ASSIGNMENT (sneha pahuja)

Exercise 1: Student Attendance & Eligibility System

```
int total = 15;
int attended = 6;
double percent = (double)attended / total * 100;
int eligib = Convert.ToInt32(percent);
Console.WriteLine("Display attendance: " + percent);
Console.WriteLine("Exact attendance: " + eligib);
```

Attendance is stored as int and percentage is calculated as double to preserve decimal precision. For display, it is converted to int using Convert.ToInt32(), which rounds instead of truncating. Truncation would underestimate attendance, while rounding provides a fair representation.

Exercise 2: Online Examination Result Processing

```
int eng = 78;
int maths = 85;
int phy = 92;
int chem = 90;
decimal avg = (eng + maths + phy + chem) / 4m;
Console.WriteLine($"Average: {avg:F2}");
int schol = Convert.ToInt32(avg);
Console.WriteLine("Scholarship:" + schol);
```

Marks are stored as int and averaged as decimal to preserve fractional values. For scholarship, the average is converted to int using Convert.ToInt32(), which rounds to the nearest whole number. Direct casting would truncate and may underestimate eligibility, so rounding ensures fairness while maintaining type safety.

Exercise 3: Library Fine Calculation System

```
decimal fine = 0.75m;
int overdue = 4;
decimal totalFine = fine * overdue;
double fineLog = (double)totalFine;
Console.WriteLine($"Total fine: {totalFine:F2}");
Console.WriteLine($"Logged fine: {fineLog}");
```

Daily fine is stored as decimal for exact monetary precision, while overdue days are int. Total fine is calculated as decimal for display and explicitly converted to double for logging and analytics. Decimal ensures exact money representation, while double allows faster calculations and aggregation.

Exercise 4: Banking Interest Calculation Module

```
decimal accountBalance = 10000.00m;
float annualInterestRate = 3.5f;
decimal interestRateDecimal = (decimal)annualInterestRate;
decimal monthlyInterest = accountBalance * interestRateDecimal / 100 /
12;
accountBalance += monthlyInterest;
Console.WriteLine($"Monthly interest: {monthlyInterest:F2}");
Console.WriteLine($"Updated balance: {accountBalance:F2}");
```

Account balance is decimal for precision, while interest rate is float. It is explicitly converted to decimal before calculation to avoid precision loss. Implicit conversion fails here because float to decimal is narrowing and not allowed, so explicit cast ensures safe calculation.

Exercise 5: E-Commerce Order Pricing Engine

```
double cart = 199.99;
cart += 49.99;
cart += 15.50;
```

```
decimal total = (decimal)cart;
decimal tax = 0.18m;
decimal disco = 0.10m;
decimal discountAmount = total * disco;
decimal taxableAmount = total - discountAmount;
decimal taxAmount = taxableAmount * tax;
decimal final = taxableAmount + taxAmount;
Console.WriteLine("Final Payable Amount: " + final);
```

Cart total is accumulated as double, but tax and discount are decimal to avoid floating-point errors. Double is explicitly converted to decimal before further operations. This ensures monetary calculations remain precise, and rounding or conversion errors are avoided.

avoided.

Exercise 6: Weather Monitoring & Reporting

```
short sensorValue = 253;
double temperatureCelsius = (sensorValue - 32) * 5.0 / 9.0;
double dailyAverage = temperatureCelsius;
int displayAverage = Convert.ToInt32(dailyAverage);
Console.WriteLine("Daily Average: " + displayAverage);
```

Sensor values are short and converted to Celsius as double to preserve precision. Daily average is converted to int using Convert.ToInt32(), which rounds rather than truncates. Overflow is unlikely due to realistic temperatures, and casting ensures safe dashboard display.

```
Exercise 7: University Grading Engine
double finalScore = 87.6;
byte grade;
if (finalScore < 0)
finalScore = 0;
else if (finalScore > 100)
```

```
finalScore = 100;
grade = Convert.ToByte(finalScore);
Console.WriteLine("Grade: " + grade);
```

Final score is double and converted to byte for grade storage. Validation ensures the score stays within 0–100 to prevent overflow. Convert.ToByte() rounds safely to the nearest integer, preventing invalid grades while keeping type safety.

Exercise 8: Mobile Data Usage Tracker

```
long usageBytes = 5368709120;
double usageMB = usageBytes / 1024.0 / 1024.0;
double usageGB = usageMB / 1024.0;
int monthlyMB = Convert.ToInt32(usageMB);
int monthlyGB = Convert.ToInt32(usageGB);
Console.WriteLine("Monthly MB: " + monthlyMB + "Monthly GB: " +
monthlyGB);
```

Usage is tracked in bytes (long) and converted to MB/GB as double using implicit conversion. Monthly summaries are rounded to int using Convert.ToInt32(). Implicit conversion ensures large byte values are handled safely, and rounding provides summary which can be easily understood.

Exercise 9: Warehouse Inventory Capacity Control

```
int currentItems = 500;
ushort maxCapacity = 450;
if (currentItems > (int)maxCapacity)
{
Console.WriteLine("Capacity exceeded");
}
else
{
```

```
Console.WriteLine("Capacity OK");
}
int maxCapacityInt = (int)maxCapacity;
Console.WriteLine($"Max capacity: {maxCapacityInt}");
```

Current items are int and max capacity is ushort. Explicit conversion of ushort to int avoids signed vs unsigned comparison issues. This ensures correct comparison and reporting, preventing misinterpretation of inventory limits.

Exercise 10: Payroll Salary Computation

```
int basicSalary = 30000;
double allowance = 4500.75;
double deduction = 1200.50;
decimal netSalary = (decimal)basicSalary + (decimal)allowance -
(decimal)deduction;
Console.WriteLine($"Net Salary: {netSalary}");
```

Basic salary is int, and allowances, deductions are double. All components are converted to decimal before calculation to maintain money precision. Explicit casting prevents floating-point errors and ensures accurate net salary.