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

```
import java.util.Scanner;


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


        System.out.print("Enter an integer: ");
        int intValue = scanner.nextInt();


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



        int castedInt = (int) doubleValue;


        short castedShort = (short) castedInt;


        System.out.println("Original double: " + doubleValue);
        System.out.println("Explicitly cast to int (truncation): " + castedInt);
        System.out.println("Then cast to short (possible overflow): " + castedShort);


        scanner.close();
    }
}
```

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 converted to String: " + intAsString);



        try {
            int parsedInt = Integer.parseInt(intAsString);
            System.out.println("String parsed back to int: " + parsedInt);
        } catch (NumberFormatException e) {
            System.out.println("Error: Invalid number format.");
        }

        scanner.close();
    }
}
```

3. Write two operations: x = x + 4.5; // Does this compile? Why or why not? x += 4.5; // What happens here?

1. x = x + 4.5;

This does not compile if x is declared as an int because:

- The expression x + 4.5 involves adding an int (x) and a double (4.5).
- Java promotes the int x to double, so the result is a double.
- You cannot assign a double result directly back to an int without explicit casting.
- Hence, a compilation error occurs: "possible lossy conversion from double to int."

2. x += 4.5;

This compiles even if x is an int because:

- The compound assignment operator += implicitly casts the result back to the type of x.
- It is effectively equivalent to x = (int)(x + 4.5);
- This means the fractional part (0.5) will be truncated and only the integral part added to x.
- No compilation error occurs here.

4. 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 = 10;
        // This line will cause a compile error:
        // error: incompatible types: possible lossy conversion from double to int
        // x = x + 4.5;
        // Explanation:
        // x is int, 4.5 is double, so x + 4.5 is double.
        // Assigning double to int without explicit cast causes compile error (implicit narrowing not allowed).

        // Correct usage with explicit cast - uncomment to test
        // x = (int)(x + 4.5); // Truncates 14.5 to 14, explicit narrowing cast.

        // Compound assignment operator performs implicit cast:
        x += 4.5; // Equivalent to: x = (int)(x + 4.5);
                // This compiles successfully because compound assignment implicitly casts result to int.

        System.out.println("Value of x after x += 4.5: " + x);
        // Output: 14
        // Explanation: 4.5 is added to 10 resulting in 14.5,
        // implicit cast truncates decimal part, so x becomes 14.
    }
}
```

5. Define an Animal class with a method makeSound(). 2. Define subclass Dog: o Override makeSound() (e.g. "Woof!"). o 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 {
    // Override makeSound() to print "Woof!"
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }



    public void fetch() {
        System.out.println("Dog is fetching the ball.");
    }
}

public class AnimalTest {
    public static void main(String[] args) {
        Dog d = new Dog();
        Animal a = d;


        a.makeSound();                  // Output: Woof!


:
        if (a instanceof Dog) {
            ((Dog) a).fetch();  // Calls Dog-specific fetch method
        }
    }
}
```

6. 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 truncatedFahrenheit = (int) fahrenheit;

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

        scanner.close();
    }
}
```

7. 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;


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


    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("Position in week (ordinal): " + 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 entered.");
        }

        scanner.close();
    }
}
```

8. 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;

public class CompassDirectionDemo {
    enum Direction {
        NORTH,
        SOUTH,
        EAST,
        WEST
    }

    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.");
        }

        scanner.close();
    }
}
```

9. 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.

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

        abstract double area(double... params);
    }

    public static void main(String[] args) {
        double radius = 5;
        double side = 4;
        double length = 6;
        double width = 3;
        double base = 8;
        double height = 5;


        for (Shape shape : Shape.values()) {
            double areaValue;
            switch (shape) {
                case CIRCLE:
                    areaValue = shape.area(radius);
                    System.out.println("Area of Circle with radius " + radius + " is " + areaValue);
                    break;
                case SQUARE:
                    areaValue = shape.area(side);
                    System.out.println("Area of Square with side " + side + " is " + areaValue);
                    break;
                case RECTANGLE:
                    areaValue = shape.area(length, width);
                    System.out.println("Area of Rectangle with length " + length + " and width " +
    width + " is " + areaValue);
                    break;
                case TRIANGLE:
                    areaValue = shape.area(base, height);
                    System.out.println("Area of Triangle with base " + base + " and height " + height +
    " is " + areaValue);
                    break;
                default:
                    System.out.println("Unknown shape");
            }
          }
       }
    }
```

10. 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;

public class CardDeckDemo {
   enum Suit {
      CLUBS, DIAMONDS, HEARTS, SPADES
   }

   enum Rank {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    EIGHT, NINE, TEN, JACK, QUEEN, KING
}

static class Card {
    private final Suit suit;
    private final Rank rank;

    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}

static 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 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();
    }
}
```

11. 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.

```
public class PriorityLevelDemo {
    enum PriorityLevel {
        LOW(1),
        MEDIUM(3),
        HIGH(5),
        CRITICAL(8);

        private final int severityCode;

        PriorityLevel(int severityCode) {
            this.severityCode = severityCode;
        }

        public int getSeverityCode() {
            return severityCode;
        }

        public boolean isUrgent() {
            return severityCode >= 5;  // threshold for urgency
        }
    }

    public static void main(String[] args) {
        for (PriorityLevel level : PriorityLevel.values()) {
            System.out.printf("Priority: %s, Severity code: %d, Is urgent? %b%n",
                    level, level.getSeverityCode(), level.isUrgent());
        }
    }
}
```

12. 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.

```
public class TrafficLightStateMachine {
  interface State {
    State next();
  }

  enum TrafficLight implements State {
    RED {
      @Override
      public State next() {
        return GREEN;
      }
    },
    GREEN {
      @Override
      public State next() {
        return YELLOW;
      }
    },
    YELLOW {
      @Override
      public State next() {
        return RED;
      }
    }
  }
```

```
    public static void main(String[] args) {
        State current = TrafficLight.RED;
        System.out.println("Starting state: " + current);

        for (int i = 1; i <= 6; i++) {
            current = current.next();
            System.out.println("Transition " + i + ": " + current);
        }
    }
}
```

13. 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.

```
public class GameSetup {
    enum Difficulty {
        EASY,
        MEDIUM,
        HARD
    }

    static class Game {
        private final Difficulty difficulty;
        private final int bullets;

        public Game(Difficulty difficulty) {
            this.difficulty = difficulty;
            this.bullets = bulletsForDifficulty(difficulty);
            System.out.println("Game difficulty: " + difficulty);
            System.out.println("Bullets assigned: " + bullets);
        }

        private int bulletsForDifficulty(Difficulty diff) {
            return switch (diff) {
                case EASY -> 3000;
                case MEDIUM -> 2000;
                case HARD -> 1000;
            };
        }
    }

    public static void main(String[] args) {
        new Game(Difficulty.EASY);
        new Game(Difficulty.MEDIUM);
        new Game(Difficulty.HARD);
    }
}
```

14. 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.

```
public class CalculatorOperationDemo {

    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 -> a / b;
            };
        }
    }

        enum OperationOverride {
        PLUS {
            @Override
            public double eval(double a, double b) {
                return a + b;
            }
        },
        MINUS {
            @Override
            public double eval(double a, double b) {
                return a - b;
            }
        },
        TIMES {
            @Override
            public double eval(double a, double b) {
                return a * b;
            }
        },
        DIVIDE {
            @Override
            public double eval(double a, double b) {
                return a / b;
            }
        };

        public abstract double eval(double a, double b);
    }

    public static void main(String[] args) {
        double a = 10, b = 5;



        System.out.println("Using switch-based eval:");
        for (OperationSwitch op : OperationSwitch.values()) {
            System.out.printf("%s: %f%n", op, op.eval(a, b));
        }



        System.out.println("\nUsing constant-specific overrides:");
        for (OperationOverride op : OperationOverride.values()) {
            System.out.printf("%s: %f%n", op, op.eval(a, b));
        }
    }
}
```

15. 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

```
public class KnowledgeLevelDemo {
  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 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: %s%n", score, level);
        }
    }
}
```

16. : Division & Array Access

Write a Java class ExceptionDemo with a main method that:

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

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) {
        // Division by zero
        try {
            int a = 10;
            int b = 0;
            int result = a / b;  // risky: division by zero
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Division by zero is not allowed!");
        } finally {
            System.out.println("Operation completed.");
        }

        // Array index out of bounds
        try {
            int[] arr = {1, 2, 3};
            int invalidAccess = arr[5];  // risky: accessing invalid index
            System.out.println("Accessed element: " + invalidAccess);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds!");
        } finally {
            System.out.println("Operation completed.");
        }
    }
}
```

17. **2: Throw and Handle Custom Exception Create a class OddChecker:**

    Implement a static method:

public static void checkOdd(int n) throws

OddNumberException { /* ... */ }

If n is odd, throw a custom checked exception

OddNumberException with message "Odd number: " + n.

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); }

}

```
// Custom checked exception class
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, 5, 8, 11, 14};


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

18. 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()"

1. **Include a finally block that prints "Cleanup done." regardless of outcom**e.

m

```
}

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 {
    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 = "sample.txt"; // replace with actual file path

    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.");
    }
  }
}
```

19. Write a class MultiExceptionDemo:

- **In a single try block, perform: o Opening a file o Parsing its first line as integer o 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: o File not found o Problem reading file o Invalid number format o Division by zero**
- **Finally, print "Execution completed".**

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;


public class MultiExceptionDemo {

  public static void main(String[] args) {

    String filename = "sample.txt"; // Change to your file path


    try (BufferedReader 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");
        } 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");
        }
    }
}
```