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.

```
import java.util.Scanner;
public class TypeCastingDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer value: ");
    int intValue = scanner.nextInt();

    double widenedDouble = intValue;
    System.out.println("Original int value: " + intValue);
    System.out.println("Implicitly widened to double: " + widenedDouble);
    System.out.print("Enter a double value: ");
    double doubleValue = scanner.nextDouble();

    int intFromDouble = (int) doubleValue;

    short shortFromInt = (short) intFromDouble;

    System.out.println("Original double value: " + doubleValue);
    System.out.println("Explicitly cast to int (truncation): " + intFromDouble);
    System.out.println("Explicitly cast to short (possible overflow): " +
shortFromInt);

    scanner.close();
  }
}
Output:
Enter an integer value: 12345
Original int value: 12345
Implicitly widened to double: 12345.0
Enter a double value: 12345.6789
Original double value: 12345.6789
Explicitly cast to int (truncation): 12345
Explicitly cast to short (possible overflow): 12345
```

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

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

    System.out.print("Enter an integer: ");
    int originalInt = scanner.nextInt();
    String intAsString = String.valueOf(originalInt);
    System.out.println("Integer as String: " + intAsString);
    try {
      int parsedInt = Integer.parseInt(intAsString);
      System.out.println("Parsed integer: " + parsedInt);
    } catch (NumberFormatException e) {
      System.out.println("Error: String is not a valid integer.");
    }
    System.out.print("Enter a string to parse as integer: ");
    scanner.nextLine();  // consume leftover newline
    String userInput = scanner.nextLine();
```

```
        try {
            int userParsedInt = Integer.parseInt(userInput);
            System.out.println("Parsed integer from user input: " + userParsedInt);
        } catch (NumberFormatException e) {
            System.out.println("Error: '" + userInput + "' is not a valid integer.");
        }

        scanner.close();
    }
}
```

Output:

```
Enter an integer: 1234
Integer as String: 1234
Parsed integer: 1234
Enter a string to parse as integer: 56abc
Error: '56abc' is not a valid integer.
```

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).

```
public class CompoundAssignmentDemo {
    public static void main(String[] args) {
        int x = 5;
        // x = x + 4.5; // This line causes a compile error — uncomment to see it.

        x += 4.5; // This compiles successfully, because compound assignment operator
+= implicitly casts the result back to the type of x (here, int).

        System.out.println("Value of x after 'x += 4.5': " + x);
    }
}
```

Output:

```
Value of x after 'x += 4.5': 9
```

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();

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

class Dog extends Animal {
    public void makeSound() {
        System.out.println("Woof!");
    }
    public void fetch() {
        System.out.println("Dog is fetching the ball");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        Animal a = d;  // upcasting: Dog object referred by Animal reference

        a.makeSound(); // Calls Dog's overridden makeSound() due to polymorphism

        // a.fetch();  // This would not compile because Animal does not have fetch()

        // To call fetch(), you need to downcast back to Dog
        if (a instanceof Dog) {
            ((Dog) a).fetch();  // downcasting to call fetch()
        }
    }
}
```

Output:

```
Woof!
Dog is fetching the ball
```

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.

```
import java.util.Scanner;

public class TemperatureConverter {

  public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);


    System.out.print("Enter temperature in Celsius: ");

    double celsius = scanner.nextDouble();



    double fahrenheit = celsius * 9 / 5 + 32;


    int fahrenheitInt = (int) fahrenheit;


    System.out.println("Precise Fahrenheit value (double): " + fahrenheit);

    System.out.println("Truncated Fahrenheit value (int): " + fahrenheitInt);

    scanner.close();

  }

}
```

Output:

```
Enter temperature in Celsius: 37.0

Precise Fahrenheit value (double): 98.6

Truncated Fahrenheit value (int): 98
```

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.

```
import java.util.Scanner;

enum DaysOfWeek {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}

public class DaysOfWeekDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a day of the week (e.g., Monday): ");
        String input = scanner.nextLine().trim().toUpperCase();

        try {
            DaysOfWeek day = DaysOfWeek.valueOf(input);
            System.out.println(day + " is day number " + day.ordinal());
            switch (day) {
                case SATURDAY, SUNDAY-> System.out.println(day + " is a weekend day.");
                default-> System.out.println(day + " is a weekday.");
            }

        } catch (IllegalArgumentException e) {
            System.out.println("Invalid day name entered. Please enter a valid day.");
        }

        scanner.close();
    }
}
```

Output:

```
Enter a day of the week (e.g., Monday): Friday
FRIDAY is day number 5
FRIDAY is a weekday.
```

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.

```
import java.util.Scanner;
enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST
}

public class CompassDirectionDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a direction (NORTH, SOUTH, EAST, WEST): ");
        String input = scanner.nextLine().trim().toUpperCase();

        try {
            Direction direction = Direction.valueOf(input);
            switch (direction) {
                case NORTH-> System.out.println("Move north");
                case SOUTH-> System.out.println("Move south");
                case EAST-> System.out.println("Move east");
                case WEST-> System.out.println("Move west");
            }

        } catch (IllegalArgumentException e) {
            System.out.println("Invalid direction entered. Please enter NORTH, SOUTH,
EAST, or WEST.");
        }

        scanner.close();
    }
}
```

Output:

```
Enter a direction (NORTH, SOUTH, EAST, WEST): east
Move east
```

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.

```
import java.util.Scanner;

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) {

        double circleRadius = 5;

        double squareSide = 4;

        double rectangleLength = 6;

        double rectangleWidth = 3;

        double triangleBase = 4;

        double triangleHeight = 5;


        for (Shape shape : Shape.values()) {

            double area;

            switch (shape) {

                case CIRCLE:

                    area = shape.area(circleRadius);

                    System.out.printf("Area of %s with radius %.2f = %.2f%n", shape,
circleRadius, area);

                    break;

                case SQUARE:

                    area = shape.area(squareSide);

                    System.out.printf("Area of %s with side %.2f = %.2f%n", shape, squareSide,
area);
```

```
                break;
            case RECTANGLE:
                area = shape.area(rectangleLength, rectangleWidth);
                System.out.printf("Area of %s with length %.2f and width %.2f = %.2f%n",
shape, rectangleLength, rectangleWidth, area);
                break;
            case TRIANGLE:
                area = shape.area(triangleBase, triangleHeight);
                System.out.printf("Area of %s with base %.2f and height %.2f = %.2f%n",
shape, triangleBase, triangleHeight, area);
                break;
            }
        }
    }
}
```

Output:

```
Area of CIRCLE with radius 5.00 = 78.54
Area of SQUARE with side 4.00 = 16.00
Area of RECTANGLE with length 6.00 and width 3.00 = 18.00
Area of TRIANGLE with base 4.00 and height 5.00 = 10.00
```

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.

```
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 Suit getSuit() {
           return suit;
       }

       public Rank getRank() {
           return rank;
       }
       public String toString() {
           return rank + " of " + suit;
       }
   }

   class Deck {
       private final List<Card> cards;

       public Deck() {
           cards = new ArrayList<>();
           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 CardDeckDemo {
       public static void main(String[] args) {
           Deck deck = new Deck();
           System.out.println("Original deck:");
           deck.printDeck();
           deck.shuffle();
           System.out.println("\nShuffled deck:");
           deck.printDeck();
       }
   }
```

```
Output:
Original deck:
ACE of CLUBS
TWO of CLUBS
...
KING of SPADES

Shuffled deck:
TEN of HEARTS
QUEEN of DIAMONDS
ACE of SPADES
...
```

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.

```
enum PriorityLevel {
    LOW(1),
    MEDIUM(3),

    HIGH(5),

    CRITICAL(10);


    private final int severity;


    PriorityLevel(int severity) {

        this.severity = severity;

    }


    public int getSeverity() {

        return severity;

    }


    public boolean isUrgent() {

        return severity >= 5;

    }

    public String toString() {

        return this.name() + " (severity " + severity + ")";

    }

}


public class PriorityLevelDemo {

    public static void main(String[] args) {

        for (PriorityLevel level : PriorityLevel.values()) {

            System.out.printf("%s- Urgent? %b%n", level, level.isUrgent());

        }

    }

}
```

```
Output:
LOW (severity 1)- Urgent? false
MEDIUM (severity 3)- Urgent? false
HIGH (severity 5)- Urgent? true
CRITICAL (severity 10)- Urgent? true
```

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.

```java
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 TrafficLightSimulator {
    public static void main(String[] args) {
        State current = TrafficLight.RED;

        System.out.println("Starting at: " + current);
        for (int i = 0; i < 6; i++) {
            current = current.next();
            System.out.println("Transition to: " + current);
        }
    }
}
```

Output:

```
Starting at: RED
Transition to: GREEN
Transition to: YELLOW
Transition to: RED
Transition to: GREEN
Transition to: YELLOW
Transition to: RED
```

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.

```
enum Difficulty {
  EASY,
  MEDIUM,
  HARD
}

class Game {
  private Difficulty difficulty;
  private int bullets;

  public Game(Difficulty difficulty) {
    this.difficulty = difficulty;
    setupGame();
  }

  private void setupGame() {
    switch (difficulty) {
      case EASY-> bullets = 3000;
      case MEDIUM-> bullets = 2000;
      case HARD-> bullets = 1000;
      default-> bullets = 0;
    }
  }

  public void printSetup() {
    System.out.printf("Difficulty: %s, Bullets: %d%n", difficulty, bullets);
  }
}


public class GameDemo {
    public static void main(String[] args) {
        Game easyGame = new Game(Difficulty.EASY);
        easyGame.printSetup();


        Game mediumGame = new Game(Difficulty.MEDIUM);
        mediumGame.printSetup();


        Game hardGame = new Game(Difficulty.HARD);
        hardGame.printSetup();
    }
}
```

Output:

```
Difficulty: EASY, Bullets: 3000
Difficulty: MEDIUM, Bullets: 2000
Difficulty: HARD, Bullets: 1000
```

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.

```
enum OperationSwitch {
    PLUS, MINUS, TIMES, DIVIDE;


    public double eval(double a, double b) {
        return switch (this) {
            case PLUS-> a + b;
            case MINUS-> a- b;
            case TIMES-> a * b;
            case DIVIDE-> {
                if (b == 0) throw new ArithmeticException("Division by zero");
                yield a / b;
            }
        };
    }
}
public class CalculatorDemo {
    public static void main(String[] args) {
        double a = 10, b = 5;

        for (OperationSwitch op : OperationSwitch.values()) {
            System.out.printf("%s: %.2f%n", op, op.eval(a, b));
        }
    }
}
```

Output:

```
PLUS: 15.00
MINUS: 5.00
TIMES: 50.00
DIVIDE: 2.00
```

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.

```
enum KnowledgeLevel {

   BEGINNER,

   ADVANCED,

   PROFESSIONAL,

   MASTER;


   public static KnowledgeLevel fromScore(int score) {

      if (score >= 0 && score <= 3) {

         return BEGINNER;

      } else if (score <= 6) {

         return ADVANCED;

      } else if (score <= 9) {

         return PROFESSIONAL;

      } else if (score == 10) {

         return MASTER;

      } else {

         throw new IllegalArgumentException("Score must be between 0 and 10");

      }

   }

}


public class KnowledgeLevelDemo {
    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.printf("Score: %d-> Level: %s%n", score, level);
        }
    }
}
```

Output:

```
Score: 0-> Level: BEGINNER
Score: 3-> Level: BEGINNER
Score: 4-> Level: ADVANCED
Score: 6-> Level: ADVANCED
Score: 7-> Level: PROFESSIONAL
Score: 9-> Level: PROFESSIONAL
Score: 10-> Level: MASTER
```

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.");
}

public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int result = a / b;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Division by zero is not allowed!");
        } finally {
            System.out.println("Operation completed.");
        }

        System.out.println();
        try {
            int[] array = {1, 2, 3};
            int value = array[5];
            System.out.println("Value: " + value);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds!");
        } finally {
            System.out.println("Operation completed.");
        }
    }
}
```

Output:

```
Division by zero is not allowed!
Operation completed.

Array index is out of bounds!
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); }
}
```

```
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[] testValues = {2, 7, 10, 15, 22};

        for (int value : testValues) {
            try {
                checkOdd(value);
                System.out.println(value + " is even.");
            } catch (OddNumberException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Output:

```
2 is even.
Odd number: 7
10 is even.
Odd number: 15
22 is even.
```

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.

```
public class FileReadDemo {


  public static void readFile(String filename) throws FileNotFoundException,
IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
      String firstLine = reader.readLine();
      System.out.println("First line: " + firstLine);
    }
  }


  public static void main(String[] args) {
    String filename = "example.txt";  // Change this to a valid filename to test


    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.");
        }
    }
}
```

Output:

```
First line: Hello, world!
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".

```
import java.io.*;
public class MultiExceptionDemo {
  public static void main(String[] args) {
    String filename = "numbers.txt";  // Change as needed

    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
      // Read first line
      String line = reader.readLine();

      // Parse integer from line
      int number = Integer.parseInt(line);

      // Divide 100 by the parsed number
      int result = 100 / number;

      System.out.println("Result: " + result);

    } catch (FileNotFoundException e) {
      System.out.println("File not found");
    } 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");
    }
  }
}
```

Output:

```
Valid number (25):
Result: 4
Execution completed

Number zero (0):
Division by zero
Execution completed

Invalid number (abc):
Invalid number format
Execution completed

File missing:
File not found
Execution completed
```