c = new Calculator();
      System.out.println("Int sum: " + c.add(5, 10));
      System.out.println("Double sum: " + c.add(5.5, 2.3));
   }
}
```

---

9. Control Statements

Types:

- **Decision Making**: if, if-else, switch.

- **Looping**: for, while, do-while.
- **Jump**: break, continue, return.

1. Decision Making Statements
Used to make choices in code.
(a) if syntax:
```
if (condition) {
    // code
}
```
Example:
```
int num = 10;
if (num > 0) {
    System.out.println("Positive");
}
```
(b) if-else syntax:
```
if (condition) {
    // code if true
} else {
    // code if false
}
```
Example:
```
int num = -5;
if (num >= 0)
    System.out.println("Positive");
else
    System.out.println("Negative");
```
(c) switch syntax:
```
switch(expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code
}
```
Example:
```
int day = 3;
switch(day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
```

```
    default: System.out.println("Other Day");
}
```

---

2. Looping Statements: Used to execute code repeatedly.

(a) for syntax:

```
for(initialization; condition; update) {
    // code
}
```

Example:

```
for(int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

(b) while syntax:

```
while(condition) {
    // code
}
```

Example:

```
int i = 1;
while(i <= 5) {
    System.out.println(i);
    i++;
}
```

(c) do-while syntax:

```
do {
    // code
} while(condition);
```

Example:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while(i <= 5);
```

---

3. Jump Statements

Used to change flow of execution.

(a) break

- Exits loop or switch.

```
for(int i = 1; i <= 5; i++) {
    if(i == 3) break;
    System.out.println(i);
}
```

(b) continue

- Skips current iteration, goes to next.

```
for(int i = 1; i <= 5; i++) {
    if(i == 3) continue;
    System.out.println(i);
}
```

(c) return

- Exits from method and returns value.

```
public class ReturnExample {
    static int square(int n) {
        return n * n; // returns value
    }
    public static void main(String[] args) {
        System.out.println("Square: " + square(4));
    }
}
```

Example:

```
public class ControlStatements {
   public static void main(String[] args) {
      int n = 5;

      // if-else
      if (n % 2 == 0) System.out.println("Even");
      else
          System.out.println("Odd");
            // for loop
      for (int i = 1; i <= n; i++)
          {
         System.out.print(i + " ");
          }
      System.out.println();

      // switch
      switch (n)
          {
         case 1:
          System.out.println("One");
break;
          case 5:
          System.out.println("Five");
break;
default:
          System.out.println("Other");
```

```
    }
  }
}
```

---

10. Inheritance

- Allows one class to acquire properties of another using **extends**.

Example:

```
class Animal {
  void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
  void bark()
{
System.out.println("Barking...");
}
}
public class InheritanceExample {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.eat();   // from parent
    d.bark();  // from child
  }}
```

UNIT-2

- Explain types/forms of inheritance in java with example (Program).
- Explain Super keyword in Java with example (Program).
- Explain final and abstract keyword in java with example (Program).
- Explain Method overriding in Java with examples (Program).
- Explain the concept of interfaces in Java with examples (Program).
- Explain the concepts of packages with Example Program.
- Explain java.io package with Example (Program).
- Explain the String class in java with Example (Program).
- Explain inner classes in java with Examples (Program).
- Explain the concept of abstract class with Example (Program).

Unit-2 Answers

1. Types/Forms of Inheritance in Java

• **Inheritance** is the process by which one class (**child/derived/subclass**) acquires the properties and behaviors (fields & methods) of another class (**parent/base/superclass**).

• Achieved using the **`extends` keyword**. Supports **code reusability** and **polymorphism**.

Java supports:

1. **Single Inheritance** – One parent, one child.
2. **Multilevel Inheritance** – Parent → Child → Grandchild.
3. **Hierarchical Inheritance** – One parent, multiple children.
   (*Multiple & Hybrid inheritance are not directly supported with classes, but with interfaces they are.*)

1. Single Inheritance

```
class Parent {
    // parent class members
}

class Child extends Parent {
    // child class members
}
```

---

2. Multilevel Inheritance

```
class GrandParent {
    // grandparent class members
}

class Parent extends GrandParent {
    // parent class members
}

class Child extends Parent {
    // child class members
}
```

---

3. Hierarchical Inheritance

```
class Parent {
    // parent class members
}

class Child1 extends Parent {
    // child1 class members
}

class Child2 extends Parent {
    // child2 class members
}
```

---

4. Multiple Inheritance (via Interfaces)

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    public void methodA() { /* implementation */ }
    public void methodB() { /* implementation */ }
}
```

---

5. Hybrid Inheritance

```
interface A {
    void methodA();
}

class B {
    void methodB() { /* implementation */ }
}

class C extends B implements A {
    public void methodA() { /* implementation */ }
}
```

Example:

```
// Single Inheritance
class A {
   void showA()
{
System.out.println("Class A method");
}
}
class B extends A {
   void showB()
{
System.out.println("Class B method");
}
}
public class InheritanceDemo {
   public static void main(String[] args) {
      B obj = new B();
      obj.showA();
      obj.showB();
   }
}
```

---

2. super Keyword

- The **`super` keyword** in Java is a reference variable used to refer to the **immediate parent class object**.
- It helps in **accessing parent class members** (variables, methods, constructors).

Uses of `super` keyword

1. **Access parent class variables** (when child and parent have same variable name).
2. **Call parent class methods** (when overridden in child class).
3. **Call parent class constructor**.

Example:

```
class Parent {
   int num = 100;
   void display()
{
System.out.println("Parent method");
```

```
}
}
class Child extends Parent {
   int num = 200;
   void display() {
      super.display(); // calls parent method
      System.out.println("Child num: " + num);
      System.out.println("Parent num: " + super.num); // parent variable
   }
}
public class SuperExample {
   public static void main(String[] args) {
      Child c = new Child();
      c.display();
   }
}
```

---

3. final and abstract Keyword

The `final` keyword can be used with **variables, methods, and classes**.

Usage

- **Final Variable** → value cannot be changed (constant).
- **Final Method** → cannot be overridden in subclass.
- **Final Class** → cannot be inherited.
- **final**: prevents **inheritance, method overriding, or variable modification**.

The `abstract` keyword is used to declare **abstract classes** and **abstract methods**.

Rules

1. An **abstract method** has **no body** → `abstract void methodName();`
2. An **abstract class** may contain abstract and non-abstract methods.
3. We **cannot create objects** of an abstract class.
4. A **subclass must implement** all abstract methods.

Example:

```
// final example
class FinalExample {
   final int value = 50;
   final void display()
```

```
{
System.out.println("Final method");
}
}
// abstract example
abstract class Shape {
   abstract void draw(); // abstract method
}
class Circle extends Shape {
   void draw()
{
System.out.println("Drawing Circle");
}
}
public class FinalAbstractDemo {
   public static void main(String[] args) {
      Circle c = new Circle();
      c.draw();
   }
}
```

---

4. Method Overriding

- **Method Overriding** occurs when a **subclass provides a specific implementation** of a method that is **already defined in its superclass**.
- The method in subclass **must have the same name, return type, and parameters** as in superclass.
- It is a way to achieve **runtime polymorphism (dynamic method dispatch)**.

Rules for Method Overriding

1. The method **must be inherited** from a superclass.
2. Method name, return type, and parameters **must be the same**.
3. The **access modifier** in subclass must be the same or **more accessible**.
4. The method in superclass **cannot be `final` or `static`**.
5. It is always between **superclass and subclass**.

Syntax

```
class Parent {
    void display() {
        System.out.println("This is parent class method");
```

```
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("This is child class method");
    }
}
```

Example:

```
class Animal {
   void sound()
{
System.out.println("Animal makes sound");
}
}
class Dog extends Animal {
   void sound()
{
System.out.println("Dog barks");
}
}
public class OverridingExample {
   public static void main(String[] args) {
      Animal a = new Dog(); // runtime polymorphism
      a.sound();
   }
}
```

---

5. Interfaces

- An **interface** is a **blueprint of a class**. It contains **abstract methods** (implicitly `public abstract`) and **constants** (implicitly `public static final`).
- A class implements an interface using the `implements` keyword.
- Supports **multiple inheritance** (since a class can implement multiple interfaces).

Syntax

```
interface InterfaceName {
    // abstract method
    void method1();
```

```
    // constant
    int VALUE = 100;
}

class ClassName implements InterfaceName {
    public void method1() {
        // method body
    }
}
```

Example:

```
interface Vehicle
{
   void start();
}
class Car implements Vehicle
{
   public void start()
{
System.out.println("Car starts with key");
}
}
public class InterfaceExample
{
   public static void main(String[] args)
{
      Vehicle v = new Car();
      v.start();
   }
}
```

---

6. Packages

- A **package** is a way to **group related classes, interfaces, and sub-packages**.
- Think of it like a **folder in a computer** that organizes files.
- Helps in **code reusability, organization, and avoiding name conflicts**.

Types of Packages

1. **Built-in Packages** → Already provided by Java (e.g., `java.util`, `java.io`, `java.sql`).
2. **User-defined Packages** → Created by programmers for project organization.

Syntax

Declaring a Package

```
package packagename;
```

Importing a Package

```
import packagename.ClassName;
```

or

```
import packagename.*;   // imports all classes from package
```

Example:

```
// File: mypackage/MyClass.java
package mypackage;
public class MyClass
{
   public void display()
{
      System.out.println("Hello from MyClass in mypackage");
   }
}

// File: TestPackage.java
import mypackage.MyClass;
public class TestPackage
{
   public static void main(String[] args)
{
      MyClass obj = new MyClass();
      obj.display();
   }
}
```

---

7. java.io Package

- The **`java.io` package** provides **classes and interfaces** for **input and output (I/O)** in Java.
- It allows programs to **read data from sources** (keyboard, files, network) and **write data to destinations** (console, files, network).

Main Classes in `java.io`

1. **File** – represents a file or directory path.
2. **FileInputStream / FileOutputStream** – for reading/writing binary data.
3. **FileReader / FileWriter** – for reading/writing character data.
4. **BufferedReader / BufferedWriter** – for efficient text reading/writing.
5. **PrintWriter** – for printing formatted text to files.
6. **ObjectInputStream / ObjectOutputStream** – for serialization (reading/writing objects).

Syntax

```
import java.io.*;

class MyClass {
    // use I/O classes like File, FileReader, etc.
}
```

Example (reading file):

```
import java.io.*;

public class IOExample {
   public static void main(String[] args) {
      try {
         FileWriter fw = new FileWriter("test.txt");
         fw.write("Hello Java IO");
         fw.close();
         FileReader fr = new FileReader("test.txt");
         int i;
         while ((i = fr.read()) != -1)
         {
            System.out.print((char) i);
         }
         fr.close();
      }
catch (IOException e)
{
         System.out.println(e);
```

```
    }
  }
}
```

---

8. String Class

- A string is a **sequence of characters**, enclosed in double quotes `" "`.
- Strings in Java are **immutable** (cannot be changed once created).
- The `String` class provides **many useful methods** such as:
  - `length()` → returns length of string
  - `concat()` → joins two strings
  - `substring()` → extracts part of string
  - `equals()` → compares strings
  - `toUpperCase() / toLowerCase()` → changes case

Syntax

```
String s1 = "Hello";              // Using string literal
String s2 = new String("World");  // Using new keyword
```

Example:

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

{
      String s1 = "Hello";
      String s2 = "World";
      System.out.println("Concatenation: " + s1.concat(" " + s2));
      System.out.println("Length: " + s1.length());
      System.out.println("Substring: " + s2.substring(1,4));
      System.out.println("Equals: " + s1.equals("Hello"));
   }
}
```

---

9. Inner Classes

- A class inside another class. Types: **Member, Static, Local, Anonymous**.
- Defined inside another class, but **outside methods**.
- Accesses both static and non-static members of the outer class.
- Can access only **static members** of the outer class directly.

- Defined **inside a method** of outer class. Scope is **local to that method** only.

Example (Member Inner Class):

```
class Outer {
   private String msg = "Hello from Inner Class";
   class Inner {
      void show() { System.out.println(msg); }
   }
}
public class InnerClassExample {
   public static void main(String[] args) {
      Outer o = new Outer();
      Outer.Inner in = o.new Inner();
      in.show();
   }
}
```

---

10. Abstract Class

- Abstract class: cannot be instantiated, may have abstract and non-abstract methods.
- An **abstract class** is a class that is declared using the keyword `abstract`.
- It can have **abstract methods** (without body) and **non-abstract methods** (with body).
- You **cannot create objects** of an abstract class.
- It is used for **inheritance and abstraction**.

Syntax

```
abstract class ClassName {
    // abstract method (no body)
    abstract void methodName();

    // normal method (with body)
    void normalMethod() {
        System.out.println("This is a normal method.");
    }
}
```

Example:

```
abstract class Animal
{
```

```
    abstract void sound();
    void sleep()
{
System.out.println("Sleeping...");
}
}
class Cat extends Animal {
    void sound() { System.out.println("Meow"); }
}
public class AbstractExample {
    public static void main(String[] args) {
        Animal a = new Cat();
        a.sound();
        a.sleep();
    }
}
```

UNIT-3

- What is the use of Exception Handling in Java? How are Exceptions handled with try and catch blocks? Explain with a program.
- Explain any five predefined Exceptions in Java with example (Program).
- Explain throw keyword in java with example (Program).
- Explain throws keyword in Java in Java with examples (Program).
- Explain finally block in java with example program.

Unit-3 Answers

1. Use of Exception Handling in Java

- Exception Handling is used to handle **runtime errors** so the normal flow of the program is not disrupted.
- It provides a way to transfer control from one part of the program to another using constructs like **try, catch, throw, throws, and finally**.

How try and catch work

- **try block** contains the code that might throw an exception.
- **catch block** handles the exception if it occurs.

Syntax

```
try {
   // Code that may throw an exception
    }
catch (ExceptionType e)
{
   // Code to handle the exception
}
```

Example:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int a = 10, b = 0;
            int result = a / b; // may cause
ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is
not allowed!");
        }
        System.out.println("Program continues...");
    }
}
```

Output:

```
Error: Division by zero is not allowed!
Program continues...
```

---

2. Five Predefined Exceptions in Java

Some commonly used predefined exceptions:

1. **ArithmeticException** – division by zero.
2. **ArrayIndexOutOfBoundsException** – invalid array index.
3. **NullPointerException** – accessing methods/variables on `null`.
4. **NumberFormatException** – invalid string conversion to number.
5. **ClassCastException** – invalid type casting.

Example:

```
public class PredefinedExceptions {
    public static void main(String[] args) {
        try {
            // 1. ArithmeticException
            int x = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e);
        }

        try {
            // 2. ArrayIndexOutOfBoundsException
            int arr[] = new int[3];
            System.out.println(arr[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught: " + e);
        }

        try {
            // 3. NullPointerException
            String str = null;
            System.out.println(str.length());
        } catch (NullPointerException e) {
            System.out.println("Caught: " + e);
        }

        try {
            // 4. NumberFormatException
            int num = Integer.parseInt("abc");
        } catch (NumberFormatException e) {
            System.out.println("Caught: " + e);
        }

        try {
            // 5. ClassCastException
            Object obj = new Integer(10);
            String s = (String) obj;
        } catch (ClassCastException e) {
            System.out.println("Caught: " + e);
        }
    }
}
```

3. throw Keyword

- The `throw` keyword is used to **explicitly throw an exception** inside a method or block of code.

Example:

```
public class ThrowExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to
vote!");
        } else {
            System.out.println("Eligible to vote!");
        }
    }

    public static void main(String[] args) {
        checkAge(15);  // throws exception
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException:
Not eligible to vote!
```

---

4. throws Keyword

- The `throws` keyword is used in a method declaration to indicate that the method might throw certain exceptions.
- It is mainly used for **checked exceptions**.

Example:

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader fr = new FileReader("test.txt"); // file
may not exist
        BufferedReader br = new BufferedReader(fr);
        System.out.println(br.readLine());
```

```
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Caught Exception: " + e);
        }
    }
}
```

---

5. finally Block

- The `finally` block always executes **whether exception occurs or not**.
- It is generally used to **close resources** (like files, DB connections).

Example:

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0;
            System.out.println("Result: " + data);
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e);
        } finally {
            System.out.println("Finally block executed
(cleanup code).");
        }
    }
}
```

Output:

```
Caught: java.lang.ArithmeticException: / by zero
Finally block executed (cleanup code).
```