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.

ANS:

```
package Day_6;

import java.util.Scanner;

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


      System.out.print("Enter an int: ");
      int intValue = sc.nextInt();
      double widened = intValue;
      System.out.println("Int value: " + intValue);
      System.out.println("Widened double: " + widened);


      System.out.print("Enter a double: ");
      double doubleValue = sc.nextDouble();
      int narrowedInt = (int) doubleValue;
      short narrowedShort = (short) narrowedInt;
      System.out.println("Double value: " + doubleValue);
      System.out.println("Casted to int: " + narrowedInt);
      System.out.println("Casted to short: " + narrowedShort);
   }
}
```

Output:

```
Widened double: 6.0
Enter a double: 23.2
Double value: 23.2
Casted to int: 23
Casted to short: 23
```

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

ANS:

```java
package Day_6;

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 number = sc.nextInt();

      String str = String.valueOf(number);
      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");
      }
   }
}
```

Output:

```
Enter an integer: 1234
Converted to String: 1234
Parsed back to int: 1234
```

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).

ANS:

```java
package Day_6;

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

```
        x += 4.5;
        System.out.println("After x += 4.5, x = " + x); // implicit narrowing (x = (int)(x + 4.5))
    }
}
```

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();
```

ANS:

```
package Day_6;

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

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }

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

public class CastingInheritance {
    public static void main(String[] args) {
        Dog d = new Dog();
        Animal a = d; // upcasting
        a.makeSound(); // calls Dog's method
```

```
    }
}
```
Output:

Woof!

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.

ANS:

```
package Day_5;


import java.util.Scanner;


public class TemperatureConverter {
    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 truncatedF = (int) fahrenheit;


        System.out.println("Fahrenheit (double): " + fahrenheit);
        System.out.println("Fahrenheit (truncated int): " + truncatedF);
    }
}
```
Output:

Enter temperature in Celsius: 32.2
Fahrenheit (double): 89.96000000000001
Fahrenheit (truncated int): 89

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.

```
package Day_6;

import java.util.Scanner;

public class DaysOfWeek {
        enum Week {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter day name: ");
    String input = sc.next();

    try {
        Week day = Week.valueOf(input);

      System.out.println("Day: " + day);
      System.out.println("Position: " + day.ordinal());

      if (day == Week.Saturday || day == Week.Sunday) {
        System.out.println(day + " is a Weekend.");
      } else {
        System.out.println(day + " is a Weekday.");
      }
    } catch (IllegalArgumentException e) {
      System.out.println("Invalid day entered.");
    }
  }
}
```

Output:

Enter day name: Sunday
Day: Sunday

Position: 6
Sunday is a Weekend

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.

```
package Day_6;

import java.util.Scanner;

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

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter direction: ");
        String input = sc.next();

        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 entered.");
        }
    }
}
```

Output:

Enter direction: 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.

```
package Day_6;

enum Shape {
    CIRCLE {
        public double area(double... params) {
            double r = params[0];
            return Math.PI * r * r;
        }
    },
    SQUARE {
        public double area(double... params) {
            double side = params[0];
            return side * side;
        }
    },
    RECTANGLE {
        public double area(double... params) {
            return params[0] * params[1];
        }
    },
    TRIANGLE {
        public double area(double... params) {
            return 0.5 * params[0] * params[1];
        }
    };

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

public class ShapeCalculator {
    public static void main(String[] args) {
        System.out.println("Circle area: " + Shape.CIRCLE.area(5));
        System.out.println("Square area: " + Shape.SQUARE.area(4));
        System.out.println("Rectangle area: " + Shape.RECTANGLE.area(4, 6));
```

```
        System.out.println("Triangle area: " + Shape.TRIANGLE.area(3, 7));
    }
}
```

Output:

```
Circle area: 78.53981633974483
Square area: 16.0
Rectangle area: 24.0
Triangle area: 10.5
```

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.

```
package Day_6;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CardGame {
    enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    enum Rank { ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
KING }

    static class Card {
        Suit suit;
        Rank rank;

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

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

    public static void main(String[] args) {
        List<Card> deck = new ArrayList<>();
        for (Suit s : Suit.values()) {
```

```
            for (Rank r : Rank.values()) {
                deck.add(new Card(s, r));
            }
        }
        Collections.shuffle(deck);
        for (Card c : deck) {
            System.out.println(c);
        }
    }
}
```

Output:

```
THREE of HEARTS
FOUR of SPADES
FIVE of SPADES
KING of CLUBS
QUEEN of DIAMONDS
```

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.

```
package Day_6;

public class PriorityLevel {
    enum Level {
        LOW(1), MEDIUM(2), HIGH(3), CRITICAL(4);

        private int severity;

        Level(int severity) {
            this.severity = severity;
        }

        public boolean isUrgent() {
            return severity >= 3;
        }

        public int getSeverity() {
            return severity;
        }
    }

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

```
        for (Level p : Level.values()) {
            System.out.println(p + " (Severity " + p.getSeverity() + ") Urgent: " + p.isUrgent());
        }
    }
}
```

Output:

```
LOW (Severity 1) Urgent: false
MEDIUM (Severity 2) Urgent: false
HIGH (Severity 3) Urgent: true
CRITICAL (Severity 4) 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.

```
package Day_6;

public class TrafficLightDemo {
    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 static void main(String[] args) {
        State light = TrafficLight.RED;
        for (int i = 0; i < 6; i++) {
            System.out.println(light);
            light = light.next();
        }
    }
}
```

Output:

```
RED
```

GREEN
YELLOW
RED
GREEN
YELLOW

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.

```
package Day_6;

public class GameDifficulty {
   enum Difficulty { EASY, MEDIUM, HARD }

   public static void startGame(Difficulty diff) {
      switch (diff) {
         case EASY:
            System.out.println("Bullets: 3000");
            break;
         case MEDIUM:
            System.out.println("Bullets: 2000");
            break;
         case HARD:
            System.out.println("Bullets: 1000");
            break;
      }
   }

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

Output:

Bullets: 3000
Bullets: 2000
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.

```
package Day_6;

public class OperationExample {
   enum Operation {
      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) { return a / b; }
      };

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

   public static void main(String[] args) {
      System.out.println(Operation.PLUS.eval(5, 3));
      System.out.println(Operation.MINUS.eval(5, 3));
      System.out.println(Operation.TIMES.eval(5, 3));
      System.out.println(Operation.DIVIDE.eval(6, 2));
   }
}
```

Output:

```
8.0
2.0
15.0
3.0
```

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.

```
package Day_6;

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 <= 6) return ADVANCED;
         else if (score <= 9) return PROFESSIONAL;
         else return MASTER;
      }
   }

   public static void main(String[] args) {
      int[] scores = {0, 4, 7, 10};
      for (int s : scores) {
         System.out.println("Score: " + s + ": " + KnowledgeLevel.fromScore(s));
      }
   }
}
```

Output:

```
Score: 0: BEGINNER
Score: 4: ADVANCED
Score: 7: PROFESSIONAL
Score: 10: 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.");
}
```

```
package Day_6;

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

      try {
         int[] arr = new int[3];
         System.out.println(arr[5]); // Out of bounds
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array index out of bounds!");
      } finally {
         System.out.println("Operation completed.");
      }
   }
}
```

Output:

```
Division by zero is not allowed!
Operation completed.
Array index 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); }
}
```

```
package Day_6;

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[] nums = {2, 5, 8};

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

Output:

```
2 is even
Odd number: 5
8 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.

```
package Day_6;

import java.io.*;

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 = "Test.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");
      }
   }
}
```

Output:

File not found: sample.txt
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".

```
package Day_6;

import java.io.*;

public class MultiExceptionDemo {
   public static void main(String[] args) {
      String filename = "Test.txt";

      try {

         BufferedReader br = new BufferedReader(new FileReader(filename));
```

```
            String line = br.readLine();
            int num = Integer.parseInt(line);


            int result = 100 / num;
            System.out.println("Result: " + result);

            br.close();
        }
        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:

```
Result: 0
Execution completed
```