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 TypeCastingExample {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Step 1: 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("Implicitly widened to double: " + doubleValue);

// Step 2: Read double and cast to int, then to short

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

double inputDouble = sc.nextDouble();

int castedInt = (int) inputDouble; // explicit narrowing

short castedShort = (short) castedInt; // explicit narrowing

System.out.println("Original double value: " + inputDouble);

System.out.println("After casting to int: " + castedInt);

System.out.println("After casting to short: " + castedShort);

sc.close();

}

}

Output:

Enter an integer: 42

Integer value: 42

Implicitly widened to double: 42.0

Enter a double value: 123.456

Original double value: 123.456

After casting to int: 123

After casting to short: 123

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

Code:

import java.util.Scanner;

public class IntStringConversion {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

try {

// Read an int value from the user

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

int num = sc.nextInt();

// Convert int to String

String strValue = String.valueOf(num);

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

// Convert String back to int

int parsedInt = Integer.parseInt(strValue);

System.out.println("String converted back to int: " + parsedInt);

} catch (NumberFormatException e) {

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

}

}

}

Output:

Enter an integer: 123

Integer as String: 123

String converted back to int: 123

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).

**Code:**

public class CompoundAssignmentDemo {

public static void main(String[] args) {

int x = 5;

// x = x + 4.5;

// Reason: x is int, but x + 4.5 produces a double.

// Java will not automatically convert double to int here (explicit cast required).

x += 4.5;

// Explanation:

// x += 4.5 is equivalent to x = (int)(x + 4.5);

// Implicit narrowing conversion happens automatically for compound assignments.

System.out.println("Value of x after compound assignment: " + x);

// x = (int)(5 + 4.5) → (int)9.5 → 9

}

}

Output:

Value of x after compound assignment: 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();

**code:**

class Animal {

void makeSound() {

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

}

}

// Subclass

class Dog extends Animal {

@Override

void makeSound() {

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

}

void fetch() {

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

}

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

Animal a = d;

a.makeSound();

// a.fetch(); // ❌ Compile-time error: Animal type doesn't have fetch()

// Downcasting to access Dog-specific method

Dog downcastedDog = (Dog) a;

downcastedDog.fetch();

}

}

Output:

Woof!

Dog is fetching the ball!

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.

Code:

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 truncatedFahrenheit = (int) fahrenheit; // Casting to int (precision loss)

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

System.out.println("Temperature in Fahrenheit (truncated): " + truncatedFahrenheit);

// Comment: Casting double to int removes fractional part (precision loss).

}

}

Output:

Enter temperature in Celsius: 37.5

Temperature in Fahrenheit (precise): 99.5

Temperature in Fahrenheit (truncated): 99

**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 DaysExample {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a day name (e.g., MONDAY): ");

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

try {

// Convert string to enum constant

DaysOfWeek day = DaysOfWeek.valueOf(input);

// Print position (ordinal starts from 0)

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

// Check weekend or weekday

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 entered. Please check spelling.");

}

sc.close();

}

}

Output:

Enter a day name (e.g., MONDAY): Saturday

Position (ordinal): 5

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.

Code:

import java.util.Scanner;

enum Direction {

NORTH, SOUTH, EAST, WEST

}

public class CompassDirections {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

String input = sc.nextLine().trim().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 (NORTH, SOUTH, EAST, WEST): 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:

* enum Shape {
* CIRCLE {
* @Override
* double area(double... params) {
* if (params.length != 1) throw new IllegalArgumentException("Circle requires radius");
* double radius = params[0];
* return Math.PI \* radius \* radius;
* }
* },
* SQUARE {
* @Override
* double area(double... params) {
* if (params.length != 1) throw new IllegalArgumentException("Square requires side length");
* double side = params[0];
* return side \* side;
* }
* },
* RECTANGLE {
* @Override
* double area(double... params) {
* if (params.length != 2) throw new IllegalArgumentException("Rectangle requires length and width");
* return params[0] \* params[1];
* }
* },
* TRIANGLE {
* @Override
* double area(double... params) {
* if (params.length != 2) throw new IllegalArgumentException("Triangle requires base and height");
* return 0.5 \* params[0] \* params[1];
* }
* };
* abstract double area(double... params);
* }
* public class ShapeAreaTest {
* public static void main(String[] args) {
* // Sample inputs for each shape
* Object[][] sampleInputs = {
* {Shape.CIRCLE, new double[]{5}}, // radius
* {Shape.SQUARE, new double[]{4}}, // side
* {Shape.RECTANGLE, new double[]{6, 3}}, // length, width
* {Shape.TRIANGLE, new double[]{4, 7}} // base, height
* };
* // Loop through and print areas
* for (Object[] data : sampleInputs) {
* Shape shape = (Shape) data[0];
* double[] params = (double[]) data[1];
* System.out.println(shape + " area = " + shape.area(params));
* }
* }
* }

Output:

CIRCLE area = 78.53981633974483

SQUARE area = 16.0

RECTANGLE area = 18.0

TRIANGLE area = 14.0

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:

package Practice\_assign;

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;

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);

}

}

}

// Main class

public class CardGame {

public static void main(String[] args) {

Deck deck = new Deck();

System.***out***.println("Deck before shuffling:");

deck.printDeck();

System.***out***.println("\nShuffled deck:");

deck.shuffle();

deck.printDeck();

}

}

Output:

4 of HEARTS

7 of CLUBS

KING of SPADES

2 of DIAMONDS

9 of HEARTS

JACK of SPADES

3 of CLUBS

ACE of HEARTS

8 of DIAMONDS

...

5 of SPADES

10 of CLUBS

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.

Code:

package Practice\_assign;

enum *PriorityLevel* {

***LOW***(1),

***MEDIUM***(3),

***HIGH***(5),

***CRITICAL***(8);

private final int severityCode;

PriorityLevel(int severityCode) {

this.severityCode = severityCode;

}

public int getSeverityCode() {

return severityCode;

}

public boolean isUrgent() {

return severityCode >= 5; // threshold for urgency

}

}

public class PriorityTest {

public static void main(String[] args) {

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

System.***out***.println(

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

" | Urgent: " + level.isUrgent()

);

}

}

}

Output:

LOW | Severity Code: 1 | Urgent: false

MEDIUM | Severity Code: 3 | Urgent: false

HIGH | Severity Code: 5 | Urgent: true

CRITICAL | Severity Code: 8 | 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 {

@Override

public State next() {

return GREEN;

}

},

GREEN {

@Override

public State next() {

return YELLOW;

}

},

YELLOW {

@Override

public State next() {

return RED;

}

};

}

public class TrafficLightTest {

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.

Code:

package Practice\_assign;

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 + " → Bullets: " + bullets);

}

}

public class DifficultyTest {

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

Code:

package Practice\_assign;

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("Division by zero");

return a / b;

default:

throw new AssertionError("Unknown operation: " + this);

}

}

}

public class TestSwitchEnum {

public static void main(String[] args) {

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

System.out.println(op + " → " + op.eval(10, 5));

}

}

}

enum OperationOverride {

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) {

if (b == 0) throw new ArithmeticException("Division by zero");

return a / b;

}

};

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

}

public class TestOverrideEnum {

public static void main(String[] args) {

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

System.out.println(op + " → " + op.eval(10, 5));

}

}

}

Output:

Switch-based:

5.0 + 3.0 = 8.0

5.0 - 3.0 = 2.0

5.0 \* 3.0 = 15.0

5.0 / 3.0 = 1.6666666666666667

Constant-specific:

5.0 + 3.0 = 8.0

5.0 - 3.0 = 2.0

5.0 \* 3.0 = 15.0

5.0 / 3.0 = 1.6666666666666667

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:

package Practice\_assign;

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 must be between 0 and 10");

}

}

}

public class KnowledgeTest {

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);

}

}

}

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:

package Practice\_assign;

public class ExceptionDemo {

public static void main(String[] args) {

// First risky operation: Division by zero

try {

int num1 = 10;

int num2 = 0;

int result = num1 / num2;

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Division by zero is not allowed!");

} finally {

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

}

// Second risky operation: Array index out of bounds

try {

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

System.out.println("Accessing element: " + numbers[5]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Invalid array index!");

} finally {

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

}

}

}

Output:

Division by zero is not allowed!

Operation completed.

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 { /\* ... \*/ }

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); }

}

Code:

package Practice\_assign;

// Custom checked exception

class OddNumberException extends Exception {

public OddNumberException(String message) {

super(message);

}

}

public class OddChecker {

// Static method to check for odd numbers

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[] numbers = {2, 5, 8, 11, 14};

for (int num : numbers) {

try {

checkOdd(num);

} catch (OddNumberException e) {

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

}

}

}

}

Output:

2 is even.

Caught Exception: Odd number: 5

8 is even.

Caught Exception: Odd number: 11

14 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.

Code:

package Practice\_assign;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;

public class FileReadDemo {

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

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

String firstLine = br.readLine();

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

br.close();

}

public static void main(String[] args) {

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

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: Hello World

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".

Code:

package Practice\_assign;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;

public class MultiExceptionDemo {

public static void main(String[] args) {

String filename = "data.txt"; // Change this to your file path

try {

// Open file

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

// Read first line

String line = br.readLine();

br.close();

// Parse to integer

int num = Integer.parseInt(line);

// Division

int result = 100 / num;

System.out.println("Result: " + result);

} 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:

File not found

Execution completed