**Welcome to my class!**

I'm Maverick!
We will be covering Part I of Object-Oriented Programming Concepts in C#. The concepts we will be covering are listed below:

01 Classes and Objects

02 Methods

03 Fields

04 Properties

05 Constructor

06 Control Flow

# Concept 1: Classes and Objects

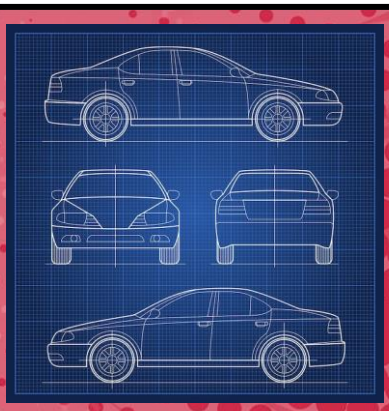
In this lesson, we will be covering classes and objects.

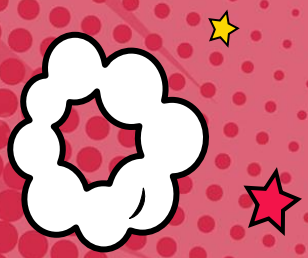
Firstly, I want you all to think of a class as a blueprint for creating things.

Imagine this blueprint(class) is of a car.

Inside this blueprint, we have attributes like 'colour' that determine what colour it should be, or 'speed' that determines how fast it is.

The blueprint also includes methods, which are instructions that tell you what the car can do. Like startEngine, to start the engine of the car.





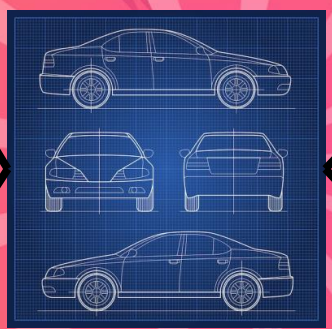
In programming, the class defines the structure for creating objects.

When you create (or instantiate) an object, like this blue car, we are essentially assembling the specific attributes according to the class's instructions to create a unique car object that behaves, and functions as defined by the class.

Below is a diagram of classes and objects.

Attributes:
Model
Colour

Class



Objects

Lamborghini
Yellow

Toyota
Red

Method:
startEngine()
Stop()

# Concept 2: Methods

In this lesson, we will be covering **methods**.

As mentioned earlier in Concept 1, **methods** are actions that an object can do. They are the verbs of a noun (**object**). In the context of cars (**class**), a Toyota, the noun, can ignite the engine and accelerate, which is the verb.

In programming, a **method** is also known as a **function**, and it tends to be used interchangeably. Now why would you use methods? It is to reuse code as many times as you like after the code has been defined once.

In this example, **Accelerate()** is the name of our method. **Static** means this method belongs to the **class Car**, not to any other objects. And the **void** keyword indicates this method doesn't return any value.

```
1 reference
class Car
{
    3 references
    static void Accelerate()
    {
        // code to increase speed
    }
}
```


To use(or call) a method, we simply write its name followed by **parentheses ()** and a **semicolon ;**

When we run this code, it will output **"The car is accelerating!"** to the console, indicating that the **Accelerate()** method has been executed.

Just like how you can press the gas pedal multiple times to accelerate, you can call a method multiple times.

```
class Car
{
    3 references
    static void Accelerate()
    {
        Console.WriteLine("The car is accelerating!");
    }
    0 references
    static void Main(string[] args)
    {
        Accelerate(); // First acceleration
        Accelerate(); // Second acceleration
        Accelerate(); // Third acceleration
        //Output will repeat Console.Writeline three times
    }
}
```

## Concept 3: Fields

In this lesson, we will be covering fields and how it works in C#.

We will continue to use cars to explain this concept so that it is relatable and understandable.

Now, tell me what you would see in a car's dashboard in the driver's seat… That's right, you would see important information like fuel gauge and speed.

Fields are like instruments on a car's dashboard. They hold important information about the car's state.

In C#, we define fields inside a class to store data. For a car, we might have fields like 'Speed' and 'fuelLevel.'

```
Public class Car
{
    //Field to store the car's speed
    public int speed;
    //Field to store the car's fuel level
    public float fuelLevel;
}
```

There are two ways to use fields.

The first way create objects that all use the same default fields.

50km/h

50km/h

The second way involves modifying each object's fields. Allowing objects to have unique fields.

50km/h 

80km/h 

## Concept 4: Properties

In this lesson, we will be covering **properties**. But using a different analogy.

Before we begin, I want to briefly touch upon the concept of **encapsulation**. Encapsulation, in essence, is trying to keep sensitive data hidden from users. This is achieved by declaring fields as **private**.

Locked House
(Private field)

Imagine the **private field** 'house' as a locked house. Now imagine the **property** as a house key.

Because that's exactly what a property does. A control access to the 'house' property field

Key
(Property)

In C#, **private variables/fields** can only be accessed within the same class. This means other classes can't use that field. But if we do need access to that field, we can do it through **property**.

A **property** is like a combination of a **variable** and **method**, it has two methods called '**get**' and '**set**'.

```
public class House
{
    private string location; // Field to store the house's location

    public string Location // Property to control access to location field
    {
        get { return location; } // Getter method
        set { location = value; } // Setter method
    }
}
```

The **Location property** is associated with the **location field**.

The **get** method returns the value of the variable location and the **set** method assigns a value to the location variable.

The **value** keyword represents the value we assign to the property.

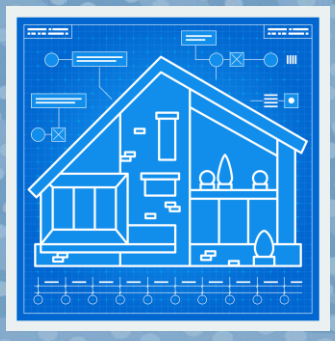
```
public class Realtor
{
    public void ShowHouseDetails(House house)
    {
        Console.WriteLine("House Location: " + house.Location);
    }
}

public class Program
{
    static void Main(string[] args)
    {
        House myHouse = new House();
        myHouse.Location = "Los Angeles";

        Realtor realtor = new Realtor();
        realtor.ShowHouseDetails(myHouse);
    }
}
```

Other classes outside the **class House** can now also use the **location field** through the **Location property**.

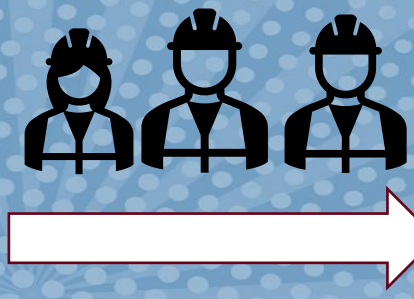
# Concept 5: Constructor

In this lesson, we're learning about **constructors**, and we will be using an analogy of houses.

Firstly, let's pretend that you're an architect and you want to build a neighbourhood. You write up a blueprint(**class**) to build a house(**object**). But who's going to build the houses? That would be the construction team (**constructor**).



Blueprint (Class)



Construction Team (Constructor)

House (Object)

In programming, each time we call the **constructor**, it's like telling the construction team to build another house, creating objects from our class. A constructor is a '**special method**.'

If you tell them to build 5 houses according to the blueprint, you will have 5 houses based on the same blueprint. The houses are made from the same mold, but they are **independent instances**. By using parameters, **constructors** can create objects with custom attributes like size and colours. If you make a mistake, fixing them is easy.

The advantage that constructor brings, is that it offers **ease of reproduction**.

Constructors help reduce the amount of code written, making it **less cluttered**. It can also be used to set **initial values** for fields when no parameters are set.

```
1 reference
class Program
{
    0 references
    static void Main(string[] args)
    {
        Car Ford = new Car();
        Ford.model = "Mustang";
        Ford.color = "red";
        Ford.year = 1969;

        Car Opel = new Car();
        Opel.model = "Astra";
        Opel.color = "white";
        Opel.year = 2005;

        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}
```

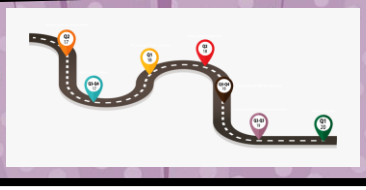
No Constructor

```
1 reference
class Program
{
    0 references
    static void Main(string[] args)
    {
        Car Ford = new Car("Mustang", "Red", 1969);
        Car Opel = new Car("Astra", "White", 2005);

        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}
```

With Constructor (Less clutter)

# Concept 6: Control Flow

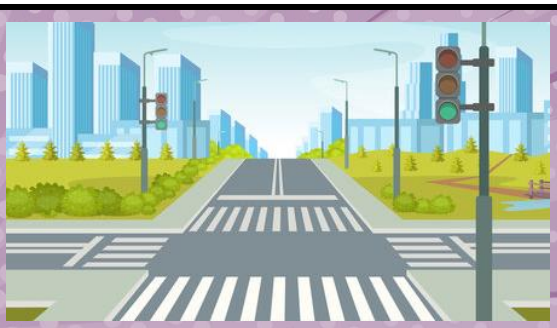
For this lesson, we will be learning **control flow**. In short, control flows refer to the **order** instructions occur. There are **three main types of control flows**.



The first type of control flow is **sequence**. The sequence is like following a roadmap – each step leads to the next. This code snippet will print out the code in order of **top to bottom**

```
0 references
class CarJourney
{
    0 references
    static void Main()
    {
        StartCar(); // Step 1: Start the car
        DriveStraight(); // Step 2: Drive straight
        TurnRight(); // Step 3: Turn right
    }
    1 reference
    static void StartCar()
    {
        Console.WriteLine("Car started.");
    }
    1 reference
    static void DriveStraight()
    {
        Console.WriteLine("Driving straight.");
    }
    1 reference
    static void TurnRight()
    {
        Console.WriteLine("Turned right.");
    }
}
```

```
0 references
class Crossroads
{
    0 references
    static void Main()
    {
        bool isGreenLight = true; // Assume green light initially

        if (isGreenLight)
        {
            GoStraight();
        }
        else
        {
            TurnLeft();
        }
    }
    1 reference
    static void GoStraight()
    {
        Console.WriteLine("Traffic light is green. Going straight.");
    }
    1 reference
    static void TurnLeft()
    {
        Console.WriteLine("Traffic light is red. Turning left.");
    }
}
```



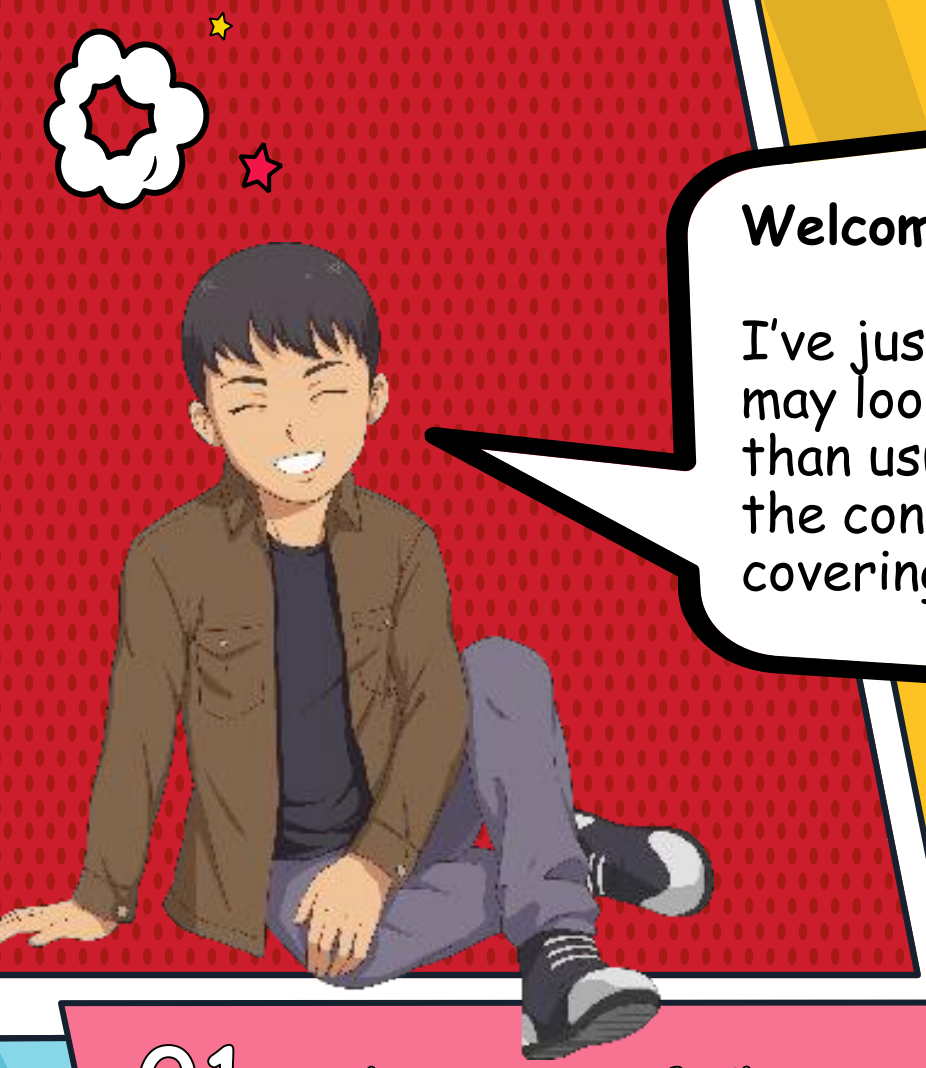
The second type of control flow is **selection**. Selection (If-Else) is **making choices**, it's like reaching a crossroads and deciding where to go.

In this code snippet, the car will go straight if the light is green, else left if it is red. It depends on whether isGreenLight is true or false.

The third type of control flow is **repetition**, aka, a **loop**. A loop is doing something over and over until the right condition is achieved. In the context of cars, it's driving in circles until you find the right exit.

There are two types of loops, **for loop** and **while loop**. The difference is that the for-loop is used when you know exactly how many times you want to repeat something and vice versa.

```
0 references
class Roundabout
{
    0 references
    static void Main()
    {
        int exits = 0;

        while (exits < 3) // Loop until the third exit is reached
        {
            DriveInCircle();
            exits++;
        }
    }

    1 reference
    static void DriveInCircle()
    {
        Console.WriteLine("Driving in circles around the roundabout.");
    }
}
```



**Welcome back for Part 2!**

I've just got a haircut, so I may look a little different than usual. Anyhow, here are the concepts we will be covering:

01 Inheritance of Classes

02 Inheritance of Interfaces

03 Polymorphism

04 Delegates

05 Lambdas

## Concept 1: Inheritance of Classes

In this lesson, we will be covering class inheritance.

But first, what is inheritance? By definition, it is the act of inheriting something or passing on something.

Let's use an analogy about cars again for starters. What do we know about cars? A car can drive, brake, and it has wheels.

Now what differs a sports car from a normal car? A sports car has and can do everything a normal car can do, but it has its unique features like being able to be turbocharged.

In this analogy, the Car is the parent class and the SportsCar is the child class. The child class inherits attributes and methods from the parent's class. But the child class can also have its unique attributes as well.


Parent Class:
Car
- Drive()
- Brake()
- Honk()

Child Class:
SportsCar
- Drive()
- Brake()
- Honk()
- TurboCharge()

```
using system;
// Base class: Car
1 reference
class Car
{
    0 references
    public void Drive()
    {
        Console.WriteLine("Car is driving.");
    }

    0 references
    public void Brake()
    {
        Console.WriteLine("Car is braking.");
    }

    0 references
    public void Honk()
    {
        Console.WriteLine("Car is honking.");
    }
}
// Derived class: SportsCar
0 references
class SportsCar : Car
{
    0 references
    public void Turbocharge()
    {
        Console.WriteLine("Sports car is turbocharged.");
    }
}
```

In the world of programming, inheritance works similarly. We have a base or parent class that contains common attributes (fields/properties) and methods (functions/procedures), and then we can create derived or child classes that inherit from this base class.

In this example, the Car class serves as the parent class, containing common methods like Drive, Brake, and Honk. The SportsCar class is a child class that inherits from Car and gains access to these methods. Additionally, the SportsCar class has its own unique method of Turbocharge.

This inheritance allows us to reuse code efficiently. We don't need to redefine the common methods in the SportsCar class because it inherits them from the Car class. However, we can still add specialised functionality specific to sports cars, like turbocharging, in the child class. Additionally, if you want to change something inherited from the parent class, you can override it.

# Concept 2: Inheritance of Interfaces

In this lesson, we will be covering **interface inheritance**. But first, let's go through what an interface is. An **interface** is a blueprint for a contract that defines a set of **methods** and **properties** without providing any implementation details. It serves as a template for classes to follow, but how that method/property is implemented is up to them.

Another way to think about it is to treat the interface as a set of rules, and how everybody follows it is different as long as they're following it

While the inheritance of classes allows for the creation of families of related types, interfaces enable us to define behavior contracts that can be implemented by various unrelated classes.

Let's delve into interfaces using the analogy of vehicles. Imagine we have different types of vehicles: cars, bicycles, and boats. While these vehicles are unrelated in terms of their implementation, they share a common ability to move and stop.

```
using system;
// Interface: IMovable
3 references
interface IMovable
{
    3 references
    void Move();
    3 references
    void Stop();
}
// Classes implementing the IMovable interface
0 references
class Car : IMovable
{
    1 reference
    public void Move()
    {
        Console.WriteLine("Car is moving.");
    }
    1 reference
    public void Stop()
    {
        Console.WriteLine("Car has stopped.");
    }
}
0 references
class Bicycle : IMovable
{
    1 reference
    public void Move()
    {
        Console.WriteLine("Bicycle is moving.");
    }

    1 reference
    public void Stop()
    {
        Console.WriteLine("Bicycle has stopped.");
    }
}
0 references
class Boat : IMovable
{
    1 reference
    public void Move()
    {
        Console.WriteLine("Boat is sailing.");
    }

    1 reference
    public void Stop()
    {
        Console.WriteLine("Boat has anchored.");
    }
}
```


In this example, the IMovable interface defines methods Move() and Stop(), representing common behaviours shared among different types of vehicles. Each vehicle class implements these methods according to their respective functionalities. Using **interfaces** achieves two things:

**Abstraction:** By defining interfaces, we can hide the implementation details of each class and focus only on the common interactions provided by the interface. This promotes code clarity and security. This is because an interface by default is **abstract** and **public**.

**Multiple Inheritance:** While *C#* does not support multiple inheritance with classes, it does support it with interfaces. This means a class can implement multiple interfaces, enabling it to exhibit behaviours from different sources.

The example here demonstrates **polymorphism**, which will be discussed next.

## Concept 3: Polymorphism

What is **poly**? Poly means 'many'. Then what is **morphism**? Morphism means shape, form or structure. When you put them together, it means '**many forms**.'

**Polymorphism** refers to how a single method can behave differently depending on the type of object it is called upon. The last concept discussed, perfectly demonstrates this. But I will provide another analogy.

Imagine you have a baseball, an American football, and a basketball. Despite their differences, they all share a common characteristic – they can be thrown. However, each one is thrown distinctly. For instance, you wouldn't throw a football like a baseball pitch, nor would you toss a baseball with a football spin.

In this code, we have a base class Ball with an abstract method Throw(), representing the action of throwing the ball. Each derived class (Baseball, AmericanFootball, Basketball) provides its own implementation of the Throw() method, reflecting how each type of ball is thrown differently.

```
using system;
// Base class: Ball
3 references
abstract class Ball
{
    3 references
    public abstract void Throw();
}
// Derived class: Baseball
0 references
class Baseball : Ball
{
    1 reference
    public override void Throw()
    {
        Console.WriteLine("Throwing a baseball with a pitch.");
    }
}
// Derived class: AmericanFootball
0 references
class AmericanFootball : Ball
{
    1 reference
    public override void Throw()
    {
        Console.WriteLine("Throwing an American football with a spiral.");
    }
}
// Derived class: Basketball
0 references
class Basketball : Ball
{
    1 reference
    public override void Throw()
    {
        Console.WriteLine("Throwing a basketball with a one-handed shot.");
    }
}
```

# Concept 4: Delegates

We will be covering **delegates** in this lesson. What are delegates? By definition, it means entrusting a task or responsibility to another person.

In programming, delegates serve a similar purpose. They allow us to pass around **references** to **functions**, enabling us to choose different **implementations** (or **means**) for accomplishing a **task**, depending on the **context** or **requirements** of the program.

Imagine you're a student getting ready for school in the morning. The means of transportation from home to school can vary depending on different factors like time constraints, personal preferences, and available resources. But the ultimate goal is to get to school **(task)**.

The transport **(methods or functions)** available include: your Mum driving you because she's fast; your Dad driving you because he gives you a free snack stop; taking the school bus because you can socialise with friends; or walking to school to enjoy the scenery.

It's almost school time, so you asked your Mum **(delegate)** to drive you to arrive on time.

```
using System;
// Define a delegate named TransportationDelegate
1 reference
delegate void TransportationDelegate(string task);
0 references
class Program
{
    // Method to represent driving by mother
    1 reference
    static void DriveByMother(string task)
    {
        Console.WriteLine($"Mother is driving you {task}.");
    }
    // Method to represent driving by father
    0 references
    static void DriveByFather(string task)
    {
        Console.WriteLine($"Father is driving you {task}.");
    }
    // Method to represent taking the school bus
    0 references
    static void TakeSchoolBus(string task)
    {
        Console.WriteLine($"Taking the school bus {task}.");
    }
    // Method to represent walking
    0 references
    static void Walk(string task)
    {
        Console.WriteLine($"Walking {task}.");
    }
    0 references
    static void Main(string[] args)
    {
        // Instantiate the TransportationDelegate and assign the method DriveByMother to it
        TransportationDelegate transportationDelegate = DriveByMother;

        // Use the delegate to perform the task of getting to school
        transportationDelegate("to school");
    }
}
```

Dad

Mum


Bus

Walking

# Concept 5: Lambdas

In this lesson, we're learning about **Lambdas**. In programming, lambdas are a concise way to represent **anonymous functions** or **inline code blocks**. They allow us to define and use small, **one-time-use functions** without the need to **explicitly name them** or define them elsewhere in the code. Lambdas are particularly useful when we need to pass simple logic or behaviour as arguments to **methods**, such as **sorting**, **filtering**, or **mapping data**.

There's no simple way to explain this. Just know that lambda makes your code look **cleaner by reducing codes**. Another thing to note is **closure**, closure allows variables outside the lambda to be used inside the lambda, making it easy to provide additional data.

Ok! Let's move on to some examples. Imagine you're a student organising a study group with your classmates. Each classmate has their own way of studying, some prefer reading, others like to watch videos, and some prefer to solve practice problems. You want to filter out the classmates who prefer studying by solving practice problems so you can form a group specifically for that activity.

```
using System;
using System.Collections.Generic;
0 references
class Program
{
    // Method to check if a classmate prefers studying by solving practice problems
    1 reference
    static bool PrefersPracticeProblems(string studyPreference)
    {
        return studyPreference == "practice problems";
    }
    0 references
    static void Main(string[] args)
    {
        List<string> classmates = new List<string> { "Alice", "Bob", "Charlie", "Diana", "Eve" };
        Dictionary<string, string> studyPreferences = new Dictionary<string, string>
        {
            { "Alice", "reading" },
            { "Bob", "practice problems" },
            { "Charlie", "watching videos" },
            { "Diana", "practice problems" },
            { "Eve", "reading" }
        };
        // Filter classmates who prefer studying by solving practice problems using a named method
        List<string> practiceProblemLovers = new List<string>();
        foreach (var classmate in classmates)
        {
            if (PrefersPracticeProblems(studyPreferences[classmate]))
            {
                practiceProblemLovers.Add(classmate);
            }
        }
        // Output classmates who prefer studying by solving practice problems
        foreach (var classmate in practiceProblemLovers)
        {
            Console.WriteLine(classmate);
        }
    }
}
```

In the first approach, we define a separate **method** (PrefersPracticeProblems) to check each classmate's study preference. This leads to more **explicit and verbose code**, with the filtering logic **encapsulated** within the named method and each classmate's preference checked individually within a **loop**.

```
using System;
using System.Collections.Generic;
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        List<string> classmates = new List<string> { "Alice", "Bob", "Charlie", "Diana", "Eve" };
        Dictionary<string, string> studyPreferences = new Dictionary<string, string>
        {
            { "Alice", "reading" },
            { "Bob", "practice problems" },
            { "Charlie", "watching videos" },
            { "Diana", "practice problems" },
            { "Eve", "reading" }
        };
        // Filter classmates who prefer studying by solving practice problems using a lambda expression
        List<string> practiceProblemLovers = classmates.FindAll(classmate => studyPreferences[classmate] == "practice problems");
        // Output classmates who prefer studying by solving practice problems
        foreach (var classmate in practiceProblemLovers)
        {
            Console.WriteLine(classmate);
        }
    }
}
```

In contrast, the second approach utilises **lambdas**, where the filtering logic is expressed **directly within** the FindAll **method** call. This results in more **concise and expressive code**, as the logic for filtering classmates who prefer studying by solving practice problems is written **inline** without the need for a **separate named method**.

Lambdas offers a streamlined and readable implementation, enhancing code clarity and maintainability.

**Welcome back for Part 3!**

I just dyed my hair. Anyhow, here's the content that will be covered in this session:

01 OOP Four Key Concepts Summary

02 Data Structure: Arrays

03 Data Structure: Lists

04 Data Structure: Dictionaries

05 Writing Clean Code

06 UML Class Diagram

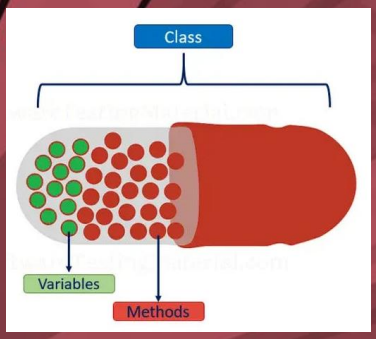
# Concept 1: Summary of OOP Four Key Concepts

Although the four key concepts of Object-oriented Programming (OOP) have been covered in Parts 1 and 2 of this series, it hasn't been explicitly stated yet. So, we will be summarising these four key concepts here.

Just to remind you all once again, they are abstraction, encapsulation, inheritance and polymorphism. Regardless of what languages you use in the future, these four key concepts can be applied to them.

Think of encapsulation as a capsule that contains all the ingredients needed to treat a condition.

In programming, encapsulation is like a protective capsule for data and methods, bundling them together within a class. Just as a capsule keeps its contents secure and prevents them from being tampered with, encapsulation hides the internal state of an object from the outside world, exposing only what is necessary through public methods. This ensures that the object's data is safe from unintended interference and misuse.

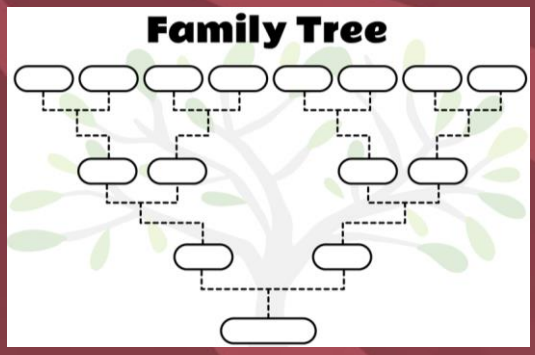



Abstraction is like using a TV remote. You press buttons to change channels or adjust the volume without needing to know how the TV processes your commands.

In OOP, abstraction involves creating simple, user-friendly interfaces for complex systems. It hides the detailed implementation and shows only the necessary features of an object. This way, you can interact with objects at a high level without getting bogged down by their internal complexities.

Inheritance is like a family tree where children inherit traits from their parents.

In OOP, inheritance allows a new class (child) to inherit properties and methods from an existing class (parent). This promotes code reuse and establishes a natural hierarchy. For example, a child might inherit their mother's eye colour and their father's height. In the context of OOP, inheritance works in a similar way.





Polymorphism is like a person who can play multiple roles – a teacher, a musician, and a chef, depending on the situation.

In OOP, polymorphism allows objects to be treated as instances of their parent class rather than their actual class. Imagine a base class called "Animal" that has a method called "makeSound." Different animals inherit from this base class, such as "Dog," "Cat," and "Bird." When you call the "makeSound" method on an animal object, each animal will respond in its way: the dog will bark, the cat will meow, and the bird will chirp.

# Concept 2: Data Structure - Arrays

In this lesson, we will be covering some commonly used data structures, starting with arrays. But before we talk about arrays, let's discuss what a data structure is. The easiest way to think of data structure is to think of it as a way to organise and store data. It helps you to access and manipulate data in your program efficiently.

Let's use an analogy of a bookshelf to explain arrays.

Imagine you have a bookshelf with a fixed number of slots and each slot can hold only one book. In addition, all the books in this bookshelf are stored in an ordered manner, from the first slot to the last slot.

When you have a bookshelf with exactly 10 slots, it will always only have 10 slots – you can't add or remove any slots. And each slot can only hold one book, not a vase or a toy. If you want book number 5, you go directly to book number 5 to retrieve it.

From this analogy, we can summarise a few key points. Arrays are fixed in size, have a uniform data type and ordered collection, and allow direct access.

```
using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Declaring an array of integers with 5 elements
        int[] numbers = new int[5];

        // Initializing the array elements
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Accessing elements
        int fifthNumber = numbers[4]; // 50

        // Printing accessed elements
        Console.WriteLine("Fifth number: " + fifthNumber);

        // Looping through the array
        Console.WriteLine("All numbers in the array:");
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine(numbers[i]);
        }
    }
}
```

To expand upon the analogy in programming terms, when you create an array of size 5, it will always have 5 slots and you can't change the size once it's been created. And each of these elements in the array is of the same data type, meaning, an integer array can only hold integers. When storing elements in an array, the elements are indexed in a specific order starting from [0]. If you want to directly access the 5$^{th}$ element of the array using its index, then you must directly refer to [4].

An array is a simple and efficient data structure for small collections with a known size that doesn't change, much like how a bookshelf allows you to access a fixed number of books.

# Concept 3: Data Structure - Lists

In this lesson, we will be covering another common data structure called lists. We will continue to use the bookshelf analogy for lists as well. Now, imagine you have a bookshelf, except it is expandable. Meaning you can add or remove slots as needed. This is the crucial difference between lists and arrays. Unlike arrays, lists can grow or shrink to accommodate more or fewer elements.

```
using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Declaring a list of integers
        List<int> numbers = new List<int>();

        // Adding elements to the list
        numbers.Add(10);
        numbers.Add(20);
        numbers.Add(30);

        // Inserting an element at a specific position
        numbers.Insert(1, 15); // List becomes [10, 15, 20, 30]

        // Removing elements from the list
        numbers.Remove(20); // List becomes [10, 15, 30]

        // Accessing elements
        int firstNumber = numbers[0]; // 10
        int secondNumber = numbers[1]; // 15

        // 10 + 15 = 25
        Console.WriteLine(firstNumber + secondNumber);

        // Looping through the list
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

For this expandable bookshelf, you can add more slots if you have more books or remove the slots if you have fewer books. This expandable bookshelf allows you to easily reorganise the books, add new ones or remove old ones without worrying about running out of space or having too much-unused space since the bookshelf adjusts accordingly.

Similar to the fixed bookshelf, the books are ordered collections. Allowing you to go directly to any slot to retrieve a specific book.

In summary, lists allow dynamic sizing, flexible data management, direct access and ordered collection.

In programming, a list can dynamically grow or shrink as you add or remove elements – you don't need to define its size like arrays. Lists allow you to easily add new elements, remove existing ones and modify elements without worrying about their size, the list will adjust accordingly. And just like arrays, elements can be directly accessed by their index. Except, a list will start from the beginning of the ordered collection and move down the list until it finds the desired element's index.

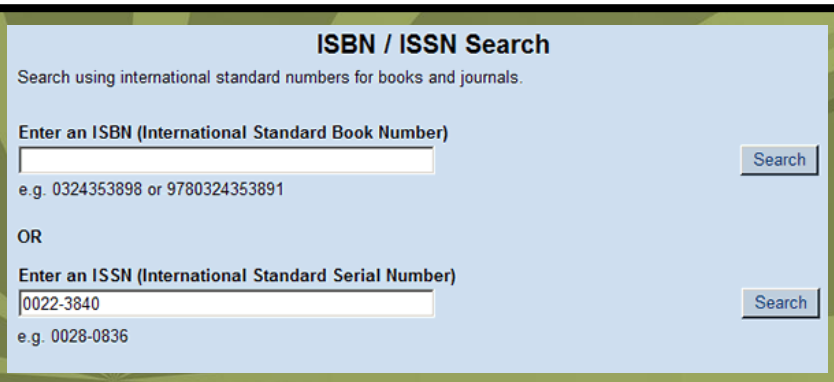
A list is a data structure that is ideal for collections of data that is not fixed and can change frequently. The drawback when compared to arrays, is that a list is a lot less efficient due to O(n) time complexity.

# Concept 4: Data Structure - Dictionaries

In this lesson, we will be discussing a data structure called **dictionaries**. We will still use the bookshelf analogy for this one as well, but instead of the bookshelf, I want you to imagine a library catalogue. This catalogue helps you find any book quickly by looking up its unique identifier like the ISBN. In programming, it's called a dictionary.

In the library catalogue, each book is associated with a **unique identifier (key)** and is linked to **specific information (value)**. Similar to an array, you can quickly find a book by looking up its unique identifier without searching through all the books like lists. Because each key is unique, there can be **no duplicate** entries in the catalogue either.

But just like lists, dictionaries have **dynamic sizing**, allowing you to add new books or remove old ones when necessary. This means that you won't have to worry about running out of space or having any unused space.




In programming, you don't need to define the **dictionary size** beforehand, as it can **dynamically grow or shrink** as you **add or remove key-value pairs**. Additionally, every **key** is **unique** in the dictionary, allowing you to **quickly retrieve** a value by its key.

A dictionary is a data structure that is ideal for when you need to associate unique keys with specific values and require efficiency. This is because the dictionary offers **O(1) average time complexity.**

So far, only 3 common data structures have been introduced, but there are many other data structures out there. It is important to be aware that **each data structure is useful for different purposes.**

```
using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Creating a dictionary with string keys (ISBN) and string values (Book Titles)
        Dictionary<string, string> libraryCatalog = new Dictionary<string, string>();

        // Adding books to the catalog
        libraryCatalog["978-0132350884"] = "Clean Code";
        libraryCatalog["978-0201485677"] = "Refactoring";
        libraryCatalog["978-0321127426"] = "Design Patterns";

        // Accessing a book by its ISBN
        if (libraryCatalog.ContainsKey("978-0132350884"))
        {
            string bookTitle = libraryCatalog["978-0132350884"];
            Console.WriteLine($"Found book: {bookTitle}");
        }

        // Removing a book from the catalog
        libraryCatalog.Remove("978-0321127426");

        // Displaying all books in the catalog
        foreach (var entry in libraryCatalog)
        {
            Console.WriteLine($"ISBN: {entry.Key}, Title: {entry.Value}");
        }
    }
}
```

# Concept 5: Writing Clean Codes

In this lesson, we will be discussing how to **write clean codes.**

Now imagine that you're in a library, surrounded by books neatly organised on shelves. Each book represents a piece of code, and the library resembles your program. Writing clean code is like maintaining an organised library.

To write clean codes, there are a few key points you need to keep in mind to ensure that it is well-structured, readable, maintainable and can be easily expanded upon by other developers in the future.

**The syntax is like grammar and spelling.** Similar to how English has grammatical rules to convey its message clearly, clean code needs to follow syntax rules. In C#, naming conventions typically use PascalCase for class and method names. Whereas camelCase if used for local variables. Furthermore, knowing the correct usage of keywords, operators and punctuation marks is similar to following the grammar rules of a language.

**Coding conventions are library rules.** Like how libraries have rules in place to organise books effectively, coding conventions serve as the rules for structuring code consistently and coherently. There are three core rules to follow, namely, **naming conventions**, **formatting standards**, and **commenting practices**.

Naming conventions just ensure classes, methods and variables are named relating to what it does. Formatting standards involve using correct indentation, spacing and placement of braces to ensure uniformity. And commenting is used to provide an understanding of that code.

**Polished implementation of code is like a well-written book.** If your program follows the syntax rules and coding conventions, then it is like reading a well-written book for other people.

**Error-free code is like a book without missing pages or printing errors.** Similar to how published books can have proofreading errors, codes must be tested for errors. And if errors can happen in your code, ensure exception handling is included.

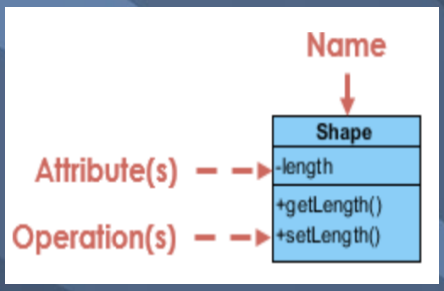
```
using System;

// This class represents a simple book in our library
2 references
public class Book
{
    // These are properties of the Book class
    2 references
    public string Title { get; set; }    // Represents the title of the book in PascalCase
    2 references
    public string Author { get; set; }   // Represents the author of the book in PascalCase

    // This method prints out information about the book
    1 reference
    public void PrintInfo()  // Method name follows PascalCase convention
    {
        // Printing out the title and author of the book
        Console.WriteLine($"Title: {Title}");
        Console.WriteLine($"Author: {Author}");
    }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Creating an instance of the Book class
        Book myBook = new Book();

        // Handling potential errors when setting properties
        try
        {
            // Prompting the user to enter the title of the book
            Console.Write("Enter the title of the book: ");
            string title = Console.ReadLine(); // Local variable follows camelCase convention

            // Validating the title input
            if (string.IsNullOrWhiteSpace(title))
            {
                throw new ArgumentException("Title cannot be empty or whitespace.");
            }

            // Setting the title property of the book
            myBook.Title = title;

            // Prompting the user to enter the author of the book
            Console.Write("Enter the author of the book: ");
            string author = Console.ReadLine(); // Local variable follows camelCase convention

            // Validating the author input
            if (string.IsNullOrWhiteSpace(author))
            {
                throw new ArgumentException("Author cannot be empty or whitespace.");
            }

            // Setting the author property of the book
            myBook.Author = author;

            // Printing out the information about the book
            myBook.PrintInfo();
        }
        catch (ArgumentException ex)
        {
            // Handling the exception and printing the error message
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

For some more examples of writing clean codes. You can look through all the other previous concept code examples in Parts 1, 2 and 3 of this comic series. They all follow the key points discussed in this article.

# Concept 6: UML Class Diagram

In this lesson, we will be discussing **UML Class Diagram**, specifically LucidChart style for C# programs.

**Unified Modelling Language**, or UML, was set up as a standardised model to describe OOP software architecture. They are widely used in software engineering to visualise the design of a system before implementation.

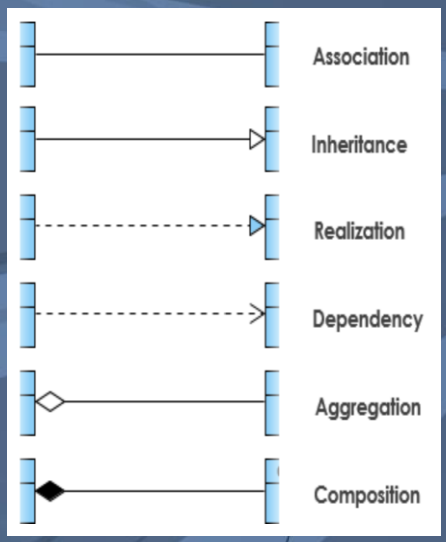



**Upper section:** Contains the name of the class.

**Middle section:** Contains the attributes and fields of the class.

**Bottom section:** Includes class methods and properties.

**Member access modifiers:** Public (+), Private (-), Protected (#), Package (~), Derived (/), Static (Underlined)




**Association:** Represents a relationship between two classes, indicating that instances of one class are connected to instances of another class.

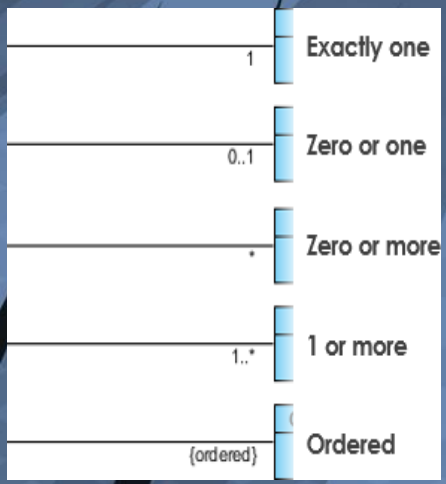
**Inheritance:** Depicts an "is-a" relationship between classes, where one class (subclass) inherits attributes and methods from another class (superclass).

**Realisation:** Represents the implementation of an interface by a class, indicating that the class provides the behaviour specified by the interface.

**Dependency:** Indicates that one class relies on another class, typically through method parameters, return types, or temporary associations.

**Aggregation:** Illustrates a whole-part relationship between classes, where one class (composite) contains or owns another class (component), typically representing a "has-a" relationship.

**Composition:** Depicts a strong form of aggregation where the lifetime of the component is managed by the composite class, representing a more exclusive ownership relationship.




**Multiplicity:** Multiplicity, also called cardinality, indicates the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees.

# Welcome to Part 4!

Part 4 will be all about applying most of the concepts we learned to build a simple console C# application.

**The console application we're building is called the 'Video Game Rental System', which keeps track of multiple users who borrow and return from a list of video games. There will be three parts to building this application.**

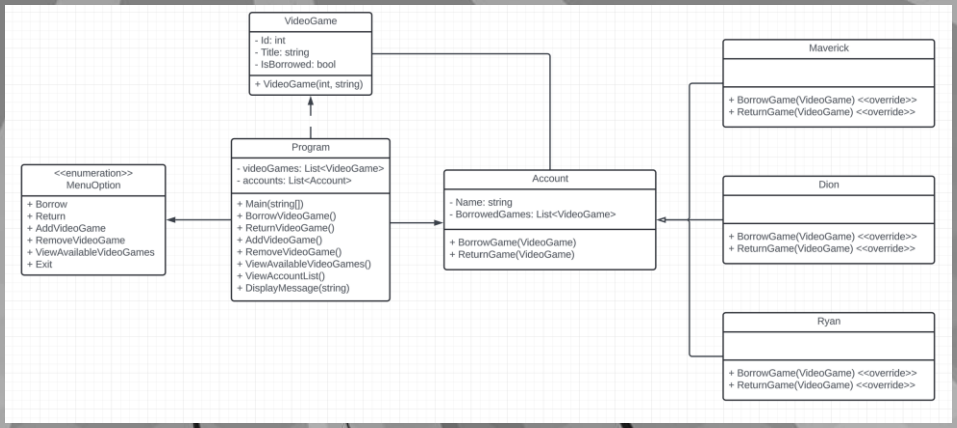
**Iteration 1: Borrowing, Returning and Account**
**Iteration 2: Remove and Add Video Games**
**Iteration 3: Multiple Accounts**

**Try to follow the provided instructions in each iteration and build the system yourself. But examples are provided if help is necessary. Remember to test as you go!**

**Here's the provided final UML class diagram based on LucidChart that you will try to implement:**

**Here's the list of concepts covered in this tutorial:**

- Class and Objects
- Methods
- Fields and Properties
- Constructor
- Control Flow
- Inheritance of Classes
- Polymorphism
- Data Structure: Lists
- Writing Clean Code
- UML Class Diagram



# Iteration 1: Borrowing, Returning and Account

**Step 1:** Setup project in your preferred IDE (VisualStudioCode is chosen for this) go into the terminal and cd to your chosen directory.
- Now set up the project in the terminal with: "dotnet new console", then "dotnet restore".

**Step 2:** Name the project "VideoGameRentalSystem" and set up the file structure with the following files:
- Program.cs
- Account.cs
- VideoGame.cs

**Step 3:** Initialise sample data for video games and accounts.
- Create a list of 'VideoGame' objects called 'videoGames' with sample data.
- Create a single 'Account' object called 'maverick' (whatever you like) to represent the user for later testing.

**Step 4:** Implement Main Method and Menu Loop.
- Implement the Main method to serve as the program's entry point.
- Implement a loop in the Main method to display the main menu and handle user input.
- Include error handling for invalid user input.

**Step 5:** Implement Borrowing and Returning functionalities to the system.
- Implement the 'BorrowVideoGame()' method to allow the user to borrow a video game.
- Implement the 'ReturnVideoGame()' method to allow the user to return a borrowed video game.
- Implement the 'ViewAvailableVideoGames()' method to display available video games.

**Step 6:** Testing
- Run the program and test all the functionalities work as intended.
- Check that error handling as intended.
- Fix any errors or add any extra bit of code for improved user experience.

**Check List:**

1. Display menu options with borrow, return, view available video games, and exit.
2. Borrowing video games function works.
3. Returning video games function works.
4. View available video games function works.
5. Error handling is included at every input stage.

# Video Game Sharing System (Iteration 1)

## All Files

Code\VideoGame.cs

Code\Account.cs

Code\Program.cs

**VideoGame.cs**

```
class VideoGame
{
    public int Id { get; private set; }
    public string Title { get; private set; }
    public bool IsBorrowed { get; set; }

    public VideoGame(int id, string title)
    {
        Id = id;
        Title = title;
        IsBorrowed = false;
    }
}
```

**Account.cs**

```
class Account
{
    public string Name { get; private set; }
    public List<VideoGame> BorrowedGames { get; private set; }

    public Account(string name)
    {
        Name = name;
        BorrowedGames = new List<VideoGame>();
    }

    public void BorrowGame(VideoGame game)
    {
        if (!game.IsBorrowed)
        {
            game.IsBorrowed = true;
            BorrowedGames.Add(game);
        }
    }

    public void ReturnGame(VideoGame game)
    {
        if (BorrowedGames.Contains(game))
        {
            game.IsBorrowed = false;
            BorrowedGames.Remove(game);
        }
    }
}
```

**Program.cs**

```
// Program.cs
using System;
```

```
using System.Collections.Generic;
using System.Linq;

class Program
{
    public enum MenuOptions
    {
        Borrow = 1,
        Return,
        ViewAvailableVideoGames,
        Exit
    }

    static List<VideoGame> videoGames = new List<VideoGame>
    {
        new VideoGame(1, "The Legend of Zelda: Breath of the Wild"),
        new VideoGame(2, "Super Mario Odyssey"),
        new VideoGame(3, "Red Dead Redemption 2")
    };

    static Account maverick = new Account("Maverick");

    static void Main(string[] args)
    {
        while (true)
        {
            Console.WriteLine("**********************");
            Console.WriteLine("Select an option");
            foreach (var option in Enum.GetValues(typeof(MenuOptions)))
            {
                Console.WriteLine($"{(int)option}. {option}");
            }
            Console.WriteLine("**********************");
            Console.Write("Enter chosen option: ");

            if (Enum.TryParse<MenuOptions>(Console.ReadLine(), out MenuOptions
selectedOption))
            {
                switch (selectedOption)
                {
                    case MenuOptions.Borrow:
                        BorrowVideoGame();
                        break;
                    case MenuOptions.Return:
                        ReturnVideoGame();
                        break;
                    case MenuOptions.ViewAvailableVideoGames:
                        ViewAvailableVideoGames();
                        break;
                    case MenuOptions.Exit:
                        Environment.Exit(0);
                        break;
                    default:
                        DisplayMessage("Invalid option, try again.");
                        break;
                }
            }
            else
            {
                DisplayMessage("Invalid input, try again.");
```

```
            }
        }
    }

    static void BorrowVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter the ID of the game you want to borrow:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && !game.IsBorrowed)
            {
                maverick.BorrowGame(game);
                DisplayMessage($"{game.Title} has been borrowed by {maverick.Name}.");
            }
            else
            {
                DisplayMessage("Game not available or invalid ID.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void ReturnVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter the ID of the game you want to return:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && game.IsBorrowed && maverick.BorrowedGames.Contains(game))
            {
                maverick.ReturnGame(game);
                DisplayMessage($"{game.Title} has been returned by {maverick.Name}.");
            }
            else
            {
                DisplayMessage("Invalid ID or the game was not borrowed by you.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void ViewAvailableVideoGames()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Available Video Games:");
        foreach (var game in videoGames.Where(g => !g.IsBorrowed))
        {
            Console.WriteLine($"{game.Id} - {game.Title}");
        }
```

```
122         Console.WriteLine("**********************");
123     }
124 
125     static void DisplayMessage(string message)
126     {
127         Console.WriteLine("**********************");
128         Console.WriteLine(message);
129         Console.WriteLine("**********************");
130     }
131 }
```

# Iteration 2: Remove/Add Video Games

**Step 1:** Extend the 'MenuOptions' enum.
- Add an option for adding videogame, called 'AddVideoGame'
- Add an option for removing videogame, called 'RemoveVideoGame'

**Step 2:** Implement the add and remove functionalities to the system
- Implement the 'AddVideoGame' method to add new video games to the list.
- Implement the 'RemoveVideoGame' method to remove old video games from the list.

**Step 3:** Update the Menu Loop
- Update the switch statement to include the new menu option for 'AddVideoGame()'
- Update the switch statement to include the new menu option for 'RemoveVideoGame()'

**Step 4:** Testing
- Run the program and test all the functionalities work as intended.
- Check that error handling as intended.
- Fix any errors or add any extra bit of code for improved user experience.

You might be wondering, why am I writing codes in iterations? There are a few reasons for this, but let's use an analogy of building a house for example.

You can't build a house in one day, and the same goes with programming. When you program, you want to split it up into smaller, manageable chunks to focus on, much like focusing on one part of the house at a time. This would allow the opportunity for continuous testing at each iteration to fix any bugs or issues encountered. This allows you to fix any issues before moving on to the next step.

Another way to think about it is to imagine you trying to complete building your desired program in one go. This would often result in many bugs, and when you try to fix one bug, more bugs may appear, making the program unusable anymore. So it's important to address any issues early on.

**Check List:**
1. Display menu options now also include adding and removing video game options.
2. Adding video games function works.
3. Removing video games function works.
4. Error handling in the new methods.

# Video Game Sharing System (Iteration 2)

## All Files

Code\VideoGame.cs

Code\Account.cs

Code\Program.cs

**VideoGame.cs**

```
class VideoGame
{
    public int Id { get; private set; }
    public string Title { get; private set; }
    public bool IsBorrowed { get; set; }

    public VideoGame(int id, string title)
    {
        Id = id;
        Title = title;
        IsBorrowed = false;
    }
}
```

**Account.cs**

```
class Account
{
    public string Name { get; private set; }
    public List<VideoGame> BorrowedGames { get; private set; }

    public Account(string name)
    {
        Name = name;
        BorrowedGames = new List<VideoGame>();
    }

    public void BorrowGame(VideoGame game)
    {
        if (!game.IsBorrowed)
        {
            game.IsBorrowed = true;
            BorrowedGames.Add(game);
        }
    }

    public void ReturnGame(VideoGame game)
    {
        if (BorrowedGames.Contains(game))
        {
            game.IsBorrowed = false;
            BorrowedGames.Remove(game);
        }
    }
}
```

**Program.cs**

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;

class Program
{
    public enum MenuOptions
    {
        Borrow = 1,
        Return,
        ViewAvailableVideoGames,
        AddVideoGame,
        RemoveVideoGame,
        Exit
    }

    static List<VideoGame> videoGames = new List<VideoGame>
    {
        new VideoGame(1, "The Legend of Zelda: Breath of the Wild"),
        new VideoGame(2, "Super Mario Odyssey"),
        new VideoGame(3, "Red Dead Redemption 2")
    };

    static Account maverick = new Account("Maverick");

    static void Main(string[] args)
    {
        while (true)
        {
            Console.WriteLine("**********************");
            Console.WriteLine("Select an option");
            foreach (var option in Enum.GetValues(typeof(MenuOptions)))
            {
                Console.WriteLine($"{(int)option}. {option}");
            }
            Console.WriteLine("**********************");
            Console.Write("Enter chosen option: ");

            if (Enum.TryParse<MenuOptions>(Console.ReadLine(), out MenuOptions
selectedOption))
            {
                switch (selectedOption)
                {
                    case MenuOptions.Borrow:
                        BorrowVideoGame();
                        break;
                    case MenuOptions.Return:
                        ReturnVideoGame();
                        break;
                    case MenuOptions.ViewAvailableVideoGames:
                        ViewAvailableVideoGames();
                        break;
                    case MenuOptions.AddVideoGame:
                        AddVideoGame();
                        break;
                    case MenuOptions.RemoveVideoGame:
                        RemoveVideoGame();
                        break;
                    case MenuOptions.Exit:
                        Environment.Exit(0);
                        break;
                    default:
```

```
                        DisplayMessage("Invalid option, try again.");
                        break;
                }
            }
            else
            {
                DisplayMessage("Invalid input, try again.");
            }
        }
    }

    static void BorrowVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter the ID of the game you want to borrow:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && !game.IsBorrowed)
            {
                maverick.BorrowGame(game);
                DisplayMessage($"{game.Title} has been borrowed by {maverick.Name}.");
            }
            else
            {
                DisplayMessage("Game not available or invalid ID.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void ReturnVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter the ID of the game you want to return:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && game.IsBorrowed && maverick.BorrowedGames.Contains(game))
            {
                maverick.ReturnGame(game);
                DisplayMessage($"{game.Title} has been returned by {maverick.Name}.");
            }
            else
            {
                DisplayMessage("Invalid ID or the game was not borrowed by you.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void ViewAvailableVideoGames()
```

```
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Available Video Games:");
        foreach (var game in videoGames.Where(g => !g.IsBorrowed))
        {
            Console.WriteLine($"{game.Id} - {game.Title}");
        }
        Console.WriteLine("**********************");
    }

    static void AddVideoGame()
    {
        Console.WriteLine("Enter the title of the new video game:");
        string title = Console.ReadLine();

        int newId = videoGames.Count > 0 ? videoGames.Max(g => g.Id) + 1 : 1;

        videoGames.Add(new VideoGame(newId, title));
        Console.WriteLine($"Video game '{title}' added with ID {newId}.");
    }

    static void RemoveVideoGame()
    {
        Console.WriteLine("Enter the ID of the video game to remove:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            var gameToRemove = videoGames.FirstOrDefault(g => g.Id == gameId);
            if (gameToRemove != null)
            {
                videoGames.Remove(gameToRemove);
                Console.WriteLine($"Video game '{gameToRemove.Title}' with ID {gameId}
removed.");
            }
            else
            {
                DisplayMessage("Video game not found.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void DisplayMessage(string message)
    {
        Console.WriteLine("**********************");
        Console.WriteLine(message);
        Console.WriteLine("**********************");
    }
}
```

# Iteration 3: Multiple Accounts

**Step 1:** Extend the 'MenuOptions' enum.
- Add an option for viewing a list of accounts called 'ViewAccountList'

**Step 2:** Implement the view account functionality to the system
- Implement the 'ViewAccountList' method to view accounts and their borrowed video games.

**Step 3:** Update the Menu Loop
- Update the switch statement to include the new menu option for 'ViewAccountList()'

**Step 4:** Apply inheritance and polymorphism to Account class.
- Turn the iteration 2 Account class into an abstract parent class with abstract methods – 'BorrowGame' and 'ReturnGame'.
- Add three derived classes with whatever name preferred and implement override abstract methods.

**Step 5:** Modify 'BorrowVideoGame' and 'ReturnVideoGame'.
- Modify 'BorrowVideoGame to prompt users to enter their name to select their account.
- Modify 'ReturnVideoGame to prompt users to enter their name to select their account.

**Step 6:** Add error handling and testing
- Add errorhandling to user prompts as required.
- Do further testing for any other errors and bugs to fix them.

This will be the final iteration of this simple console application. Don't forget to follow the rules for writing clean codes established in Part 3 of this series! The provided examples of each iteration followed all the rules established in that article but didn't include any comments. That will be up to you to provide the necessary comments for anybody else reading your code. This is especially useful if you plan to add any future functions to this system. Maybe you can add a user registration option or even a password to log in to that account, you can also add dates of video games borrowed or returned and save it in a log. The possibilities are up to you.

**Check List:**

1. Display menu options now include an option to view a list of accounts with their borrowed video games.

2. Borrowing the game and returning the game now requires selecting the appropriate user.

3. Error handling for user's name input stage.

# Video Game Sharing System (Iteration 3)

## All Files

Code\VideoGame.cs

Code\Account.cs

Code\Program.cs

**Code\VideoGame.cs**

```
public class VideoGame
{
    public int Id { get; private set; }
    public string Title { get; private set; }
    public bool IsBorrowed { get; set; }

    public VideoGame(int id, string title)
    {
        Id   id;
        Title   title;
        IsBorrowed   false;
    }
}
```

**Code\Account.cs**

```
using System.Collections.Generic;

public abstract class Account
{
    public string Name { get; private set; }
    public List<VideoGame> BorrowedGames { get; private set; }

    protected Account(string name)
    (
        Name   name;
        BorrowedGames   new List<VideoGame>();
    }

    public abstract void BorrowGame(VideoGame game);
    public abstract void ReturnGame(VideoGame game);
}

public class Maverick : Account
{
    public Maverick() : base("Maverick”) { }

    public override void BorrowGame(VideoGame game)
    (
        if (!game.IsBorrowed)
        {
            game.IsBorrowed   true;
            BorrowedGames.Add(game);
        }
    }

    public override void ReturnGame(VideoGame game)
    {
        if (BorrowedGames.Contains(game))
        {
```

```
                game.IsBorrowed    false;
36              BorrowedGames.Remove(game);
37          }
38      }
39  }
40
41  pub1ic class Ryan : Account
42  (
43      public Ryan() : base(”Ryan") { }
44
45      public override void BorrowGame(VideoGame game)
46      {
47          if (!game.IsBorrowed)
48          {
49              game.IsBorrowed     true;
50              BorrowedGames.Add(game);
51          }
52      }
53
54      public override void ReturnGame(VideoGame game)
55      (
56          if (BorrowedGames.Contains(game))
57          {
58              game.IsBorrowed     false;
59              BorrowedGames.Remove(game);
60          }
61      }
62  }
63
64  pub11c class D1on : Account
65  (
66      public Dion() : base(”Dion") { }
67
68      public override void BorrowGame(VideoGame game)
69      {
70          if (!game.IsBorrowed)
71          {
72              game.IsBorrowed     true;
73              BorrowedGames.Add(game);
74          }
75      }
76
77      public override void ReturnGame(VideoGame game)
78      {
79          if (BorrowedGames.Contains(game))
80          {
81              game.IsBorrowed     false;
82              BorrowedGames.Remove(game);
83          }
84      }
85  }
```

**Code\Program.cs**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  class Program
6  {
```

```
    public enum MenuOptions
    {
        Borrow = 1,
        Return,
        AddVideoGame,
        RemoveVideoGame,
        ViewAvailableVideoGames,
        ViewAccountList,
        Exit
    }

    static List<VideoGame> videoGames = new List<VideoGame>
    {
        new VideoGame(1, "The Legend of Zelda: Breath of the Wild"),
        new VideoGame(2, "Super Mario Odyssey"),
        new VideoGame(3, "Red Dead Redemption 2"),
        new VideoGame(4, "The Witcher 3: Wild Hunt"),
        new VideoGame(5, "God of War"),
        new VideoGame(6, "Horizon Zero Dawn"),
        new VideoGame(7, "Spider-Man"),
        new VideoGame(8, "Cyberpunk 2077"),
        new VideoGame(9, "Assassin's Creed Valhalla"),
        new VideoGame(10, "Ghost of Tsushima")
    };

    static List<Account> accounts = new List<Account>
    {
        new Maverick(),
        new Ryan(),
        new Dion()
    };

    static void Main(string[] args)
    {
        while (true)
        {
            Console.WriteLine("**********************");
            Console.WriteLine("Select an option");
            foreach (var option in Enum.GetValues(typeof(MenuOptions)))
            {
                Console.WriteLine($"{(int)option}. {option}");
            }
            Console.WriteLine("**********************");
            Console.Write("Enter chosen option: ");

            if (Enum.TryParse<MenuOptions>(Console.ReadLine(), out MenuOptions
selectedOption))
            {
                switch (selectedOption)
                {
                    case MenuOptions.Borrow:
                        BorrowVideoGame();
                        break;
                    case MenuOptions.Return:
                        ReturnVideoGame();
                        break;
                    case MenuOptions.AddVideoGame:
                        AddVideoGame();
                        break;
                    case MenuOptions.RemoveVideoGame:
```

```
                        RemoveVideoGame();
                        break;
                    case MenuOptions.ViewAvailableVideoGames:
                        ViewAvailableVideoGames();
                        break;
                    case MenuOptions.ViewAccountList:
                        ViewAccountList();
                        break;
                    case MenuOptions.Exit:
                        Environment.Exit(0);
                        break;
                    default:
                        DisplayMessage("Invalid option, try again.");
                        break;
                }
            }
            else
            {
                DisplayMessage("Invalid input, try again.");
            }
        }
    }

    static void BorrowVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter your name (Maverick, Ryan, Dion):");
        string name = Console.ReadLine();
        Account account = accounts.FirstOrDefault(a => a.Name.Equals(name,
StringComparison.OrdinalIgnoreCase));

        if (account == null)
        {
            DisplayMessage("Account not found.");
            return;
        }

        ViewAvailableVideoGames();

        Console.WriteLine("Enter the ID of the game you want to borrow:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && !game.IsBorrowed)
            {
                account.BorrowGame(game);
                DisplayMessage($"{game.Title} has been borrowed by {account.Name}.");
            }
            else
            {
                DisplayMessage("Game not available or invalid ID.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

```

```
    static void ReturnVideoGame()
    {
        Console.WriteLine("**********************");
        Console.WriteLine("Enter your name (Maverick, Ryan, Dion):");
        string name = Console.ReadLine();
        Account account = accounts.FirstOrDefault(a => a.Name.Equals(name,
StringComparison.OrdinalIgnoreCase));

        if (account == null)
        {
            DisplayMessage("Account not found.");
            return;
        }

        Console.WriteLine("Enter the ID of the game you want to return:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            VideoGame game = videoGames.FirstOrDefault(g => g.Id == gameId);

            if (game != null && game.IsBorrowed && account.BorrowedGames.Contains(game))
            {
                account.ReturnGame(game);
                DisplayMessage($"{game.Title} has been returned by {account.Name}.");
            }
            else
            {
                DisplayMessage("Invalid ID or the game was not borrowed by you.");
            }
        }
        else
        {
            DisplayMessage("Invalid ID.");
        }
    }

    static void AddVideoGame()
    {
        Console.WriteLine("Enter the title of the new video game:");
        string title = Console.ReadLine();


        int newId = videoGames.Count > 0 ? videoGames.Max(g => g.Id) + 1 : 1;


        videoGames.Add(new VideoGame(newId, title));
        Console.WriteLine($"Video game '{title}' added with ID {newId}.");
    }

    static void RemoveVideoGame()
    {
        Console.WriteLine("Enter the ID of the video game to remove:");
        if (int.TryParse(Console.ReadLine(), out int gameId))
        {
            var gameToRemove = videoGames.FirstOrDefault(g => g.Id == gameId);
            if (gameToRemove != null)
            {
                videoGames.Remove(gameToRemove);
                Console.WriteLine($"Video game '{gameToRemove.Title}' with ID {gameId}
removed.");
            }
```

```
183                 else
184                 {
185                     DisplayMessage(”Video game not found.");
186                 }
187             }
188             else
189             {
190                 DisplayMessage(“Invalid ID.");
191             }
192         }
193
194         stat1c vold V1ewAva11ab1eVideoGames()
195         (
196             Console.WriteLine("**********************”);
197             Console.WriteLine("Available Video Games:”);
198             foreach (var game in videoGames.Where(g => !g.IsBorrowed))
199             {
200               Console.WriteLine($”{game.Id}       {game.Title}");
201             }
202             Console.WriteLine("**********************”);
203         }
204
205         static void ViewAccountList()
206         {
207             Console.WriteLine("**********************”);
208             foreach (var account in accounts)
209             {
210                 Console.WriteLine($”Account: {account.Name}");
211                 Console.WriteLine(”Borrowed Games:”);
212                 foreach (var game in account.BorrowedGames)
213
214                     Console.WriteLine($”{game.Id}
                    }                                          {game.Title}”
                        );
215             }
216                 Console.WriteLine();
217         }
218             Console.WriteLine("**********************”);
219
220
221         static void DisplayMessage(string message)
222         (
223             Console.WriteLine("**********************”);
224         }   Console.WriteLine(message);
225             Console.WriteLine("**********************”);
226
227 }
```