DAY 6

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.

```
class Typecasting {
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a number : ");
                int num = sc.nextInt();
                double d = num;
                System.out.println("Integer value : " + num);
                System.out.println("Widened to double : " + d);
                System.out.println("Enter a double : ");
                double num2 = sc.nextDouble();
                int intValue = (int) num2; // explicit cast
                short shortValue = (short) intValue;
                System.out.println("Double Value : " + num2);
                public System.out.println("Narrowed to int : " + intValue);
                System.out.println("Further narrowed to short: " +
shortValue);

        }
}
```

Output:

```
Enter a number :
45
Integer value : 45
Widened to double : 45.0
```

Enter a double :

54

Double Value : 54.0

Narrowed to int : 54

Further narrowed to short: 54

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

```
public class QuestionTwo {
   public static void main(String[] args) {
      int num = 1234;
      // int to String
      String str = String.valueOf(num);
      System.out.println("Converted int to String: " + str);
      try {
         // String to  int
         int parsedNum = Integer.parseInt(str);
         System.out.println("Converted String back to int: " + parsedNum);
      }
      catch (NumberFormatException e) {
         System.out.println("Error: String is not a valid integer!");
      }
   }
}
```

Output:

Converted int to String: 1234

Converted String back to int: 1234

Compound Assignment Behaviour

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

x = x + 4.5

x += 4.5;

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

```
public class CompoundAssignment {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int x=5;
        // x = x + 4.5; // Compile-time error: cannot assign double to int
    // Explanation: x + 4.5 results in a double. Cannot assign directly to int.
    x += 4.5; // Valid due to implicit narrowing with compound assignment
    // Explanation: x += 4.5 is equivalent to x = (int)(x + 4.5)
    System.out.println(x);
    }
}
```

Output:

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{
        void makeSound() {
            System.out.println("Making sound");
```

```
        }
}
class Dog extends Animal{
        void makeSound() {
            System.out.println("Woof..");
        }
        void fetch() {
            System.out.println("Dog fetches the ball.");
        }
}
public class InheritanceCasting {
        public static void main(String[] args) {
            // TODO Auto-generated method stub
            Dog d = new Dog();
    Animal a = d;
    a.makeSound();
        }
}
```

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.

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

```
        // TODO Auto-generated method stub
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter temperature in Celsius :");
        double tc = sc.nextDouble();
        double fr = tc * 9/5 + 32;
                    int temp = (int)fr;
        System.out.println("Fahrenheit (double): " +fr);
    System.out.println("Fahrenheit (truncated to int): " +temp);
  //Casting to int drops the decimal part — precision is lost
    }
}
```

Output:

```
Enter temperature in Celsius :
35
Fahrenheit (double): 95.0
Fahrenheit (truncated to int): 95
```

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.

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

```
}
public class weekday {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String input = sc.nextLine().toUpperCase();
      DaysOfWeek day = DaysOfWeek.valueOf(input);
      System.out.println("Position: " + day.ordinal());
      switch (day) {
         case SATURDAY:
         case SUNDAY:
            System.out.println("It's a weekend");
            break;
         default:
            System.out.println("It's a weekday");
      }
   }
}
```

Output:

```
Enter Day Name: wednesday
Day ordinal position: 2
It's 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.

```
enum Direction {
   NORTH, SOUTH, EAST, WEST
}
```

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

      Scanner sc = new Scanner(System.in);
      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");
      }
   }
}
```

Output:

north

Move north

- ---

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.

```
enum Shape {
   CIRCLE {
      double area(double... p) { return Math.PI * p[0] * p[0]; }
   },
   SQUARE {
      double area(double... p) { return p[0] * p[0]; }
   },
   RECTANGLE {
      double area(double... p) { return p[0] * p[1]; }
   },
   TRIANGLE {
      double area(double... p) { return 0.5 * p[0] * p[1]; }
   };
   abstract double area(double... p);
}

public class ShapeTest {
   public static void main(String[] args) {
      for (Shape s : Shape.values()) {
         double result = 0;
         switch (s) {
            case CIRCLE: result = s.area(5); break;
            case SQUARE: result = s.area(4); break;
            case RECTANGLE: result = s.area(4, 6); break;
            case TRIANGLE: result = s.area(4, 5); break;
         }
```

```
            System.out.println(s + " area: " + result);
        }
    }
}
```

Output:

```
CIRCLE area: 78.53981633974483
SQUARE area: 16.0
RECTANGLE area: 24.0
TRIANGLE area: 10.0
```

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;

public class CardGameDemo {

    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;

    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 static void main(String[] args) {
        Deck deck = new Deck();
        System.out.println("Deck created with 52 cards.\n");
        System.out.println("Shuffling deck...\n");
        deck.shuffle();
        deck.printDeck();
    }
}
```

Output:

```
Deck created with 52 cards.

Shuffling deck...

SEVEN of SPADES
FIVE of DIAMONDS
QUEEN of SPADES
TEN of HEARTS
EIGHT of CLUBS
THREE of DIAMONDS
EIGHT of DIAMONDS
SIX of SPADES
SEVEN of HEARTS
SIX of CLUBS
EIGHT of SPADES
KING of DIAMONDS
ACE of DIAMONDS
JACK of DIAMONDS
QUEEN of HEARTS
TWO of SPADES
FIVE of CLUBS
TWO of CLUBS
NINE of SPADES
EIGHT of HEARTS
THREE of HEARTS
TEN of CLUBS
SEVEN of CLUBS
ACE of HEARTS
NINE of DIAMONDS
JACK of HEARTS
QUEEN of CLUBS
SIX of DIAMONDS
KING of CLUBS
TEN of SPADES
JACK of SPADES
JACK of CLUBS
KING of HEARTS
FOUR of CLUBS
FOUR of SPADES
NINE of CLUBS
ACE of SPADES
SEVEN of DIAMONDS
THREE of SPADES
NINE of HEARTS
FOUR of DIAMONDS
THREE of CLUBS
FIVE of HEARTS
TWO of HEARTS
KING of SPADES
QUEEN of DIAMONDS
```

TWO of DIAMONDS	SIX of HEARTS	ACE of CLUBS
FOUR of HEARTS	TEN of DIAMONDS	FIVE 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(2), HIGH(3), CRITICAL(4);
   private final int severity;
   PriorityLevel(int severity) {
      this.severity = severity;
   }
   public boolean isUrgent() {
      return severity >= 3;
   }

   public String getDescription() {
      return name() + " (Severity: " + severity + ")";
   }
}
public class PriorityTest {
   public static void main(String[] args) {
      for (PriorityLevel p : PriorityLevel.values()) {
         System.out.println(p.getDescription() + " | 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.

```
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);
            current = current.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_typeConversion;


enum Difficulty {
    EASY, MEDIUM, HARD
}


class Game {
    private Difficulty difficulty;
    private int bullets;


    Game(Difficulty difficulty) {
```

```
        this.difficulty = difficulty;
        switch (difficulty) {
            case EASY:
                bullets = 3000;
                break;
            case MEDIUM:
                bullets = 2000;
                break;
            case HARD:
                bullets = 1000;
                break;
        }
    }

    public void start() {
        System.out.println("Difficulty: " + difficulty);
        System.out.println("Bullets: " + bullets);
    }
}

public class GameTest {
    public static void main(String[] args) {
        Game g1 = new Game(Difficulty.EASY);
        g1.start();
        Game g2 = new Game(Difficulty.MEDIUM);
        g2.start();
        Game g3 = new Game(Difficulty.HARD);
        g3.start();
    }
}
```

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.

Version 1(Using switch)

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


   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: return a / b;
         default: throw new AssertionError("Unknown op");
      }
   }
}


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

```
        System.out.println("Switch Version:");
        for (OperationSwitch op : OperationSwitch.values()) {
            System.out.println(op + ": " + op.eval(10, 5));
        }
    }
}
```

Output

```
Switch Version:
PLUS: 15.0
MINUS: 5.0
TIMES: 50.0
DIVIDE: 2.0
```

Version 2(constant-specific method overrides)

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

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

public class CalculatorOverrideTest {
    public static void main(String[] args) {
        System.out.println("\nOverride Version:");
        for (OperationOverride op : OperationOverride.values()) {
            System.out.println(op + ": " + op.eval(10, 5));
        }
    }
}
```

Output

Override Version:

PLUS: 15.0

MINUS: 5.0

TIMES: 50.0

DIVIDE: 2.0

- ---

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_typeConversion;

enum KnowledgeLevel {
    BEGINNER, ADVANCED, PROFESSIONAL, MASTER;
    static KnowledgeLevel fromScore(int score) {
        if (score >= 0 && score <= 3) return BEGINNER;
        if (score >= 4 && score <= 6) return ADVANCED;
        if (score >= 7 && score <= 9) return PROFESSIONAL;
        if (score == 10) return MASTER;
        return null;
    }
}
public class KnowledgeTest {
    public static void main(String[] args) {
        int[] scores = {0, 3, 4, 6, 7, 9, 10};
        for (int s : scores) {
            System.out.println(s + " ->" + KnowledgeLevel.fromScore(s));
        }
    }
```

```
}
```

Output:

```
0 -> BEGINNER

3 -> BEGINNER

4 -> ADVANCED

6 -> ADVANCED

7 -> PROFESSIONAL

9 -> PROFESSIONAL

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.".

```
public class ExceptionDemo {
   public static void main(String[] args) {
      try {
         int a = 10 / 0;
      } catch (ArithmeticException e) {
         System.out.println("Division by zero is not allowed!");
      } finally {
         System.out.println("Operation completed.");
      }
      try {
         int[] arr = {1, 2, 3};
         System.out.println(arr[5]);
```

```
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index is out of range!");
            } finally {
                System.out.println("Operation completed.");
            }
        }
    }
```

Output:

```
Division by zero is not allowed!
Operation completed.
Array index is out of range!
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.

```
package Day_6_typeConversion;


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, 5, 8, 11};
        for (int n : numbers) {
            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
Odd number: 11
```

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.

```
import java.io.*;
public class FileReadDemo {
   public static void readFile(String filename) throws FileNotFoundException, IOException {
      BufferedReader br = new BufferedReader(new FileReader(filename));
      System.out.println("First line: " + br.readLine());
      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:

```
First line: null
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".

```
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);
         System.out.println("Result: " + (100 / num));
         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:

```
Invalid number format
Execution completed
```