Name: Shashank Chauhan

UID: 22BCS10698

Batch: 4

.NET 8 / C# 12 – Detailed Case Study Exercises Value Types & Type Conversions

Exercise 1:

```
using System;
class Ex1
{
   static void Main()
   {
      int attended = 68, total  = 80;
      double percent  = (double) attended / total * 100;
      int result = (int)Math.Round(percent );
      Console.WriteLine("Attendance %: " + result);
   }
}
```

Truncation simply removes the decimal part, which can underestimate values and lead to unfair or incorrect decisions.

Rounding adjusts the value to the nearest whole number, giving a more accurate and balanced result, so it is preferred in eligibility and grading systems.

Exercise 2:

```
using System;
class Ex2
{
   static void Main()
   {
      int marks = 423, sub = 5;
```

```
        double avg = Math.Round((double)marks / sub, 2);
        int i = (int)Math.Round(avg);


        Console.WriteLine("Average: " + avg);
    }
}
```

The conversion flow starts with exact types like int for input, moves to double for averaging or calculations, and finally converts to int only for display or decisions.

Precision loss occurs when decimal values are dropped during this conversion, so rounding is used instead of truncation to minimize errors.

Exercise 3:

```
using System;
class Ex3
{
    static void Main()
    {
        decimal finday = 2.5m;
        int days = 6;
        decimal t = finday* days;
        double l = (double)t;
        Console.WriteLine("Fine: " + t);
    }
}
```

Data types are chosen according to usage int for whole values, decimal for monetary calculations to maintain accuracy, and double for analytical purposes where speed is important.

Type conversion is done automatically when it is safe, and manually through casting when there is a possibility of data loss or precision issues.

Exercise 4:

```
using System;
class Ex4
{
    static void Main()
```

```
    {
        decimal balance = 50000m;
        float rate = 7.5f;
        decimal interest = balance * (decimal)rate / 100 / 12;
        balance += interest;
        Console.WriteLine("Balance: " + balance);
    }
}
```

Safe conversions are done using explicit casting or conversion methods when there is a possibility of losing precision, such as while converting float or double to decimal.

Implicit conversion is not allowed in such cases because C# prevents automatic conversion between types with different precision to avoid hidden data loss, so explicit casting is required.

Exercise 5:

```
using System;
class Ex5
{
    static void Main()
    {
        double cartTotal = 1299.99;
        decimal finalAmount = (decimal)cartTotal * 1.08m;
        Console.WriteLine("Final Amount: " + finalAmount);
    }
}
```

A conversion strategy uses broader data types during calculations and converts the final result into more accurate types like decimal.

Precision issues can occur when converting from float or double, so explicit casting is used and rounding is applied only at the end.

Exercise 6:

```
using System;
class Ex6
```

```
{
   static void Main()
   {
      short s = 320;
      double t = s / 10.0;
      int d = (int)Math.Round(t);
      Console.WriteLine("Temp: " + d + " C");
   }
}
```

Overflow happens when a value goes beyond the limit of its data type during calculations or conversions, which can cause wrong results or errors.

Casting issues occur when converting from a larger or decimal type to a smaller one, leading to data loss. These problems can be avoided by validating values and using safe or wider data types.

Exercise 7:

```
using System;
class Ex7
{
   static void Main()
   {
      double s = 86.5;
      byte g;
      if (s >= 80)
         g = 9;
      else
         g = 8;
      Console.WriteLine("Grade: " + g);
   }
}
```

Validation ensures the value is within an acceptable range before conversion, preventing runtime errors or incorrect data.

Casting choices are made to avoid data loss or overflow; instead of direct casting, controlled logic or methods like Convert are used to safely transform values.

Exercise 8:

```
using System;

class Ex8

{

   static void Main()

   {

      long b = 5368709120;

      double g = b / 1024.0 / 1024 / 1024;

      int r = (int)Math.Round(g);

      Console.WriteLine("Usage: " + r + " GB");

   }

}
```

Implicit conversions are automatically handled by C# when the conversion is safe and does not cause data loss, for example converting an int to a double. These conversions generally move from a smaller to a larger data type.

When converting decimal values to integers, rounding methods are applied. Math.Round() gives the nearest whole number, while Math.Floor() and Math.Ceiling() round down and up respectively. Rounding is preferred over simple truncation to keep results accurate..

Exercise 9:

```
using System;

class Ex9

{

   static void Main()

   {

      int i = 5200;

      ushort m = 6000;

      Console.WriteLine("Within Limit: " + (i <= m));

   }

}
```

Signed types can store negative values, but unsigned types cannot.

When a signed value is converted to unsigned, a negative number can turn into a very large positive number, causing incorrect comparisons or overflows.

To avoid this, I try not to mix signed and unsigned types and prefer converting both values to a wider signed type before comparison.

Exercise 10:

```
using System;

class Ex10

{

   static void Main()

   {

      int b = 30000;

      double a = 4500.75, d = 1200.25;

      decimal n = (decimal)b + (decimal)a - (decimal)d;

      Console.WriteLine("Net Salary: " + n);

   }

}
```