Capgemini training – Assignment 1(Day 2)

```
using System;
class Program
{
    static void Main()
    {
        Exercise1();
        Exercise2();
        Exercise3();
        Exercise4();
        Exercise5();
        Exercise6();
        Exercise7();
        Exercise8();
        Exercise9();
        Exercise10();
    }
    // Exercise 1: Student Attendance & Eligibility
    static void Exercise1()
    {
        Console.Write("Enter classes attended: ");
        int attended = int.Parse(Console.ReadLine());
        Console.Write("Enter total classes: ");
        int total = int.Parse(Console.ReadLine());

        double percentage = (double)attended / total * 100;
        int displayPercentage = (int)Math.Round(percentage);
        Console.WriteLine($"Attendance Percentage: {displayPercentage}%");


    }
```

Q) Explain rounding vs truncation impact

When converting attendance percentage from double to int, **truncation** simply removes the decimal part (74.9 → 74), which may wrongly reduce eligibility. **Rounding** converts to the nearest whole number (74.9 → 75), giving a fair and more accurate attendance value. Hence, rounding is preferred for eligibility display.

// Exercise 2: Online Examination Result Processing

```
static void Exercise2()
{
    Console.Write("Enter marks 1: ");
    int m1 = int.Parse(Console.ReadLine());

    Console.Write("Enter marks 2: ");
    int m2 = int.Parse(Console.ReadLine());

    Console.Write("Enter marks 3: ");
    int m3 = int.Parse(Console.ReadLine());

    double average = (m1 + m2 + m3) / 3.0;
    Console.WriteLine($"Average: {average:F2}");
    int scholarshipScore = (int)Math.Round(average);
    Console.WriteLine($"Scholarship Score: {scholarshipScore}");
}
```

Q) Design the conversion flow and explain precision loss scenarios.

Conversion Flow:

Marks for each subject are stored as int since they are whole numbers. The final average is calculated as double to support decimal values and is rounded to two decimal places for display. For scholarship eligibility, the rounded average is then converted to int.

Precision Loss Scenarios:

Precision loss occurs when converting the double average to int because decimal values are removed. Truncation can reduce the score unfairly (85.9 → 85), while rounding minimizes this loss (85.9 → 86), making rounding the safer choice.

// Exercise 3: Library Fine Calculation

```
static void Exercise3()
{
    Console.Write("Enter fine per day: ");
    decimal finePerDay = decimal.Parse(Console.ReadLine());
    Console.Write("Enter days overdue: ");
    int daysLate = int.Parse(Console.ReadLine());
    decimal totalFine = finePerDay * daysLate;
    double analyticsFine = (double)totalFine;
    Console.WriteLine($"Total Fine: {totalFine}");
    Console.WriteLine($"Logged Fine (double): {analyticsFine}");

}
```

Q) Explain why different types are used and how conversions occur.

Different data types are used based on accuracy requirements. The fine per day is stored as decimal to ensure precise financial calculations, while overdue days are stored as int because they are whole numbers. The total fine is calculated in decimal to avoid rounding errors. For analytics purposes, the total fine is explicitly converted to double, where minor precision loss is acceptable and performance is better.

// Exercise 4: Banking Interest Calculation

```
static void Exercise4()
{
    Console.Write("Enter account balance: ");
    decimal balance = decimal.Parse(Console.ReadLine());

    Console.Write("Enter interest rate (%): ");
    float interestRate = float.Parse(Console.ReadLine());

    decimal interest = balance * (decimal)interestRate / 100;
    balance += interest;
```

```
    Console.WriteLine($"Updated Balance: {balance}");

}
```

Q) Demonstrate safe conversions and explain why implicit conversion may fail.

In the banking module, the account balance is stored as decimal to maintain financial accuracy, while the interest rate is received as float from an external API. A safe conversion is done by explicitly converting the float interest rate to decimal before calculation. Implicit conversion from float to decimal is not allowed because float has lower precision and may introduce inaccuracies. C# enforces explicit casting to prevent unintended data loss in financial computations.

// Exercise 5: E-Commerce Order Pricing

```
static void Exercise5()
{
    Console.Write("Enter cart total: ");
    double cartTotal = double.Parse(Console.ReadLine());

    Console.Write("Enter tax rate (decimal): ");
    decimal taxRate = decimal.Parse(Console.ReadLine());
    Console.Write("Enter discount: ");
    decimal discount = decimal.Parse(Console.ReadLine());

    decimal finalAmount = (decimal)cartTotal +
        ((decimal)cartTotal * taxRate) – discount;
    Console.WriteLine($"Final Payable Amount: {finalAmount}");

}
```

Q) Explain conversion strategy and precision risks.

In the e-commerce pricing engine, the cart total is accumulated as double due to frequent calculations, but tax and discount values are handled as decimal to ensure monetary accuracy. The conversion strategy involves explicitly converting the double cart total to decimal before applying tax and discounts,

and storing the final payable amount as decimal. Precision risks arise if double is used for financial values, as floating-point representation can cause rounding errors, which is why decimal is preferred for the final amount.

// Exercise 6: Weather Monitoring

```
static void Exercise6()
{
    Console.Write("Enter temperature in Kelvin: ");
    short sensorReading = short.Parse(Console.ReadLine());

    double celsius = sensorReading - 273.15;
    int displayTemp = (int)Math.Round(celsius);

    Console.WriteLine($"Temperature in Celsius: {displayTemp}°C");

}
```

Q) Discuss overflow and casting concerns.

In the weather monitoring system, sensor readings are stored as short, which has a limited range. While converting short to double is safe and does not cause overflow, issues can arise when converting the calculated average back to int. If values exceed the int range or are not properly handled, incorrect results may occur. Using rounding during casting ensures accurate display values and prevents data loss due to truncation.

// Exercise 7: University Grading Engine

```
static void Exercise7()
{
    Console.Write("Enter final score: ");
    double finalScore = double.Parse(Console.ReadLine());
    byte grade;
    if (finalScore >= 90) grade = 10;
    else if (finalScore >= 80) grade = 9;
    else if (finalScore >= 70) grade = 8;
    else grade = 7;
    Console.WriteLine($"Grade Point: {grade}");
```

}

Q) Explain validation and casting choices.

In the university grading engine, the final score is calculated as a double to preserve precision. Grades are stored as byte because they represent small, predefined numeric values. Before casting, validation is performed to ensure the score falls within valid grading boundaries. This prevents overflow and incorrect grade assignment, making the conversion safe and reliable.

// Exercise 8: Mobile Data Usage Tracker

```
static void Exercise8()
{
    Console.Write("Enter data usage in bytes: ");
    long bytesUsed = long.Parse(Console.ReadLine());

    double mb = bytesUsed / (1024.0 * 1024);
    double gb = bytesUsed / (1024.0 * 1024 * 1024);
    int roundedGB = (int)Math.Round(gb);

    Console.WriteLine($"Usage: {mb:F2} MB");
    Console.WriteLine($"Rounded Usage: {roundedGB} GB");

}
```

Q) Explain implicit conversions and rounding methods.

In the data usage tracker, usage is stored as long (bytes) and is **implicitly converted** to double when calculating MB and GB because converting from a smaller/integer type to a larger floating-point type is safe in C#. Since these values can be fractional, **rounding methods** such as Math.Round() are used to convert them to the nearest integer for monthly summaries, ensuring readable and accurate display values.

// Exercise 9: Warehouse Inventory Capacity

```
static void Exercise9()
{
    Console.Write("Enter current item count: ");
    int items = int.Parse(Console.ReadLine());
    Console.Write("Enter maximum capacity: ");
```

```
    ushort capacity = ushort.Parse(Console.ReadLine());
    bool isOverLimit = items > capacity;
    Console.WriteLine($"Over Capacity: {isOverLimit}");
}
```

Q) Explain signed vs unsigned conversion risks.

In the warehouse inventory system, item count is stored as int (signed) while maximum capacity is stored as ushort (unsigned). Signed vs unsigned conversion risks arise because unsigned types cannot represent negative values, and improper casting may lead to incorrect comparisons or overflow. To avoid this, the unsigned value is safely converted to a signed type before comparison, ensuring accurate and predictable results.

```
// Exercise 10: Payroll Salary Computation
static void Exercise10()
{
    Console.Write("Enter basic salary: ");
    int basicSalary = int.Parse(Console.ReadLine());

    Console.Write("Enter allowance: ");
    double allowance = double.Parse(Console.ReadLine());

    Console.Write("Enter deduction: ");
    double deduction = double.Parse(Console.ReadLine());

    decimal netSalary =
        basicSalary +
        (decimal)allowance -
        (decimal)deduction;

    Console.WriteLine($"Net Salary: {netSalary}");
```

```
    }
}
```

Q) Design type conversion flow and justify choices.

In the payroll system, the basic salary is stored as int because it is a whole number, while allowances and deductions are stored as double to handle fractional values. During calculation, these values are combined and then converted to decimal to compute and store the net salary. Using decimal for the final amount ensures high precision and avoids floating-point rounding errors, which is essential for acc urate payroll processing.