C# Course

▼ Day 1: Introduction to C# and .NET

▼ Introduction to C# and .NET

- What is C#?
  - Brief History:
    - C# was developed by Microsoft in 2000
    - Part of the .NET initiative, designed to be a simple, modern, and object-oriented language.
  - Importance of C# in Software Development:
    - Widely used for enterprise applications, web development, game development (Unity), and more.
    - Continues to evolve with regular updates and new features, reflecting the needs of modern software development.
- NET Framework vs .NET Core vs .NET 5+
  - **.NET Framework:**
    - Released in 2002, it was Windows-only, focusing on desktop applications, web applications (ASP.NET), and services.
  - **.NET Core:**
    - Introduced in 2016 as a cross-platform, open-source framework.
    - Emphasis on modularity, lightweight, and cloud-first architecture.
    - Key features: Cross-platform support (Windows, Linux, macOS), performance improvements, and microservices support.

- **.NET 5, 6, 7, and 8:**
  - The unification of .NET Framework and .NET Core.
  - New features with each version, focusing on performance, security, and productivity.
  - .NET 8: Latest version at the time of your course, with improvements in cloud-native development, ARM64 support, and more.

- **.NET Architecture Overview**
  - **Common Language Runtime (CLR):**
    - The heart of the .NET framework, responsible for executing managed code.
    - Memory management, exception handling, and security checks.
    - Garbage Collector
  - **Just-In-Time (JIT) Compilation:**
    - Converts IL (Intermediate Language) code into native machine code.
  - **Intermediate Language (IL) Code:**
    - Overview of IL as the CPU-independent set of instructions.
    - How IL allows for cross-language interoperability within the .NET ecosystem.
  - **Base Class Library (BCL):**
    - Provides a set of standard classes that serve as the foundation for .NET applications.
    - Discuss commonly used namespaces and classes within BCL.

▼ Understanding the C# Compiler

- **C# Compiler Evolution**
  - **Early Compilers:**
    - How C# compilers have evolved from the first versions to the present.
    - Introduction to Roslyn in 2014: A fully managed and open-source C# compiler.
    - The importance of Roslyn in modern development, providing rich code analysis APIs.

- **Compilation Process**
  - **Source Code to IL:**
    - Step-by-step explanation of how C# source code is transformed into IL code.
    - The role of the C# compiler in checking syntax, semantic errors, and optimization.
  - IL to Machine Code via JIT:
    - How the CLR uses JIT to convert IL to native code at runtime.
    - Advantages of JIT: Just-in-time optimizations, platform independence.
    - Discussion on Ahead-Of-Time (AOT) compilation, where applicable.
  - Understanding CLR, JIT, and IL:
    - Detailed discussion on the roles these components play in executing a .NET application.
    - Introduction to tools like ILDASM (IL Disassembler) for inspecting IL code.

▼ Deep Dive into Primitive Data Types in C#

- Primitive Data Types Overview
  - Value Types vs Reference Types
    - **▼ Value Types:**
      - **Primitive Value Types**
        1. **Integral Types**
           - `byte` : 8-bit unsigned integer (`0` to `255`)
           - `sbyte` : 8-bit signed integer (`128` to `127`)
           - `short` : 16-bit signed integer (`32,768` to `32,767`)
           - `ushort` : 16-bit unsigned integer (`0` to `65,535`)
           - `int` : 32-bit signed integer (`2,147,483,648` to `2,147,483,647`)
           - `uint` : 32-bit unsigned integer (`0` to `4,294,967,295`)
           - `long` : 64-bit signed integer (`9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`)
           - `ulong` : 64-bit unsigned integer (`0` to `18,446,744,073,709,551,615`)
           - `nint` : Platform-specific signed integer (size depends on the platform: 32-bit on 32-bit systems, 64-bit on 64-bit systems)
           - `nuint` : Platform-specific unsigned integer (size depends on the platform: 32-bit on 32-bit systems, 64-bit on 64-bit systems)
        2. **Floating-Point Types**
           - `float` : 32-bit single-precision floating-point (`±1.5 × 10^-45` to `±3.4 × 10^38` , approximately 7 digits of precision)

- `double` : 64-bit double-precision floating-point (`±5.0 × 10^-324` to `±1.7 × 10^308`, approximately 15-16 digits of precision)
- `decimal` : 128-bit precise decimal type with a range of (`±1.0 × 10^-28` to `±7.9228 × 10^28`, 28-29 significant digits), used for financial and monetary calculations

3. **Boolean Type**
    - `bool` : Represents a boolean value (`true` or `false`)
4. **Character Type**
    - `char` : 16-bit Unicode character (`U+0000` to `U+FFFF`)
5. **Special Value Types**
    - `DateTime` : Represents date and time values
    - `TimeSpan` : Represents a time interval
    - `Guid` : Represents a globally unique identifier (GUID)

- Value types are stored directly on the stack.
    - In a nutshell, the stack is a collection of memory blocks that is used to keep track of the current thread. The stack is used to obtain basic property data. Because the stack is only used for basic data, accessing it is relatively quick.
    - **Value Type Allocated on stack**
        - **When you create a variable that is a value type, a part of memory called Stack is allocated for that.**
    - **Memory allocation done automatically**
        - Allocating a stack for a variable is done automatically. So we don’t have to worry.
    - **Immediately removed when out of the scope**
        - If the a variable goes out of the scope, it will immediately removed by runtime or CLR.

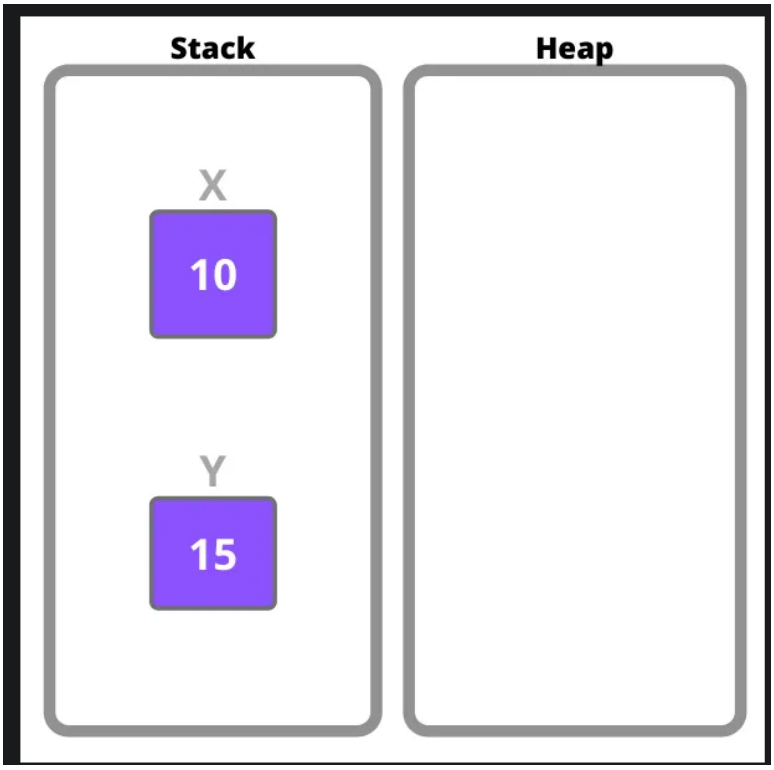
- **Value Types in Action**
  - `int x = 10;`
- Then, I am going to create another variable called y and copying x to y
  -

```
int y = x;
Console.WriteLine(y); //output => 10
```

- Now I am going to assign 15 to y.
  -

```
y = 15;
Console.WriteLine(y); //output => 15
```

- Finally, let’s see what happened to x
  -



- **Reference Types**
  - Definition
    - Reference types are types that store references (pointers) to the actual data, which is stored on the heap. Unlike value types, which hold their data directly, reference types only contain a reference to the location in memory where the data is stored.
  - **Memory Allocation:**
    - When a reference type is instantiated, memory is allocated on the heap for the data, and a reference to that memory location is stored on the stack.
    - If a reference type variable is assigned to another variable, only the reference is copied, not the data itself.

This means that both variables point to the same memory location on the heap.

- Types of Reference Types in .NET
  - **Class**
    - **What It Is:** A class defines a blueprint for creating objects. Each object (instance of a class) is stored on the heap, and the reference to the object is stored on the stack.
    - **Example**
      -

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

Person person1 = new Person();
```

    - **Memory Management:** The object `person1` is allocated on the heap, and `person1` holds the reference to that memory location on the stack.
  - **Array**
    - **What It Is:** Arrays are collections of elements of the same type. Arrays are reference types even if they contain value types.
    - Example

```
int[] numbers = new int[5];
```

    - **Memory Management:** The array `numbers` is stored on the heap, and the reference to the array is held in

the `numbers` variable on the stack.

- **String**
  - **What It Is:** In .NET, strings are immutable reference types. Any modification to a string creates a new string object on the heap.
  - Example

```
string greeting = "Hello, World!";
```

  - **Memory Management:** The string `"Hello, World!"` is stored on the heap, and `greeting` holds the reference to that string on the stack.
- Delegates
- Interfaces

▼ Day 3: Classes ，Optional Parameters ,Access Modifiers ,Static Class, Static Methods, and Static Fields

▼ Access Modifiers

- public
  - **Visibility:** The member is accessible from any other code in the same assembly or another assembly that references it.
- private
  - **Visibility:** The member is only accessible within the same class or struct.
- protected
  - **Visibility:** The member is accessible within the same class and by derived classes.
- internal

    - **Visibility:** The member is accessible within the same assembly but not from another assembly.
- protected internal
    - **Visibility:** The member is accessible from any class in the same assembly or any derived class in another assembly.
- private protected
    - The member is accessible within the same class or in derived classes, but only within the same assembly.

▼ Classes

- A class is a blueprint for creating objects (instances). Classes encapsulate data for the object and methods to manipulate that data.:

▼ Field: Is Variable That Declared In class Or Struct And Store data directly and are typically used for internal data Storage within an object

- When to Use Fields Over Properties?
    - Private/Internal Fields for Simple Data Storage
    - Simple, Internal Data: If a field is used purely for internal calculations or as a private helper, and there’s no need to expose it or control access to it, using a private field is appropriate.

▼ Property is member that provide flexible to read, write, or compute the value of a private field. Properties are methods in disguise, but they appear to be public fields.

- Why Use Properties Over Fields?
    - Encapsulation and Data Validation:
        - Control Over Access: Properties allow you to control how the internal state of an object is accessed or modified. For example, you might want to allow reading but restrict or control writing to a field.

        - Validation: Properties enable you to include validation logic to ensure that only valid data is assigned to your fields. This is particularly useful for maintaining the integrity of your object's state.
    - Computed or Derived Values
        - Lazy Initialization or Computation: Sometimes, the value of a property might depend on other fields or might be expensive to calculate. Properties allow you to compute the value on demand.
    - Readonly Access
        - Immutable Objects: You can create properties with only a get accessor, effectively making the property read-only. This is useful in scenarios where the value should not change after the object is created.
    - Consistency and Future-Proofing
        - Consistency: Using properties ensures that all access to the field goes through the same interface, making it easier to change how the data is stored or accessed in the future without modifying the public API.
    - Future-Proofing
        - You can start with an auto-implemented property and later add logic to the get or set accessors without breaking the public interface

▼ Methods

- **Definition:** Methods define the actions that an object can perform. They contain code that can manipulate the object's data.
- Methods can be public, private, or protected.

▼ Constructors

- Constructors are special methods used to initialize objects. They have the same name as the class and do not have a return type.
- **Parameterless Constructor:** Initializes fields with default values.
- **Parameterized Constructor:** Initializes fields with specific values provided as arguments.

▼ Optional Parameters vs. Default Parameters

- Optional Parameters:
  - Parameters that have default values defined in the method signature
  - **Use Case:** Use when you want to provide default behavior while still allowing the caller to override values.

▼ Simple Equality Comparer

- Equality in C# can be based on reference or value. When creating custom classes, you often need to define how two instances of your class should be compared.
- Reference Equality vs. Value Equality:
  - **Reference Equality:** Two references point to the same object in memory.
  - **Value Equality:** Two objects are considered equal if their values are the same.

▼ Introduction to Records

- Records were introduced in C# 9.0 as a way to define immutable, value-based data types. They simplify many common tasks associated with data models.
- What Makes Records Different from Classes:
  - **Immutability:** Records are immutable by default, meaning their state cannot change after they are created.
  - **Value Equality:** Records automatically implement value equality, meaning two records are equal if their data is the same.
  - **Built-in Methods:** Records automatically provide `Equals`, `GetHashCode`, and `ToString` implementations.
  - `with` **Expressions:** Create new instances of a record with some properties changed.
  -

▼ Static Class, Static Methods, and Static Fields

- Static Classes:
  - **Definition:** A class that cannot be instantiated. It can only contain static members.
  - **Use Case:** Use for utility or helper classes where you don't need to store instance data.
- Static Methods:
  - **Definition:** Methods that belong to the class itself rather than an instance. They can be called without creating an object of the class.
  - **Use Case:** Use when the method does not need to access instance data.
- Static Fields
  - **Definition:** Fields that belong to the class rather than instances. All instances of the class share the same value of the static field.

    - **Use Case:** Use when you need a value to be shared across all instances or when storing class-wide settings or data.
-

▼ Day 4: Record In depth ,Equality ,Operator Overloading, Implicit and Explicit Casting, Passing Value Types by Value and Reference, `params` Array, Exception Handling , Enums

▼ Records vs Classes

▼ Overview of Classes

- **Reference Types**: Classes in C# are reference types. When you create an instance of a class, you're actually creating a reference to an object in memory.
- **State and Behavior**: Classes are used to define objects with both state (fields/properties) and behavior (methods).
- **Mutability**: By default, classes are mutable, meaning you can change their state after creation. This is useful but can lead to bugs if not managed carefully.
- **Memory Allocation**: When you create a class instance, it's stored in the heap, and the reference to this object is stored in the stack.

▼ Overview of Records

- **Introduced in C# 9.0**: Records are a new feature in C# designed to represent immutable data.
- **Value-Like Semantics**: Unlike classes, records are compared based on their values (the data they contain), not their references.

- **Immutable by Default**: Once created, the data in a record cannot be changed (unless you explicitly allow it). This makes them ideal for data transfer objects (DTOs) or other scenarios where data integrity is crucial.

- **Deep Dive into Differences**:
  - **Equality**:
    - **Classes**: By default, two class instances are equal if they refer to the same object in memory.
    - **Records**: Two record instances are equal if their values are equal, regardless of whether they are the same object in memory.
  - **Use Cases**:
    - **Classes**: Use when you need objects with mutable state and behavior (e.g., service classes, business logic).
    - **Records**: Use for data that should remain unchanged after creation (e.g., DTOs, configuration objects).
  -

▼ Implementing Equality for Reference Types

- **Overview**:
  - By default, reference types are compared by their memory addresses. If two references point to the same object, they are considered equal.
- **Overriding `Equals` and `GetHashCode`**:
  - **Why Override?**:
    - For custom objects, you may want to define equality based on the values of the properties rather than the reference.

- `Equals` Method:
  - This method is used to compare two objects for equality.
- `GetHashCode` **Method**:
  - This method returns an integer that represents the object's hash code, which is used in collections like dictionaries or hash sets

- Always override `GetHashCode` when overriding `Equals`.
- Use `HashCode.Combine` for a simple way to create a hash code.

▼ Operator Overloading (`+`, `-`, `==`, `!=`)

- Operator overloading allows you to define how operators like `+`, `-`, `==`, and `!=` behave for your custom classes or structs.
- syntax : The syntax for overloading an operator is straightforward. You define a method with the `operator` keyword followed by the operator you want to overload.

▼ Implicit and Explicit Casting

- **Implicit Casting**: Allows for automatic type conversion without data loss.
- **Explicit Casting**: Requires manual type conversion, typically when there might be data loss or when converting between incompatible types.
-

▼ Passing Value Types by Value and Reference

- **Passing by Value**:
  - **Default Behavior**: Value types (e.g., `int`, `struct`) are passed by value, meaning a copy of the data is made.

    - **Impact on Memory**: Since a copy is made, changes to the parameter inside the method do not affect the original value.

- **Passing by Reference**:
    - **Using `ref` and `out`**: These keywords allow you to pass a value type by reference, so changes to the parameter will affect the original value.
    - **Impact on Memory**: No new copy is created; the method operates on the original value.
    - Value Type Passed by `ref`
        - **Memory Allocation**: When you pass a value type by `ref`, no copy is made. Instead, the method receives a reference (address) to the original value.
        - **Method Operation**: The method operates directly on the original value, so changes made within the method affect the original value.
    - Value Type Passed by `out`:
        - **Memory Allocation**: Similar to `ref`, when you pass a value type by `out`, no copy is made. The method receives a reference to the original value.
        - **Initialization Requirement**: Unlike `ref`, the `out` parameter does not need to be initialized before being passed to the method. However, the method is required to assign a value to the `out` parameter before it returns.
    - `in` Keyword:
        - **Behavior**: Passes a variable by reference for read-only access. It ensures the method cannot modify the passed argument.
        - **Use Case**: When passing large structs to avoid copying, but without allowing modifications.
    - Memory-Wise Passing of Parameters

▼ Using `params` Array

- The `params` keyword allows a method to accept a variable number of arguments as an array.
- The `params` keyword must be the last parameter in the method signature.
- Advantages of Using `params`
  - **Simplified Method Calls**:
    - **Without `params`**: If you don't use `params`, the caller must explicitly create an array to pass multiple arguments.
    - **With `params`**: The caller can simply pass a comma-separated list of arguments without needing to create an array.

```
LogMessage("Info", new object[] { "Detail1", "Detail2

// With params
LogMessage("Info", "Detail1", "Detail2", "Detail3");
```

  - **Flexibility**:
    - **With `params`**: The method can be called with zero, one, or multiple arguments without the caller needing to worry about array syntax.
    - **Without `params`**: The caller must always provide an array, even if it's empty or contains a single element.
    - Code

```
// With params
LogMessage("Info");
LogMessage("Info", "Detail1");
```

```
// Without params
LogMessage("Info", new object[] { }); // Empty array
LogMessage("Info", new object[] { "Detail1" }); // Si
```

- **Readability**:
  - **With** `params`: The intention of accepting a variable number of arguments is clear to anyone reading the method signature. It conveys that the method is designed to handle multiple values in a natural way.
  - **Without** `params`: It's less clear that the method is intended to handle a variable number of arguments, and it might suggest that the method expects an array as a cohesive unit rather than individual elements.

▼ Exception Handling with `try-catch`

- Exception handling is critical for writing robust, error-resistant applications. The `try-catch` block is the primary tool in C# for managing exceptions.

▼ Enums in C# (Excluding Bit Flag Enums)

- Enums are a way to define named constants that represent a set of related values. They provide a type-safe way to work with a set of predefined values.
- Code

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
```

```
    Thursday,
    Friday,
    Saturday
}
```

- **Advantages**
  - Improves code readability by replacing magic numbers with meaningful names.
  - Enums provide compile-time checking, reducing errors.
- **Use Cases**
  - Using enums for representing states, options, or predefined sets of values.
-

▼ Day 5 : Introduction to Object-Oriented Programming

- What Is Object-Oriented Programming :
  - Object-Oriented Programming (OOP) is a programming paradigm that uses the concept of "objects" to design software. Objects are instances of classes, which are blueprints for creating these objects. OOP focuses on organizing code by grouping related variables and functions into classes, making it easier to manage, understand, and extend.

▼ Why Do We Need OOP

- **Modularity**: Code is easier to maintain and understand when it is divided into smaller, self-contained units (classes).
- **Reusability**: Classes and objects can be reused across programs, reducing redundancy.
- **Scalability**: OOP makes it easier to scale large applications by dividing responsibilities among different classes.

- **Real-World Mapping**: OOP closely models real-world entities and relationships, making it intuitive to design complex systems.

▼ Core Concepts Of OOP

▼ Classes and Objects

- **Class**: A blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have.
- **Object**: An instance of a class. It is created based on the structure defined by the class.
- Example

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    public void StartEngine()
    {
        Console.WriteLine("Engine started.");
    }
}

// Creating an object of the Car class
Car myCar = new Car();
myCar.Make = "Toyota";
myCar.Model = "Corolla";
myCar.Year = 2020;
myCar.StartEngine();
```

-

▼ Encapsulation

- Encapsulation is the practice of bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of the object's components, which can prevent unintended interference and misuse.
- Example

```
using System;

public class BankAccount
{
    private decimal balance;
    private string accountNumber;

    public BankAccount(string accountNumber, decimal ini
    {
        this.accountNumber = accountNumber;
        balance = initialBalance > 0 ? initialBalance :
    }

    public string AccountNumber => accountNumber;

    public decimal Balance
    {
        get => balance;
        private set => balance = value;
    }

    public void Deposit(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Deposit amount
        }
        balance += amount;
        Console.WriteLine($"Deposited {amount:C}. New ba
```

```
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Withdrawal amou
        }
        if (amount > balance)
        {
            throw new InvalidOperationException("Insuffi
        }
        balance -= amount;
        Console.WriteLine($"Withdrew {amount:C}. New bal
    }

    public void Transfer(BankAccount targetAccount, deci
    {
        if (targetAccount == null)
        {
            throw new ArgumentNullException(nameof(targe
        }
        if (amount <= 0)
        {
            throw new ArgumentException("Transfer amount
        }
        if (amount > balance)
        {
            throw new InvalidOperationException("Insuffi
        }

        Withdraw(amount);
        targetAccount.Deposit(amount);
        Console.WriteLine($"Transferred {amount:C} to ac
    }
}
```

```
public class Program
{
    public static void Main()
    {
        var account1 = new BankAccount("12345", 1000);
        var account2 = new BankAccount("67890", 500);

        account1.Deposit(200);
        account1.Withdraw(100);
        account1.Transfer(account2, 300);
    }
}
```

- **Encapsulation:** The `BankAccount` class encapsulates the `balance` and `accountNumber` fields, which are private and cannot be accessed directly from outside the class. Public methods such as `Deposit`, `Withdraw`, and `Transfer` allow controlled interaction with the `balance` field, ensuring that the balance is modified only through these methods.
- **Access Modifiers:** The use of `private` for the fields and `public` for methods exemplifies encapsulation, ensuring that the internal state of the object is protected from unintended external modifications.

▼ Inheritance

- Inheritance allows a class to inherit properties and methods from another class. The class that inherits is called a **subclass** or **derived class**, and the class being inherited from is called a **superclass** or **base class**.
- Example

```
public class Vehicle
{
    public int Speed { get; set; }

    public void Drive()
    {
        Console.WriteLine("The vehicle is driving.");
    }
}

public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }
}

// Usage
Car myCar = new Car();
myCar.Speed = 100;
myCar.NumberOfDoors = 4;
myCar.Drive(); // Outputs: The vehicle is driving.
```

- `new`, `override`, and No Keyword in C#
- In C#, the `new`, `override`, and no keyword modifiers are used to define how methods in derived classes relate to methods in base classes.
- **`override`**: Used when you want to override a method in the base class with a new implementation in the derived class. The base class method must be marked with `virtual` or `abstract`.
- **`new`**: Used to hide a method in the base class with a new method in the derived class. This does not override the base method but instead hides it, meaning the base class version is still accessible if accessed through a base class reference.

- **No Keyword:** If you define a method in a derived class that has the same name and signature as one in the base class but don't use `override` or `new`, it will generate a warning, and the base class method will still be accessible.

```
using System;

public class Shape
{
    // A virtual method that can be overridden by derive
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a Shape");
    }

    // A non-virtual method that can be hidden by derive
    public void Describe()
    {
        Console.WriteLine("This is a shape");
    }
}

public class Rectangle : Shape
{
    // Overriding the Draw method from the base class
    public override void Draw()
    {
        Console.WriteLine("Drawing a Rectangle");
    }

    // Using the 'new' keyword to hide the Describe meth
    public new void Describe()
    {
        Console.WriteLine("This is a rectangle");
    }
```

```
}

public class Circle : Shape
{
    // Overriding the Draw method from the base class
    public override void Draw()
    {
        Console.WriteLine("Drawing a Circle");
    }

    // A method with the same signature as in the base c
    // This will generate a compiler warning
    public void Describe()
    {
        Console.WriteLine("This is a circle");
    }
}

class Program
{
    static void Main()
    {
        // Creating instances
        Shape shape = new Shape();
        Rectangle rectangle = new Rectangle();
        Circle circle = new Circle();

        // Draw method call
        shape.Draw();        // Output: Drawing a Shape
        rectangle.Draw();    // Output: Drawing a Rectan
        circle.Draw();       // Output: Drawing a Circle

        // Describe method call
        shape.Describe();        // Output: This is a sh
        rectangle.Describe();    // Output: This is a re
        circle.Describe();       // Output: This is a ci
```

```
        // Using a base class reference to a derived cla
        Shape shapeRefRect = rectangle;
        Shape shapeRefCirc = circle;

        shapeRefRect.Draw();     // Output: Drawing a Re
        shapeRefRect.Describe(); // Output: This is a sh

        shapeRefCirc.Draw();     // Output: Drawing a Ci
        shapeRefCirc.Describe(); // Output: This is a sh
    }
}
```

▼ Polymorphism

- Polymorphism allows methods to do different things based on the object it is acting upon. It enables one interface to be used for a general class of actions, with specific actions determined by the exact nature of the situation.
-

```
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Some generic animal sound");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark");
    }
```

```
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow");
    }
}

// Usage
Animal myDog = new Dog();
Animal myCat = new Cat();
myDog.MakeSound(); // Outputs: Bark
myCat.MakeSound(); // Outputs: Meow
```

▼ Abstraction

- Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It allows focusing on what an object does instead of how it does it.
- Example

```
public abstract class Shape
{
    public abstract double GetArea();
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override double GetArea()
    {
        return Math.PI * Radius * Radius;
    }
```

```
}

// Usage
Shape myCircle = new Circle { Radius = 5 };
Console.WriteLine(myCircle.GetArea()); // Outputs: 78.54
```

▼ Interfaces

- An interface defines a contract that can be implemented by classes. It contains only the signatures of methods, properties, events, or indexers. A class or struct that implements an interface must implement all the members of the interface.
- Example

```
public interface IDrivable
{
    void Drive();
}

public class Car : IDrivable
{
    public void Drive()
    {
        Console.WriteLine("The car is driving.");
    }
}

// Usage
IDrivable myCar = new Car();
myCar.Drive(); // Outputs: The car is driving.
```

▼ Story About OOP

- Imagine you are a game developer tasked with creating a racing game.
  - **Classes and Objects**: You need to create different types of vehicles (Cars, Motorcycles, Trucks). Each type of vehicle has common

attributes like speed and methods like drive. You create a `Vehicle` class that acts as a blueprint for all vehicles.

- **Encapsulation**: You want to make sure the `speed` of a vehicle cannot be directly manipulated by other parts of the program. Instead, you provide methods like `Accelerate` and `Brake` to control the speed.
- **Inheritance**: You realize that all vehicles share some common features, but also have unique characteristics. You create a base `Vehicle` class and derive `Car`, `Motorcycle`, and `Truck` classes from it, inheriting the common features.
- **Polymorphism**: You want to ensure that when you call a `Drive` method, the correct behavior is executed based on the type of vehicle. A `Car` might have a different driving animation compared to a `Truck`. Polymorphism allows this flexibility.
- **Abstraction**: The game has complex physics calculations, but as a game developer, you don't need to worry about the implementation details. The physics engine provides an abstraction layer that you can use without knowing how it works.
- **Interfaces**: You want to create different types of vehicles that can be controlled by the player. You define an `IDrivable` interface that all drivable vehicles must implement. This way, you can write code that controls any vehicle without knowing its specific type.

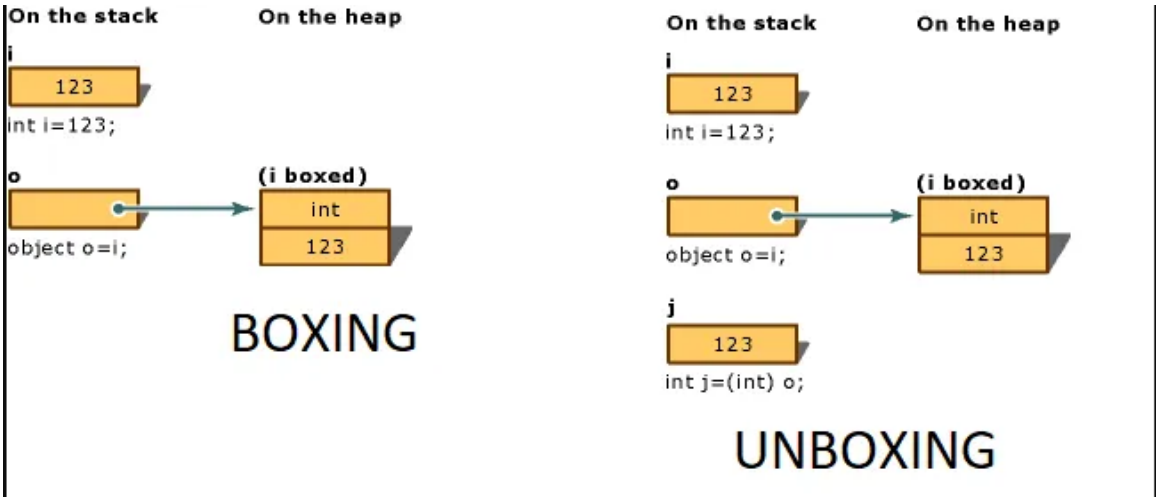
- **Why OOP is Beneficial**
  - **Improved Maintenance**: With OOP, if you need to change the way `Drive` works, you can do it in the `Vehicle` class and automatically apply it to all vehicles.
  - **Reusability**: Once you create a `Vehicle` class, you can reuse it across different games or projects.
  - **Better Collaboration**: OOP allows multiple developers to work on different parts of a project independently. One developer can work on the `Vehicle` class while another works on the `Physics` class.
  - **Real-World Mapping**: OOP allows you to model real-world entities and interactions in a way that is intuitive and easy to understand.

- Examples for Each Concept
  - **Inheritance**: A game might have a `Character` base class and derive classes like `Warrior`, `Mage`, and `Archer`, each with their own unique abilities.
  - **Polymorphism**: A `Spell` class might have different effects based on the specific spell type (`Fireball`, `Heal`, `Shield`), all derived from a base `Spell` class.
  - **Encapsulation**: A `BankAccount` class that hides the balance and provides methods for depositing and withdrawing money.
  - **Abstraction**: A `Shape` class where the specific details of calculating the area are hidden from the user.
  - **Interfaces**: An `IDamageable` interface that different game entities (like `Enemy`, `Player`, `Obstacle`) implement to receive damage.

▼ Day 6: boxing Vs Unboxing , Nullable Value Type , Generics

▼ boxing Vs unboxing

-



- In simple terms:

    - Boxing implies storing a value type as an object = box value type in System.Object
    - Unboxing = read the value from an object = unbox value from the object
- Now the official definitions:
    - 1. Boxing
        - Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. .NET CLR boxes a value type i.e., wraps it in System.Object instance and stores it on the managed(garbage-collected) heap. Boxing is an implicit operation.
    - Unboxing
        - Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface.
          Unboxing extracts the value type from the object.
        - An unboxing operation consists of:
        - Checking the object instance to make sure that it is a boxed value of the given value type.
        - Copying the value from the instance into the value-type variable.
        - Attempting to unbox null causes a NullReferenceException.
        - Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.
- **Problems with Boxing and Unboxing**
    - **Performance Overhead**: Boxing and unboxing involve additional memory allocation and CPU cycles.
    - **Type Safety**: Unboxing requires explicit casting, which can lead to runtime errors if the types don’t match.
- Performance -Wise

- Boxing and unboxing operations affect the performance as they are computationally expensive processes.
  - When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally.

▼ Nullable Value Types

- Definition ：Value types in C# have an inherent limitation: by definition, they represent a value and therefore cannot be null. But what if you genuinely need to represent the absence of a value? This is where nullable value types come in, introduced with C# 2.0.
- **What are Nullable Value Types?**
  - In C#, value types (e.g., `int`, `double`, `bool`, `struct`) are types that have a value or are uninitialized. They cannot be set to null. Nullable value types bridge this gap by allowing a value type to be `null`.
  - **Declaring a Nullable Value Type**

```
int? nullableInt = null;
double? nullableDouble = 123.45;
bool? flag = null;
```

  - **Value and HasValue Properties:**
    - The `Nullable<T>` struct provides two handy properties:
      - • `HasValue`: Returns `true` if the nullable type contains a value; `false` if it contains `null`.
      - • `Value`: Returns the value of the nullable type if `HasValue` is `true`. If `HasValue` is `false`, accessing `Value` will

throw an `InvalidOperationException`.

- **Null Coalescing and Nullable Value Types**
  - The null coalescing operator (`??`) is particularly useful with nullable value types. It returns the left-hand operand if it's not null; otherwise, it returns the right operand.
  -

```
int? a = null;
int b = a ?? -1;  // b will be set to -1
```

- **Null Conditional Operator**
  - The null conditional operator (`?.`) allows for short-circuiting when dealing with nullable value types:
- Conversion
  - You can convert between a nullable value type and its underlying type:
- Nullable value types are still stored on the stack, but with an additional flag indicating whether they have a value or are `null`.

```
int? a = 5; // Nullable int with value 5
int? b = null; // Nullable int with value null
int c = (int)a; // Unboxing a nullable value ty
```

- **Casting Nullable to Non-Nullable**:
  -

```
int? nullableInt = 5;
int nonNullableInt = nullableInt.Value; // Expl
```

- **Handling Null Values**:

```
int? nullableInt = null;
int nonNullableInt = nullableInt ?? 0; // Use t
```

▼ Generics

- **Introduction**
  - **Generics**: Allow you to define classes, methods, delegates, and interfaces with a placeholder for the data type.
- Benefits of Generics:
  - **Type Safety**: Compile-time checking, reducing runtime errors.
  - **Performance**: Avoids boxing/unboxing as the types are specified at compile time.
  - **Code Reusability**: Write a method or class once and reuse it with different data types.
  - Example

```
public class Box<T>
{
    private T _value;
    public Box(T value)
    {
        _value = value;
    }
    public T GetValue()
    {
        return _value;
    }
}

Box<int> intBox = new Box<int>(123);
Box<string> strBox = new Box<string>("Hello");
```

```
public T Add<T>(T a, T b) where T : struct
{
    return a + b; // Assume operator + is defined for T
}
```

- Generic Constraints
  - `where` **Clause**
    - `where T : struct` (T must be a value type)
    - `where T : class` (T must be a reference type)
    - `where T : new()` (T must have a parameterless constructor)
    - `where T : SomeBaseClass` (T must inherit from `SomeBaseClass`)
  - Example

```
public class GenericExample<T> where T : class, new()
{
    public T CreateInstance()
    {
        return new T(); // Requires T to have a paramete
    }
}
```

- **Memory-Wise**:
  - **No Boxing/Unboxing**: Since types are determined at compile time, generics avoid the overhead associated with boxing/unboxing.
  - **Performance**: Generics are optimized at runtime, providing both type safety and performance benefits.

▼ Day Seven : Understanding Generics Collections in C#

▼ General Talk

- In this session, you will explore generic collections in C#. You'll understand why generic collections are essential, how they differ from arrays, and learn about key collection types like `List<T>`, `HashSet<T>`, `Dictionary<TKey, TValue>`, and others. We will also dive into deferred vs immediate execution, `IEnumerable`, and how `foreach` works in .NET 8. The session will be filled with practical examples and deep explanations.
- Introduction to Generics Collections :
  - **What are Generics Collections?**
    - Explain how generic collections are type-safe and how they allow for code reuse without sacrificing performance.
  - Why Use Generics Collections Over Arrays?
    - **Arrays**: Fixed size, no type safety when using non-generic collections.
    - **Generics Collections**: Flexible size, type-safe, and more powerful methods.
  - Commonly Used Generic Collections:
    - Briefly introduce `List<T>`, `HashSet<T>`, `Dictionary<TKey, TValue>`, `Queue<T>`, and `Stack<T>`.
- Deep Dive into `List<T>`
  - **What is a `List<T>`?**
    - Dynamic array-like collection.
  - Key Methods and Properties
    - `Add()`, `Remove()`, `Contains()`, `Count`, `Insert()`, `IndexOf()`.
  - Examples :
    - Create a list of integers and demonstrate adding, removing, and finding elements.
    - Show how `List<T>` automatically resizes as you add elements
  - Performance Considerations

        - Discuss resizing and memory allocation.
    - Understanding `HashSet<T>`
        - What is a `HashSet<T>`
            - A collection that contains only unique elements.
        - Key Methods and Properties
            - `Add()`, `Remove()`, `Contains()`, `Count`, `UnionWith()`, `IntersectWith()`.
        - Examples:
            - Create a `HashSet<string>` to store unique words from a text.
            - Demonstrate the use of `UnionWith` and `IntersectWith` for set operations.
    - Differences from `List<T>`
        - No duplicates, optimized for set operations.
- **Exploring `Dictionary<TKey, TValue>`**
    - **What is a `Dictionary<TKey, TValue>`?**
        - A collection of key-value pairs.
    - **Key Methods and Properties**
        - `Add()`, `Remove()`, `ContainsKey()`, `TryGetValue()`, `Keys`, `Values`.
    - **Examples**:
        - Create a dictionary to map student IDs to their names.
        - Show how to safely retrieve values using `TryGetValue`.
    - **Performance Considerations**
        - Discuss hash tables and the O(1) lookup time.
- Deferred vs Immediate Execution
    - **What is Deferred Execution?**

        - Execution of a query is delayed until the results are actually iterated over.
        - Examples: LINQ queries that are not executed until `ToList()`, `ToArray()`, `FirstOrDefault()` are called.
    - **What is Immediate Execution?**
        - Execution happens as soon as the query is defined.
        - Examples: Methods like `Count()`, `Sum()`, and `Average()` in LINQ.
- How `foreach` Works in .NET
    - **Understanding the `foreach` Loop**
        - How `foreach` iterates over collections.
        - Role of `GetEnumerator()` and `MoveNext()`.

▼ What is a Collection

- A **collection** in C# is a class that holds multiple objects, typically of the same type. It allows you to manage a group of related objects together. Instead of handling each book individually, you can now store them all in a collection—a shelf, if you will.

▼ Base for All Collections: `IEnumerable`

- Before diving into specific collections, let's talk about what makes collections tick: the `IEnumerable` **interface**. This interface is the foundation upon which all collections in C# are built. It provides the ability to iterate over a collection, allowing you to access each item one by one using a `foreach` loop.

▼ The Power of `List<T>`

- Now that we've built our own collection, let's dive into the most commonly used collection in C#: `List<T>`.
- **Adding and Removing Items**:

```
var books = new List<Book>();
books.Add(new Book { Title = "Clean Code", Author = "Robert
books.RemoveAt(0); // Removes the first book
```

- **Finding Items**:

```
var cleanCode = books.FirstOrDefault(b => b.Title == "Clean
```

- Indexing:

```
var firstBook = books[0]; // Access the first book
```

- **Deep Dive Example**:

```
var libraryList = new List<Book>
{
    new Book { Title = "C# in Depth", Author = "Jon Skeet"
    new Book { Title = "The Pragmatic Programmer", Author =
    new Book { Title = "Clean Code", Author = "Robert C. Ma
};

// Adding a new book
libraryList.Add(new Book { Title = "Code Complete", Author

// Finding a book
var pragmaticProgrammer = libraryList.FirstOrDefault(b => b
Console.WriteLine($"Found: {pragmaticProgrammer.Title} by {

// Removing a book
libraryList.Remove(pragmaticProgrammer);

// Listing all books
foreach (var book in libraryList)
```

```
{
    Console.WriteLine($"{book.Title} by {book.Author}");
}
```

▼ Exploring `Dictionary<TKey, TValue>`

- In a physical library, books are often organized by unique identifiers, like an ISBN number. This is akin to how a `Dictionary<TKey, TValue>` works in C#. It stores data in key-value pairs, allowing for fast lookups based on a unique key.
- Example

```
var bookDictionary = new Dictionary<string, Book>
{
    { "978-0132350884", new Book { Title = "Clean Code", Au
    { "978-0137081073", new Book { Title = "The Pragmatic P
};

// Adding a new entry
bookDictionary.Add("978-0134757599", new Book { Title = "Re

// Accessing a book by its key
var cleanCodeBook = bookDictionary["978-0132350884"];
Console.WriteLine($"{cleanCodeBook.Title} by {cleanCodeBook
```

▼ The Uniqueness of `HashSet<T>`

- Imagine a collection where no two books are exactly the same. This is the world of the `HashSet<T>`. It's a collection that ensures all elements are unique.

```
var uniqueBooks = new HashSet<Book>
{
    new Book { Title = "Clean Code", Author = "Robert C
    new Book { Title = "Clean Code", Author = "Robert C
```

```
};
Console.WriteLine($"Total unique books: {uniqueBooks.Cou
```

-
- **Union and Intersection**:

```
var csharpBooks = new HashSet<string> { "C# in Depth", "
var programmingBooks = new HashSet<string> { "Clean Code

csharpBooks.UnionWith(programmingBooks);
Console.WriteLine("Books after union:");
foreach (var book in csharpBooks)
{
    Console.WriteLine(book);
}
```

- Deep Dive
  -

```
var bookSet1 = new HashSet<Book>
{
    new Book { Title = "C# in Depth", Author = "Jon Skee
    new Book { Title = "Clean Code", Author = "Robert C
};

var bookSet2 = new HashSet<Book>
{
    new Book { Title = "The Pragmatic Programmer", Autho
    new Book { Title = "Clean Code", Author = "Robert C
};

bookSet1.IntersectWith(bookSet2);
Console.WriteLine("Books in both sets:");
foreach (var book in bookSet1)
```

```
{
    Console.WriteLine($"{book.Title} by {book.Author}")
}
```

▼ Deferred vs Immediate Execution

- Deferred execution in LINQ is like a lazy librarian who waits until you actually request a book before fetching it. Immediate execution is like a librarian who retrieves the book as soon as you ask for it, whether you need it now or later.
- Examples of Deferred Execution

```
var bookQuery = libraryList.Where(b => b.Title.Contains

// Deferred execution: The query is only executed when w
foreach (var book in bookQuery)
{
    Console.WriteLine(book.Title);
}
```

- **Examples of Immediate Execution**:
-

▼ Day Eight : Delegate

- **What is a Delegate?**
  - A delegate is a type that represents references to methods. With delegates, you can pass methods as parameters, assign them to variables, and call them dynamically at runtime.
- **Benefits of Delegates:**
  - **Reusability**: Write a single method that can be used with different operations.
  - **Flexibility**: Pass different methods as arguments to a single method.

- **Why Are Delegates Useful:**
  - A common scenario in programming is when the part of the code that knows the action that needs to be executed isn't the same part of the code that performs the execution. In such situations, you need a way to encapsulate an action inside an object and pass it around.
  - How to solve this issue? C# delegate to the rescue! By instantiating a delegate, you can express an action as an object and hand that object over (or delegate it) to the code that's actually able to execute the action.
  - In C#, delegates are particularly useful for event handling. Through delegates, you subscribe to an event. Delegates are also essential in LINQ, which is honestly a big part of what makes programming in C# so pleasurable.
- **Instantiating a Delegate:**
  - `StrToInt myAction = delegate(string s) {  return s.Length; };`
- As you can see, it's possible to use the **delegate** keyword to create a function on the fly and assign it to the delegate instance. But why stop there? We can go a step further:
  - `StrToInt myAction = s => s.Length;`
- **Types of Delegates:**
  - **Single cast**
    - You can leverage different types of delegates in C#. The ones you've seen so far are **single-cast:** delegates that point to a single method. Having a delegate hold two or more methods is possible and often quite valuable.
  - **Multicast:**
    - Delegates that hold more than one method are \ called **multicast delegates**. You can use the plus (+) operator to add more methods to a delegate instance, and the minus (-) operator to remove a method. When you invoke the delegate instance, it executes all of

the methods in its invocation likes in the order in which they were added.

- Let’s see a multicast delegate in action. First, consider the following delegate declaration:
  - public delegate void DisplaySomething(int number);
- DisplaySomething displayEvenOrOdd = x ⇒ Console.WriteLine(x % 2 == 0 ? "even" : "odd");
  - Now we’ll create another instance that takes a number and displays its value in binary:
    - DisplaySomething displayBinary = x ⇒ Console.WriteLine(Convert.ToString(x, 2));
  - Let’s now create yet another instance that displays as many asterisks as the number informed:
    - DisplaySomething displayNAsteriks = x ⇒ Console.WriteLine(new String('*', x));
  - DisplaySomething displayAll = displayEvenOrOdd + displayBinary + displayNAsteriks;
  - displayNAsteriks;
  - displayAll(10);

- **Generic Delegates:**
  - **Func: Take To 16 input Parameter And Return One Type**
  - Action Take To 16 Input Parameter And Return Void
  - Predicate Take One Generic Parameter And Return Bool