# 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.

Code:

import java.util.Scanner;

public class TypeConversion {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// implicit widening

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

int intValue = scanner.nextInt();

double doubleValue = intValue;

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

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

//explicit narrowing

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

double inputDouble = scanner.nextDouble();

int castedInt = (int) inputDouble; //truncates decimal

short castedShort = (short) inputDouble;//may truncate or overflow

System.out.println("Double: " + inputDouble);

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

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

String intAsString = String.valueOf(intValue);

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

try {

int parsedInt = Integer.parseInt(intAsString);

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

} catch (NumberFormatException e) {

System.out.println("Error parsing String to int: " + e.getMessage());

}

scanner.close();

}

}

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

Code:

public class IntStringSimple {

public static void main(String[] args) {

int num = 123;

String str = String.valueOf(num);

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

try {

int parsed = Integer.parseInt(str);

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

} catch (NumberFormatException e) {

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

}

try {

int bad = Integer.parseInt("12ab");

System.out.println("Parsed: " + bad);

} catch (NumberFormatException e) {

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

}

}

}

3. 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?

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

Code:

public class CompoundAssignment {

public static void main(String[] args) {

int x = 5;

// Operation 1: x = x + 4.5;

// This will NOT compile because 4.5 is a double to covert double to int we need to do explicit casting.

// Operation 2: x += 4.5;

// This compile and runs. The compound assignment operator (+=) implicitly casts the double result to int, truncating the decimal part.

x += 4.5; // x becomes 9 (5 + 4.5 = 9.5, truncated to 9)

System.out.println("After x += 4.5, x = " + x);

}

}

4. 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 {

public void makeSound() {

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

}

}

class Dog extends Animal {

@Override

public void makeSound() {

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

}

public void fetch() {

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

}

}

public class AnimalDogCasting {

public static void main(String[] args) {

Dog d = new Dog();

Animal a = d; // Upcasting (implicit)

a.makeSound();

// a.fetch() would cause a compile-time error because Animal does not have a fetch method.

}

}

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 scanner = new Scanner(System.in);

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

double celsius = scanner.nextDouble();

// Convert to Fahrenheit

double fahrenheit = celsius \* 9.0 / 5.0 + 32;

int fahrenheitInt = (int) fahrenheit; // Cast to int, truncates decimal

//result

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

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

scanner.close();

}

}

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 DaysOfWeekEnum {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a day of the week: ");

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

try {

DaysOfWeek day = DaysOfWeek.valueOf(input);

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

// Check if it's a weekend day

boolean isWeekend = day == DaysOfWeek.SATURDAY || day == DaysOfWeek.SUNDAY;

System.out.println("Is weekend? " + isWeekend);

} catch (IllegalArgumentException e) {

System.out.println("Invalid day: " + input);

}

scanner.close();

}

}

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 DirectionEnum {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

String input = scanner.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: " + input);

}

scanner.close();

}

}

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");

return Math.PI \* params[0] \* params[0];

}

},

SQUARE {

@Override

double area(double... params) {

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

return params[0] \* params[0];

}

},

RECTANGLE {

@Override

double area(double... params) {

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

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 ShapeEnum {

public static void main(String[] args) {

// Test each shape with sample inputs

System.out.printf("Circle area (radius=5): %.2f%n", Shape.CIRCLE.area(5));

System.out.printf("Square area (side=4): %.2f%n", Shape.SQUARE.area(4));

System.out.printf("Rectangle area (width=4, height=6): %.2f%n", Shape.RECTANGLE.area(4, 6));

System.out.printf("Triangle area (base=3, height=8): %.2f%n", Shape.TRIANGLE.area(3, 8));

}

}

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.ArrayList;

import java.util.Collections;

import java.util.List;

enum Suit {

CLUBS, DIAMONDS, HEARTS, SPADES

}

enum Rank {

ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING

}

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;

}

}

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 class CardDeck {

public static void main(String[] args) {

Deck deck = new Deck();

System.out.println("Original deck:");

deck.printDeck();

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

deck.shuffle();

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.

Code:

enum PriorityLevel {

LOW(1), MEDIUM(5), HIGH(8), CRITICAL(10);

private final int severity;

PriorityLevel(int severity) {

this.severity = severity;

}

public int getSeverity() {

return severity;

}

public boolean isUrgent() {

return severity >= 8;

}

}

public class PriorityLevelEnum {

public static void main(String[] args) {

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

System.out.printf("Priority: %s, Severity: %d, Urgent: %b%n",

level, level.getSeverity(), level.isUrgent());

}

}

}

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 TrafficLightEnum {

public static void main(String[] args) {

State current = TrafficLight.RED;

System.out.println("Starting state: " + current);

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

current = current.next();

System.out.println("Transition " + (i + 1) + ": " + current);

}

}

}

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 {

private final int bullets;

public Game(Difficulty difficulty) {

switch (difficulty) {

case EASY:

bullets = 3000;

break;

case MEDIUM:

bullets = 2000;

break;

case HARD:

bullets = 1000;

break;

default:

bullets = 1000; // Fallback

}

System.out.println("Game initialized with " + bullets + " bullets for " + difficulty);

}

}

public class GameDifficulty {

public static void main(String[] args) {

new Game(Difficulty.EASY);

new Game(Difficulty.MEDIUM);

new Game(Difficulty.HARD);

}

}

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:

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 IllegalStateException("Unknown operation");

}

}

}

enum OperationOverride {

PLUS {

@Override

double eval(double a, double b) {

return a + b;

}

},

MINUS {

@Override

double eval(double a, double b) {

return a - b;

}

},

TIMES {

@Override

double eval(double a, double b) {

return a \* b;

}

},

DIVIDE {

@Override

double eval(double a, double b) {

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

return a / b;

}

};

abstract double eval(double a, double b);

}

public class CalculatorOperations {

public static void main(String[] args) {

// Test switch-based version

System.out.println("Switch-based Operations:");

System.out.printf("PLUS: %.2f%n", OperationSwitch.PLUS.eval(10, 5));

System.out.printf("MINUS: %.2f%n", OperationSwitch.MINUS.eval(10, 5));

System.out.printf("TIMES: %.2f%n", OperationSwitch.TIMES.eval(10, 5));

try {

System.out.printf("DIVIDE: %.2f%n", OperationSwitch.DIVIDE.eval(10, 0));

} catch (ArithmeticException e) {

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

}

// Test override-based version

System.out.println("\nOverride-based Operations:");

System.out.printf("PLUS: %.2f%n", OperationOverride.PLUS.eval(10, 5));

System.out.printf("MINUS: %.2f%n", OperationOverride.MINUS.eval(10, 5));

System.out.printf("TIMES: %.2f%n", OperationOverride.TIMES.eval(10, 5));

try {

System.out.printf("DIVIDE: %.2f%n", OperationOverride.DIVIDE.eval(10, 0));

} catch (ArithmeticException e) {

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

}

}

}

9. 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) throw new IllegalArgumentException("Score cannot be negative");

if (score <= 3) return BEGINNER;

else if (score <= 6) return ADVANCED;

else if (score <= 9) return PROFESSIONAL;

else return MASTER;

}

}

public class KnowledgeLevelEnum {

public static void main(String[] args) {

// Test boundary conditions

int[] scores = {0, 3, 4, 6, 7, 9, 10};

for (int score : scores) {

try {

KnowledgeLevel level = KnowledgeLevel.fromScore(score);

System.out.println("Score " + score + ": " + level);

} catch (IllegalArgumentException e) {

System.out.println("Error for score " + score + ": " + e.getMessage());

}

}

}

}

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) {

// Division by zero

try {

int result = 10 / 0;

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[] array = {1, 2, 3};

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

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out of bounds!");

} 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); }

}

Code:

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, 3, 4, 5};

for (int n : numbers) {

try {

checkOdd(n);

System.out.println(n + " is even");

} catch (OddNumberException e) {

System.out.println(e.getMessage());

}

}

}

}

3.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.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;

public class FileReadDemo {

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

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

return reader.readLine();

}

}

public static void main(String[] args) {

String filename = "test.txt";

try {

String line = readFile(filename);

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

} 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:
  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.BufferedReader;

import java.io.FileReader;

import java.io.FileNotFoundException;

import java.io.IOException;

public class MultiExceptionDemo {

public static void main(String[] args) {

try {

// Open file

BufferedReader reader = new BufferedReader(new FileReader("test.txt"));

// Parse first line as integer

String line = reader.readLine();

int number = Integer.parseInt(line);

// Divide by the parsed number

int result = 100 / number;

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

reader.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");

}

}

}