1.Write a program to:

- Read an int value from user input.
- Assign it to a double (implicit widening) and print both.
- Read a double, explicitly cast it to int, then to short, and print results—demonstrate truncation or overflow.

A.
```
package Type_casting;
import java.util.Scanner;
public class Conversion {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int intVal = sc.nextInt();
        double widened = intVal;
        System.out.println("Integer: " + intVal);
        System.out.println("Widened to double: " + widened);
        System.out.print("Enter a double: ");
        double doubleVal = sc.nextDouble();
        int castedInt = (int) doubleVal;
        short castedShort = (short) castedInt;
        System.out.println("Original double: " + doubleVal);
        System.out.println("Casted to int (truncation): " + castedInt);
        System.out.println("Casted to short (possible overflow): " + castedShort);
        sc.close();
    }
}
```

2.Convert an int to String using String.valueOf(...), then back with Integer.parseInt(...). Handle NumberFormatException.

A.
```
import java.util.Scanner;
public class IntStringConversion {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        System.out.print("Enter an integer: ");
        int original = sc.nextInt();
        String str = String.valueOf(original);
        System.out.println("Converted to String: " + str);
        try {
            int parsed = Integer.parseInt(str);
            System.out.println("Parsed back to int: " + parsed);
        } catch (NumberFormatException e) {
            System.out.println("Invalid number format.");
        }
        sc.close();
    }
}
```

3.Compound Assignment Behaviour

1. Initialize int x = 5;.
2. Write two operations:

x = x + 4.5; // Does this compile? Why or why not?

x += 4.5; // What happens here?

3. Print results and explain behavior in comments (implicit narrowing, compile error vs. successful assignment).

A.
```
public class CompoundAssignment {
    public static void main(String[] args) {
        int x = 5;
        // x = x + 4.5; // Compile Error: possible lossy conversion from double to int
        x += 4.5; // Works: implicit narrowing happens, double is cast to int
        System.out.println("x after x += 4.5: " + x); // Output: 9
    }
}
```

- x + 4.5 becomes double, and assigning double to int loses precision. Java doesn’t allow this without an explicit cast.
- Compound assignment does implicit narrowing: (int)(x + 4.5) is done automatically by the compiler.

4.Object Casting with Inheritance

1. Define an Animal class with a method makeSound().
2. Define subclass Dog:
    - Override makeSound() (e.g. "Woof!").
    - Add method fetch().
3. In main:

Dog d = new Dog();

Animal a = d; // upcasting

a.makeSound();

A.
```
package day6;
class Animal {
   void makeSound() {
      System.out.println("Some generic animal sound");
   }
}
class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.println("Woof!");
   }
   void fetch() {
      System.out.println("Dog is fetching.");
   }
}
public class Object_Casting_with_Inheritance {
   public static void main(String[] args) {
      Dog d = new Dog();
      Animal a = d;  // upcasting
      a.makeSound(); // calls Dog's overridden method
```

```
    }
}
```

Mini-Project – Temperature Converter

1. Prompt user for a temperature in Celsius (double).
2. Convert it to Fahrenheit:

```
double fahrenheit = celsius * 9/5 + 32;
```

3. Then cast that fahrenheit to int for display.
4. Print both the precise (double) and truncated (int) values, and comment on precision loss.

A.
```
package day6;

import java.util.Scanner;

public class TempConverter {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter temperature in Celsius: ");
        double celsius = sc.nextDouble();
        double fahrenheit = celsius * 9 / 5 + 32;
        int truncatedFahrenheit = (int) fahrenheit;
        System.out.println("Precise Fahrenheit: " + fahrenheit);
        System.out.println("Truncated Fahrenheit (int): " + truncatedFahrenheit);
        System.out.println("// Truncation loses the decimal precision when casting double to int.");
        sc.close();
    }
}
```

Enum:

1: Days of the Week

Define an enum DaysOfWeek with seven constants. Then in main(), prompt the user to input a day name and:

- Print its position via ordinal().

Confirm if it's a weekend day using a switch or if-statement.

A. package day6;

```
import java.util.Scanner;
enum DaysOfWeek { SUNDAY,MONDAY,TUESDAY,
WEDNESDAY,THURSDAY,FRIDAY,SATURDAY}
public class EnumDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a day of the week: ");
        String input = sc.nextLine().toUpperCase();
        try {
            DaysOfWeek day = DaysOfWeek.valueOf(input);
            System.out.println(day + " is day number " + day.ordinal());
            if(day == DaysOfWeek.SATURDAY || day == DaysOfWeek.SUNDAY) {
                System.out.println(day + " is a weekend.");
            } else {
                System.out.println(day + " is a weekday.");
            }
        } catch (IllegalArgumentException e) {
            System.out.println("Invalid day name.");
        }
        sc.close();
    }
}
```

2: Compass Directions

Create an enum Direction with the values NORTH, SOUTH, EAST, WEST. Write code to:

- Read a Direction from a string using valueOf().

Use switch or if to print movement (e.g. “Move north”).
Test invalid inputs with proper error handling.

A. package day6;

```
import java.util.Scanner;
```

```
enum Direction {NORTH, SOUTH, EAST, WEST}
public class DirectionExample {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter a direction (NORTH, SOUTH, EAST, WEST): ");
      String input = sc.nextLine().toUpperCase();
      try {
         Direction dir = Direction.valueOf(input);
         switch (dir) {
            case NORTH:
               System.out.println("Move north");
               break;
            case SOUTH:
               System.out.println("Move south");
               break;
            case EAST:
               System.out.println("Move east");
               break;
            case WEST:
               System.out.println("Move west");
               break;
         }
      } catch (IllegalArgumentException e) {
         System.out.println("Invalid direction. Please enter NORTH, SOUTH, EAST,
or WEST.");
      }
      sc.close();
   }
}
```

3: Shape Area Calculator

Define enum Shape (CIRCLE, SQUARE, RECTANGLE, TRIANGLE) where each constant:

- Overrides a method double area(double... params) to compute its area.
- E.g., CIRCLE expects radius, TRIANGLE expects base and height. Loop over all constants with sample inputs and print results.

A. pacakage day6;

```
enum Shape {
    CIRCLE {
        double area(double... params) {
            double radius = params[0];
            return Math.PI * radius * radius;
        }
    },
    SQUARE {
        double area(double... params) {
            double side = params[0];
            return side * side;
        }
    },
    RECTANGLE {
        double area(double... params) {
            double length = params[0];
            double width = params[1];
            return length * width;
        }
    },
    TRIANGLE {
        double area(double... params) {
            double base = params[0];
            double height = params[1];
            return 0.5 * base * height;
        }
    };
```

```
    abstract double area(double... params);
}
public class ShapeAreaCalculator {
    public static void main(String[] args) {
        System.out.println("Area of Shapes with Sample Inputs:");
        for (Shape shape : Shape.values()) {
            double result = 0.0;
            switch (shape) {
                case CIRCLE:
                    result = shape.area(5); // radius = 5
                    break;
                case SQUARE:
                    result = shape.area(4); // side = 4
                    break;
                case RECTANGLE:
                    result = shape.area(6, 3); // length = 6, width = 3
                    break;
                case TRIANGLE:
                    result = shape.area(10, 2); // base = 10, height = 2
                    break;
            }
            System.out.println(shape + " area: " + result);
        }
    }
}
```

4.Card Suit & Rank

Redesign a Card class using two enums: Suit (CLUBS, DIAMONDS, HEARTS, SPADES) and Rank (ACE…KING).
Then implement a Deck class to:

- Create all 52 cards.
- Shuffle and print the order.

A.
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}
enum Rank {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    EIGHT, NINE, TEN, JACK, QUEEN, KING
}
class Card {
    private final Suit suit;
    private final Rank rank;
    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }
    public String toString() {
        return rank + " of " + suit;
    }
}
class Deck {
    private final List<Card> cards = new ArrayList<>();
    public Deck() {
        for (Suit suit : Suit.values()) {
            for (Rank rank : Rank.values()) {
                cards.add(new Card(suit, rank));
            }
        }
    }
```

```
    public void shuffle() {
        Collections.shuffle(cards);
    }
    public void printDeck() {
        for (Card card : cards) {
            System.out.println(card);
        }
    }
}
public class CardGame {
    public static void main(String[] args) {
        Deck deck = new Deck();
        System.out.println("Shuffled deck of 52 cards:");
        deck.shuffle();
        deck.printDeck();
    }
}
```

5: Priority Levels with Extra Data

Implement enum PriorityLevel with constants (LOW, MEDIUM, HIGH, CRITICAL), each having:

- A numeric severity code.
- A boolean isUrgent() if severity ≥ some threshold.
  Print descriptions and check urgency.

A.
```
package day6;
enum PriorityLevel { LOW(1),MEDIUM(2), HIGH(3),CRITICAL(4);
    private final int severity;
    PriorityLevel(int severity) {
        this.severity = severity;
    }
    public int getSeverity() {
        return severity;
    }
```

```
    public boolean isUrgent() {
        return severity >= 3; // HIGH or CRITICAL is considered urgent
    }
    public String toString() {
        return name() + " (Severity: " + severity + ", Urgent: " + isUrgent() + ")";
    }
}
public class PriorityLevelDemo {
    public static void main(String[] args) {
        for (PriorityLevel level : PriorityLevel.values()) {
            System.out.println(level);
        }
    }
}
```

6: Traffic Light State Machine

Implement enum TrafficLight implementing interface State, with constants RED, GREEN, YELLOW.
Each must override State next() to transition in the cycle.
Simulate and print six transitions starting from RED.

A.

```
package day6;
interface State {
    State next();
}
enum TrafficLight implements State {
    RED {
        public State next() {
            return GREEN;
        }
    },
    GREEN {
        public State next() {
            return YELLOW;
        }
```

```
    },
    YELLOW {
        public State next() {
            return RED;
        }
    };
}
public class TrafficSimulation {
    public static void main(String[] args) {
        State current = TrafficLight.RED;

        for (int i = 0; i < 6; i++) {
            System.out.println("Current Light: " + current);
            current = current.next();
        }
    }
}
```

7: Difficulty Level & Game Setup

Define enum Difficulty with EASY, MEDIUM, HARD.
Write a Game class that takes a Difficulty and prints logic like:

- EASY → 3000 bullets, MEDIUM → 2000, HARD → 1000.
  Use a switch(diff) inside constructor or method.

A.
```
package day6;
enum Difficulty {
    EASY,
    MEDIUM,
    HARD
}
public class Game {
    private int bullets;
    public Game(Difficulty diff) {
        switch (diff) {
```

```
            case EASY:
                bullets = 3000;
                break;
            case MEDIUM:
                bullets = 2000;
                break;
            case HARD:
                bullets = 1000;
                break;
        }
        System.out.println("Difficulty: " + diff);
        System.out.println("Starting bullets: " + bullets);
    }
    public static void main(String[] args) {
        new Game(Difficulty.EASY);
        new Game(Difficulty.MEDIUM);
        new Game(Difficulty.HARD);
    }
}
```

8: Calculator Operations Enum

Create enum Operation (PLUS, MINUS, TIMES, DIVIDE) with an eval(double a, double b) method.
Implement two versions:

- One using a switch(this) inside eval.
- Another using constant-specific method overrides for eval. Compare both designs.

A.
```
package day6;
// Version 1: Using switch(this) inside eval
enum OperationSwitch {
    PLUS, MINUS, TIMES, DIVIDE;
    public double eval(double a, double b) {
        switch (this) {
```

```
            case PLUS:
                return a + b;
            case MINUS:
                return a - b;
            case TIMES:
                return a * b;
            case DIVIDE:
                if (b == 0) {
                    System.out.println("Warning: Division by zero");
                    return 0;
                }
                return a / b;
            default:
                throw new IllegalStateException("Unknown operation: " + this);
        }
    }
}
// Version 2: Using constant-specific method overrides
enum OperationOverride {
    PLUS {
        public double eval(double a, double b) {
            return a + b;
        }
    },
    MINUS {
        public double eval(double a, double b) {
            return a - b;
        }
    },
    TIMES {
        public double eval(double a, double b) {
```

```
            return a * b;
        }
    },
    DIVIDE {
        public double eval(double a, double b) {
            if (b == 0) {
                System.out.println("Warning: Division by zero");
                return 0;
            }
            return a / b;
        }
    };
    public abstract double eval(double a, double b);
}
// Main class to test and compare both implementations
public class OperationTest {
    public static void main(String[] args) {
        double a = 10, b = 5;
        System.out.println("=== Using OperationSwitch (switch inside eval) ===");
        for (OperationSwitch op : OperationSwitch.values()) {
            System.out.println(op + ": " + a + " and " + b + " => " + op.eval(a, b));
        }
        System.out.println("\n=== Using OperationOverride (constant-specific
methods) ===");
        for (OperationOverride op : OperationOverride.values()) {
            System.out.println(op + ": " + a + " and " + b + " => " + op.eval(a, b));
        }
    }
}
```

10: Knowledge Level from Score Range

Define enum KnowledgeLevel with constants BEGINNER, ADVANCED, PROFESSIONAL, MASTER.
Use a static method fromScore(int score) to return the appropriate enum:

- 0–3 → BEGINNER, 4–6 → ADVANCED, 7–9 → PROFESSIONAL, 10 → MASTER.
  Then print the level and test boundary conditions.

A. package day6;

```
enum KnowledgeLevel {
   BEGINNER, ADVANCED, PROFESSIONAL, MASTER;
   public static KnowledgeLevel fromScore(int score) {
      if (score >= 0 && score <= 3) return BEGINNER;
      else if (score >= 4 && score <= 6) return ADVANCED;
      else if (score >= 7 && score <= 9) return PROFESSIONAL;
      else if (score == 10) return MASTER;
      else throw new IllegalArgumentException("Score must be between 0 and 10");
   }
}
public class KnowledgeLevelTest {
   public static void main(String[] args) {
      int[] testScores = {0,3,4,6,7,9,10};
      for (int score : testScores) {
         KnowledgeLevel level = KnowledgeLevel.fromScore(score);
         System.out.println("Score: "+score+" => Level: "+level);
      }
   }
}
```

Exception handling

1: Division & Array Access

Write a Java class ExceptionDemo with a main method that:

1. Attempts to divide an integer by zero and access an array out of bounds.

2. Wrap each risky operation in its own try-catch:
    - Catch only the specific exception types: ArithmeticException and ArrayIndexOutOfBoundsException.
    - In each catch, print a user-friendly message.
3. Add a finally block after each try-catch that prints "Operation completed.".

Example structure:

```
try {
   // division or array access
} catch (ArithmeticException e) {
   System.out.println("Division by zero is not allowed!");
} finally {
   System.out.println("Operation completed.");
}
```

A.
```
package day6;
public class ExceptionDemo {
   public static void main(String[] args) {
      try {
         int result = 10 / 0;
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Division by zero is not allowed!");
      } finally {
         System.out.println("Operation completed.");
      }
      int[] arr = {1, 2, 3};
      try {
         int val = arr[5];
         System.out.println("Value: " + val);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array index out of bounds!");
```

```
        } finally {
            System.out.println("Operation completed.");
        }
    }
}
```

2: Throw and Handle Custom Exception

Create a class OddChecker:

1. Implement a static method:

public static void checkOdd(int n) throws OddNumberException { /* ... */ }

2. If n is odd, throw a custom checked exception OddNumberException with message "Odd number: " + n.
3. In main:
    - Call checkOdd with different values (including odd and even).
    - Handle exceptions with try-catch, printing e.getMessage() when caught.

Define the exception like:

```
public class OddNumberException extends Exception {
    public OddNumberException(String message) { super(message); }
}
```

A.
```
package day6;
public class OddNumberException extends Exception {
    public OddNumberException(String message) {
        super(message);
    }
}
public class OddChecker {
    public static void checkOdd(int n) throws OddNumberException {
        if (n % 2 != 0) {
            throw new OddNumberException("Odd number: " + n);
        }
```

```
    }
    public static void main(String[] args) {
        int[] numbers = {2, 3, 4, 5, 6};
        for (int num : numbers) {
            try {
                checkOdd(num);
                System.out.println(num + " is even.");
            } catch (OddNumberException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

File Handling with Multiple Catches

Create a class FileReadDemo:

1. In main, call a method readFile(String filename) that declares throws FileNotFoundException, IOException.
2. In readFile, use FileReader (or BufferedReader) to open and read the first line of the file.
3. Handle exceptions in main using separate catch blocks:
    - catch (FileNotFoundException e) → print "File not found: " + filename
    - catch (IOException e) → print "Error reading file: " + e.getMessage()"
4. Include a finally block that prints "Cleanup done." regardless of outcome.

A. package day6;

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
public class FileReadDemo {
   public static void readFile(String filename) throws FileNotFoundException, IOException {
      BufferedReader br = new BufferedReader(new FileReader(filename));
      String line = br.readLine();
      System.out.println("First line: " + line);
      br.close();
   }
   public static void main(String[] args) {
      String filename = "my.txt";
      try {
         readFile(filename);
      } catch (FileNotFoundException e) {
         System.out.println("File not found: " + filename);
      } catch (IOException e) {
         System.out.println("Error reading file: " + e.getMessage());
      } finally {
         System.out.println("Cleanup done.");
      }
   }
}
```

4: Multi-Exception in One Try Block

Write a class MultiExceptionDemo:

- In a single try block, perform:
  - Opening a file
  - Parsing its first line as integer
  - Dividing 100 by that integer
- Use multiple catch blocks in this order:

1. FileNotFoundException
2. IOException
3. NumberFormatException

4. ArithmeticException

- In each catch, print a tailored message:
  - File not found
  - Problem reading file
  - Invalid number format
  - Division by zero
- Finally, print "Execution completed".

A.
```
package Exceptions;
import java.io.*;
public class MultiExceptionDemo {
   public static void main(String[] args) {
      String filename = "my.txt";
      BufferedReader reader = null;
      try {
         reader = new BufferedReader(new FileReader(filename));
         String line = reader.readLine();
         int number = Integer.parseInt(line);
         int result = 100 / number;
         System.out.println("Result: " + result);
      }
      catch (FileNotFoundException e) {
         System.out.println("File not found: " + filename);
      }
      catch (IOException e) {
         System.out.println("Problem reading file");
      }
      catch (NumberFormatException e) {
         System.out.println("Invalid number format");
      }
      catch (ArithmeticException e) {
         System.out.println("Division by zero");
```

```
        }
        finally {
            System.out.println("Execution completed");
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                }
            }
        }
    }
}
```