WEEK-4 NOTES

▼ C# OOP Types and Keywords

Class

A class is a blueprint for creating objects. It defines the structure (fields, properties) and behavior (methods) of objects. Classes support OOP features like **inheritance**, **polymorphism**, and **encapsulation**.

Key points:

- Defined using the `class` keyword.
- Supports inheritance (except for sealed classes).
- Can contain constructors, fields, properties, methods, events, and nested types.
- Objects of a class are stored in the heap.

Example:

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

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public void Greet()
    {
```

```
        Console.WriteLine($"Hello, my name is {Name} and I am {Age} yea
rs old.");
    }
}

class Program
{
    static void Main()
    {
        Person p = new Person("Nisharg",21);
        p.Greet();
    }
}
```

Output:

```
Hello, my name is Nisharg and I am 21 years old.
```

Struct

A struct is similar to a class but is a **value type** stored in the stack. It is useful for small data objects that do not require inheritance.

Key points:

- Defined using the `struct` keyword.
- Value type → stored in the stack.
- No inheritance (cannot inherit other structs/classes except interfaces).
- Can have methods, fields, properties, and constructors.
- Default parameterless constructor is provided automatically.

Example:

```
public struct Point
{
```

```
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public void Display()
    {
        Console.WriteLine($"Point is at ({X}, {Y})");
    }
}

class Program
{
    static void Main()
    {
        Point p1 = new Point(5, 10);
        p1.Display();
    }
}
```

Output:

```
Point is at (5, 10)
```

Interface

An interface is a **contract** that defines a set of methods, properties, events, or indexers without implementing them. Classes or structs that implement the interface must provide the implementation.

Key points:

- Defined using the `interface` keyword.
- Members are **implicitly public and abstract**.

- Cannot contain fields.
- Supports multiple inheritance.
- From C# 8+, interfaces can have default implementations and static members.

Example:

```
public interface IShape
{
    double Area();
    double Perimeter();
}

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

    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Area()
    {
        return Math.PI * Radius * Radius;
    }

    public double Perimeter()
    {
        return 2 * Math.PI * Radius;
    }
}

class Program
{
    static void Main()
    {
        IShape shape = new Circle(5);
```

```
        Console.WriteLine($"Area: {shape.Area()}");
        Console.WriteLine($"Perimeter: {shape.Perimeter()}");
    }
}
```

Output:

```
Area: 78.53981633974483
Perimeter: 31.41592653589793
```

Static Keyword

The `static` keyword is used to declare members that belong to the type itself rather than to an instance of the type.

Key points:

- Static members belong to the class, not instances.
- Static classes cannot be instantiated.
- Static members are accessed using the class name.
- Useful for utility methods, constants, and shared data.

Example of Static Member:

```
public class MathHelper
{
    public static double Pi = 3.14159;

    public static double Square(double number)
    {
        return number * number;
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine(MathHelper.Pi);
```

```
        Console.WriteLine(MathHelper.Square(4));
    }
}
```

Output:

```
3.14159
16
```

Example of Static Class:

```
public static class Utilities
{
    public static void PrintMessage(string message)
    {
        Console.WriteLine(message);
    }
}

class Program
{
    static void Main()
    {
        Utilities.PrintMessage("Hello World");
    }
}
```

Output:

```
Hello World
```

Abstract Keyword

The `abstract` keyword is used to define classes and members that must be implemented in derived classes.

Key points:

- Abstract class cannot be instantiated.

- Abstract methods have no body and must be overridden in derived classes.
- Abstract classes can contain normal methods and fields.

Example:

```
public abstract class Animal
{
    public abstract void MakeSound();

    public void Eat()
    {
        Console.WriteLine("Eating");
    }
}

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

class Program
{
    static void Main()
    {
        Animal animal = new Dog();
        animal.MakeSound(); // Bark
        animal.Eat();       // Eating
    }
}
```

Output:

```
Bark
Eating
```

Sealed Keyword

The `sealed` keyword is used to prevent a class from being inherited or a method from being overridden.

Key points:

- Sealed classes cannot be inherited.
- Sealed methods cannot be overridden in derived classes.

Example of Sealed Class:

```
public sealed class FinalClass
{
    public void Display()
    {
        Console.WriteLine("This is a sealed class.");
    }
}

// class Derived : FinalClass {} //  Error: cannot inherit sealed class
```

Example of Sealed Method:

```
public class BaseClass
{
    public virtual void Show()
    {
        Console.WriteLine("Base class method");
    }
}

public class DerivedClass : BaseClass
{
    public sealed override void Show()
    {
        Console.WriteLine("Derived class sealed method");
    }
}
```

```
public class SubDerivedClass : DerivedClass
{
    // public override void Show() {} //  Error: cannot override sealed met
hod
}
```

`virtual` Keyword

The `virtual` keyword is used to allow a method or property in a base class to be **overridden** in a derived class.

Key points:

- Marks a method, property, indexer, or event as overridable.
- Allows derived classes to provide their own implementation.
- Virtual members have a base implementation unless overridden.

Example:

```
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Generic Animal Sound");
    }
}

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

class Program
```

```
{
    static void Main()
    {
        Animal a = new Dog();
        a.MakeSound(); // Bark
    }
}
```

Output:

```
Bark
```

`override` Keyword

The `override` keyword is used in a derived class to **replace** the base class implementation of a virtual method or property.

Key points:

- Must be used with members declared as `virtual`, `abstract`, or `override` in the base class.
- Changes the behavior for the derived class.

Example:

```
public class BaseClass
{
    public virtual void Display()
    {
        Console.WriteLine("Base class display");
    }
}

public class DerivedClass : BaseClass
{
    public override void Display()
    {
        Console.WriteLine("Derived class display");
    }
}
```

```
class Program
{
    static void Main()
    {
        BaseClass obj = new DerivedClass();
        obj.Display(); // Derived class display
    }
}
```

new Keyword (Method Hiding)

The `new` keyword is used to **hide** a base class member instead of overriding it.

This is called **method hiding**. It does not replace the base member for polymorphism — it creates a separate method.

Key points:

- Used for hiding members with the same name in base class.
- Base class reference still uses the base member unless cast to derived type.

Example:

```
public class BaseClass
{
    public void Display()
    {
        Console.WriteLine("Base class display");
    }
}

public class DerivedClass : BaseClass
{
    public new void Display()
    {
        Console.WriteLine("Derived class display");
    }
```

```
}

class Program
{
    static void Main()
    {
        BaseClass obj1 = new DerivedClass();
        obj1.Display(); // Base class display

        DerivedClass obj2 = new DerivedClass();
        obj2.Display(); // Derived class display
    }
}
```

`readonly` Keyword

The `readonly` keyword is used to make fields immutable after initialization.

Key points:

- `readonly` fields can be initialized during declaration or in a constructor.
- Cannot be modified elsewhere.

Example:

```
public class Circle
{
    public readonly double Pi = 3.14159;
    public readonly double Radius;

    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Area()
    {
        return Pi * Radius * Radius;
```

```
    }
}

class Program
{
    static void Main()
    {
            Circle c = new Circle(5);
            Console.WriteLine(c.Area()); // 78.53975
    }
}
```

Note: `readonly` is different from `const` because it allows runtime initialization.

const Keyword

The `const` keyword defines constant values that are known at compile time and cannot change.

Key points:

- Must be initialized during declaration.
- Implicitly `static`.
- Cannot be changed after declaration.

Example:

```
public class MathConstants
{
    public const double Pi = 3.14159;
}

class Program
{
    static void Main()
    {
        Console.WriteLine(MathConstants.Pi); // 3.14159
```

```
    }
}
```

partial Keyword

The `partial` keyword allows splitting a class, struct, or interface into multiple files.

Useful in large projects and auto-generated code (e.g., designer code in Windows Forms).

Key points:

- All parts must use the `partial` keyword.
- All parts must be in the same namespace and assembly.
- Combines at compile time into a single type.

Example:

File: `Person.Part1.cs`

```
public partial class Person
{
    public string Name { get; set; }
}
```

File: `Person.Part2.cs`

```
public partial class Person
{
    public int Age { get; set; }

    public void Display()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}
```

Usage:

```
class Program
{
    static void Main()
    {
        Person p = new Person { Name = "Nisharg", Age = 25 };
        p.Display();
    }
}
```

Output:

```
Name: Nisharg, Age: 25
```

C# Generics

What are Generics?

Generics in C# allow you to define **classes, methods, interfaces, and delegates with type parameters** so they can work with any data type while providing **type safety** and **code reusability.**

Instead of creating multiple versions of the same class for different types, we create one generic definition.

Advantages of Generics:

- Type safety (errors detected at compile time)
- Code reusability (write once, use for any type)
- Performance (no boxing/unboxing)

Generic Classes

A generic class uses **type parameters** to work with any data type.

Syntax:

```
public class ClassName<T>
{
```

```
    // T is a placeholder type parameter
}
```

Example:

```
public class Box<T>
{
    private T _value;

    public void Add(T value)
    {
        _value = value;
    }

    public T Get()
    {
        return _value;
    }
}

class Program
{
    static void Main()
    {
        Box<int> intBox = new Box<int>();
        intBox.Add(100);
        Console.WriteLine(intBox.Get()); // 100

        Box<string> strBox = new Box<string>();
        strBox.Add("Hello World");
        Console.WriteLine(strBox.Get()); // Hello World
    }
}
```

Explanation:

Here `T` is a placeholder for any type. We create a `Box<int>` and a `Box<string>` without rewriting the class.

Generic Methods

A generic method defines its own type parameters. Useful when a method needs to work with different types without making the whole class generic.

Syntax:

```
public ReturnType MethodName<T>(T param)
```

Example:

```
public class Utilities
{
    public static void Swap<T>(ref T a, ref T b)
    {
        T temp = a;
        a = b;
        b = temp;
    }
}

class Program
{
    static void Main()
    {
        int x = 5, y = 10;
        Utilities.Swap(ref x, ref y);
        Console.WriteLine($"x: {x}, y: {y}"); // x: 10, y: 5

        string s1 = "A", s2 = "B";
        Utilities.Swap(ref s1, ref s2);
        Console.WriteLine($"s1: {s1}, s2: {s2}"); // s1: B, s2: A
    }
}
```

Explanation:

Here, the method `Swap<T>` can swap values of any type.

Generic Constraints

Constraints allow restricting what types can be used for generic parameters.

Syntax:

```
where T : constraint
```

Common Constraints:

- `where T : class` → must be a reference type
- `where T : struct` → must be a value type
- `where T : new()` → must have a parameterless constructor
- `where T : BaseClass` → must inherit from `BaseClass`
- `where T : IInterface` → must implement interface

Example:

```
public class Repository<T> where T : new()
{
    public T CreateInstance()
    {
        return new T(); // Requires parameterless constructor
    }
}

class Sample
{
    public int Id { get; set; }
    public Sample() { Id = 100; }
}

class Program
{
    static void Main()
    {
        Repository<Sample> repo = new Repository<Sample>();
```

```
        Sample s = repo.CreateInstance();
        Console.WriteLine(s.Id); // 100
    }
}
```

C# Generic Collections (`System.Collections.Generic`)

The `System.Collections.Generic` namespace provides **type-safe collections** that work with any data type using generics. Unlike non-generic collections, they avoid boxing/unboxing and provide better performance.

1. List

A `List<T>` is a **dynamic array** that grows automatically as items are added.

Key Properties:

- `Count` → number of elements in the list.
- `Capacity` → number of elements the list can hold before resizing.
- `Item[index]` → gets or sets the element at the specified index.

Key Methods:

- `Add(item)` → adds an item.
- `AddRange(collection)` → adds multiple items.
- `Insert(index, item)` → inserts at index.
- `Remove(item)` → removes first occurrence.
- `RemoveAt(index)` → removes item at index.
- `Contains(item)` → checks existence.
- `IndexOf(item)` → returns index of item.
- `Sort()` → sorts the list.

- `Reverse()` → reverses order.
- `Clear()` → removes all items.

Example:

```
List<int> numbers = new List<int> { 1, 2, 3 };
numbers.Add(4);
numbers.Insert(1, 99);
numbers.Remove(3);

Console.WriteLine(numbers.Count);   // 3
Console.WriteLine(numbers[1]);      // 99
```

2. Dictionary<TKey, TValue>

A `Dictionary<TKey, TValue>` stores data in **key-value pairs** with **fast lookups.**

Key Properties:

- `Count` → number of pairs.
- `Keys` → collection of keys.
- `Values` → collection of values.

Key Methods:

- `Add(key, value)` → adds entry.
- `Remove(key)` → removes by key.
- `ContainsKey(key)` → checks if key exists.
- `ContainsValue(value)` → checks if value exists.
- `TryGetValue(key, out value)` → safe retrieval.
- `Clear()` → removes all pairs.

Example:

```
Dictionary<int, string> students = new Dictionary<int, string>();
students.Add(1, "Alice");
students[2] = "Bob";
```

```
Console.WriteLine(students[1]);           // Alice
Console.WriteLine(students.ContainsKey(2)); // True
```

3. SortedDictionary<TKey, TValue>

Similar to `Dictionary`, but **automatically sorts** by key.

Key Properties:

- `Count`, `Keys`, `Values` (same as Dictionary).

Key Methods:

- Same as `Dictionary`, but items are always ordered by key.

Example:

```
SortedDictionary<int, string> sd = new SortedDictionary<int, string>();
sd.Add(3, "C");
sd.Add(1, "A");
sd.Add(2, "B");

foreach (var kv in sd)
    Console.WriteLine($"{kv.Key} : {kv.Value}");
```

Output:

```
1 : A
2 : B
3 : C
```

4. SortedList<TKey, TValue>

Similar to `SortedDictionary`, but stored internally as **arrays** (less memory, faster lookups but slower insertions).

Key Properties:

- `Count`, `Keys`, `Values`.
- `Capacity`.

Key Methods:

- `Add(key, value)`, `Remove(key)`, `ContainsKey(key)` (like Dictionary).
- `IndexOfKey(key)` → gets index of key.
- `IndexOfValue(value)` → gets index of value.

Example:

```
SortedList<int, string> sl = new SortedList<int, string>();
sl.Add(2, "B");
sl.Add(1, "A");

Console.WriteLine(sl[1]); // A
```

5. HashSet

A `HashSet<T>` is a collection of **unique values**, unordered.

Key Properties:

- `Count`
- `Comparer`

Key Methods:

- `Add(item)` → adds if not exists.
- `Remove(item)` → removes.
- `Contains(item)` → checks existence.
- `UnionWith(collection)` → union set.
- `IntersectWith(collection)` → intersection.
- `ExceptWith(collection)` → removes items found in another set.
- `Clear()`

Example:

```
HashSet<int> set = new HashSet<int> { 1, 2, 2, 3 };
set.Add(4);
```

```
foreach (var i in set)
    Console.WriteLine(i); // 1,2,3,4 (no duplicates)
```

6. Queue

A `Queue<T>` is a **FIFO (First In, First Out)** collection.

Key Properties:

- `Count`.

Key Methods:

- `Enqueue(item)` → adds item to end.
- `Dequeue()` → removes and returns first item.
- `Peek()` → returns first item without removing.
- `Contains(item)` → checks if exists.
- `Clear()`.

Example:

```
Queue<string> q = new Queue<string>();
q.Enqueue("A");
q.Enqueue("B");

Console.WriteLine(q.Dequeue()); // A
Console.WriteLine(q.Peek());    // B
```

7. Stack

A `Stack<T>` is a **LIFO (Last In, First Out)** collection.

Key Properties:

- `Count`.

Key Methods:

- `Push(item)` → adds item on top.
- `Pop()` → removes and returns top item.
- `Peek()` → returns top item without removing.

- `Contains(item)`.
- `Clear()`.

Example:

```
Stack<int> st = new Stack<int>();
st.Push(1);
st.Push(2);

Console.WriteLine(st.Pop());  // 2
Console.WriteLine(st.Peek()); // 1
```

8. LinkedList

A `LinkedList<T>` is a **doubly-linked list**.

Key Properties:

- `Count`.
- `First` → first node.
- `Last` → last node.

Key Methods:

- `AddFirst(item)` → adds at beginning.
- `AddLast(item)` → adds at end.
- `AddBefore(node, item)` → insert before a node.
- `AddAfter(node, item)` → insert after a node.
- `Remove(item)`, `RemoveFirst()`, `RemoveLast()`.
- `Find(item)` → finds node with value.

Example:

```
LinkedList<string> list = new LinkedList<string>();
list.AddLast("A");
list.AddLast("B");
list.AddFirst("Start");
```

```
foreach (var i in list)
    Console.WriteLine(i);
```

Output:

```
Start
A
B
```

9. KeyValuePair<TKey, TValue>

A struct representing a **key-value pair** used in dictionaries.

Properties:

- `Key`
- `Value`

Example:

```
KeyValuePair<int, string> kv = new KeyValuePair<int, string>(1, "One");
Console.WriteLine(kv.Key);   // 1
Console.WriteLine(kv.Value); // One
```

▼ Data serialization with JSON, XML

What is Serialization?

- **Serialization** is the process of converting an object into a format (JSON, XML, Binary, etc.) that can be stored or transmitted.
- **Deserialization** is the reverse process: converting the serialized data back into an object.
- Used for:
  - Saving application state
  - Sending data over a network (APIs)

- Storing structured data in files

JSON Serialization

Namespace

`System.Text.Json` (recommended in .NET 5+)

Older: `Newtonsoft.Json` (via NuGet)

Important Classes

- `JsonSerializer` – handles serialization and deserialization.
- `JsonSerializerOptions` – configures serialization behavior.

Basic Example

```
using System;
using System.Text.Json;

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsActive { get; set; }
}

class Program
{
    static void Main()
    {
        Student s = new Student { Id = 1, Name = "Nisharg", IsActive = true
};

        // Serialization: Object → JSON
        string json = JsonSerializer.Serialize(s);
        Console.WriteLine(json);
        // {"Id":1,"Name":"Nisharg","IsActive":true}

        // Deserialization: JSON → Object
```

```
        Student copy = JsonSerializer.Deserialize<Student>(json);
        Console.WriteLine(copy.Name); // Nisharg
    }
}
```

JsonSerializerOptions

```
var options = new JsonSerializerOptions
{
    WriteIndented = true, // pretty printing
    PropertyNameCaseInsensitive = true, // ignore case when deserializin
g
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase // id → "id"
};

string json = JsonSerializer.Serialize(s, options);
```

Attributes for JSON

- `[JsonPropertyName("customName")]` → Rename property in JSON
- `[JsonIgnore]` → Exclude property during serialization

```
using System.Text.Json.Serialization;

public class Student
{
    [JsonPropertyName("student_id")]
    public int Id { get; set; }

    public string Name { get; set; }

    [JsonIgnore]
    public bool IsActive { get; set; }
}
```

Collections with JSON

```
List<Student> students = new List<Student>
{
    new Student { Id = 1, Name = "A" },
    new Student { Id = 2, Name = "B" }
};

string json = JsonSerializer.Serialize(students);
var deserialized = JsonSerializer.Deserialize<List<Student>>(json);
```

When we serialize enum it stored as integer and when deserialize it it gets as an integer

```
string jsoenum = JsonSerializer.Serialize(book, new JsonSerializerOptions
{
    Converters =
    {
        new JsonStringEnumConverter()
    }
});
```

XML Serialization

Namespace

`System.Xml.Serialization`

Important Classes

- `XmlSerializer` – handles XML serialization/deserialization.
- `XmlElement`, `XmlAttribute` – attributes to control XML output.

Basic Example

```
using System;
using System.IO;
using System.Xml.Serialization;
```

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Program
{
    static void Main()
    {
        Student s = new Student { Id = 1, Name = "Nisharg" };

        // Serialization: Object → XML
        XmlSerializer serializer = new XmlSerializer(typeof(Student));
        using (StringWriter writer = new StringWriter())
        {
            serializer.Serialize(writer, s);
            string xml = writer.ToString();
            Console.WriteLine(xml);
        }

        // Deserialization: XML → Object
        string xmlData = "<Student><Id>2</Id><Name>John</Name></St
udent>";
        using (StringReader reader = new StringReader(xmlData))
        {
            Student copy = (Student)serializer.Deserialize(reader);
            Console.WriteLine(copy.Name); // John
        }
    }
}
```

Attributes in XML

- `[XmlElement("CustomName")]` → Rename element
- `[XmlAttribute]` → Make property an attribute
- `[XmlIgnore]` → Ignore during serialization

- `[XmlArray]` and `[XmlArrayItem]` → Customize collections

```
public class Student
{
    [XmlAttribute]
    public int Id { get; set; }

    [XmlElement("FullName")]
    public string Name { get; set; }

    [XmlIgnore]
    public bool IsActive { get; set; }
}
```

Collections with XML

```
public class Classroom
{
    [XmlArray("Students")]
    [XmlArrayItem("Student")]
    public List<Student> Students { get; set; }
}
```

This generates:

```xml
<Classroom>
  <Students>
    <Student>
      <Id>1</Id>
      <Name>A</Name>
    </Student>
    <Student>
      <Id>2</Id>
      <Name>B</Name>
    </Student>
  </Students>
</Classroom>
```

▼ Lambda

What is a Lambda Expression?

- A **lambda expression** is an **inline, anonymous function** that can be used wherever a delegate type is expected.
- It is a more concise replacement for **anonymous methods.**
- Built using the **lambda operator (`=>`)**, pronounced "goes to".

Syntax Forms

```
(parameters) => expression
(parameters) => { statement_block }
```

- **Expression Lambda** → has one expression (value returned automatically).
- **Statement Lambda** → contains multiple statements in `{}` (explicit `return` required if non-void).

Examples:

```
x => x * x              // one param, one expression
(x, y) => x + y         // multiple params
() => Console.WriteLine("Hi") // no params
(int x) => x + 1        // explicitly typed param
```

Delegate Compatibility

Lambdas are **syntactic sugar for delegates.**

With `Func`

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(4)); // 16
```

With `Action`

```
Action<string> greet = name => Console.WriteLine("Hello " + name);
greet("Nisharg"); // Hello Nisharg
```

With `Predicate`

```
Predicate<int> isEven = n => n % 2 == 0;
Console.WriteLine(isEven(7)); // False
```

Expression vs Statement Lambdas

Expression Lambda

```
Func<int, int> increment = x => x + 1;
```

- No `{}`
- Single expression
- Return is **implicit**

Statement Lambda

```
Func<int, int> increment = x =>
{
    int y = x + 1;
    return y;
};
```

- Can include multiple statements
- Must explicitly `return`

Parameter Rules

- Parentheses **optional** if there's exactly one parameter and no type specified:

```
n ⇒ n * n
```

- Must use parentheses for:
  - Multiple parameters: `(x, y) ⇒ x + y`
  - No parameters: `() ⇒ 42`
  - Typed parameters: `(int x, int y) ⇒ x + y`

Closures in Lambda

A lambda can **capture variables** from the enclosing scope.

```
int factor = 5;
Func<int, int> multiply = n ⇒ n * factor;

Console.WriteLine(multiply(10)); // 50
```

Notes:

- Captured variables live as long as the delegate does.
- This is called a **closure**.
- Be careful inside loops.

Example:

```
var actions = new List<Action>();
for (int i = 0; i < 3; i++)
    actions.Add(() ⇒ Console.WriteLine(i));

foreach (var action in actions) action();
// Prints: 3, 3, 3 (not 0,1,2) because `i` is captured, not copied.
```

Lambda with LINQ

Most LINQ methods take delegates, so lambdas are perfect.

Filtering

```
var evens = numbers.Where(n => n % 2 == 0);
```

Projection

```
var names = students.Select(s => s.Name);
```

Ordering

```
var ordered = students.OrderBy(s => s.Age);
```

Grouping

```
var groups = students.GroupBy(s => s.Department);
```

▼ Extension methods

An **extension method** is a static method that the compiler lets you call as if it were an instance method on the type of its first parameter.

- The extension method must be a **static method** inside a **static class**.
- The first parameter **must be** prefixed with `this` and its type is the *receiver*. The `this` parameter **must be the first parameter**.
- The extension type can be a class, struct, interface, array, generic type, etc.
- The extension method can be generic and may include additional parameters after the `this` parameter.

example:

```
namespace MyExtensions
{
    public static class StringExtensions
    {
        public static bool IsNullOrEmpty(this string s)
```

```
        {
            return string.IsNullOrEmpty(s);
        }
    }
}
```

Usage:

```
using MyExtensions;

string name = null;
bool empty = name.IsNullOrEmpty(); // calls StringExtensions.IsNullOrE
mpty(name)
```

Examples

A. Simple helper on `int`

```
public static class IntExtensions
{
    public static bool IsEven(this int value) => value % 2 == 0;
}
```

B. Generic collection helper (`AddRange`)

```
public static class CollectionExtensions
{
    public static void AddRange<T>(this ICollection<T> coll, IEnumerable
<T> items)
    {
        if (coll == null) throw new ArgumentNullException(nameof(coll));
        foreach (var it in items) coll.Add(it);
    }
}
```

Calling extension methods (two equivalent forms)

Given `s.IsNullOrEmpty()` , both forms are equivalent:

```
// instance-like
bool b = s.IsNullOrEmpty();

// explicit static call
bool b2 = StringExtensions.IsNullOrEmpty(s);
```

Overloading / precedence / ambiguities

- If the type already has an instance method with the same signature, the instance method is chosen. The extension method is *not* used.
- If two imported namespaces provide extension methods with the same signature, the call is **ambiguous** and results in a compile-time error (unless overload resolution favors one). Avoid polluting broad namespaces to reduce collisions.

Example ambiguous situation:

```
using A; // defines FooExtensions.Foo(this string)
using B; // defines FooExtensions.Foo(this string)

"hello".Foo(); // compile-time error: call is ambiguous
```

Limitations and important rules

- Extension methods **cannot** add fields or true instance properties/constructors. They are static methods — you cannot store state on the extended type via an extension method.
- Extension methods **cannot** access private or protected members of the extended type. They only use the public/internal API surface.

- Historically, the `this` parameter **cannot** be declared with `ref`/`out` modifiers (the compiler will report an error).
- Extension methods do **not** become part of the original type's metadata; they are static methods defined in a separate type.

Generics and constraints

Extension methods work perfectly with generics and constraints:

```
public static class EnumerableExtensions
{
    public static T? FirstOrNull<T>(this IEnumerable<T> source) where T : class
    {
        foreach (var s in source) return s;
        return null;
    }
}
```

Extension methods for interfaces

Because extension methods are resolved by compiler when the namespace is in scope, you can "extend" interfaces and all their implementors:

```
public interface IRepository { }
public static class RepositoryExtensions
{
    public static void Ping(this IRepository repo) => Console.WriteLine("pong");
}
```

Any type implementing `IRepository` can call `.Ping()` provided the extension's namespace is imported. This is a way to add helper behavior for any implementor without modifying the interface

Null receiver behavior

Because `x.Foo()` is translated to `StaticClass.Foo(x)`, *you can call extension methods on a `null` reference* and the method will receive `null` as the parameter. This is useful if you want null-safe helpers:

```
public static bool IsNullOrEmpty(this string? s) => string.IsNullOrEmpty
(s);

string? maybe = null;
Console.WriteLine(maybe.IsNullOrEmpty()); // true
```

Async extension methods

Extension methods can be `async` and return `Task`/`Task<T>`:

```
public static class HttpClientExtensions
{
    public static async Task<string> GetStringOrEmptyAsync(this HttpCli
ent client, string url)
    {
        try { return await client.GetStringAsync(url); }
        catch { return string.Empty; }
    }
}
```

Example

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

namespace MyExtensions
{
    public static class IntExtensions
    {
        public static bool IsEven(this int n) => n % 2 == 0;
```

```
    }

    public static class EnumerableExtensions
    {
        public static IEnumerable<T> WhereNotNull<T>(this IEnumerable<
T?> source) where T : class
        {
            foreach (var item in source)
                if (item is not null) yield return item;
        }

        public static void AddRange<T>(this ICollection<T> coll, IEnumera
ble<T> items)
        {
            if (coll == null) throw new ArgumentNullException(nameof(coll));
            foreach (var it in items) coll.Add(it);
        }

        public static string ToCsv<T>(this IEnumerable<T> items)
        {

            return string.Join(",", items);
        }
    }
}

class Program
{
    static void Main()
    {
        using MyExtensions;

        Console.WriteLine(10.IsEven()); // True

        List<string?> names = new() { "Alice", null, "Bob" };
        var nonNull = names.WhereNotNull();
        Console.WriteLine(nonNull.ToCsv()); // Alice,Bob
```

```
        List<int> ints = new() { 1, 2 };
        ints.AddRange(new[] { 3, 4 });
        Console.WriteLine(ints.ToCsv()); // 1,2,3,4
    }
}
```

▼ Reflection

Reflection, in **C# and the .NET framework**, is a mechanism that lets you **inspect, discover, and interact with metadata of assemblies, modules, and types at runtime**.

Core Components of Reflection

Reflection is provided by the `System.Reflection` namespace.

- `Assembly` → Represents a compiled DLL/EXE. Lets you load and explore all types inside.
- `Module` → Represents a single file in an assembly (a DLL or EXE file may contain multiple modules).
- `Type` → Represents metadata for a class, struct, interface, enum, or delegate.
- `MemberInfo` → Base class for describing members of a type (methods, properties, fields, events).
- `MethodInfo` → Represents methods, including signatures and parameters.
- `ConstructorInfo` → Represents constructors.
- `PropertyInfo` → Represents properties (getter/setter).
- `FieldInfo` → Represents fields (public, private).
- `EventInfo` → Represents events.
- `ParameterInfo` → Represents metadata about method/constructor parameters.
- `CustomAttributeData` → Represents custom attributes applied to types/members.

Getting Type Metadata

You can obtain a `Type` object in three main ways:

```
// 1. At compile-time
Type t1 = typeof(string);

// 2. At runtime
string s = "Hello";
Type t2 = s.GetType();
```

Once you have a `Type`, you can query its metadata:

```
Console.WriteLine(t1.FullName); // System.String
Console.WriteLine(t1.Namespace); // System
Console.WriteLine(t1.IsClass);   // True
```

Inspecting Assemblies

```
using System.Reflection;

Assembly asm = Assembly.GetExecutingAssembly();

// List all types in the current assembly
foreach (Type type in asm.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

You can also **dynamically load** external assemblies:

```
Assembly external = Assembly.LoadFrom("MyLibrary.dll");
```

Inspecting Members

class:

```
public class Person
{
```

```
    public string Name { get; set; }
    private int age;

    public Person(string name, int age) { Name = name; this.age = age; }

    public void Greet() => Console.WriteLine($"Hi, I'm {Name}, {age} year
s old.");

    private void Secret() => Console.WriteLine("This is private!");
}
```

You can inspect it:

```
Type t = typeof(Person);

// Properties
foreach (var prop in t.GetProperties())
    Console.WriteLine($"Property: {prop.Name} ({prop.PropertyType})");

// Methods (public + non-public)
foreach (var method in t.GetMethods(BindingFlags.Public | BindingFlag
s.NonPublic | BindingFlags.Instance))
    Console.WriteLine($"Method: {method.Name}");

// Constructors
foreach (var ctor in t.GetConstructors())
{
    Console.WriteLine($"Constructor: {ctor}");
    foreach (var param in ctor.GetParameters())
        Console.WriteLine($"  Param: {param.Name} ({param.ParameterTy
pe})");
}
```

Dynamic Instantiation & Invocation

Create an object dynamically:

```
object personObj = Activator.CreateInstance(typeof(Person), "Alice", 2
5);
```

Call a method dynamically:

```
MethodInfo greet = typeof(Person).GetMethod("Greet");
greet.Invoke(personObj, null);
```

Access private method:

```
MethodInfo secret = typeof(Person).GetMethod("Secret", BindingFlags.
NonPublic | BindingFlags.Instance);
secret.Invoke(personObj, null);
```

Get/Set property dynamically:

```
PropertyInfo prop = typeof(Person).GetProperty("Name");
Console.WriteLine(prop.GetValue(personObj)); // Alice
prop.SetValue(personObj, "Bob");
```

▼ HttpClient

What is HttpClient?

`HttpClient` is the primary class in **System.Net.Http** to:

- Send HTTP requests (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`, `HEAD`, etc.).
- Receive HTTP responses.
- Work with both **high-level helpers** (`GetAsync`, `PostAsJsonAsync`) and **low-level pipeline control** (`SendAsync`).

It integrates with:

- **Handlers** (for logging, retry, decompression, cookies, etc.).

- **IHttpClientFactory** (in ASP.NET Core) to manage lifecycle and pooling.

Constructors

Constructor	Description
`HttpClient()`	Uses a default `HttpClientHandler`. Handler is disposed when client is disposed.
`HttpClient(HttpMessageHandler handler)`	Uses a custom handler (cookies, proxy, decompression). Handler disposed with client.
`HttpClient(HttpMessageHandler handler, bool disposeHandler)`	Lets you decide whether the handler is disposed with the client. Useful when sharing a handler across multiple clients.

Properties

Property	Type	Usage
`BaseAddress`	`Uri`	Sets base URL for relative requests.
`DefaultRequestHeaders`	`HttpRequestHeaders`	Headers applied to all requests (e.g. `Accept`, `Authorization`).
`DefaultRequestVersion`	`Version`	HTTP version used by default (1.1 / 2 / 3).
`DefaultVersionPolicy`	`HttpVersionPolicy`	Controls fallback (e.g. RequestVersionOrLower).
`Timeout`	`TimeSpan`	Request timeout (throws `TaskCanceledException` if exceeded).
`MaxResponseContentBufferSize`	`long`	Maximum buffer size for responses. Less used now (streaming preferred).
`DefaultProxy`	`IWebProxy?`	Configures proxy at client level.

Methods

Convenience Methods

- `GetAsync(string)`
- `GetStringAsync(string)`
- `GetStreamAsync(string)`
- `GetByteArrayAsync(string)`
- `PostAsync(string, HttpContent)`
- `PutAsync(string, HttpContent)`
- `DeleteAsync(string)`
- `PatchAsync(string, HttpContent)`

- `SendAsync(HttpRequestMessage, HttpCompletionOption, CancellationToken)`

  Gives control over:
  - HTTP method
  - Headers
  - Content
  - Completion behavior (buffered or streamed)

- `CancelPendingRequests()` → cancels all outstanding requests.
- `Dispose()` → releases resources.

Error & Exception Behavior

- Non-success status codes (e.g. 404, 500) **do not throw automatically.**
  - Use `.IsSuccessStatusCode` OR
  - `.EnsureSuccessStatusCode()` → throws `HttpRequestException`.
- Failures:
  - Network issues → `HttpRequestException`
  - Timeout → `TaskCanceledException` / `OperationCanceledException`

- Cancellations (via `CancellationToken`) → `OperationCanceledException`

Handlers & Advanced Config

HttpClient uses **pipeline of HttpMessageHandler objects**:

- `HttpClientHandler` → default, supports:
  - Proxy
  - Cookies (`CookieContainer`)
  - Automatic decompression
  - Redirects
  - Client certificates
- `SocketsHttpHandler` → modern, more efficient (connection pooling, DNS caching).
- `DelegatingHandler` → custom middleware-like handler (logging, retry, auth, correlation IDs).

Example:

```
var handler = new HttpClientHandler {
    AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate,
    AllowAutoRedirect = true,
    UseCookies = true,
    CookieContainer = new CookieContainer()
};
using var client = new HttpClient(handler) { Timeout = TimeSpan.FromSeconds(30) };
```

JSON Helpers (System.Net.Http.Json)

Provides `System.Net.Http.Json` extension methods (with `System.Text.Json`).

- `GetFromJsonAsync<T>()`
- `PostAsJsonAsync<T>()`
- `ReadFromJsonAsync<T>()`

- `PutAsJsonAsync<T>()`
- `PatchAsJsonAsync<T>()`

Example:

```
using System.Net.Http.Json;

var todo = await client.GetFromJsonAsync<Todo>("todos/1");
await client.PostAsJsonAsync("todos", new Todo { Title = "New task" });
```

Examples

Minimal GET

```
static readonly HttpClient client = new() { BaseAddress = new Uri("https://jsonplaceholder.typicode.com/") };

string body = await client.GetStringAsync("posts/1");
Console.WriteLine(body);
```

POST JSON

```
using System.Net.Http.Json;

record PostDto(int UserId, string Title, string Body);

var response = await client.PostAsJsonAsync("posts", new PostDto(1, "Hello", "Body text"));
response.EnsureSuccessStatusCode();

var created = await response.Content.ReadFromJsonAsync<PostDto>();
Console.WriteLine($"Created: {created.Title}");
```

Custom Request with Headers + Cancellation

```
var json = JsonSerializer.Serialize(new { Name = "x", Qty = 3 });
using var request = new HttpRequestMessage(HttpMethod.Post, "https://api.example.com/items") {
    Content = new StringContent(json, Encoding.UTF8, "application/json")
};
request.Headers.Add("X-Request-ID", Guid.NewGuid().ToString());

using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
using var response = await client.SendAsync(request, HttpCompletionOption.ResponseHeadersRead, cts.Token);
response.EnsureSuccessStatusCode();
string result = await response.Content.ReadAsStringAsync();
```

Stream Large File

```
using var response = await client.GetAsync(largeFileUrl, HttpCompletionOption.ResponseHeadersRead);
using var remoteStream = await response.Content.ReadAsStreamAsync();
using var file = File.Create("bigfile.dat");
await remoteStream.CopyToAsync(file);
```

▼ RestClient (RestSharp Library)

- Part of the **RestSharp** library (a popular .NET HTTP client).
- Provides a **higher-level abstraction over HttpClient.**
- Specially designed for **consuming REST APIs.**
- Handles **serialization, deserialization, headers, query parameters, authentication** more easily.

Namespaces

```
using RestSharp;
```

Install package (NuGet):

```
dotnet add package RestSharp
```

Core Classes

- `RestClient` → Represents the HTTP client that communicates with the server.
- `RestRequest` → Represents an HTTP request (URL, parameters, headers, body, etc.).
- `RestResponse` → Represents an HTTP response (status code, headers, body, etc.).

Creating a RestClient

```
var client = new RestClient("https://jsonplaceholder.typicode.com/");
```

You can also configure it with options:

```
var options = new RestClientOptions("https://jsonplaceholder.typicode.com/")
{
    ThrowOnAnyError = true,
    Timeout = 5000
};
var client = new RestClient(options);
```

1. GET Request

```
var client = new RestClient("https://jsonplaceholder.typicode.com/");
var request = new RestRequest("posts/1", Method.Get);

RestResponse response = await client.ExecuteAsync(request);
```

```
Console.WriteLine(response.Content); // Raw JSON response
```

2. POST Request

```
var client = new RestClient("https://jsonplaceholder.typicode.com/");
var request = new RestRequest("posts", Method.Post);

request.AddJsonBody(new
{
    title = "foo",
    body = "bar",
    userId = 1
});

RestResponse response = await client.ExecuteAsync(request);
Console.WriteLine(response.Content);
```

3. PUT Request

```
var request = new RestRequest("posts/1", Method.Put);
request.AddJsonBody(new
{
    id = 1,
    title = "updated title",
    body = "updated body",
    userId = 1
});

var response = await client.ExecuteAsync(request);
Console.WriteLine(response.Content);
```

4. DELETE Request

```
var request = new RestRequest("posts/1", Method.Delete);
var response = await client.ExecuteAsync(request);
```

```
Console.WriteLine(response.StatusCode); // Should be 200 or 204
Adding Headers & Query Parameters
```

```
var request = new RestRequest("posts", Method.Get);

// Add query parameter
request.AddParameter("userId", 1);

// Add custom header
request.AddHeader("Authorization", "Bearer mytoken");

var response = await client.ExecuteAsync(request);
Console.WriteLine(response.Content);
```

Deserialization (JSON → C# Object)

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Body { get; set; }
    public int UserId { get; set; }
}

var request = new RestRequest("posts/1", Method.Get);
var response = await client.ExecuteAsync<Post>(request);

Post post = response.Data;
Console.WriteLine($"Post Title: {post.Title}");
```

▼ EPPlus

```
using OfficeOpenXml;
using OfficeOpenXml.Style;
```

```
Install-Package EPPlus
```

Key Features

1. Create new Excel files (`.xlsx`) from scratch.
2. Open and edit existing Excel files.
3. Work with **worksheets**, **rows**, **columns**, and **cells**.
4. Apply **formulas**, **styles**, and **formatting**.
5. Support for **tables, charts, pivot tables, images**.
6. Export data from C# objects, lists, or `DataTable` to Excel.
7. Read values back from Excel into your program.

Core Classes

- `ExcelPackage` → Represents the entire Excel file (the workbook).
- `ExcelWorkbook` → Represents the workbook inside the package.
- `ExcelWorksheet` → Represents a single worksheet inside the workbook.
- `ExcelRange` → Represents a range of cells (single cell or multiple cells).

Example

1. Create a New Excel File

```
using OfficeOpenXml;
using System;
using System.IO;

namespace EPPlusDemo
{
    internal class Program
    {
```

```
        static void Main(string[] args)
        {
            ExcelPackage.LicenseContext = LicenseContext.NonCommercia
l; // Required for non-commercial use

            string filePath = "Demo.xlsx";

            using (ExcelPackage package = new ExcelPackage())
            {
                // Add a worksheet
                ExcelWorksheet sheet = package.Workbook.Worksheets.Add
("MySheet");

                // Write data
                sheet.Cells[1, 1].Value = "ID";
                sheet.Cells[1, 2].Value = "Name";
                sheet.Cells[1, 3].Value = "Score";

                sheet.Cells[2, 1].Value = 1;
                sheet.Cells[2, 2].Value = "Alice";
                sheet.Cells[2, 3].Value = 95;

                sheet.Cells[3, 1].Value = 2;
                sheet.Cells[3, 2].Value = "Bob";
                sheet.Cells[3, 3].Formula = "SUM(50,40)"; // Using formula

                // Save to file
                FileInfo excelFile = new FileInfo(filePath);
                package.SaveAs(excelFile);
            }

            Console.WriteLine("Excel file created successfully!");
        }
    }
}
```

2. Reading from an Existing Excel File

```
using OfficeOpenXml;
using System;
using System.IO;

namespace EPPlusDemo
{
    internal class ReadExcel
    {
        static void Main(string[] args)
        {
            ExcelPackage.LicenseContext = LicenseContext.NonCommercia
l;

            FileInfo file = new FileInfo("Demo.xlsx");

            using (ExcelPackage package = new ExcelPackage(file))
            {
                ExcelWorksheet sheet = package.Workbook.Worksheets["My
Sheet"];

                int row = 2; // Reading Alice's row
                string name = sheet.Cells[row, 2].Text;  // Get as string
                int score = int.Parse(sheet.Cells[row, 3].Value.ToString()); // G
et as int
                Console.WriteLine($"{name} scored {score}");
            }
        }
    }
}
```

3. Formatting Cells

```
sheet.Cells[1, 1, 1, 3].Style.Font.Bold = true; // Bold header
sheet.Cells[1, 1, 1, 3].Style.Fill.PatternType = ExcelFillStyle.Solid;
sheet.Cells[1, 1, 1, 3].Style.Fill.BackgroundColor.SetColor(System.Drawin
g.Color.LightBlue);
```

```
sheet.Cells[2, 3].Style.Numberformat.Format = "0.00"; // Format as deci
mal
```

4. Looping Through Rows

```
int rowCount = sheet.Dimension.Rows;
int colCount = sheet.Dimension.Columns;

for (int row = 1; row <= rowCount; row++)
{
    for (int col = 1; col <= colCount; col++)
    {
        Console.Write(sheet.Cells[row, col].Text + "\t");
    }
    Console.WriteLine();
}
```