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.**

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

Program:

```
public class TypeCasting_Implicit
{
        public static void main(String[] args)
        {
        int a= 25;
        double b=a;
        System.out.println("Int value: " +a);
        System.out.println("Implicitly casted to double: " + b);
        double c= 12345.678;
        int d= (int) c;
        short s = (short) d;
        System.out.println("Original double: " + c);
        System.out.println("After casting to int: " + d);
        System.out.println("After casting to short: " + s);
        }
}
```

Output:

```
Int value: 25
Implicitly casted to double: 25.0
Original double: 12345.678
After casting to int: 12345
After casting to short: 12345
```

2.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).**

Program:

```
public class CompoundAssignment
{
        public static void main(String[] args)
        {
                int x = 5;
                // 1. Normal addition assignment
                // x = x + 4.5; //  Compile Error: x + 4.5 becomes double, cannot assign to
        without //explicit cast
                // 2. Compound assignment
                 x += 4.5; // Implicit narrowing from double to int
                System.out.println("Value of x after compound assignment: " + x);
           }
}
```

Output:

```
Value of x after compound assignment: 9
```

3.**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();
```

Program:

```
class Animal
{
    public void makeSound()
    {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal
{
    @Override
    public void makeSound()
    {
        System.out.println("Woof!");
    }
public void fetch()
{
   System.out.println("Dog is fetching...");
 }
}
public class Object_Casting_Inheritance
{
    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 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.**

Program:

```
public class MiniProject_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 Fahrenheit1 = (int) fahrenheit;
        System.out.println("Fahrenheit : " + fahrenheit);
        System.out.println("Fahrenheit (Changed to int): " Fahrenheit1);
        sc.close();
         }
}
```

OutPut:

```
Enter temperature in Celsius: 25
```

Fahrenheit: 77.0

Fahrenheit (Changed to int): 77

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.**

Program:

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

public class Enum_DaysOfWeek
{
    public static void main(String[] args)
    {
        String input = "SATURDAY";
        if (input.equals("SATURDAY") || input.equals("SUNDAY"))
        {
                System.out.println("It's a weekend.");
         }
        else if (input.equals("MONDAY") || input.equals("TUESDAY") ||
                    input.equals("WEDNESDAY") || input.equals("THURSDAY") ||
                    input.equals("FRIDAY"))
        {
                System.out.println("It's a weekday.");
         }
        else
        {
                System.out.println("Invalid day entered.");
         }

    }

}
```

OutPut:

```
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.

Program:

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

public class CompassDemo

{

    public static void main(String[] args)

    {

        String input = "NORTH";

        try

        {

        Direction dir = Direction.valueOf(input.toUpperCase());


        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: " + input);
        }
      }
    }
```

OutPut:

```
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.**

Program:

```
enum Shape {CIRCLE, SQUARE, RECTANGLE, TRIANGLE;
double area(double a)
{
     if (this == CIRCLE)
    {
            return 3.14 * a * a;
     }
    else if (this == SQUARE)
    {
        return a * a;
     }
     return 0;
  }

  double area(double a, double b)
{
     if (this == RECTANGLE)
    {
            return a * b;
     }
```

```
        else if (this == TRIANGLE)
        {
            return 0.5 * a * b;
         }
         return 0;
       }
    }
    public class Enum_Shape
    {
        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(6, 3));
         System.out.println("Triangle area: " + Shape.TRIANGLE.area(8, 2.5));

        }

    }
```

OutPut:

```
Circle area: 78.5

Square area: 16.0

Rectangle area: 18.0

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.**

Program:

```
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;
        }
}
public class Cards
{
      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:

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

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

Program:

```
enum PriorityLevel {LOW(1), MEDIUM(2), HIGH(3), CRITICAL(4)};
int severity;
PriorityLevel(int s)
{
     severity = s;
  }
}
public class Enum_Priority
{
    public static void main(String[] args)
    {
            PriorityLevel[] levels = PriorityLevel.values();
            for (int i = 0; i < levels.length; i++)
            {
                    PriorityLevel p = levels[i];
                    System.out.print(p + " - Severity: " + p.severity + " - ");
                    if (p.severity >= 3)
                    {
                            System.out.println("Urgent");
                    }
                    else
                    {
                            System.out.println("Not Urgent");
                    }
            }

    }

}
```

Output:

```
LOW - Severity: 1 - Not Urgent

MEDIUM - Severity: 2 - Not Urgent

HIGH - Severity: 3 - Urgent

CRITICAL - Severity: 4 - Urgent
```

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.

Program:

```
enum TrafficLight { RED, GREEN, YELLOW }

public class Enum_Traffic {

public static void main(String[] args) {

TrafficLight current = TrafficLight.RED;

                    for (int i = 1; i <= 6; i++) {

                        System.out.println("Current light: " + current);

                        if (current == TrafficLight.RED) {

                            current = TrafficLight.GREEN;

                        } else if (current == TrafficLight.GREEN) {

                            current = TrafficLight.YELLOW;

                        } else {

                            current = TrafficLight.RED;

                        }

                    }

            }

}
```

Output:

```
Current light: RED

Current light: GREEN

Current light: YELLOW

Current light: RED

Current light: GREEN

Current light: 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.

Program:

```
enum Difficulty { EASY, MEDIUM, HARD }
public class Game
{
        public static void main(String[] args)
        {
              Difficulty diff = Difficulty.MEDIUM;
              switch (diff)
                 {
                                case EASY:
                        System.out.println("3000 bullets");
                        break;
                                case MEDIUM:
                                System.out.println("2000 bullets");
                                break;
                                case HARD:
                                System.out.println("1000 bullets");
                                break;
                        }
                }
        }
```

Output:

```
2000 bullets
```

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 Operation1  {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;
            }
        return 0;
    }
}
enum Operation2
{
    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 CalculatorEasy {
    public static void main(String[] args) {
        // Test switch version
        System.out.println("Switch PLUS: " + Operation1.PLUS.eval(5, 3));
        System.out.println("Switch DIVIDE: " + Operation1.DIVIDE.eval(10, 2));
        // Test override version

        System.out.println("Override MINUS: " + Operation2.MINUS.eval(5, 3));

        System.out.println("Override TIMES: " + Operation2.TIMES.eval(4, 2));

    }

}
```

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.

Program:

```
num KnowledgeLevel {BEGINNER, ADVANCED, PROFESSIONAL, MASTER};
static KnowledgeLevel fromScore(int score)
{
         if (score <= 3) return BEGINNER;
         else if (score <= 6) return ADVANCED;
         else if (score <= 9) return PROFESSIONAL;
         else return MASTER;
       }
    }

    public class Enum_Knowledge
    {
        public static void main(String[] args)
        {
                System.out.println(KnowledgeLevel.fromScore(2));
                System.out.println(KnowledgeLevel.fromScore(5));
                System.out.println(KnowledgeLevel.fromScore(8));
                System.out.println(KnowledgeLevel.fromScore(10));

        }

    }
```

OutPut:

```
BEGINNER

ADVANCED
```

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.");
}
```

Program:

```
public class Exception_DivisionArray
{
        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.");

                }

                try

                {

                                int[] arr = {1, 2, 3};

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

                }

                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); }
}
```

Program:

```
class OddNumberException extends Exception
{
        public OddNumberException(String message)
        {
                super(message);
        }
}
public class Exception_ThrowAndHandle
{
        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.");
        }
    }
    public static void main(String[] args)
    {
        int[] num = {3, 4, 7, 8};
        for (int n : num)
    {
            try
        {
            checkOdd(n);
            }
        catch (OddNumberException e)
        {
            System.out.println(e.getMessage());
            }
    }
 }
}
```

Output:

```
Odd number: 3
4 is even.
Odd number: 7
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.**

Program:

```
public class Exception_FileHandling

{

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

        {

                FileReader fr = new FileReader(filename);

                int ch;

                System.out.print("File content: ");

                while ((ch = fr.read()) != -1) {

                 System.out.print((char) ch);

                break;

             }

             fr.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("\nCleanup done.");
                }
          }
}
```

OutPut:

```
File content: H
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".**

Program:

```
public class Exception_MultiException
{
    public static void main(String[] args)
    {
        String filename = "data.txt";
         try
        {
                FileReader fr = new FileReader(filename);
                int ch;
                String numberStr = "";
                while ((ch = fr.read()) != -1 && ch != '\n') {
                numberStr += (char) ch;
            }

            int number = Integer.parseInt(numberStr.trim());
            int result = 100 / number;
            System.out.println("Result: " + result);
            fr.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.");
        }
    }
}
```