Coding Challenges

The objective of these progressive coding challenges is to consolidate your understanding of C#. You will be directed to them by the course tutor at appropriate moments. The underlying scenario is based around the creation of a simple banking application that allows a user to find their current account balance and also make credits and debits to the account.

NOTE: The challenges will probably be set as an alternative to one of the exercises. It is highly unlikely, for there to be time for you to tackle both whilst the course is in progress. However, do feel free to tackle uncompleted tasks and exercises after the course has completed.

Bank Account Application

TASK 1: To be tackled after the Functions session

Overview:

Create a console application that includes functions that return a user's bank account balance, make credits to and debits from their account. Add code to the Main function that calls the functions and displays the balance after each transaction.

Use Cases:

- As a user I want to be able to create a bank account object and specify an account number, account name and initial balance.
- As a user I want to be able to find what my current bank balance is.
- As a user I want to be able to make credits and debits to my account.
- As a developer I want to ensure the credit, debit and balance determination logic only occurs once in my code.

Enhancement (if you have time):

- As a user I want to be able to specify the amount of money I want to credit or debit by being prompted when to do this.
- As a developer I want to make sure the code to do these things only occurs once but also ensures a number of credits and debits can be made before the program terminates.

TASK 2: To be tackled after the Flow Control session

Overview:

Create a simple console-based menu that gives the user the options of checking their current balance and making credits and debits to their account. The user should be instructed to enter "1" for the balance, "2" to credit the account, "3" to debit the account (don't worry about allowing the user to repeat the process, we will deal with that in the next challenge). Ensure the selections trigger calls to the appropriate function.

Use Cases:

As a user I want to be able to select an option from a menu that allows me to request either a bank balance or credit or debit an account.

As a developer I want to make sure the code that displays the menu and deals with the response only occurs once in my logic.

Enhancement (if you have time):

As a developer I want to ensure the credit and debit values entered by a user are within an acceptable range of sensible values. Negative values for crediting and debiting should never be permitted.

TASK 3: To be tackled after the Arrays, Loops and Collections session

Overview:

The banking program logic needs to be adjusted so the user can interact with the menu as many times as they like with the program terminating only after they enter an upper or lowercase "q". In addition. When the program starts it should prompt the user for an account number. If the number is wrongly entered more than three times the program should terminate.

Use Cases:

As a user I'd like to interact with the applications menu as many times as I like such that the program only terminates if I enter an upper or lowercase "q".

As a user, when the program first starts, I'd like to be prompted to enter my account number. If I enter a wrong account number I'd like to be able to try again for a maximum of three attempts.

As a developer if the user enters a valid account number I'd like to retrieve their bank balance from a Dictionary that has a key of string (for the account number) and a value of decimal (for the balance).

Enhancement (if you have time):

As a user I'd like to be prompted for a credit or debit amount and if I enter an invalid value I'd like to be repeatedly prompted until I either enter an acceptable value or a "c" (upper or lower case) that will indicate I want to cancel the operation and return to the main menu.

TASK 4: To be tackled after the Unit testing session

Overview:

The banking program logic needs to include a second project that is configured with a set of unit tests that can be run to confirm the Code Under Test (CUT) works as it is expected to.

Use Cases:

As a developer I'd like to add a set of unit tests to the program so as to confirm all the code I've **written as functions** works as expected.

Coding Challenge: Coffee Shop Order System

Scenario

You are tasked with creating a simple console-based coffee shop order system. The system should allow customers to order drinks, track the orders, and generate a simple receipt.

Requirements

1. **Classes (OO basics, no inheritance required)**
   - Create a `Drink` class with properties:
     - `Name` (string)
     - `Price` (decimal)
   - Create an `Order` class with properties:
     - `Drinks` (List of `Drink`)
   - The `Order` class should have methods:
     - `AddDrink(Drink drink)` – adds a drink to the order
     - `GetTotal()` – returns the total price of the order
2. **Collections**
   - Maintain a `Dictionary<string, Drink>` as a menu, where the key is the drink name and the value is the `Drink` object.
3. **Loops**
   - Use a loop to continuously prompt the user for input (selecting drinks) until they type `"done"`.
4. **If-Else**
   - If the user types a valid drink name, add it to the order.
   - If the user types an invalid name, show a message: `"Drink not found, please try again."`
5. **Unit Testing**
   Write unit tests for:
   - Adding drinks to the order.
   - Calculating the total.
   - Handling an empty order.

Example Console Flow

```
Welcome to ChatGPT Coffee Shop!
Menu:
- Espresso ($2.50)
- Latte ($3.50)
- Cappuccino ($3.00)

Type the name of the drink to order, or "done" to finish:

> Latte
```

```
Latte added to your order.
> Espresso
Espresso added to your order.
> Mocha
Drink not found, please try again.
> done

Your total is: $6.00
Thank you for your order!
```

The Personal Organizer App

As the course progresses you will be challenged to create a “Personal Organiser” app. You will start small but progress to an app that keeps track of tasks that the user needs to get done. Each challenge includes:

- Overview (what the challenge is about)
- Use Cases (examples of expected behavior)
- Extension Tasks (optional stretch goals)

Challenge 1: Unit Converter (Functions & Basic I/O)

Overview:
Start your personal organizer by writing functions that create and display tasks. A *task* will be represented simply as a text string (no class yet). The goal is to practice writing and calling functions, passing arguments, and returning values.

What you’ll do:

- Write a function that takes a description (string) and deadline (DateTime) and combines them into a single task string.
- Write another function that prints a formatted task to the console.
- In Main, ask the user for a description and a priority, create a task, and display it.

Use Cases:

1. Input: Description = "Buy groceries", Deadline = “01/01/2027”
   → Output: "Task: Buy groceries by 01/01/2027”
2. Input: Description = "Do laundry", Deadline = “01/02/2027”
   → Output: "Task: Do laundry by 01/02/2027”
3. Input: Description = "Finish report", Deadline = “01/03/2027”
   → Output: "Task: Finish report by 01/03/2027”

Extension Tasks:

4. Add a function to mark a task as completed ("Task: Buy groceries by 01/01/2027 - Completed").

- Print the dates in different formats e.g. 01 Jan 2027 or Fri 01 Jan 2027)

Challenge 2: Task Prioritiser (Conditionals)

Overview:
Extend the organizer to accept tasks with a priority level (low, medium, high). Use conditional expressions to categorize tasks and display a message depending on their urgency.

Use Cases:

1. Input: Task = "Finish report", Deadline = “01/01/2027”, Priority = High → Output: "ALERT: High-priority Task: Finish report by 01/01/2025 "
2. Input: Task = "Do laundry", Deadline = “01/01/2027”, Priority = Low → Output: "Low-priority Task: Do laundry by 01/01/2025 "
3. Input: Task = "Pay bills", Deadline = “01/01/2027”, Priority = Medium → Output: "Medium-priority Task: Pay bills by 01/01/2025 "

Extension Tasks:

- Add a “due today” flag and highlight urgent tasks.
- Validate that priority input is only “low/medium/high.”
- Allow multiple tasks in one run.

Challenge 3: To-Do List Manager (Loops + Collections)

Overview:
Build a program that maintains a list of tasks. The user can repeatedly add, view, or remove tasks until they choose to exit. Use loops for continuous interaction and List<string> for task storage.

Use Cases:

1. User selects "Add Task → 'Buy groceries'" → 01/01/2027 → Low→ Program stores it.
2. User selects "Add Task → 'Walk the dog'" → 01/02/2027 → Medium →Program stores it.
3. User selects "View Tasks" → Output:

```
1. Low-priority task: Buy groceries by Fri 01 Jan 2027
```

```
2. Medium-priority task: Walk the dog by Fri 01 Jan 2027
```

4. User selects "Remove Task → 1" → Remaining list:

```
1. Medium-priority task: Walk the dog by Fri 01 Jan 2027
```

Extension Tasks:

- Prevent duplicate tasks.
- Add a “Clear all tasks” option.
- Save tasks to a file and reload them when the program starts.

Challenge 4: Task Class (Basic OOP)

Overview:
Introduce TaskItem and TaskManager classes to make tasks more structured.

Each TaskItem should have the following properties:

- Id (auto-assigned)
- Description
- Deadline (date)
- Priority (low, medium, high)
- IsCompleted (default: false)

And the following methods:

- An override of ToString() returns a string of the following forms:

  "{id} – ALERT: High-priority task: {description} by {deadline} – Completed”

  "{id} – Medium-priority task: {description} by {deadline} – Not Completed”

  "{id} – Low-priority task: {description} by {deadline} – Completed”
- MarkComplete() sets IsCompleted to true.

The TaskManager class should encapsulate a List<TaskItem> and expose the following methods:

- Add (takes description, deadline and priority as parameters and adds a new TaskItem to its collection).
- Remove (takes a TaskItem's Id as a parameter and removes the specified item from the list).
- MarkComplete (takes a TaskItem's Id as a parameter and sets the specified item's IsCompleted property to true).
- User selects "Add Task → 'Buy groceries'" → 01/01/2027 → Low→ Program stores it.

Use Cases:

1. User selects "Add Task → 'Buy groceries'" → 01/01/2027 → High→ Program stores it.
2. User selects "Add Task → 'Walk the dog'" → 01/02/2027 → Medium →Program stores it.
3. User selects "View Tasks" → Output:

```
1. ALERT: High-priority task: Buy groceries by Fri 01 Jan 2027 – Not Completed
2. Medium-priority task: Walk the dog by Fri 01 Jan 20272027 – Not Completed
```

4. User selects "Mark Complete" → 1 → Program stores it.
5. User selects "View Tasks" → Output:

```
1. ALERT: High-priority task: Buy groceries by Fri 01 Jan 2027 – Completed
2. Medium-priority task: Walk the dog by Fri 01 Jan 20272027 – Not Completed
```

Extension Tasks:

- Allow editing a task description, dead line and/or priority.
- Display tasks sorted by priority.

---

1. **Challenge 5: Unit Testing (with xUnit or NUnit)**

Overview:
Write automated tests for the Task class and task list operations. Ensure that methods work correctly under different conditions.

Use Cases (Test Scenarios):

1. When calling MarkComplete() on a new task → IsCompleted == true.

2. When adding a task to the list → list size increases by 1.
3. When removing a task by ID → task no longer appears in the list.
4. When trying to remove a non-existent task → list remains unchanged.

Extension Tasks:

- Test invalid inputs (e.g., empty description).
- Test sorting by priority.
- Add a helper method IsOverdue() (if due date was added) and write tests for it.

C# Coding Challenge: Vehicle Rental System

Scenario

You are building a **Vehicle Rental System** for a rental company. Customers can rent cars or motorcycles. The system should manage availability, allow searching/filtering, and persist data using **Entity Framework Core**.

Requirements

1. **OO Classes with Inheritance & Abstraction**
   - Create an **abstract class Vehicle** with properties:
     - Id (int)
     - Model (string)
     - DailyRate (decimal)
     - IsRented (bool)
     - abstract void Rent() and abstract void Return() methods.
   - Create two concrete classes:
     - Car (with extra property NumberOfDoors)
     - Motorcycle (with extra property HasSidecar)
2. **Interfaces**
   - Define an interface IRentable with Rent() and Return() methods.
   - Both Car and Motorcycle implement IRentable.
3. **Collections**
   - Maintain a List<Vehicle> for in-memory rental operations (pre-populate with example vehicles)
4. **Unit Testing Part 1**
   - Write tests for:
     - Renting a vehicle (marks IsRented = true).
     - Returning a vehicle.
5. **UI project**
   - Implement a console menu loop:
     - 1. View all vehicles
     - 2. Rent a vehicle
     - 3. Return a vehicle
     - 4. Exit
   - Use if-else to handle invalid input.

Example Console Flow

```
Welcome to Vehicle Rental System!
1. View all vehicles
2. Rent a vehicle
3. Return a vehicle
4. Exit
> 1
```

```
Vehicles:
1. Car: Toyota Corolla (4 doors, $40/day, Available)
2. Motorcycle: Harley Davidson (Sidecar: No, $55/day, Available)

> 2
Enter vehicle ID to rent:
1
You rented Toyota Corolla.

> 2
Enter vehicle ID to rent:
1
Error: Vehicle is not available.

> 3
Enter vehicle ID to return: 1
Toyota Corolla returned successfully.

> 3
Enter vehicle ID to return: 1
Error: Vehicle is not currently rented so can't be returned.

> 4
Goodbye!
```

6. **LINQ**
    - Use LINQ queries to:
        - List all available vehicles
        - Find available cars under a certain daily rate.
        - List all currently rented vehicles.
7. **Unit Testing Part 2**
    - Write tests for:
        - LINQ query for filtering by price.
        - Exception handling when renting an unavailable vehicle.
8. **UI project**
    - Implement the new features into the existing code:

```
Welcome to Vehicle Rental System!
1. View all vehicles
2. View available vehicles
3. Rent a vehicle
4. Return a vehicle
5. Search vehicles under price
6. View rented vehicles
7. Exit
```

9. **Exception Handling**
    - Create a custom exception VehicleNotAvailableException that is thrown when trying to rent an already rented vehicle.
    - Handle database save exceptions gracefully.
10. **Unit Testing Part 3**
    - Write tests for:
        - Exception handling when renting an unavailable vehicle.

11. **UI project**
    - Implement code that handles the exceptions and displays appropriate messages

12. **Database Access (Entity Framework Core)**
    - Use EF Core with a RentalDbContext that has DbSet<Car> and DbSet<Motorcycle>.
    - Seed some vehicles into the database.
    - Persist rent/return actions.
        -

Example Console Flow

```
Welcome to Vehicle Rental System!
1. View all vehicles
2. View available vehicles
3. Rent a vehicle
4. Return a vehicle
5. Search vehicles under price
6. View rented vehicles
7. Exit
> 1

Vehicles:
1. Car: Toyota Corolla (4 doors, $40/day, Available)
2. Motorcycle: Harley Davidson (Sidecar: No, $55/day, Available)

> 3
Enter vehicle ID to rent:
1
You rented Toyota Corolla.

> 3
Enter vehicle ID to rent:
1
Error: Vehicle is not available.

> 5
Enter max price:
50
Found: Toyota Corolla - $40/day (Not available)

> 7
Goodbye!
```