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.

Program:

import java.util.Scanner;

public class TypeCastingDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Read int and assign to double (implicit widening)

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

int intValue = sc.nextInt();

double doubleValue = intValue; // Implicit widening

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

System.out.println("Double value (widened): " + doubleValue);

// Read double and cast to int, then short

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

double dValue = sc.nextDouble();

int intFromDouble = (int) dValue; // Explicit narrowing

short shortFromInt = (short) intFromDouble; // Further narrowing

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

System.out.println("Int from double: " + intFromDouble);

System.out.println("Short from int: " + shortFromInt);

sc.close();

}

}

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

Program:

public class IntStringConversion {

public static void main(String[] args) {

try {

int num = 123;

// Convert int to String

String strValue = String.valueOf(num);

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

// Convert String back to int

int parsedNum = Integer.parseInt(strValue);

System.out.println("Parsed int value: " + parsedNum);

// Example of handling NumberFormatException

String invalidNumber = "123abc";

int invalidParsed = Integer.parseInt(invalidNumber); // Will throw exception

System.out.println("Parsed invalid number: " + invalidParsed);

} catch (NumberFormatException e) {

System.out.println("Error: Invalid number format - " + e.getMessage());

}

}

}

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?

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

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();

Mini‑Project – Temperature Converter

1. Prompt user for a temperature in Celsius (double).
2. Convert it to Fahrenheit:

double fahrenheit = celsius \* 9/5 + 32;

1. Then cast that fahrenheit to int for display.
2. Print both the precise (double) and truncated (int) values, and comment on precision loss.

Program:

import java.util.Scanner;

public class TemperatureConverter {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// 1. Prompt user for temperature in Celsius

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

double celsius = sc.nextDouble();

// 2. Convert to Fahrenheit

double fahrenheit = celsius \* 9 / 5 + 32;

// 3. Cast to int (truncation)

int fahrenheitInt = (int) fahrenheit;

// 4. Print both values

System.out.println("Temperature in Fahrenheit (double): " + fahrenheit);

System.out.println("Temperature in Fahrenheit (int, truncated): " + fahrenheitInt);

// Comment on precision loss

System.out.println("Note: Casting to int removes decimal precision, keeping only the whole number.");

sc.close();

}

}

Output:

Enter temperature in Celsius: 36.6

Temperature in Fahrenheit (double): 97.88

Temperature in Fahrenheit (int, truncated): 97

Note: Casting to int removes decimal precision, keeping only the whole number.

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:

import java.util.Scanner;

enum Direction {

NORTH, SOUTH, EAST, WEST

}

public class DirectionDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter direction (NORTH, SOUTH, EAST, WEST): ");

String input = sc.nextLine().toUpperCase();

try {

// Read a Direction from string

Direction dir = Direction.valueOf(input);

// Use switch to print movement

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 direction (NORTH, SOUTH, EAST, WEST): east

Move east

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.

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 {

@Override

double area(double... params) {

if (params.length != 1) throw new IllegalArgumentException("Circle needs radius");

double radius = params[0];

return Math.PI \* radius \* radius;

}

},

SQUARE {

@Override

double area(double... params) {

if (params.length != 1) throw new IllegalArgumentException("Square needs side length");

double side = params[0];

return side \* side;

}

},

RECTANGLE {

@Override

double area(double... params) {

if (params.length != 2) throw new IllegalArgumentException("Rectangle needs width and height");

double width = params[0];

double height = params[1];

return width \* height;

}

},

TRIANGLE {

@Override

double area(double... params) {

if (params.length != 2) throw new IllegalArgumentException("Triangle needs base and height");

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) {

// Loop over all shapes with sample inputs

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

double area = 0;

switch (shape) {

case CIRCLE:

area = shape.area(5); // radius

break;

case SQUARE:

area = shape.area(4); // side

break;

case RECTANGLE:

area = shape.area(5, 3); // width, height

break;

case TRIANGLE:

area = shape.area(6, 4); // base, height

break;

}

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

}

}

}

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:

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

// Enum for Suit

enum Suit {

CLUBS, DIAMONDS, HEARTS, SPADES

}

// Enum for Rank

enum Rank {

ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,

EIGHT, NINE, TEN, JACK, QUEEN, KING

}

// Card class

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;

}

}

// Deck class

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);

}

}

}

// Main class

public class CardDeckDemo {

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();

}

}

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);

private final int severityCode;

PriorityLevel(int severityCode) {

this.severityCode = severityCode;

}

public int getSeverityCode() {

return severityCode;

}

public boolean isUrgent() {

return severityCode >= 3; // Threshold: 3 or more is urgent

}

public String getDescription() {

return name() + " (Severity Code: " + severityCode + ", Urgent: " + isUrgent() + ")";

}

}

public class PriorityLevelDemo {

public static void main(String[] args) {

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

System.out.println(level.getDescription());

}

}

}

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:

// State interface

interface State {

State next();

}

// Enum implementing the State interface

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 class TrafficLightDemo {

public static void main(String[] args) {

State current = TrafficLight.RED; // Start from RED

// Simulate 6 transitions

for (int i = 0; i < 6; i++) {

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

current = current.next();

}

}

}

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

}

class Game {

private Difficulty difficulty;

private int bullets;

public Game(Difficulty difficulty) {

this.difficulty = difficulty;

setupGame();

}

private void setupGame() {

switch (difficulty) {

case EASY:

bullets = 3000;

break;

case MEDIUM:

bullets = 2000;

break;

case HARD:

bullets = 1000;

break;

}

System.out.println("Game started with difficulty: " + difficulty + ", Bullets: " + bullets);

}

}

public class GameSetupDemo {

public static void main(String[] args) {

new Game(Difficulty.EASY);

new Game(Difficulty.MEDIUM);

new Game(Difficulty.HARD);

}

}

Output:

Game started with difficulty: EASY, Bullets: 3000

Game started with difficulty: MEDIUM, Bullets: 2000

Game started with 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.

Program:

enum OperationSwitch {

PLUS, MINUS, TIMES, DIVIDE;

public 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:

if (b == 0) throw new ArithmeticException("Cannot divide by zero");

return a / b;

default: throw new AssertionError("Unknown operation");

}

}

}

public class CalculatorSwitchDemo {

public static void main(String[] args) {

double a = 10, b = 5;

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

System.out.println(op + ": " + op.eval(a, b));

}

}

}

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.

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.");

}

2: Throw and Handle Custom Exception

Create a class OddChecker:

1. Implement a static method:

public static void checkOdd(int n) throws OddNumberException { /\* ... \*/ }

1. If n is odd, throw a custom checked exception OddNumberException with message "Odd number: " + n.
2. 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); }

}

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:

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class FileReadDemo {

// Method to read the first line from a file

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

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {

String firstLine = br.readLine();

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

}

}

public static void main(String[] args) {

String filename = "test.txt"; // Change file name if needed

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.");

}

}

}

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:
  + FileNotFoundException
  + IOException
  + NumberFormatException
  + 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:

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class MultiExceptionDemo {

public static void main(String[] args) {

String filename = "test.txt"; // Change this to your file name

try {

// Open the file

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

// Read first line

String firstLine = br.readLine();

// Parse first line as integer

int number = Integer.parseInt(firstLine);

// Divide 100 by that integer

int result = 100 / number;

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

System.out.println("100 / " + number + " = " + 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");

}

}

}