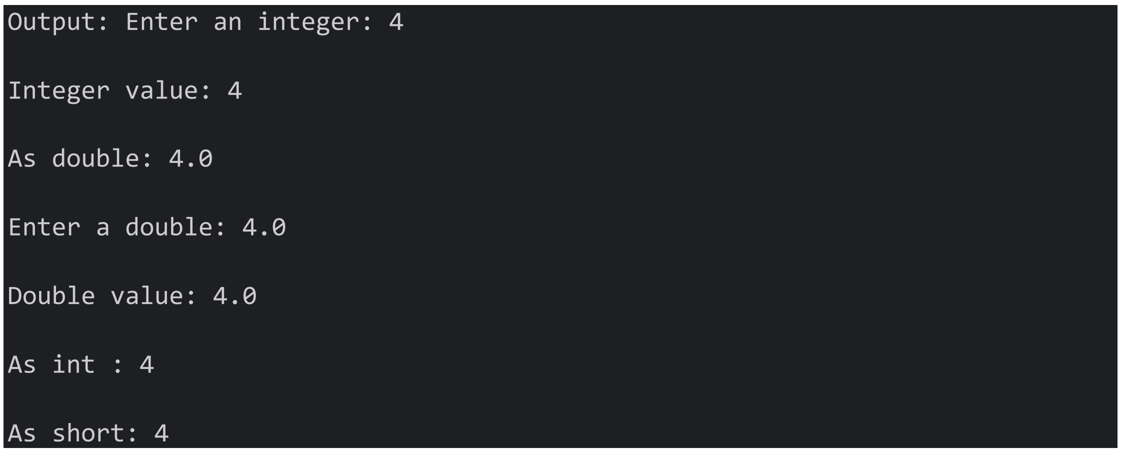
1. Write a program to:
   - Read an int value from user input.
   - Assign it to a double (implicit widening) and print both.
   - Read a double, explicitly cast it to int, then to short, and print results—demonstrate truncation or overflow

Code:

```
import java.util.Scanner;

public class Typecast {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.print("Enter an integer: ");

    int intVal = sc.nextInt();

    double doubleVal = intVal;

    System.out.println("Integer value: " + intVal);

    System.out.println("As double: " + doubleVal);

    System.out.print("\nEnter a double: ");

    double doubleInput = sc.nextDouble();

    int intFromDouble = (int) doubleInput;

    short shortFromInt = (short) intFromDouble;
```

```
        System.out.println("Double value: " + doubleInput);

        System.out.println("As int : " + intFromDouble);

        System.out.println("As short: " + shortFromInt);

        sc.close();

    }

}
```



```
Output: Enter an integer: 4

Integer value: 4

As double: 4.0

Enter a double: 4.0

Double value: 4.0

As int : 4

As short: 4
```

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

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?
```

Code:

```
public class compoundassignment {

    public static void main(String[] args) {

        try {

            int num = 123;

            String str = String.valueOf(num);

            System.out.println("String value: " + str);



            int parsed = Integer.parseInt(str);

            System.out.println("Parsed integer: " + parsed);

        } catch (NumberFormatException e) {

            System.out.println("Invalid number format!");

        }

        int x = 5;

        x += 4.5;

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

    }

}
```

```
Output: String value: 123

Parsed integer: 123

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

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

   X= X + 4.5; → Needs explicit cast because it's a direct assignment from double to int.

   X+= 4.5; → Works without cast because compound assignment automatically narrows.

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();
   ```

Code:

```
class Animal {

   void makeSound() {

       System.out.println("Some animal sound");
```

```
    }

}

class Dog extends Animal {

    void makeSound() {

        System.out.println("Woof!");

    }

    void fetch() {

        System.out.println("Dog is fetching the ball");

    }

}

public class objectcasting {

    public static void main(String[] args) {

        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 6ahrenheit = 6ahrenh * 9/5 + 32;

3. Then cast that 6ahrenheit to int for display.
4. Print both the precise (double) and truncated (int) values, and comment on precision loss.

Code:

```
import java.util.Scanner;

public class temperatureconv {

   public static void main(String[] args) {

      Scanner sc = new Scanner(System.in);

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

      double ahrenh = sc.nextDouble();

      double ahrenheit = ahrenh * 9 / 5 + 32;

      int fahrenheitInt = (int) ahrenheit;

      System.out.println(“Fahrenheit (precise): “ + ahrenheit);

      System.out.println(“Fahrenheit (truncated): “ + fahrenheitInt);

      sc.close();

   }

}
```

```
Output: Enter temperature in Celsius: 45

Fahrenheit (precise): 113.0

Fahrenheit (truncated): 113
```

Enum

1: Days of the Week

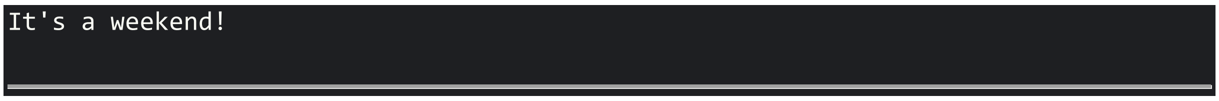
Define an enum DaysOfWeek with seven constants. Then in main(), prompt the user to input a day name and:

- Print its position via ordinal().
- Confirm if it's a weekend day using a switch or if-statement.

Code :

```
import java.util.Scanner;

enum DaysOfWeek {

   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY

}

public class Enumexamp {

   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("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.");

            }



        } catch (IllegalArgumentException e) {

            System.out.println("Invalid day name.");

        }

        sc.close();

    }

}
```

```
Output: Enter a day of the week: sunday

Position: 6
```


```
It's 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.

```
import java.util.Scanner;

public class directiontest {

    enum Direction {

        NORTH, SOUTH, EAST, WEST

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter a direction : ");

        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();
    }
}
```

```
Output: Enter a 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.

Code :

```
import java.util.*;

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 radius = 5;

        double side = 4;

        double length = 6, width = 3;

        double base = 8, height = 4;

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

            switch (shape) {

                case CIRCLE:

                    System.out.println("CIRCLE area: " + shape.area(radius));

                    break;

                case SQUARE:

                    System.out.println("SQUARE area: " + shape.area(side));

                    break;

                case RECTANGLE:

                    System.out.println("RECTANGLE area: " + shape.area(length, width));

                    break;

                case TRIANGLE:
```

```
                System.out.println("TRIANGLE area: " + shape.area(base, height));

                break;

            }

        }

    }

}
```

```
•   Output: CIRCLE area: 78.53981633974483
•   SQUARE area: 16.0
•   RECTANGLE area: 18.0
•   TRIANGLE area: 16.0
```

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.

Code :

```
import java.util.*;

enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}
enum Rank {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    EIGHT, NINE, TEN, JACK, QUEEN, KING
}
class Card {
    Suit suit;
    Rank rank;
    Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public String toString() {
        return rank + " of " + suit;
    }
}
class Deck {
    List<Card> cards = new ArrayList<>();
    Deck() {
        for (Suit s : Suit.values()) {
            for (Rank r : Rank.values()) {
                cards.add(new Card(s, r));
            }
        }
    }
    void shuffle() {
        Collections.shuffle(cards);
    }

    void printDeck() {
        for (Card c : cards) {
            System.out.println(c);
        }
    }
}
public class Cardgame {
    public static void main(String[] args) {
        Deck deck = new Deck();

        deck.shuffle();

        deck.printDeck();

    }

}
```

```
Output: EIGHT of DIAMONDS

FIVE of HEARTS

SIX of DIAMONDS

THREE of CLUBS

SEVEN of HEARTS

THREE of SPADES

ACE of DIAMONDS

TEN of HEARTS

SIX of SPADES

FIVE of DIAMONDS

JACK of CLUBS

SEVEN of SPADES

QUEEN of DIAMONDS

FOUR of HEARTS
```

5: Priority Levels with Extra Data

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

- A numeric severity code.

Code :

A boolean isUrgent() if severity ≥ some threshold.

Print descriptions and check urgency.

```
package Assignment6;

enum PriorityLevel {

   LOW(1),

   MEDIUM(2),

   HIGH(3),

   CRITICAL(4);

   private int severityCode;

   PriorityLevel(int code) {

      this.severityCode = code;

   }

   public int getSeverityCode() {

      return severityCode;

   }



   public boolean isUrgent() {

      return severityCode >= 3;

   }

}
```

```
public class prioritymain {

    public static void main(String[] args) {

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

            System.out.println(

                level + " | Severity Code: " + level.getSeverityCode() +

                " | Urgent? " + level.isUrgent()

            );

        }

    }

}
```

```
Output: LOW | Severity Code: 1 | Urgent? false

MEDIUM | Severity Code: 2 | Urgent? false

HIGH | Severity Code: 3 | Urgent? true

CRITICAL | Severity Code: 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.

Code :

```
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 trafficlightstatemachine {

    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.

Code :

```
enum Difficulty {
    EASY, MEDIUM, HARD
}
class Game {
    public Game(Difficulty diff) {
        int bullets;
        switch (diff) {
            case EASY:
                bullets = 3000;
                break;
            case MEDIUM:
                bullets = 2000;
                break;
            case HARD:
                bullets = 1000;
                break;
            default:
                bullets = 0;
        }
        System.out.println("Difficulty: " + diff);
        System.out.println("Bullets: " + bullets);

    }

}

public class gamesetup {

    public static void main(String[] args) {

        new Game(Difficulty.EASY);

        new Game(Difficulty.MEDIUM);

        new Game(Difficulty.HARD);

    }

}
```

```
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.

Code :

```
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 operation: " + this);

      }

   }

}

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

        double x = 10, y = 5;

        for (OperationSwitch op : OperationSwitch.values()) {

            System.out.println(op + ": " + op.eval(x, y));

        }

    }

}
```

```
Output: PLUS: 15.0

MINUS: 5.0

TIMES: 50.0

DIVIDE: 2.0
```

- Another using constant-specific method overrides for eval.
  Compare both designs.

Code :

```
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 mainswitch2 {
    public static void main(String[] args) {
        double x = 10, y = 5;
        for (OperationOverride op : OperationOverride.values()) {

            System.out.println(op + ": " + op.eval(x, y));

        }

    }

}
```

```
Output: 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.

Code :

```
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 out of range (0-10): " + score);
        }
    }
}
public class scorerange {
    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);
```

```
      }
   }
}
```

```
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.");
}
```

Code:

```
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[] numbers = {1, 2, 3};

            System.out.println(numbers[5]);

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("You tried to access an invalid array index!");

        } finally {

            System.out.println("Operation completed.");

        }

    }

}
```

```
Output: Division by zero is not allowed!

Operation completed.

----------------------------

You tried to access an invalid array index!

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); }

}

Code:

```
class OddNumberException extends Exception {

   public OddNumberException(String message) {

      super(message);

   }

}

public class oldchecker {

   public static void checkOdd(int n) throws OddNumberException {
      if (n % 2 != 0) {

          throw new OddNumberException("Odd number: " + n);

      } else {

          System.out.println(n + " is even. No exception thrown.");

      }

   }

   public static void main(String[] args) {

      int[] numbers = {2, 5, 8, 11};

      for (int num : numbers) {

          try {

              checkOdd(num);

          } catch (OddNumberException e) {

              System.out.println("Caught Exception ⟶ " + e.getMessage());

          }

      }

   }

}
```

```
Output: 2 is even. No exception thrown.

Caught Exception → Odd number: 5

8 is even. No exception thrown.

Caught Exception → 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.

Code :

```
import java.io.*;

public class filereaddemo {

   public static void readFile(String filename) throws FileNotFoundException, IOException {

      FileReader fr = new FileReader(filename);

      BufferedReader br = new BufferedReader(fr);

      String firstLine = br.readLine();

      System.out.println("First line: " + firstLine);
```

```
        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: test.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".

Code :

```
import java.io.*;

public class multiexceptiondemo {

   public static void main(String[] args) {

      String filename = "employee.txt";



      try {
```

```
    FileReader fr = new FileReader(filename);

    BufferedReader br = new BufferedReader(fr);

    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: Invalid number format

Execution completed
```

-