👨‍💻

.NET Development

📌 My personal accounts links

🔗 LinkedIn	https://www.linkedin.com/in/ahmed-hany-899a9a321?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=android_app
🟢WhatsApp	https://wa.me/qr/7KNUQ7ZI3KO2N1
📘Facebook	https://www.facebook.com/share/1NFM1PfSjc/

1. C#

1. Basics ⇒

1. Advanced ⇒
2. Data Structure (Collections) ⇒
3. LINQ ⇒

Basics

What is a C# and what can do with it?

C# is a programming language developed by Microsoft and a strongly type languages . It's used to build various types of applications, including:

1. **Web Applications** (websites and APIs).

2. **Desktop Applications** (Windows apps).
3. **Mobile Apps** (using Xamarin or MAUI).
4. **Games** (using Unity).
5. **Cloud and IoT Solutions**.
6. **Database-Driven Applications**.

What is a variables and how it stores ?

A **variable** is a named storage location in a computer's memory that holds a value. Variables allow programmers to store data, retrieve it later, and manipulate it during program execution.

In programming, you define variables with specific **data types** (e.g., `int`, `string`, `float`), which determine what kind of data the variable can hold.

Storage is two types

1. Volatile (Ram)
2. Un volatile (Hard)

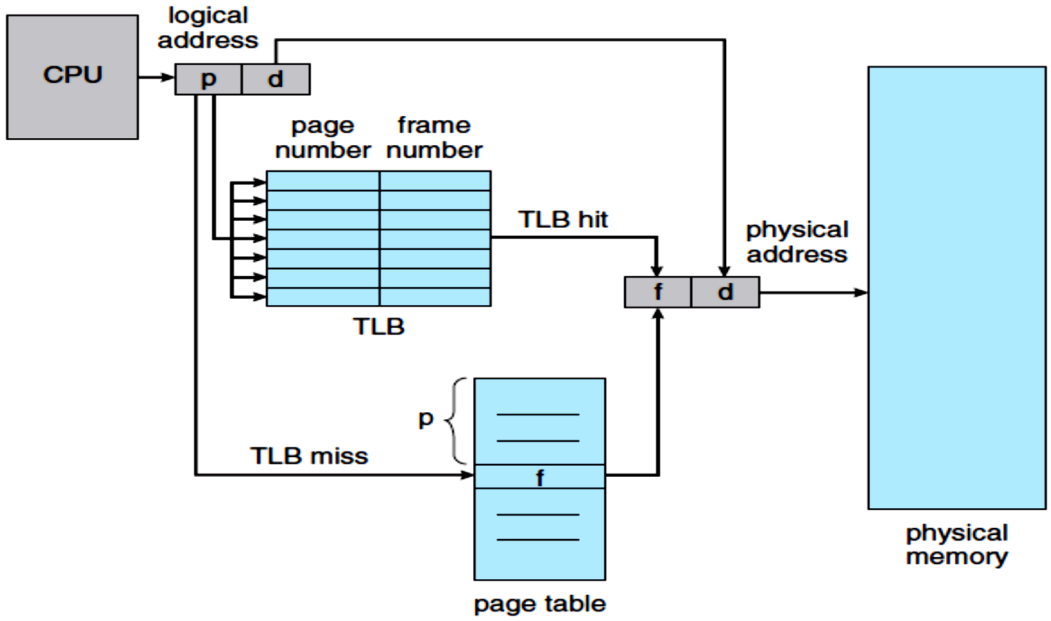
Ram is containing

1. **Stack** is used to store **local variables** and **function call information** (like function parameters, return addresses, and local variables).
2. **Heap** is used to store **dynamically allocated memory** (like objects and data that need to persist beyond the scope of a function).
3. **Intern Pool** is a special area in memory where **unique, immutable strings** are stored to save space. When a string is **interned**, it's stored only once, and any duplicate strings with the same value will refer to the same memory location, rather than creating new instances. This helps to reduce memory usage and improve performance, especially for comparing strings.

1. Example of store in a stack

```
internal class Program
{
    static void Main(string[] args)
    {
        int x; //declaration
        x = 1; //assignment

        int y = 2; //initialization


    }

}
```

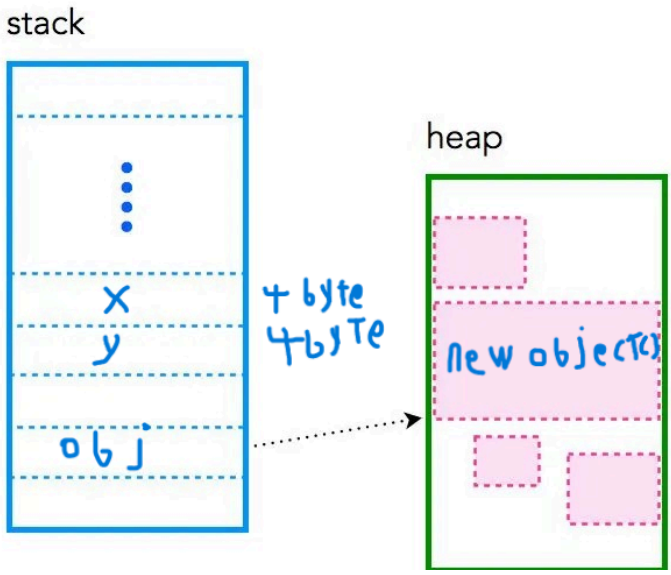
1. CPU generating a logical address
2. converting logical address to physical address



3. here x and y are local variables ⇒ put it in stack
4. Book 4byte for everyone because here datatype is integer

2. **Example of store in Heap**

1. CPU generating a logical address
2. converting logical address to physical address
3. here we store a reference type ⇒ put an identifier in stack and value in heap



What a Difference between value type and reference type?

- **Value Type**:
  - Holds the **actual value**.
  - Stored directly in **stack memory**.
  - Examples: `int`, `float`, `char`, `struct`.
  - When you assign a value type variable to another, a **copy** of the value is made.
- **Reference Type**:
  - Holds a **reference (address)** to the actual data in memory.
  - Stored in **heap memory**.

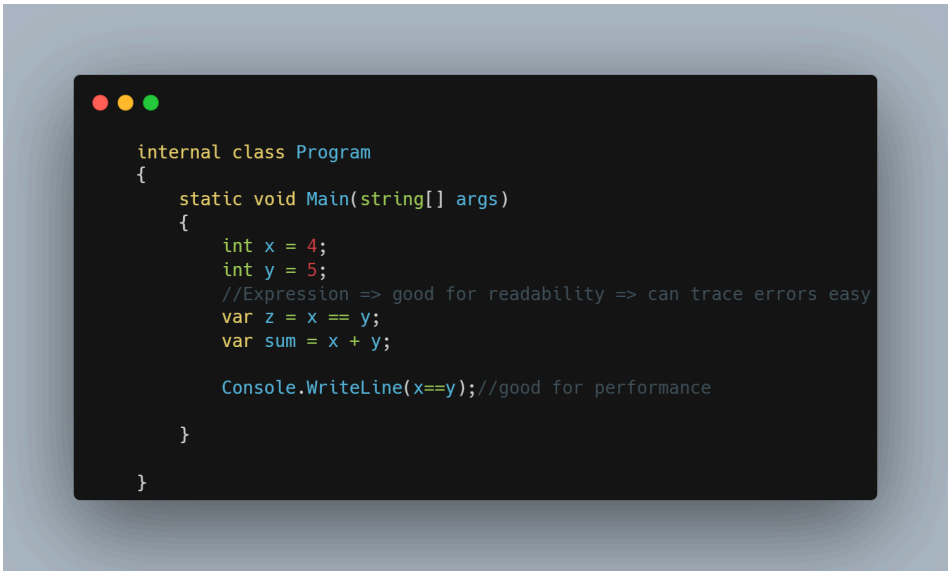
- Examples: `class`, `string`, `array`, `delegate`.
- When you assign a reference type variable to another, both variables point to the **same object** in memory.

What a Difference between var and dynamic keywords ?

Feature	`var`	`dynamic`
Type Determination	Compile-time	Runtime
Type Checking	Strong (at compile time)	Weak (at runtime)
Flexibility	Fixed type after assignment	Type can change dynamically
Error Detection	At compile time	At runtime

What is an Expression ?

An
expression is a combination of values, variables, operators, and functions that are evaluated to a single value

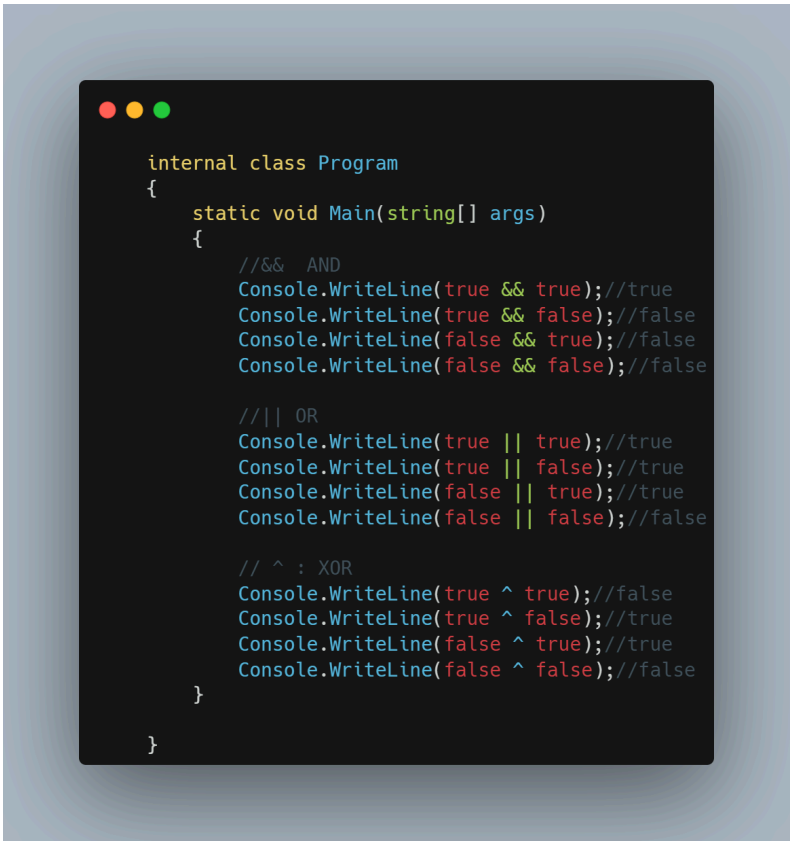


```
internal class Program
{
    static void Main(string[] args)
    {
        int x = 4;
        int y = 5;
        //Expression => good for readability => can trace errors easy
        var z = x == y;
        var sum = x + y;

        Console.WriteLine(x==y);//good for performance

    }

}
```

What a Difference between short circuit and long circuit?

short circuit

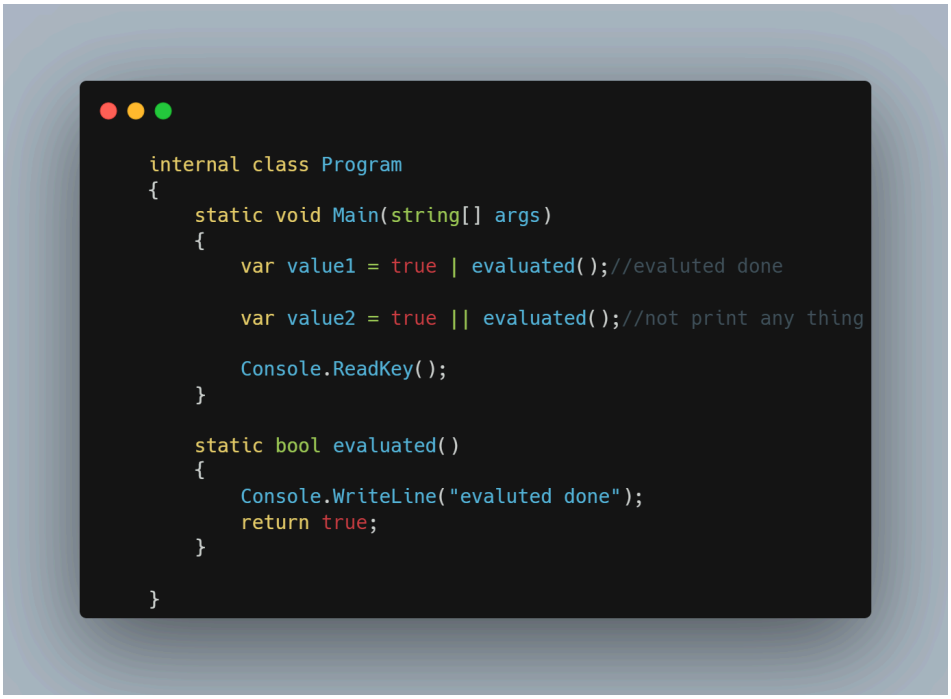
- Stops evaluating as soon as the result is determined
- Faster, avoids unnecessary evaluations.
- Applies to logical operators like



```
internal class Program
{
    static void Main(string[] args)
    {
        //&&  AND
        Console.WriteLine(true && true);//true
        Console.WriteLine(true && false);//false
        Console.WriteLine(false && true);//false
        Console.WriteLine(false && false);//false

        //|| OR
        Console.WriteLine(true || true);//true
        Console.WriteLine(true || false);//true
        Console.WriteLine(false || true);//true
        Console.WriteLine(false || false);//false

        // ^ : XOR
        Console.WriteLine(true ^ true);//false
        Console.WriteLine(true ^ false);//true
        Console.WriteLine(false ^ true);//true
        Console.WriteLine(false ^ false);//false
    }

}
```

Long Circuit

- Evaluates all conditions, even if the result is already determined.
- **Long Circuit:** Always evaluates all conditions, even if it's not needed



```
internal class Program
{
    static void Main(string[] args)
    {
        var value1 = true | evaluated();//evaluted done

        var value2 = true || evaluated();//not print any thing

        Console.ReadKey();
    }

    static bool evaluated()
    {
        Console.WriteLine("evaluted done");
        return true;
    }

}
```

`short circuit high performance comperation with long circuits`

How equality operator (==) works with Reference Types?

we know that when we work with reference types

identifier stored in stack and value stored in heap

when we compare two reference types ⇒ two types stored in deference locations

all time `reference == reference` ⇒ false because its in deference locations

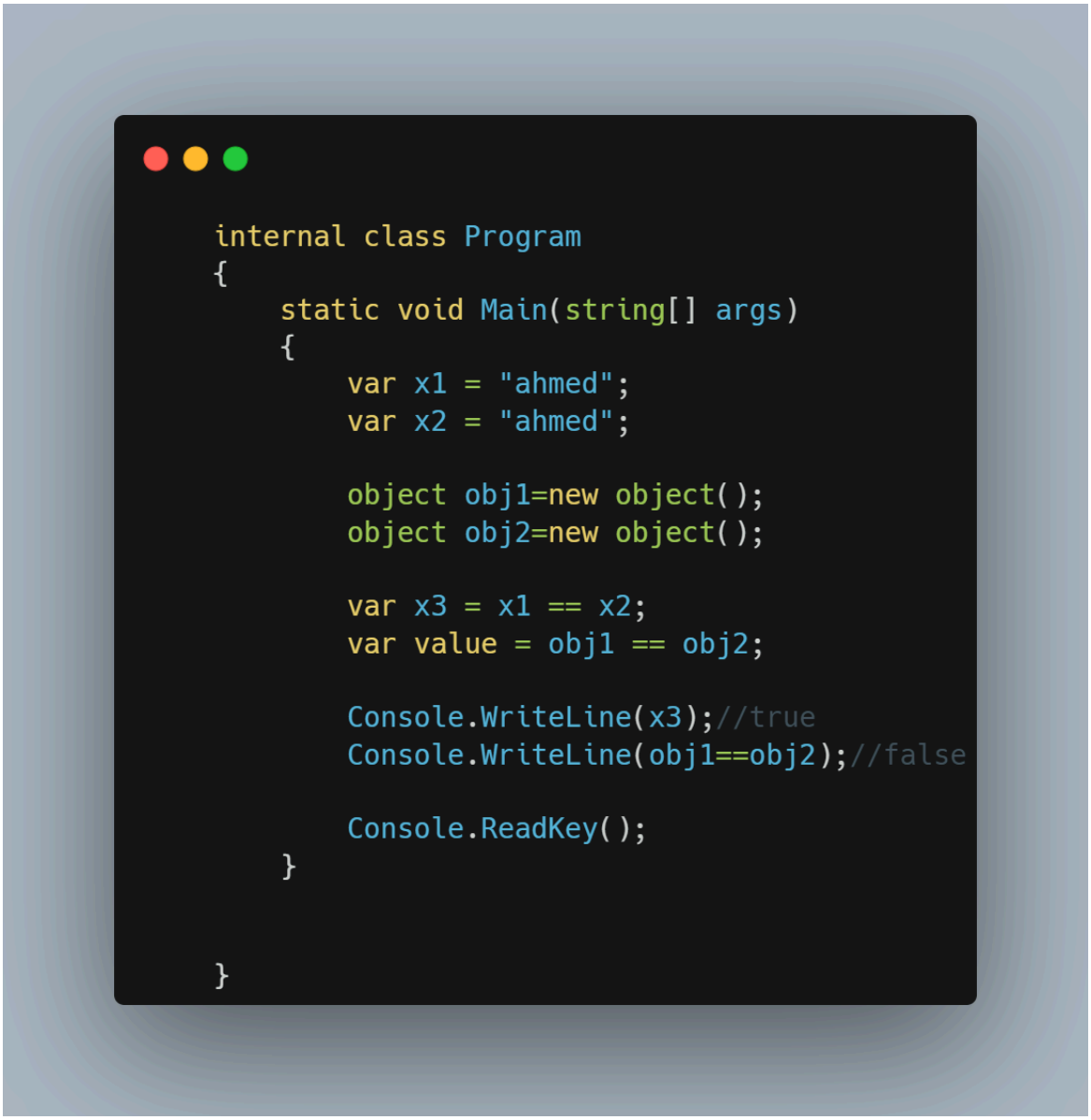
but it can be true by overriding in equals and Hash Code we explain it later

for comparation the content

a specific case with string is when we compare `( string == string )`

CLR internally calls `string.IsEqual()`

it compares the actual content of the strings , not their references.



```
internal class Program
{
    static void Main(string[] args)
    {
        var x1 = "ahmed";
        var x2 = "ahmed";

        object obj1=new object();
        object obj2=new object();

        var x3 = x1 == x2;
        var value = obj1 == obj2;

        Console.WriteLine(x3);//true
        Console.WriteLine(obj1==obj2);//false

        Console.ReadKey();
    }


}
```

What is a ternary operator?

A
ternary operator is a shorthand way to write an `if-else` statement in a single line. It evaluates a condition and returns one of two values based on whether the condition is true or false.

```
int number = 10;
string result = (number > 5) ? "Greater than 5" : "Less than or equal to 5";
```

```
Console.WriteLine(result); // Output: Greater than 5
```

```
int score = 85;
string grade = (score >= 90) ? "A" : (score >= 75) ? "B" : "C";
Console.WriteLine(grade); // Output: B
```

What a Difference between The null coalescing operator (`??`) & The null conditional operator(`?.`) & Null-Forgiving (`!`) ?

- `??` is for **providing a default** value if something is `null`.
- `?.` is for **safe access** to members or methods, returning `null` if the object is `null`.
- `!` is for **suppressing warnings** when you are certain that a nullable type won't be `null` (though it doesn't check for `null` at runtime).

```
using System.Collections;
using System.Collections.ObjectModel;

namespace Null_processes
{
   internal class Program
   {
      static void Main(string[] args)
      {
         // null coalescing operator ??
         string name1 = null;
         string result = name1 ?? "Default Name";

         Console.WriteLine(result);  // Output: Default Name

         //null conditional operator ?
         Person person = null;
            string name2 = person?.Name;

            Console.WriteLine(name2);  // Output: null

         // Null-Forgiving !
```

```
        string? name = null;
        Console.WriteLine(name!.Length);  // This will cause a runtime exception
                                          // (NullReferenceException) if 'name' is null.
    }
  }

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

what is Difference between `if` and `switch` statements?

The
`if` statement is a flexible conditional structure that can be used for a variety of conditions, including comparisons, logical checks, and even complex expressions.

The
`switch` statement is typically used when you have a **single variable** that can be compared against a set of constant values (like `int`, `char`, or `string`). It is more concise and readable for **multiple comparisons** of the same variable.

what is a types of switch statement ?

1. Basic Switch Statement (Traditional)

This is the most commonly used form of the `switch` statement, where you compare a single variable against different constant values.

```
int number = 2;

switch (number)
{
  case 1:
    Console.WriteLine("One");
    break;
  case 2:
```

```
        Console.WriteLine("Two");
        break;
    default:
        Console.WriteLine("Other number");
        break;
}
```

2. Switch with Multiple Case Labels (Fall-through Behavior)

In C#, **fall-through** behavior is not allowed, so each `case` needs to explicitly end with a `break` or another control flow statement (e.g., `return`). However, you can group multiple case labels together

```
int day = 3;
switch (day)
{
case 1:
case 2:
case 3:
Console.WriteLine("Monday to Wednesday");
break;
case 4:
case 5:
Console.WriteLine("Thursday and Friday");
break;
default:
Console.WriteLine("Weekend");
break;
}
```

3.
Switch Expression (C# 8.0 and later)

Introduced in C# 8.0, the
`switch expression` is a more concise way to use `switch`. It allows for returning values based on conditions directly, without the need for multiple lines and `break` statements.

```
int day = 3;

string result = day switch
{
1 ⇒ "Monday",
```

```
2 => "Tuesday",
3 => "Wednesday",
_ => "Invalid day"
};

Console.WriteLine(result);//Wednesday
```

4. Pattern Matching in Switch (C# 7.0 and later)

Introduced in C# 7.0, **pattern matching** allows you to match more complex patterns, such as types and ranges, within a `switch` statement.

Example (Type Pattern Matching):

```
object obj = 42;

switch (obj)
{
    case int i:
        Console.WriteLine($"Integer: {i}");
        break;
    case string s:
        Console.WriteLine($"String: {s}");
        break;
    default:
        Console.WriteLine("Unknown type");
        break;
}
```

Example (Relational Pattern Matching in C# 9.0+):

```
int number = 15;

switch (number)
{
    case int n when (n > 0 && n <= 10):
        Console.WriteLine("Number between 1 and 10");
        break;
    case int n when (n > 10):
        Console.WriteLine("Number greater than 10");
        break;
```

```
    default:
        Console.WriteLine("Non-positive number");
        break;
}
```

5. Switch with Tuple Matching (C# 7.0 and later)

Introduced in C# 7.0, **tuple matching** allows matching on multiple values in a single `switch` statement.

```
(int, int) point = (2, 3);

switch (point)
{
    case (0, 0):
        Console.WriteLine("Origin");
        break;
    case (var x, 0):
        Console.WriteLine($"X-axis, x = {x}");
        break;
    case (0, var y):
        Console.WriteLine($"Y-axis, y = {y}");
        break;
    default:
        Console.WriteLine("Point is somewhere else");
        break;
}
```

what is the Difference between `while` , `do while` and `for` ?

- `while`: Condition is checked before the loop starts. The loop may not run if the condition is false at the beginning.

```
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);
    i++;
}
```

- `do-while`: Code runs at least once, then checks the condition after. It will always execute at least one iteration.

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while (i < 5);
```

- `for`: Best when you know the number of iterations in advance, as it combines initialization, condition check, and increment/decrement in one line.

```
for (int i = 0; i < 5; i++)
{
    Console.Write(i+" ");
}
//0 1 2 3 4
```

```
    //Fibonacci sequence
    for (int counter = 0,prev=0,current=1; counter < 10; counter++)
    {
        Console.Write(prev+" ");
        var newfib = current + prev;
        prev = current;
        current = newfib;

    }//0 1 1 2 3 5 8 13 21 34
```

What is a types of Jump statements ?

- `break`: Exit loop or switch.

```
for (int i = 0; i < 5; i++)
{
    if (i == 3)
        break;  // Exits the loop when i is 3
```

```
    Console.WriteLine(i);
}
```

- `continue`: Skip the current iteration and continue to the next one.

```
for (int i = 0; i < 5; i++)
{
    if (i == 3)
        continue;  // Skips the iteration when i is 3
    Console.WriteLine(i);
}
```

- `return`: Exit method and optionally return a value.

```
int Add(int a, int b)
{
    return a + b;  // Returns the sum and exits the method
}
```

- `goto`: Jump to a specific label (use with caution).

```
int i = 0;
startLoop:
    if (i == 5)
        goto endLoop;  // Jumps to the endLoop label
    Console.WriteLine(i);
    i++;
    goto startLoop;  // Jumps to the startLoop label
endLoop:
    Console.WriteLine("End of loop");
```

What is a types of Conversions ?

- **Implicit conversion**: Happens automatically when there's no risk of data loss (e.g., from `int` to `double`).

```
int a = 10;
double b = a;  // Implicit conversion from int to double
Console.WriteLine(b);  // Output: 10.0
```

- **Explicit conversion**: You use casting or the `Convert` class when converting types that may lose data (e.g., from `double` to `int`).

```
double a = 10.5;
int b = (int)a;  // Explicit conversion (casting) from double to int
Console.WriteLine(b);  // Output: 10
```

What is the Difference between Boxing and unboxing?

- **Boxing**: Value type → Reference type (`object`).
- **Unboxing**: Reference type (`object`) → Value type.

```
using System;

class Program
{
   static void Main()
   {
      int num = 10;         // Value type
      object boxed = num;    // Boxing
      Console.WriteLine(boxed);  // Output: 10
       //here move from stack to heap
       //it easy because heap is larger than stack
      int unboxed = (int)boxed;  // Unboxing
      Console.WriteLine(unboxed);  // Output: 10
      //here move from heap(unlimited) to stack(limited)

   }
}
```

what is the Difference ways for convert from string to another type ?

1. Parse

- :Converts a string into a specific data type (`int` , `double`).
- Throws an exception if the string cannot be converted.

```
string number = "123";
int result = int.Parse(number);  // Converts string to int
Console.WriteLine(result);       // Output: 123
```

```
// Invalid input:
string invalid = "abc";
int errorResult = int.Parse(invalid);  // Throws FormatException
```

2. `TryParse`

- Safely tries to convert a string into a specific data type.
- Returns a `bool` indicating success or failure, and avoids exceptions.

```
string number = "123";
bool success = int.TryParse(number, out int result);
Console.WriteLine(success);  // Output: True
Console.WriteLine(result);   // Output: 123


        string v = "123";

        if(int.TryParse(v, out int y))
        {
           Console.WriteLine(y);//done
        }
        else
        {
           Console.WriteLine("can not converting");
        }


// Invalid input:
string invalid = "abc";
success = int.TryParse(invalid, out result);
Console.WriteLine(success);  // Output: False
Console.WriteLine(result);   // Output: 0 (default value for int)
```

3. `Convert` Class

- Provides methods to convert between different types (e.g., `string` to `int`, `double` to `string`).
- Handles `null` gracefully (returns `0` for numeric types) but throws an exception for invalid formats.

```
string number = "123";
int result = Convert.ToInt32(number);  // Converts string to int
Console.WriteLine(result);           // Output: 123

// Null input:
string nullValue = null;
int nullResult = Convert.ToInt32(nullValue);  // No exception, returns 0
Console.WriteLine(nullResult);              // Output: 0

// Invalid input:
string invalid = "abc";
int errorResult = Convert.ToInt32(invalid);  // Throws FormatException
```

What is a Bit Converter any why we use it?

`BitConverter` is a class in C# that converts **primitive data types** (like integers, floats, etc.) to **byte arrays** and vice versa.

- **Binary Data Handling**:
  - To work with raw binary data in files or memory.

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        int number = 12345;
        byte[] bytes = BitConverter.GetBytes(number);

        // Save the byte array to a binary file
        File.WriteAllBytes("data.bin", bytes);

        // Read the binary file back into a byte array
        byte[] readBytes = File.ReadAllBytes("data.bin");

        // Convert the byte array back to an integer
        int restoredNumber = BitConverter.ToInt32(readBytes, 0);
        Console.WriteLine(restoredNumber);  // Output: 12345
```

```
    }
}
```

- **Data Transmission**:
  - To send data over a network in byte form.

```
using System;
using System.Net;
using System.Net.Sockets;

class Program
{
    static void Main()
    {
        int number = 12345;
        byte[] bytes = BitConverter.GetBytes(number);

        // Sending data over a UDP socket
        UdpClient udpClient = new UdpClient();
        udpClient.Send(bytes, bytes.Length, "127.0.0.1", 8080);
        Console.WriteLine("Data sent over network.");
    }
}
```

- Encoding/Decoding:
  - To interact with hardware or systems that require data in binary format.

```
using System;

class Program
{
    static void Main()
    {
        float temperature = 36.5f;
        byte[] bytes = BitConverter.GetBytes(temperature);

        // Simulating hardware interaction: Send byte array to a device
        Console.WriteLine("Data sent to hardware: " + BitConverter.ToString(bytes));

        // Receiving data back from hardware (simulate response)
        float receivedTemp = BitConverter.ToSingle(bytes, 0);
        Console.WriteLine("Data received from hardware: " + receivedTemp);
```

```
        // Output: 36.5
    }
}
```

Starting in OOP

What Is OOP ?

Object-oriented programming (OOP) is a programming paradigm that organizes software design based on the concept of "objects", which are instances of classes to create modular, reusable, and maintainable code.

Why OOP?

OOP (Object-Oriented Programming) helps in:

1. **Modularity**: Organizes code into self-contained objects.
2. **Reusability**: Allows code (classes/methods) to be reused across projects.
3. **Flexibility**: Supports polymorphism for flexible code.
4. **Easy Maintenance**: Simplifies updates and reduces code breakage.
5. **Real-World Modeling**: Models real-world entities intuitively.

What is Class and Object ?

A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) that objects created from it will have.

An object is an instance of a class. It is a specific entity that has actual values for the properties and can perform the behaviors defined in the class.

```

```

```
namespace OOP
{
    internal class Program
    {
        static void Main(string[] args)
        {

            Employee employee = new Employee();
        }

    }
    class Employee
    {
        public int Id { get; set; }
    }

}
```

What is the Difference between Instance and Static ?

Instance:

- An **instance** refers to a specific object created from a class.
- Instance members (fields, properties, methods) belong to individual objects, meaning each object has its own copy of these members.
- Instance members are accessed using an object of the class

Static:

- **Static** members belong to the **class** itself, not to any specific instance of the class.
- They are shared across all instances of the class.
- Static members are accessed using the class name, not an object.

```
internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        Console.WriteLine(employee.Id);
        Console.WriteLine(Employee.v);
```

```
    }

}
class Employee
{
    public static int v = 5;
    public int Id = 1;
}
```

What is the Difference between Method and Function?

- **Method**:
  - A **method** is a function that is **associated with an object** or **class** in object-oriented programming (OOP).
  - Methods typically operate on the data (fields or properties) of the object or class.

  Example:

```
class Car
{
    public void Start()  // Method
    {
        Console.WriteLine("The car is starting...");
    }
}

Car myCar = new Car();
myCar.Start();  // Calling the method
```

- **Function**:
  - A **function** is a **block of code** that performs a specific task and **returns a value**.
  - Functions are not necessarily associated with objects or classes (though in OOP, they can b

  Example:

```
int Add(int a, int b)  // Function
{
    return a + b;
}
```

```
int result = Add(3, 4);  // Calling the function
```

What is Difference Between ref , on and parms Keywords?

1. `ref`:

- The `ref` keyword is used to pass a parameter **by reference** to a method.
- The variable must be **initialized before** passing it to the method.
- Changes made to the parameter inside the method are reflected outside the method.

```
static void Main(string[] args)
{
   int y = 10;
   test(ref y);//20
   Console.WriteLine(y);//20
}

static void test(ref int x)
{
   x = x + 10;
   Console.WriteLine(x);
}
```

2. `out`:

- The `out` keyword is also used to pass a parameter **by reference**.
- The variable **does not need to be initialized** before passing it.
- The method is required to **assign a value** to the `out` parameter before it finishes.

```
static void Main(string[] args)
{
   int y;
   test( out y);//10
   Console.WriteLine(y);//10
}

static void test(out int x)
{
   x =10;
```

```
    Console.WriteLine(x);
}
```

3. `params`:

- The `params` keyword allows passing **a variable number of arguments** to a method as an array.
- It should be used only for the **last parameter** in a method signature.
- You can pass **zero or more** argument

```
static void Main(string[] args)
{

    test(2,8,7,6,9,11);//2 8 7 6 9 11
    test();//nothing

}

static void test(params int[] nums)
{
    foreach (var n in nums)
    {
        Console.Write(n+" ");
    }
}
```

What is an Expression Bodied Method ?

An
expression-bodied method is a method in C# that has a single expression and is written in a concise form using the `⇒` syntax, instead of using curly braces and a `return` statement.

```
static void Main(string[] args)
{
    if (IsEven(146))
        Console.WriteLine("even");
    else
        Console.WriteLine("ood");
```

```
    //output even
}


static bool IsEven(int n) ⇒ n % 2 == 0;
```

What is Difference between Constructor and Distractor ?

A constructor is a special method in a class that is called when an object of the class is created. Its primary purpose is to initialize the object's state (set values to the fields or properties of the class).

A destructor is a special method in a class that is called when an object is destroyed or goes out of scope. Its main purpose is to clean up resources (like closing file handles or releasing unmanaged memory) that the object was using before it is garbage collected.

```
class Car
{
    public string Model;

    // Destructor
    ~Car()
    {
        Console.WriteLine("Destructor called for " + Model);
    }
}

class Program
{
    static void Main()
    {
        Car myCar = new Car();
        myCar.Model = "Toyota";

        // The destructor will be called when the object is destroyed,
        //usually when it goes out of scope.
    }
}
```

what is a types of Constructor ?

- **Default Constructor**: No parameters, automatically provides default values.

```
class Car
{
    public string Model;
    public int Year;

    // Default Constructor (automatically provided if no other constructor is defined)
    public Car()
    {
        Model = "Unknown";
        Year = 2020;
    }
}
```

- **Parameterized Constructor**: Takes parameters to initialize an object with specific values.

```
class Car
{
    public string Model;
    public int Year;

    // Parameterized Constructor
    public Car(string model, int year)
    {
        Model = model;
        Year = year;
    }
}
```

- **Static Constructor**: Initializes static members, called once for the class.

```
class Car
{
    public static int CarCount;

    // Static Constructor
    static Car()
    {
        CarCount = 0;
        Console.WriteLine("Static Constructor called");
```

```
    }
}
```

- **Copy Constructor**: Creates a new object as a copy of an existing object.

```
class Car
{
    public string Model;
    public int Year;

    // Copy Constructor
    public Car(Car other)
    {
        Model = other.Model;
        Year = other.Year;
    }
}
```

- **Private Constructor**: Restricts object creation, often used for singletons or to control instantiation.

```
class Car
{
    public string Model;
    public int Year;

    // Private Constructor
    private Car()
    {
        Model = "Unknown";
        Year = 2020;
    }

    // Static method to control object creation
    public static Car CreateCar()
    {
        return new Car();
    }
}
```

what is `this` keyword ?

The `this` **keyword** refers to the **current instance** of the class. It is used to:

1. **Access instance members** (fields, methods, properties) of the class.
2. **Differentiate between class fields and method parameters when they have the same name.**
3. **Pass the current instance as a parameter** to other methods or constructors.
4. **Invoke another constructor** in the same class.

Example :

```
csharp
CopyEdit
public class Person
{
    private string name;

    public Person(string name)
    {
        this.name = name; // Refers to the class field 'name'
    }
}
```

What is OOP PILLARS/Features ?

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

What is a Encapsulation ?

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It involves bundling the data (attributes or properties) and the methods (functions or behaviors) that operate on the data into a single unit, called a class.

What is a Difference between Field and Constant?

A
field is a variable that is declared inside a class or struct to store data. It represents an object's state or attribute and is typically used to hold information related to the object.
can not put an initial value and
can be modified after they are initialized.

Constant: A fixed value that cannot change once initialized and is shared across all instances of the class.

```
//Field
//access modifier + Datatype field_name  = intial_value
class Employee
{
private id=5;//Field
public string Name ="ahmed";//Fied
private salary; //field


//constant
public const int x = 55; //constant

}
static void Main(string[] args)
{
// x=44; error
}
```

Encapsulation

What are the Problems Before Encapsulation?

- **Data Inconsistency**:
  - Without encapsulation, data could be directly accessed and modified by any code, leading to invalid or inconsistent states.
- **Lack of Control**:

    - There was no control over how fields were accessed or modified, making it difficult to enforce rules or restrictions.
- **Poor Security**:
    - Sensitive data was exposed and could be easily misused by other parts of the program.
- **Difficult Maintenance**:
    - Code became harder to maintain as changes in data structure could impact multiple areas.
- **No Abstraction**:
    - Details of how data was stored and handled were exposed, breaking the principle of hiding implementation details.

What is an access modifiers ?

Access modifiers are keywords in C# that define the **scope** or **visibility** of a class, its members (fields, methods, properties, etc.), and other programming constructs. They control which parts of the program can access or modify the code elements.

- **Public**: Accessible from anywhere in the program.
- **Private**: Accessible only within the declaring class.
- **Protected**: Accessible within the declaring class and its derived classes.
- **Internal**: Accessible within the same assembly/project.
- **Protected Internal**: Accessible within the same assembly and from derived classes.
- **Private Protected**: Accessible within the declaring class and its derived classes in the same assembly

What is a properties and types ?

Properties are
specialized class members used to encapsulate fields, providing a way to **get** or **set** their values while maintaining control over access and validation.

Types :

1. **Full Property** : provides encapsulation of a field with additional logic, such as validation or calculation, in the getter and setter methods.

```
class Employee
{
   private const float TAX = 68.4f;
   private float salary;
   public float Salary
   {
      set
      {
         salary = value-TAX; //validation
      }
      get
      {
         return salary;
      }
   }
}
```

2. **Automatic Property:** simplify property declarations by allowing The compiler generates a private, anonymous backing field(Hidden Private Attribute) that can only be accessed through the property's get and set accessors.

```
internal class Program
{
   static void Main(string[] args)
   {
      Employee employee = new Employee();
      employee.Salary = 8000;

   }

}

class Employee
{

   public float Salary { get;set; }

}
```

3. **Indexer** :is special property

Named always with keyword this,

Can take parameter

Allows an object to be indexed like an array. Used when abject contain arrays
It is useful for classes that encapsulate collections or other array-like structures.

```
internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        for (int i = 0; i < 5; i++)
        {
            employee[i] = i+10;
        }

        Console.WriteLine(employee[3]);//13
    }

}

class Employee
{
    private int[] nums=new int[5];
    public int this[int index]
    {
        set
        {
            nums[index] = value;
        }
        get
        {
            return nums[index];
        }
    }

}
```

Abstraction

what is Abstraction

Abstraction is an

OOP principle that hides the internal implementation details of an object and only shows the necessary features to the user. It focuses on **what an object does** rather than **how it does it we can done abstraction by interface and abstract class**

What is Difference between interface and abstract class?

Interface:

An **interface** defines a contract that classes must follow, containing only method and property declarations without implementation.

```
// Defining an interface
public interface IDriveable
{
    void Start();  // Method signature
    void Stop();   // Method signature
}

// Implementing the interface in a class
public class Car : IDriveable
{
    public void Start()  // Implementing Start method
    {
        Console.WriteLine("Car is starting.");
    }

    public void Stop()   // Implementing Stop method
    {
        Console.WriteLine("Car is stopping.");
    }
}

public class Bike : IDriveable
{
    public void Start()  // Implementing Start method
    {
        Console.WriteLine("Bike is starting.");
    }

    public void Stop()   // Implementing Stop method
    {
```

```
        Console.WriteLine("Bike is stopping.");
    }
}

// Usage
class Program
{
    static void Main()
    {
        IDriveable car = new Car();
        car.Start();
        car.Stop();

        IDriveable bike = new Bike();
        bike.Start();
        bike.Stop();
    }
}
```

Abstract Class:

An **abstract class** is a base class that can include both implemented (concrete) and unimplemented (abstract) methods, used for inheritance in related classes.

```
// Defining an abstract class
public abstract class Animal
{
    public string Name { get; set; }

    // Concrete method
    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }

    // Abstract method (must be implemented in derived class)
    public abstract void MakeSound();
}

// Implementing the abstract class in a subclass
public class Dog : Animal
```

```
{
    public Dog(string name)
    {
        Name = name;
    }

    public override void MakeSound()  // Implementing abstract method
    {
        Console.WriteLine($"{Name} says Woof!");
    }
}

public class Cat : Animal
{
    public Cat(string name)
    {
        Name = name;
    }

    public override void MakeSound()  // Implementing abstract method
    {
        Console.WriteLine($"{Name} says Meow!");
    }
}

// Usage
class Program
{
    static void Main()
    {
        Animal dog = new Dog("Buddy");
        dog.Eat();
        dog.MakeSound();

        Animal cat = new Cat("Whiskers");
        cat.Eat();
        cat.MakeSound();
    }
}
```

Feature	Interface	Abstract Class
Methods	Only contains method definitions (no implementation, except default methods in modern C#)	Can contain both abstract methods (no implementation) and concrete methods (with implementation).
Properties	Can define properties without implementation.	Can have properties with or without implementation.
Fields	Cannot have fields (no storage of data).	Can have fields to store data.
Events	Can define events.	Can define and implement events.
Constructors	Cannot have constructors.	Can have constructors.
Access Modifiers	Methods and properties are implicitly `public` (cannot specify access modifiers).	Can use access modifiers (e.g., `private`, `protected`, `public`).
Inheritance	Supports multiple inheritance (a class can implement multiple interfaces).	Supports single inheritance (a class can inherit only one abstract class).
Purpose	Used to define a contract or behavior that implementing classes must follow.	Used to provide both shared implementation and a base structure for derived classes.
Default Methods	Can contain default method implementations (starting from C# 8.0).	Can contain implemented methods directly.
Fields Accessibility	Not allowed.	Fields can be `private`, `protected`, or `public`.

Inheritance

what is Inheritance?

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (called the derived or child class) to inherit properties and behaviors (methods) from an existing class (called the base or parent class). This mechanism promotes code reusability, modularity, and the organization of code in a hierarchical manner.

Why we use Inheritance?

1. Reusability (قابليه الاستخدام)
2. maintainability(قابلية الصيانه)
3. Extensibility(قابليه التمدد)

How we can apply Inheritance?

by using Type + : + inherit type

```
internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        employee.print();//hello from manger
        //can use only the contains of parent class that is
    //public,internal and protected access modifiers
    // but contains of class to reuse it in another class like main
    // must be public or internal

    }

}
public class Employee : manger { }
public class manger
{
    public void print() ⇒Console.WriteLine("hello from manger");
}
```

}

What is a Restrictions of Inheritance that can use Inheritance with it?

1. can only inherit from one class
2. can not inherit from sealed class

what is sealed keyword?

the class or method that apply sealed it can not do updates in it

1. **class sealed not support inheritance**

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

```
}

// This will cause a compile-time error:
// class Child : Parent {}
```

2. **method sealed not support override**

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

class DerivedClass : BaseClass {
    public sealed override void Display() {
        Console.WriteLine("DerivedClass Sealed Display Method");
    }
}

// Further overriding will cause a compile-time error:
// class AnotherClass : DerivedClass {
//     public override void Display() { } // Error: Cannot override sealed method
// }
```

What is Difference between Upcasting and Downcasting?

1. **Upcasting** : create a base class reference from subclass reference

```
internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        manger m = employee; // here the same of new manger();
        m.print();//only see this

    }

}
public class Employee : manger
{
    public void copy() => Console.WriteLine("hello from Employee");
}
```

```
public  class manger
{
   public void print() ⇒Console.WriteLine("hello from manger");
}
```

2. **Downcasting:** create subclass reference from base class reference

```
   internal class Program
   {
      static void Main(string[] args)
      {
         manger m =new manger();

         Employee employee =(Employee) m;

         employee.print();
         employee.copy();

      }

   }
   public class Employee : manger
   {
      public void copy() ⇒ Console.WriteLine("hello from Employee");
   }
   public  class manger
   {
      public void print() ⇒Console.WriteLine("hello from manger");
   }
```

What is `as` Keyword?

it try to convert from type to another and if it could not to cast return null

```
static void Main(string[] args)
{
  Employee1 employee1 = new Employee1();

    manger m = employee1;
```

```
    Employee2 employee2 = (Employee2)m; //Exeption

    Employee2 _employee2 = m as Employee2; // casting done

    if (_employee2 == null)  //true
        Console.WriteLine("object is null");
    //output : object is null

}
```

What is a Types of Inheritance

- **Single-Level Inheritance**:

  A single child class inherits from a single parent class.

- **Hierarchical Inheritance**:

  Multiple child classes inherit from the same parent class.

- **Multi-Level Inheritance**:

  A class inherits from a parent class, and another class inherits from that child class.

- **Multiple Inheritance**:

  A class inherits from multiple parent classes. *(Not directly supported in C#, but achievable through interfaces.)*

- **Hybrid Inheritance**:

  A combination of two or more inheritance types, such as hierarchical and multi-level inheritance. *(Not directly supported in C#, but achievable through interfaces.)*

Polymorphism

what is Polymorphism ?

Polymorphism is composed of two words - “poly” which means “many”, and “morph”which means “forms”.
Therefore Polymorphism refers to something that has many shapes.

This pillar makes the same entities such as functions, and operators perform differently in different scenarios. You can perform the same task in different ways.

what is types of between Polymorphism ?

1. **run time polymorphism(Overriding)**

Method Overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class. using two keyword New and Override

```
internal class Program
{
   static void Main(string[] args)
   {
     Employee employee = new Employee();
      employee.print();
      //output
      //hello from manger
      //hello from manger
   }

}
public class Employee : manger
{

   public override void print()
   {
      base.print();// print ⇒ "hello from manger"
      Console.WriteLine("hello from manger");

   }

}

public  class manger
{
   public virtual void print() ⇒Console.WriteLine("hello from manger");
}
```

2. **Compile time polymorphism (overloading)**

Overloading is a compile-time polymorphism feature in which an entity has multiple implementations with the same name. Overloading enhances the flexibility and readability of code by allowing the same method name to be used for different tasks depending on the input parameters.

Types of overloading

1. Method overloading

is the ability to define multiple methods with the same name but different parameter types or a different number of parameters within the same class. The compiler determines which method to call based on the arguments passed during the method call

```
public class Employee
{

    public void print(int a,double y)
    {

    }

    public void print( double y, int a)
    {

    }

    public void print(double y, int a,string c)
    {

    }
}
```

2. Constructor overloading

is the ability to define multiple constructors with the same name but different parameter lists in a class. This allows you to create objects of the class in different ways, depending on the arguments provided during object creation.

```
class Person {
    public string Name;
    public int Age;

    // Constructor 1
```

```
    public Person(string name) {
        Name = name;
        Age = 0;  // Default value
    }

    // Constructor 2
    public Person(string name, int age) {
        Name = name;
        Age = age;
    }
}

class Program {
    static void Main() {
        Person p1 = new Person("Alice"); // Calls Constructor 1
        Person p2 = new Person("Bob", 25); // Calls Constructor 2
    }
}
```

3. **Operator overloading**

is the ability to redefine or overload existing operators (such as `+`, `-`, `*`, `==`, etc.) for user-defined types (classes or structs). It allows you to specify custom behavior when operators are used with objects of your class.

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

    // Overloading the "+" operator to add two Point objects
    public static Point operator +(Point p1, Point p2) {
        return new Point { X = p1.X + p2.X, Y = p1.Y + p2.Y };
    }

    // To display the Point object
    public override string ToString() {
        return $"({X}, {Y})";
    }
}

class Program {
    static void Main() {
        Point point1 = new Point { X = 5, Y = 10 };
```

```
        Point point2 = new Point { X = 3, Y = 7 };

        Point result = point1 + point2;  // Calls the overloaded "+" operator
        Console.WriteLine(result);  // Output: (8, 17)
    }
}
```

4. **Casting operator overloading**

allows you to define how explicit type casting is performed between custom types (classes or structs) in C#. This can be done by overloading the `explicit` or `implicit` casting operators, enabling objects of one type to be converted to another in a custom manner

```
using System;

class Dollar {
    public double Amount { get; set; }

    public Dollar(double amount) {
        Amount = amount;
    }

    // Overloading implicit cast from Dollar to double
    public static implicit operator double(Dollar d) {
        return d.Amount;
    }

    // Overloading explicit cast from double to Dollar
    public static explicit operator Dollar(double amount) {
        return new Dollar(amount);
    }
}

class Program {
    static void Main() {
        Dollar dollar = new Dollar(100);

        // Implicit cast from Dollar to double
        double value = dollar;
        Console.WriteLine($"Dollar to double: {value}");  // Output: 100

        // Explicit cast from double to Dollar
```

```
        Dollar newDollar = (Dollar) 50.75;
        Console.WriteLine($"Double to Dollar: {newDollar.Amount}");  // Output: 50.75
    }
}
```

```
    internal class Program
    {
        static void Main(string[] args)
        {
            Employee employee = new Employee(80);
            test ts = employee;
            int x = employee;
        }

    }


    class Employee
    {
        public int n;
        public Employee(int n)
        {
            this.n = n;
        }

        public static implicit operator test (Employee e)
        {
            return new test();
        }

        public static implicit operator int(Employee e)
        {
            return e.n;
        }
    }

    class test
    {
```

```
}
```

```
internal class Program
{
    static void Main(string[] args)
    {
        Employee emp = (Employee)122;

        Console.WriteLine(emp.n); //n
    }

}


class Employee
{
    public int n;
    public Employee(int n)
    {
        this.n = n;
    }

    public static explicit operator Employee(int n)
    {
        return new Employee(n);
    }
}
```

5. **Indexer overloading**

define custom behavior for indexing operations on objects of a class or struct. Indexers enable instances of a class or struct to be accessed like arrays using square brackets (`[]`), and overloading them allows you to define how those indexing operations should behave based on different parameters or conditions.

```
using System;

class MyCollection {
    private int[] arr = new int[10];

    // Basic indexer: Access by single index
    public int this[int index] {
        get {
            return arr[index];
        }
        set {
            arr[index] = value;
        }
    }

    // Overloaded indexer: Access by range (start and end index)
    public int this[int start, int end] {
        get {
            return arr[start] + arr[end];  // Example: returning sum
            //of elements at two indices
        }
    }
}

class Program {
    static void Main() {
        MyCollection collection = new MyCollection();

        // Using basic indexer
        collection[0] = 10;
        collection[1] = 20;
        Console.WriteLine(collection[0]);  // Output: 10
        Console.WriteLine(collection[1]);  // Output: 20

        // Using overloaded indexer
        collection[2] = 30;
        collection[3] = 40;
        Console.WriteLine(collection[2, 3]);  // Output: 70 (sum of 30 and 40)
    }
}
```

What is Difference between class and struct?

Feature	Class	Struct
Type	Reference type	Value type
Memory	Stored on the heap	Stored on the stack
Inheritance	Supports inheritance	Does not support inheritance
Constructor	Can have a default constructor	Always has a default constructor
Nullability	Can be `null`	Cannot be `null` (unless nullable)
Boxing/Unboxing	Not applicable	Needs boxing/unboxing when used as `object`
Performance	Slower (due to heap allocation and GC)	Faster (stack allocation, no GC)
Equality	Reference-based equality	Value-based equality

How can use base Keyword in Inheritance ?

```
using System;

class Animal
{
    public Animal(string name)
    {
        Console.WriteLine($"Animal constructor called: {name}");
    }
}

class Dog : Animal
{
    public Dog(string name) : base(name)  // Calling the base class constructor
    {
        Console.WriteLine($"Dog constructor called: {name}");
    }
}

class Program
{
    static void Main()
    {
        Dog dog = new Dog("Buddy");  // This will call the base
        // constructor first, then the derived constructor
```

```
    }
}
```

What is Enum?

An E**num** (short for "enumeration") is a special data type in programming that defines a set of named values, often representing a collection of related constants. It is used to represent a variable that can take one of a limited set of values, making code more readable and easier to maintain.

what is a types of Enum?

1. Simple Enum

```
enum Days
{
    Sunday,   // 0
    Monday,   // 1
    Tuesday,  // 2
    Wednesday, // 3
    Thursday, // 4
    Friday,   // 5
    Saturday  // 6
}
Console.Write(Days.Friday);//Friday
Console.Write((int)Days.Friday);//5
```

2. Flag Enum

```
    internal class Program
    {
        static void Main(string[] args)
        {
            var day = Days.Friday | Days.Saturday;
           if(day==Days.WEEKEND)
            {
                Console.WriteLine("this is a Free Days");
            }
        }

    }
```

```
[Flags]
enum Days
{
    Sunday = 0b_0000_0000,
    Monday = 0b_0000_0001,
    Tuesday = 0b_0000_0010,
    Wednesday = 0b_0000_0100,
    Thursday = 0b_0000_1000,
    Friday = 0b_0001_0000,
    Saturday = 0b_0010_0000,
    WEEKEND=Friday | Saturday,//0b_0011_0000

}
```

How can convert from string to Enum?

by use parse or tryparse or check if value is Define in Enum or not By IsDefined

```
using System.Collections;
using System.Collections.ObjectModel;
using System.Net.Sockets;
using System.Threading.Channels;

namespace OOP_Rev
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string val = "Friday";
            if(Enum.TryParse(val,out Days day))
            {
                Console.WriteLine(day);
            }

            if (Enum.IsDefined(typeof(Days),  day))
            {
                Console.WriteLine(day);
            }

            int n = 2;
            if (Enum.IsDefined(typeof(Days), n))
```

```
        {
            Console.WriteLine((Days)n);
        }
        //output
        /*
         Friday
         Friday
         Tuesday
         */
    }

}

enum Days
{
    Sunday ,
    Monday ,
    Tuesday ,
    Wednesday,
    Thursday ,
    Friday ,
    Saturday


}




}
```

Advanced C#

What is Delegate ?

A
delegate is a type that represents a reference to a method. It allows you to store and call methods dynamically, making it possible to pass methods as arguments, return methods from other methods,

or even assign multiple methods to a delegate.

```
using System.Collections;
using System.Collections.ObjectModel;
using System.Net.Sockets;
using System.Threading.Channels;

namespace OOP_Rev
{
   internal class Program
   {
      public  delegate bool Cheker(int v);
      public delegate void saytext();
      static void Main(string[] args)
       {
         Print(IsEven, 14,() => Console.WriteLine("Hello Bro Keep Going"));
         //Hello Bro Keep Going
         //Even Number
       }

      static bool IsEven(int n)
      {
         return n%2==0;
      }

      static void Print(Cheker cheker,int n,saytext saytext)
      {
         saytext();

         if(cheker(n))
            Console.WriteLine("Even Number");
         else
         Console.WriteLine("OOD Number");
      }
   }



}
```

What is Multicast Delegate ?

A
Multicast Delegate is a delegate that can hold and call multiple methods, allowing you to execute several methods with a single delegate invocation.

```
internal class Program
{
   public delegate void Calc(float x, float y);
    static void Main(string[] args)
     {
       Calc calc = GetArea;
       calc += GetPerimeter;

       calc(5, 8);
       //RectangleArea = 40
       //SquertArea = 26

     }


    static void GetArea(float x,float y)
    {
       Console.WriteLine("RectangleArea = "+(x*y));
    }

    static void GetPerimeter(float x,float y)
    {
       Console.WriteLine($"SquertArea =  {(x+y)*2}" );
    }
}
```

What is event?

An **event** in C# is a way to provide a notification or signal to other parts of your application when something happens. It is typically used to handle user actions (like button clicks, key presses, etc.) or system-related actions (like file changes, network status updates,

```
using System;

class Program
{
    // Declare an event using a delegate
    public delegate void Notify();  // Delegate for the event

    public static event Notify OnEventOccurred;  // Event declaration

    static void Main(string[] args)
    {
        // Subscribe (attach event handler) to the event
        OnEventOccurred += EventHandlerMethod;

        // Trigger the event
        Console.WriteLine("Triggering event...");
        OnEventOccurred?.Invoke();  // Using safe call to trigger the event

        // Unsubscribe (detach event handler) from the event
        OnEventOccurred -= EventHandlerMethod;
    }

    // Method that handles the event
    static void EventHandlerMethod()
    {
        Console.WriteLine("Event occurred!");
    }
}
```

Example of File Changes Event

this case can use to tracking File Systems like (SQLite or Microsoft Access)

```
  static void Main(string[] args)
        {
string mypath = @"D:\FolderChanger";

FileSystemWatcher watcher = new FileSystemWatcher(mypath);

watcher.Created += Createdfile;
```

```
watcher.Deleted += DeletedFile;
 watcher.Changed += OnChanged;
watcher.EnableRaisingEvents = true;


Console.ReadKey();

       }

       private static void DeletedFile(object sender, FileSystemEventArgs e)
       {
      Console.WriteLine($"File  {e.Name}  Deleted");
       }

       private static void Createdfile(object sender, FileSystemEventArgs e)
       {
     Console.WriteLine($"File {e.Name} Created");
       }

   private static void OnChanged(object sender, FileSystemEventArgs e)
       {
     Console.WriteLine($"Data Changes");
       }
```

Example for check Network Event

```
  static void Main(string[] args)
       {
NetworkChange.NetworkAvailabilityChanged += NetworkChangerMethod;

Console.WriteLine("Let's Check Network");

Console.ReadKey();
       }

static void NetworkChangerMethod(object sender,NetworkAvailabilityEventArgs e)
       {
if(e.IsAvailable)
{
   Console.WriteLine("conneted");
}
else
```

```
    Console.WriteLine("unconnected");
    }
```

Common Built-in Events in C#

Event	Source Class	Description
Click	Button, LinkButton, MenuItem	Triggered when a button, link, or menu item is clicked by the user.
TextChanged	TextBox, RichTextBox, ComboBox	Triggered when the text in a text box or combo box changes.
KeyDown	Control	Triggered when a key is pressed down while the control has focus.
KeyUp	Control	Triggered when a key is released while the control has focus.
MouseClick	Control, Button, Panel	Triggered when the mouse is clicked on the control.
MouseMove	Control, Panel, PictureBox	Triggered when the mouse is moved over a control.
Resize	Control, Form	Triggered when the control or form is resized.
Load	Form, UserControl	Triggered when a form or control is loaded into memory.
Closing	Form	Triggered when the form is about to close.
Closed	Form	Triggered after a form has been closed.
MouseEnter	Control	Triggered when the mouse pointer enters the control.
MouseLeave	Control	Triggered when the mouse pointer leaves the control.
ValueChanged	TrackBar, NumericUpDown, DateTimePicker	Triggered when the value of the control changes.
CheckedChanged	CheckBox, RadioButton	Triggered when the checked state of a check box or radio button changes.
Scroll	ScrollBar, Panel, TextBox	Triggered when a scroll event occurs (i.e., the user scrolls the control).
FormClosing	Form	Triggered before a form is closed, allowing you to cancel the close operation.
FormClosed	Form	Triggered after a form is closed.
NetworkAvailabilityChanged	NetworkChange	Triggered when the network availability changes (e.g., connected or disconnected).
FileSystemWatcher.Changed	FileSystemWatcher	Triggered when a file or directory is changed.
FileSystemWatcher.Created	FileSystemWatcher	Triggered when a file or directory is created.

Event	Source Class	Description
`FileSystemWatcher.Deleted`	`FileSystemWatcher`	Triggered when a file or directory is deleted.
`FileSystemWatcher.Renamed`	`FileSystemWatcher`	Triggered when a file or directory is renamed.
`UnhandledException`	`AppDomain`	Triggered when an unhandled exception occurs in the application.
`ApplicationExit`	`Application`	Triggered when the application is about to exit.
`ThreadException`	`Application`	Triggered when an unhandled exception occurs on a thread in the application.
`Timer.Elapsed`	`Timer`	Triggered when the timer reaches the specified interval.
`StateChanged`	`Process`	Triggered when the state of a process changes (e.g., running, exited, etc.).
`Exited`	`Process`	Triggered when a process exits.
`NetworkAvailabilityChanged`	`NetworkChange`	Triggered when the network status changes (connected or disconnected).

What is `EventHandeler` ?

`EventHandler` is a built-in **delegate** in C# used to handle events. It defines the method signature for handling an event, which takes two parameters:

- `object sender`: The source of the event (the object that triggered the event).
- `EventArgs e`: Contains additional event information (typically empty if no extra data is needed).

```
using System;

class Program
{
    // تعريف الحدث باستخدام EventHandler مع بيانات مخصصة
    public static event EventHandler<MyEventArgs> MyEvent;

    static void Main(string[] args)
    {
        // الاشتراك في الحدث
        MyEvent += MyEventHandler;

        // تفعيل الحدث
        OnMyEvent("رسالة مخصصة");
```

```
        Console.ReadKey();
    }

    // دالة معالج الحدث
    static void MyEventHandler(object sender, MyEventArgs e)
    {
        Console.WriteLine($"الحدث تم تفعيله! الرسالة: {e.Message}");
    }

    // دالة لتفعيل الحدث
    static void OnMyEvent(string message)
    {
        // تحقق من وجود مستمعين للحدث ثم استدعائه
        MyEvent?.Invoke(null, new MyEventArgs(message));
    }
}

// فئة تحتوي على البيانات المخصصة للحدث
public class MyEventArgs : EventArgs
{
    public string Message { get; }

    public MyEventArgs(string message)
    {
        Message = message;
    }
}
```

What is Generics ?

Generics allow you to write classes, methods, or functions that can work with any data type while keeping the code type-safe. Instead of specifying a concrete type, you use a placeholder (like `T`), and the actual type is provided when the code is used.

1. Generic Class

```
public class Container<T>  // Generic class with placeholder 'T'
{
```

```
    private T item;

    public void SetItem(T value)
    {
        item = value;
    }

    public T GetItem()
    {
        return item;
    }
}


Container<int> intContainer = new Container<int>();
intContainer.SetItem(10);
Console.WriteLine(intContainer.GetItem());  // Output: 10

Container<string> stringContainer = new Container<string>();
stringContainer.SetItem("Hello");
Console.WriteLine(stringContainer.GetItem());  // Output: Hello
```

2. Generic Method

```
public class GenericMethodExample
{
public static T PrintValue<T>(T value)  // Generic method with placeholder 'T'
{
Console.WriteLine(value);
return value;
}
}

GenericMethodExample.PrintValue(123);  // Output: 123
GenericMethodExample.PrintValue("Hello, Generics!");  // Output: Hello, Generics!
```

Why we use Generics?

- **Code Reusability**: Generics allow you to write code that can work with any data type, so you don't have to duplicate code for each type.
- **Type Safety**: With generics, the compiler ensures that you're using the correct data types, reducing runtime errors caused by invalid type casting.
- **Performance**: Generics over come the need for boxing/unboxing (converting types to `object` and back), which can improve performance.
- **Maintainability**: With reusable and type-safe code, it becomes easier to maintain and update the codebase without breaking functionality.
- **Flexibility**: Generics let you create flexible methods, classes, and data structures that work with any type, making your code more adaptable

What is Generic Constrains ?

Generic constraints allow you to impose specific requirements on the types that can be used with a generic class, method, or interface. By using constraints, you can ensure that the type parameter meets certain conditions, such as being a reference type, implementing a particular interface, or having a specific constructor.

```
public class MyClass<T> where T : class
{
// Only reference types can be used as 'T'
T can be
class
struct
new() // Must have a parameterless constructor
SomeClass
Interface
Multiple Constraints ⇒where T : class, IDisposable, new()
}
```

Example:

```
public class MyGenericClass<T> where T : IComparable<T>
{
   public T GetMax(T value1, T value2)
   {
      if (value1.CompareTo(value2) > 0)
         return value1;
```

```
        else
            return value2;
    }
}
```

What is Generic Delegates?

Generic delegates allow you to define a delegate type that works with any data type, similar to how generic classes and methods work. By using generics, you can create flexible and reusable delegate types that can operate with different types, while maintaining type safety.

1. `Action` Delegate

`Action` is a generic delegate that represents a method that **does not return a value**. It can take up to 16 parameters of any type, but it always returns `void`.

```
// Action that takes two parameters and does not return anything.
Action<int, int> addNumbers = (a, b) =>
{
    Console.WriteLine($"Sum: {a + b}");
};

addNumbers(5, 10);  // Output: Sum: 15
```

2.
`Func` **Delegate**

`Func` is a generic delegate that represents a method that **returns a value**. It can take up to 16 parameters, and the last type parameter represents the return type.

```
// Func that takes two integers and returns their sum as an integer.
Func<int, int, int> addNumbers = (a, b) =>
{
    return a + b;
};

int result = addNumbers(5, 10);
Console.WriteLine(result);  // Output: 15
```

3. `Predicate<T>` Delegate:

   The `Predicate<T>` delegate is a built-in delegate type that represents a method that takes a single parameter of type `T` and returns a **boolean value** (`true` or `false`). It's commonly used to test or check if an object meets a specific condition.

```
using System;

class Program
{
    static bool IsEven(int number)
    {
        return number % 2 == 0;
    }

    static void Main()
    {
        // Create a Predicate to test if a number is even
        Predicate<int> isEvenPredicate = new Predicate<int>(IsEven);

        Console.WriteLine(isEvenPredicate(4));  // Output: True
        Console.WriteLine(isEvenPredicate(7));  // Output: False
    }
}
```

Trapshooting

What is a Types of Errors?

1. **Syntax Error**: An error in the code's structure, such as missing punctuation or incorrect keywords, that prevents it from being executed.

2. **Run Time Error(Exception)**: An error that occurs during the execution of a program, often due to invalid operations (dividing by zero or accessing invalid memory).
3. **Logical Error**: A mistake in the program's logic that causes it to produce incorrect results, even though it runs without errors

How Can Solve Run Time Error(Exception)?

1. **Try-Catch Blocks**:

   Use `try` to enclose risky code and `catch` to handle exceptions if they occur.
2. **Finally Block**:

   Use `finally` to execute code that must run regardless of whether an exception occurs.
3. **Validate Inputs**:

   Check inputs or conditions before performing operations to prevent exceptions.
4. **Multiple Catch Blocks**:

   Use multiple `catch` blocks to handle different types of exceptions separately.
5. **Throw Custom Exceptions**:

   Define and throw custom exceptions to handle specific errors in your program.
6. **Log the Error**:

   Log exception details to help diagnose and fix issues.
7. **Use Debugging Tools**:

   Use tools like Visual Studio to identify and fix the source of exceptions.
8. **Fix the Underlying Issue**:

   Analyze the exception message and stack trace to resolve the root cause of the error.

Simple Example of Use Try Catch Finial

```
using System;

class Program {
    static void Main() {
        try {
            Console.WriteLine("Enter a number:");
            int num = int.Parse(Console.ReadLine());
            int result = 10 / num;
            Console.WriteLine("Result: " + result);
        }
```

```
        catch (DivideByZeroException) {
            Console.WriteLine("Error: Division by zero is not allowed.");
        }
        catch (FormatException) {
            Console.WriteLine("Error: Invalid input. Please enter a valid number.");
        }
        catch (Exception e) {
            Console.WriteLine("An unexpected error occurred: " + e.Message);
        }
        finally {
            Console.WriteLine("Program execution complete.");
        }
    }
}
```

How can solve Logical Error ?and Give an Example of Logical Error:

1. Understand the expected output.
2. Review the logic of the code.
3. Add debugging statements to trace the flow.
4. Test with different inputs.
5. Fix the flawed logic.
6. Use comments to explain the logic.
7. Have someone else review the code.
8. Write unit tests to verify correctness.

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = { 10, 20, 30, 40, 50 };
        int sum = 0;

        // Logical Error: Dividing by the wrong number (4 instead of 5)
        for (int i = 0; i < numbers.Length; i++)
```

```
        {
            sum += numbers[i];
        }
        double average = sum / 4; // Incorrect: Should divide by numbers.Length (5)

        Console.WriteLine("Average: " + average); // Output will be incorrect
    }
}
```

What is Difference Between General Exception and Specific Exceptions?

1. General Exception:

- Represents a catch-all for any type of exception.
- Catches all exceptions, regardless of their type.
- Example: `catch (Exception e)`
- Use when you want to handle any unexpected error but may lose specific details about the error.

2. Specific Exceptions:

- Represent particular types of errors (divide by zero, null reference, file not found).
- Catch only the specified type of exception.
- Example: `catch (DivideByZeroException )` or `catch (NullReferenceException )`
- Use when you want to handle a specific error type and provide targeted solutions or messages.

> **use Specific Exceptions: high performance comperation with General Exception**

What is a Exception Filter?

An **exception filter** is a feature in some programming languages (like C#) that allows you to specify a condition within a `catch` block. The `catch` block will only handle the exception if the condition evaluates to `true`. This provides more control over exception handling by allowing you to filter exceptions based on specific criteria

```
int retryCount = 0;
while (true) {
   try {

      break;
   }
   catch (Exception e) when (retryCount < 3) //only start handle Ex if condition true
   {
      retryCount++;
      Console.WriteLine($"Retry attempt {retryCount}");
   }
}
```

Why you should Use `throw` Instead of `throw ex`?

1. `throw;`:
   - Keeps the **original stack trace** (shows where the error really happened).
   - Better for **debugging** because it points to the actual source of the error.
2. `throw ex;`:
   - **Resets the stack trace** (hides where the error originally happened).
   - Makes debugging harder because it points to where you rethrew the error, not the real source.

Always Use `throw;`:

It's the best practice because it keeps the error details intact and makes debugging easier. Use `throw ex;` only in rare cases where you want to hide the original error (not recommended usually).

- `throw;` كما هي، حتى (و الرسالة الأصلية `stack trace` مثل الـ) يحتفظ بكل التفاصيل الأصلية للاستثناء بعد إعادة رمي الاستثناء.
- `throw ex;` و الرسالة، خصوصًا إذا تم التلاعب `stack trace` يمكن أن يغير بعض التفاصيل مثل الـ بالاستثناء قبل رميه، مما قد يؤدي إلى فقدان بعض المعلومات الهامة.

ببساطة:

- **استخدم** `throw;` عندما ترغب في إعادة رمي الاستثناء مع الحفاظ على جميع تفاصيله الأصلية.
- **استخدم** `throw ex;` فقط إذا كنت بحاجة لتعديل الاستثناء قبل رميه (ولكن قد تفقد بعض التفاصيل الأصلية).

What is a custom exception ?

A
custom exception is a user-defined error type that you create by extending the built-in `Exception` class. It allows you to define specific error situations in your application with custom messages or additional information, making error handling more meaningful and easier to understand.

```
using System;

public class NegativeNumberException : Exception
{
    public string Location { get; set; }  // خاصية لتخزين الموقع

    // الكونستركتر الذي يقبل الرسالة والموقع
    public NegativeNumberException(string message, string location)
        : base(message)
    {
        Location = location;  // تخزين الموقع
    }
}

class Program
{
    static void Main()
    {
```

```
        try
        {
            int number = -5;
            if (number < 0)
            {
                // رمي الاستثناء مع الرسالة والموقع
                throw new NegativeNumberException("Negative numbers
                are not allowed.", "Program.Main() method");
            }
        }
        catch (NegativeNumberException ex)
        {
            // طباعة الرسالة والموقع
            Console.WriteLine($"Error: {ex.Message}");
            Console.WriteLine($"Location: {ex.Location}");
        }
    }
}
```

> Output
>
> Error: Negative numbers are not allowed.
> Location: Program .Main() method

How can Compare Content of Reference Types Exception String?

1.
    - **Override** `Equals` **and** `GetHashCode`: To compare the content of objects.

```
        static void Main()
        {
            Employee employee1 = new Employee { Age = 20, Name = "ahmed" };

            Employee employee2 = new Employee { Age = 20, Name = "ahmed" };

            Console.WriteLine(employee1.Equals(employee2));//true

            Console.WriteLine(employee1==employee2);//true
```

```
            //withoutoverriding Equals & GetHashCode result will be false
        }
    }

    class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override bool Equals(object? obj)
    {
        if (obj == null || GetType() != obj.GetType())
            return false;

        Employee other = obj as Employee;
        return other.Name.Equals(Name) && other.Age.Equals(Age);
    }


    public override int GetHashCode()
    {
        return HashCode.Combine(Name, Age);
    }

    //here for use == or !=
    public static bool operator ==(Employee a, Employee b) => a.Equals(b);
    public static bool operator !=(Employee a, Employee b) =>!( a.Equals(b));

}
```

2. **Implement `IEquatable<T>` Interface**

Implementing the `IEquatable<T>` interface allows you to provide a strongly-typed method for comparing objects.

```
class Programme
{

    static void Main()
    {
        Employee employee1 = new Employee { Age = 20, Name = "ahmed" };

        Employee employee2 = new Employee { Age = 20, Name = "ahmed" };
```

```
        Console.WriteLine(employee1.Equals(employee2));//true

        Console.WriteLine(employee1!=employee2);//false
    }
}

class Employee : IEquatable<Employee>
{
    public string Name { get; set; }
    public int Age { get; set; }

    public bool Equals(Employee? other) // implemented method
    {
        if (other == null || GetType() != other.GetType())
            return false;

        Employee employee = other as Employee;
        return Name.Equals(employee.Name) && Age.Equals(Age);
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Name, Age);
    }

    public static bool operator ==(Employee a, Employee b) => a.Equals(b);

    public static bool operator !=(Employee a, Employee b)=>!(a.Equals(b));

}
```

3. Using `IEqualityComparer<T>`

If you don't want to modify the class itself, you can create a custom comparer by implementing the `IEqualityComparer<T>` interface.

```
    class Programme
    {

        static void Main()
        {
```

```
        Employee employee1 = new Employee { Age = 20, Name = "ahmed" };

        Employee employee2 = new Employee { Age = 20, Name = "ahmed" };

        EmployeeComparer comparar= new EmployeeComparer();


        Console.WriteLine(comparar.Equals(employee1,employee2));//True


    }
}

class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class EmployeeComparer : IEqualityComparer<Employee>
{




    public int GetHashCode([DisallowNull] Employee obj)
    {
        return HashCode.Combine(obj.Name, obj.Age);
    }

    public bool Equals(Employee? x, Employee? y)
    {
        if (x == null || y == null)
        {
            return false;
        }

        return x.Name .Equals( y.Name) && x.Age .Equals( y.Age);
    }
```

```
    }
```

What is Yield Keyword and why we use it ?

The `yield` keyword in C# is used to create **iterators** for sequences of data. It allows you to return values one at a time **on-the-fly** without needing to store the entire sequence in memory. When you use `yield return`, the method pauses and resumes execution each time a value is requested, making it efficient for large or infinite sequences. Use `yield break` to stop the iteration early.

```
    static void Main()
    {
      Employee employee=new Employee(1,2,3,4);
       employee[2] = 50;


       foreach (var item in employee)
       {
          Console.Write(item+" ");
       }
       //1 2 50 4
    }
}

class Employee :IEnumerable
{
    int[] nums;

    public int this[int index]
    {
       get { return nums[index]; }
       set
       {
          nums[index] = value;
       }
    }

    public Employee() { }

    public Employee(int n1,int n2,int n3,int n4)
```

```
    {
        nums =new int[]{ n1,n2,n3,n4 };
    }

    public IEnumerator GetEnumerator()
    {
        foreach (var item in nums)
        {
            yield return item;
        }
    }
}
```

What is a `IComparer` ?

The `IComparer` interface in C# is used to provide a way to **compare two objects** of the same type. It is commonly used for **custom sorting** in collections like lists, arrays, or dictionaries when the default comparison behavior is not suitable.

```
        static void Main()
        {
            ArrayList Employees = new ArrayList()
            {
                new Employee(){Name="Ahmed",Age=35},
                new Employee(){Name="Ali",Age=28}

            };

            Employees.Sort(new Sorter());

            foreach (Employee item in Employees)
            {
                Console.WriteLine($"{item}");
            }
            //output
            //Ali ⇒  28
            //Ahmed ⇒  35
        }
    }

    class Employee
```

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

    public override string ToString()
    {
        return $"{Name} ⇒  {Age}";
    }

}

class Sorter : IComparer
{
    public int Compare(object? x, object? y)
    {
        if(x == null || y == null)
            return 0;

      Employee  X = x as Employee;
      Employee  Y = y as Employee;

        return X.Age.CompareTo(Y.Age);

    }

}
```

Why is `for` faster than `foreach` in C#?

1️⃣ `for` **uses direct indexing (** `nums[i]` **), which is very fast.**

2️⃣ `foreach` **creates an "Enumerator" (hidden extra object), which makes it slower.**

3️⃣ `for` **does not check the type of each item, but** `foreach` **does.**

4️⃣ `for` **is optimized by the compiler for better performance.**

5️⃣ **For arrays (** `int[]` **),** `for` **is always faster.**

6️⃣ **For** `List<T>` **, the difference is smaller, but** `for` **can still be better.**

✔️ Use `for` for **arrays** when you need maximum speed.

✔️ Use `foreach` for **Lists and collections** when readability is more important than performance.

What is Difference between `IEnumerable` and `IQueryable` ?

The Common Factor Between `IEnumerable` and `IQueryable`:

Both are **interfaces** in C# used to interact with collections like `List`, `Array`, and others.

Differences Between `IEnumerable` and `IQueryable`:

1. **Deferred Execution**:
   - Both `IQueryable` and `IEnumerable` support **deferred execution**, but `IEnumerable` can also support **immediate execution**.
2. **Immediate Execution**:
   - Immediate execution means the query is executed immediately after the query is completed. The result is stored in memory and can be reused in the same state. This is useful when working with data from the same source, and you want to reflect any changes in the data that occur.
3. **Deferred Execution**:
   - Deferred execution means that the query execution is delayed until an iteration (like a `foreach` loop) is performed. This approach allows for reflecting any updates in the data source (e.g., SQL Server) because the query is executed anew each time the data is accessed.
4. **Types of Deferred Execution**:
   - **Streaming**: This occurs when you don't need to access all the data at once, and you only need specific parts. For example, using `Take` or filtering.
   - **Non-streaming**: This occurs when you need to loop through the entire collection, like when performing sorting or grouping operations.
5. **Examples of Deferred Execution**:
   - `IEnumerable` can perform both types (streaming and non-streaming), but `IQueryable` is typically used for scenarios where complex queries are needed, and it is optimized for external data sources.

IEnumerable vs IQueryable - More Differences:

- `IEnumerable` is **LINQ to Objects**, meaning it works with collections directly in memory.
- `IQueryable` is **out-of-process LINQ**, meaning it can work with external data sources, and the query is converted into an **Expression Tree**, which can be sent to an external LINQ provider (e.g., SQL Server) to be executed.

  The **Expression Tree** allows the query to be translated into something that the external system can understand and process.

when use every one

- **Use** `IEnumerable` when working with small collections or simple queries that don’t require complex operations or optimizations.

- **Use** `IQueryable` when working with large collections or when you need complex queries that require optimization, as it allows for better performance through query translation (e.g., SQL translation) and deferred execution.

> 💡 شرحت الموضوع فالبوست دا قبل كدا
>
> https://www.linkedin.com/posts/ahmed-hany-899a9a321_c-activity-7228678411668373504-EHtq?utm_source=share&utm_medium=member_android

What is Method Chaining?

Method chaining is a programming technique where multiple methods are called on the same object in a single line of code, one after another. Each method returns the object itself (or another object) so that the next method can be called directly on it.

This allows for more **concise** and **readable** code, as multiple actions can be performed on the same object in a single expression.

```
class Program
{

   static  void Main( string[] args)
   {
      Employee employee = new Employee();
      employee.Method1("Hello")
         .Method2("I am ")
         .Method3("BackEnd");
   }

}

class Employee
{

   public Employee Method1(string message)
   {
      Console.WriteLine(message);
      return this;
   }
```

```
    public Employee Method2(string message)
    {
        Console.WriteLine(message);
        return this;
    }
    public Employee Method3(string message)
    {
        Console.WriteLine(message);
        return this;
    }
}
```

What is Extension Method?

An
extension method in C# is a special kind of static method that allows you to add new functionality to an existing class or type without modifying its original code. You define extension methods in a static class, and they can be called like regular instance methods on the type they extend.

```
class Programme
{

    static void Main()
    {
        Console.WriteLine(12.IsEven());//true
        Console.WriteLine((-14).IsPositive());//false
    }


}

static class  Number
{
    public static bool IsEven(this int n )
    {
        return n % 2 == 0;
    }

    public static bool IsPositive(this int n)
```

```
    {
        return n >0;
    }
}
```

What is Assembly?

An
assembly in .NET is a compiled code library used by .NET applications. It contains the code and metadata required for the application or application component to run.

```
class Program
{

    static void Main()
    {
      LibraryB.Method(); // Calls into LibraryB.dll

        Console.WriteLine(typeof(Program).Assembly.FullName);//the same get assemply
    }

    //In the same assembly as Program (, main app):
//typeof(Program).Assembly = GetExecutingAssembly() = GetEntryAssembly()
//In a different assembly ( library):
//typeof(Program).Assembly = GetEntryAssembly() (≠ GetExecutingAssembly())
//So, the specific behavior depends on whether your code is part of the main application or a
}

void Method()
{
Console.WriteLine($"Executing Assembly: {Assembly.GetExecutingAssembly().FullName}");
//Returns the assembly that contains the currently executing code
Console.WriteLine($"Entry Assembly: {Assembly.GetEntryAssembly()?.FullName}");
//Returns the entry assembly (the main executable that started the application).
Console.WriteLine($"Calling Assembly: {Assembly.GetCallingAssembly().FullName}");
//Returns the assembly that invoked the currently executing method.
}
```

How Code Executed?

1. At compile time code with any programming language
2. Code translation to **(EXE/DLL) ⇒ metadata and Intermediate language(IL) = Assembly**

IL:⇒ a language that all codes converts to it before converts to machine language

is the low-level, platform-independent code that is generated when you compile a C# (or any .NET) program. It is not directly executed by the computer's hardware but is instead executed by the .NET runtime (CLR - Common Language Runtime)

Metadata :

is data that describes other data. In the context of .NET assemblies, **metadata** provides information about the structure of the code, such as:

- **Types** (classes, structs, interfaces)
- **Methods** (method names, parameters, return types)
- **Properties and Fields**
- **Attributes** (e.g., security, versioning)

It helps the .NET runtime understand how to work with the code, what each component is, and how they relate to each other. Essentially, it acts like a map or guide to the program's content.

go to Common language Runtime(CLR) ⇒ here we had Mange Code:

In **.NET**, **managed code** runs under the **CLR** and is automatically managed, including memory management. The **Garbage Collector (GC)** automatically reclaims memory from objects that are no longer in use, preventing memory leaks and reducing the need for manual memory management.

3. **go to Runtime**

Just in time compiler(JIT) convert IL to Machine learning

4. **Operating System get a machine code and process it**

here what a OS do :

- **Load the Program**: The OS loads the machine code (usually from disk) into memory. This could involve loading an executable file or a program that was compiled into machine code.
- **Memory Allocation**: The OS allocates memory for the program. It sets up the necessary memory addresses where the program will execute, ensuring that there are no conflicts with other programs or processes running on the system.
- **Process Management**: The OS creates a process for the program. A process is an instance of a program in execution. It sets up process control blocks (PCBs) that track the state of the process, its resources, and its execution status.
- **CPU Scheduling**: The OS's scheduler decides when the program (or its processes/threads) will get CPU time. It could assign the CPU to the program immediately or wait for the appropriate

time based on priority or other scheduling policies.

- **Execution**: The CPU begins to execute the machine code instructions step by step, interpreting the binary instructions. The OS coordinates to ensure resources (like memory, I/O devices, etc.) are properly managed during execution.
- **Input/Output Operations**: If the program requires input from or output to external devices (keyboard, screen, disk, etc.), the OS handles these I/O operations through system calls. The program itself may interact with the OS to request these operations.
- **Error Handling**: The OS monitors the execution for any potential errors (e.g., invalid operations, memory issues) and may handle them by stopping the program or reporting the error to the user.
- **Interrupts**: The OS manages interrupts that may arise from hardware devices (like a keyboard press or network packet arrival) or from the program itself (e.g., system calls or exceptions). Interrupts can cause the OS to pause the current execution and deal with the interrupt before resuming.
- **Multitasking**: If the OS is running multiple programs, it may switch between them in a process called multitasking. The OS will save the current state of the program in its PCB and load the state of another program to execute.
- **Program Termination**: After the program finishes executing, the OS cleans up. This includes deallocating memory, closing files, and removing the process from the scheduler. If the program ends with an error, the OS may display an error message or take corrective action.

What is Deference between `GetType` and `typeof` ?

Feature	`GetType()`	`typeof`
When resolved	Runtime	Compile time
Requires	Object instance	Type name
Reflects	Actual runtime type of the instance	Specified type
Use case	Dynamic type checking (, `obj.GetType() == typeof(Dog)`)	Static type metadata (e.g., reflection)

When to Use:

- `GetType()` : Check the actual type of an object at runtime (handling polymorphic objects).
- `typeof` : Work with type metadata at compile time (reflection, generics)

What is Reflection

Reflection in .NET is a powerful feature that allows code to inspect, modify, and interact with its own structure at runtime. It is facilitated by the classes in the `System.Reflection` namespace, along with the `System.Type` class

💡 ببساطة الكود بعد مدخل مرحلة الCompile Time هيتم تحويلة اليAssembly ازاي بقي نعمل Access at Run Time ?

By using Reflection

What can get from Type ?

- **Type Name**: The name of the type.
- **Namespace**: The namespace where the type is defined.
- **Base Type**: The base type from which the current type inherits.
- **Properties**: Information about the properties of the type.
- **Methods**: Information about the methods of the type.
- **Fields**: Information about the fields of the type.
- **Is Class**: Whether the type is a class.
- **Is Interface**: Whether the type is an interface.
- **Is Abstract**: Whether the type is abstract.
- **Is Sealed**: Whether the type is sealed.
- **Is Public**: Whether the type is public.
- **Is Nested**: Whether the type is nested

```
static void Main(string[] args)
{
  Type type1 = DateTime.Now.GetType();//Type at RunTime
  Type type2 = typeof(DateTime);//Type at Compile Type

   Console.WriteLine(type1.Assembly);//System.Private.CoreLib, Version=9.0.0.0,
                          //Culture=neutral, PublicKeyToken=7cec85d7bea7798e
   Console.WriteLine(type1);//System.DateTime
   Console.WriteLine(type1.FullName);//System.DateTime
   Console.WriteLine(type1.Name);//DateTime
   Console.WriteLine(type1.Namespace);//System
   Console.WriteLine(type1.BaseType);//System.ValueType
   Console.WriteLine(type1.IsValueType);//true
   Console.WriteLine(type1.IsPublic);//true
```

```
    var interfaces= type1.GetInterfaces();
    foreach (var item in interfaces)
    {
        Console.WriteLine(item);
    }
    /*
     System.IComparable
     System.ISpanFormattable
     System.IFormattable
     System.IConvertible
     System.IComparable`1[System.DateTime]
     System.IEquatable`1[System.DateTime]
     System.Runtime.Serialization.ISerializable
     System.ISpanParsable`1[System.DateTime]
     System.IParsable`1[System.DateTime]
     System.IUtf8SpanFormattable
     */


}
```

How can Create Instance By using Reflection?

To create an instance of a type using Reflection in C#, you can use the `Activator.CreateInstance`

```
internal class Program
{


    static void Main(string[] args)
    {
      var instance=Activator.CreateInstance(typeof(DateTime));

        Console.WriteLine(instance);//1/1/0001 12:00:00 AM

        Employee e1 =(Employee) Activator.CreateInstance(typeof(Employee),"Ahmed");
        Console.WriteLine(e1);//Ahmed

    }

}
```

```
    class Employee
    {
        public Employee(string name)
        {
            Name = name;
        }

        public string Name { get; set; }
        public override string ToString()
        {
            return Name;
        }

    }
}
```

How can show members of Type?

```
using System.Reflection;

namespace Reflactions
{
    internal class Program
    {


        static void Main(string[] args)
        {


            Console.WriteLine("MemberInfo");
            MemberInfo[] members = typeof(BankAcount).GetMembers();

            foreach (var member in members)
            {
                Console.WriteLine(member);//here print all members
                //in class except private members
            }

            //to get a spcific members use flag enum ⇒ BindingFlags
```

```
MemberInfo[] specificmembers = typeof(BankAcount)
.GetMembers(BindingFlags.NonPublic | BindingFlags.Instance);

foreach (var member in specificmembers)
{
   Console.WriteLine(member);//here print all private members & instances
}

Console.WriteLine("Field Info----------------------");
FieldInfo[] fieldInfos = typeof(BankAcount).GetFields
(BindingFlags.NonPublic | BindingFlags.Instance);
foreach (var field in fieldInfos)
{
   Console.WriteLine(field);
}

Console.WriteLine("propertyInfo----------------------");
PropertyInfo[] propertyinfo = typeof(BankAcount).GetProperties();
foreach (var property in propertyinfo)
{
   Console.WriteLine(property);
}
Console.WriteLine("property (get)----------------------");
foreach (var property in propertyinfo)
{
   Console.WriteLine(property.GetGetMethod());
}

Console.WriteLine("property (set)----------------------");
foreach (var property in propertyinfo)
{
   Console.WriteLine(property.GetSetMethod());
}

Console.WriteLine("EventInfo------------------------");
EventInfo[] EventInfo = typeof(BankAcount).GetEvents();
foreach (var e in EventInfo)
{
   Console.WriteLine(e);
}

Console.WriteLine("ConstractorInfo---------------");
```

```
        ConstructorInfo[] Constractorinfoo =
        typeof(BankAcount).GetConstructors();
        foreach (var c in Constractorinfoo)
        {
            Console.WriteLine(c);
        }
    }

}

class BankAcount
{
    public event EventHandler OnNegativeBalance;
    private decimal coins;

    public BankAcount(string name) ⇒ Name = name;

    public string Name { get; set; }
    public decimal Coins ⇒ coins;

    public void CreateAcount()
    {

        coins = 100;
        Console.WriteLine($"hello sir
         {this.Name} Balance = {this.Coins}");
    }

    public void DeletAcount()
    {
        coins = 0;
        Console.WriteLine($"sir {this.Name} Acount Deleted");

    }

    public void Debit(decimal amount)
    {
        Console.WriteLine($"coins = {this.coins} before debit");
        coins += amount;
        Console.WriteLine($"coins = {this.coins} after debit");
    }

    public void Crdit(decimal amount)
```

```
        {
            Console.WriteLine($"coins = {this.coins} before crdit");
            coins -= amount;
            Console.WriteLine($"coins = {this.coins} after crdit");
            if (coins < 0)
                OnNegativeBalance?.Invoke(this, null);

        }

        internal static void Print(object? sender, EventArgs e)
        {
            BankAcount Acount = (BankAcount)sender;
            Console.WriteLine($"sir can not crdit again balance
            = {Acount.Coins}");
        }
    }
}
```

OutPut

Int32 GetHashCode()
Void .ctor(System.String)
System.String Name
System.Decimal Coins
System.EventHandler OnNegativeBalance
System.Object MemberwiseClone()
Void Finalize()
System.EventHandler OnNegativeBalance
System.Decimal coins
System.String <Name>k__BackingField
Field Info----------
System.EventHandler OnNegativeBalance
System.Decimal coins
System.String <Name>k__BackingField
propertyInfo----------
System.String Name
System.Decimal Coins

```
property (get)--------------------
System.String get_Name()
System.Decimal get_Coins()
property (set)--------------------
Void set_Name(System.String)

EventInfo-----------------------
System.EventHandler OnNegativeBalance
ConstractorInfo-----------------------
Void .ctor(System.String)
```

How Can Call Method at Compile Time and Run Time?

```
    static void Main(string[] args)
    {

        //Call Method at Compile Time and Run Time

        BankAcount Acount = new BankAcount("Ahmed");
        Acount.CreateAcount();//Call Method At Compile Time

        Type type= typeof(BankAcount);
        MethodInfo mymethod = type.GetMethod("CreateAcount");

        mymethod.Invoke(Acount,null);//cal method at RunTime


    }

}

class BankAcount
{
    public event EventHandler OnNegativeBalance;
    private decimal coins;

    public BankAcount(string name) => Name = name;

    public string Name { get; set; }
```

```
        public decimal Coins ⇒ coins;

        public void CreateAcount()
        {

            coins = 100;
            Console.WriteLine($"hello sir {this.Name} Balance = {this.Coins}");
        }


    }
```

How can use contains of `dll` file in your code?

by using path of the `dll` file you can load the file and

by using Reflection you can access to it

```
static void Main(string[] args)
{
    string path = "*******";
    var assemply=Assembly.LoadFile(path);

    var types=assemply.GetTypes();



}
```

What is Attribute?

attribute is a declarative tag that provides metadata about a program element (such as a class, method, property, or assembly). Attributes can be used to add additional information to code elements, which can then be queried at runtime using reflection.

Some Built-in Attributes

1. **[** `Obsolete]` `:`

```
static void Main(string[] args)
```

```
{

    print();//Error
    print2();//only there are a worning


}
[Obsolete("this method  unsupported",true)]
static void print1()
{
    Console.WriteLine("hello");
}

[Obsolete("this method will unsupported", false)]
static void print2()
{
    Console.WriteLine("hello");
}
```

2. `[DebuggerDisplay]`

is a powerful tool that allows you to customize how objects appear in the debugger, making debugging more intuitive and efficient

```
        Employee[] employee = new Employee[]
        {
            new Employee("Ahmed"),
            new Employee("omar"),
            new Employee("said"),
            //in Debuuger ⇒
            //Name = hello Iam "Ahmed"
            //Name = hello Iam "omar"
            //Name = hello Iam "said"
        };

        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(employee[i]);
        }

    }
```

```
}
[DebuggerDisplay("Name = hello Iam {Name}")]
class Employee(string name)
{
    public string Name { get; set; } = name;

    public override string ToString()
    {
        return $"sir {Name}";
    }
}
```

What is a Custom Attribute and How to Create It?

A **custom attribute** in C# is a user-defined metadata tag that can be attached to program elements such as classes, methods, properties, or fields. Custom attributes allow you to add additional information to these elements, which can be accessed at runtime using reflection.

Step 1: Define the Custom Attribute

Create a custom attribute called `MyCustomAttribute` with a single property.

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class MyCustomAttribute : Attribute{
    public string Message { get; }

    public MyCustomAttribute(string message)
    {
        Message = message;
    }
}
```

Step 2: Apply the Custom Attribute

Apply the custom attribute to a class and a method.

```
[MyCustom("This is a custom attribute on a class.")]
public class MyClass
{
    [MyCustom("This is a custom attribute on a method.")]
    public void MyMethod()
    {
        Console.WriteLine("Method executed.");
    }
}
```

Step 3: Retrieve and Display the Attribute Information

Use reflection to retrieve and display the attribute information.

```
using System;
using System.Reflection;

class Program
{
    static void Main()
    {
        // Get the custom attribute from the class
        var classAttribute = typeof(MyClass)
        .GetCustomAttribute<MyCustomAttribute>();
        if (classAttribute != null)
        {
            Console.WriteLine($"Class Attribute: {classAttribute.Message}");
        }

        // Get the custom attribute from the method
        var methodInfo = typeof(MyClass)
        .GetMethod("MyMethod");
        var methodAttribute = methodInfo.GetCustomAttribute<MyCustomAttribute>();
        if (methodAttribute != null)
        {
            Console.WriteLine($"Method Attribute: {methodAttribute.Message}");
        }
    }
}
```

what is a types of streaming?

- **Backstore**:

  This generally refers to a storage system or a data source used for holding data. In certain contexts, it may refer to the "backend" storage (like a database or file system) where data is kept, and from which it is retrieved for processing.

- **Decorator**:

  The **Decorator** pattern is a structural design pattern that allows you to add new functionality to an object without altering its structure. It involves wrapping an object with another object (the decorator) that enhances or modifies its behavior. For example, adding extra features to a graphical UI component without changing the original component itself.

- **Adaptor**:

  The **Adaptor** (or Adapter) pattern is a structural design pattern used to enable compatibility between two incompatible interfaces. It acts as a bridge, converting one interface into another that the client expects. For example, if a system expects an interface with certain methods, an adapter can be used to make a different class conform to that interface.

How can Read and Write From File?

```
 static void Main(string[] args)
{
    //here how can write bytes of string in file
    string path = @"D:\Streaming\myfile.txt";

    using(FileStream mystream=new FileStream(path,
    FileMode.Append,FileAccess.Write))
    {
      if(mystream.CanWrite)
       {
          byte[] text = System.Text.Encoding.UTF8
          .GetBytes("Hello sir Ahmed in my File \n");
          mystream.Write(text, 0, text.Length);

       }
      mystream.Close();
    }

    //here how can write string directly in file
    using(StreamWriter writer=new StreamWriter(path,true))
    {
       writer.WriteLine("I am a backend Develper");
```

```
        }

        using(StreamReader reader=new StreamReader(path))
        {
            Console.WriteLine(reader.ReadToEnd());
        }

        /*
         Hello sir Ahmed in my File
          I am a backend Develper
         */
        Console.ReadKey();

    }
```

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        // Create a new file and write some initial data
        using (FileStream fs = new FileStream("example.txt", FileMode.Create, FileAccess.Write))
        {
            byte[] initialText = System.Text.Encoding.UTF8.GetBytes("Hello, this is the initial content.");
            fs.Write(initialText, 0, initialText.Length);
        }

        // Now open the file and use Seek to move to a specific position to overwrite data
        using (FileStream fs = new FileStream("example.txt", FileMode.Open, FileAccess.Write))
        {
            // Move the file pointer to position 7 (after "Hello, ")
            fs.Seek(7, SeekOrigin.Begin);

            // Write new data starting at position 7
            byte[] newText = System.Text.Encoding.UTF8.GetBytes("world");
            fs.Write(newText, 0, newText.Length);
        }

        // Read the file to see the changes
```

```
        string content = File.ReadAllText("example.txt");
        Console.WriteLine(content); // Output: "Hello, world is the initial content."
    }
}
```

How can Compression and Decompression Data?

```
        static void Main(string[] args)
        {

            using (var mystream = File.Create("mydata.bin"))
            {
                using(var compr=new DeflateStream(mystream,CompressionMode.Compress))
                {
                    compr.WriteByte(67);
                    compr.WriteByte(66);
                }
            }

            using (var mystream = File.OpenRead("mydata.bin"))
            {
                using (var decomp = new DeflateStream(mystream, CompressionMode.Decompress))
                {
                    for(int i = 0; i <mystream.Length; i++)
                    {
                        Console.Write((char)(decomp.ReadByte())+" ");//C B ? ?
                    }
                }
            }


            Console.ReadKey();

        }
```

What is the Nullable Value Type and Nullable Reference Type?

Nullable Value Type

we use it when we use the default value of value type in own case

like we use the default value of `int` 0 to assign variable

but it not use for tell us that the variable is empty

to do this we use `int`? or `Nullable<int>`

```
static void Main(string[] args)
{

    Nullable<int> n = default;
    int x = default;
    if(n is  null)
    {
        Console.WriteLine("null");
    }
    else
    {
        Console.WriteLine("Not Null");
    }
    Console.WriteLine(x);
    //output
    //null
    //0
}
```

Nullable Reference Type

Nullable Reference Types (introduced in C# 8.0) allow reference types (like `string`, `object`,.) to be explicitly marked as nullable (`string?`), meaning they can hold `null`. When enabled, the compiler will warn you if you try to use a reference type without checking for `null` when it might be `null`.

```
    static void Main(string[] args)
    {

        string n = null;
        string x = default;
        if(n is  null)
        {
            Console.WriteLine("null");
        }
        else
        {
            Console.WriteLine("Not Null");
        }
        Console.WriteLine(x??"null");
        //output
        //null
```

```
        //null
    }
```

How Data is Store?

Data store in Disk in form of Binary Format where every character store in 1 Byte(8bits)

To use this Data it Loaded in Form of Hexadecimal in memory(Ram)

All of this came from Ascii Table

Here is a simplified version of the ASCII table:

Char	Dec	Hex	Bin
A	65	41	01000001
B	66	42	01000010
C	67	43	01000011
D	68	44	01000100
E	69	45	01000101
F	70	46	01000110
G	71	47	01000111
H	72	48	01001000
I	73	49	01001001
J	74	4A	01001010
K	75	4B	01001011
L	76	4C	01001100
M	77	4D	01001101
N	78	4E	01001110
O	79	4F	01001111
P	80	50	01010000
Q	81	51	01010001
R	82	52	01010010
S	83	53	01010011
T	84	54	01010100
U	85	55	01010101
V	86	56	01010110

Char	Dec	Hex	Bin
W	87	57	01010111
X	88	58	01011000
Y	89	59	01011001
Z	90	5A	01011010
a	97	61	01100001
b	98	62	01100010
c	99	63	01100011
d	100	64	01100100
e	101	65	01100101
f	102	66	01100110
g	103	67	01100111
h	104	68	01101000
i	105	69	01101001
j	106	6A	01101010
k	107	6B	01101011
l	108	6C	01101100
m	109	6D	01101101
n	110	6E	01101110
o	111	6F	01101111
p	112	70	01110000
q	113	71	01110001
r	114	72	01110010
s	115	73	01110011
t	116	74	01110100
u	117	75	01110101
v	118	76	01110110
w	119	77	01110111
x	120	78	01111000
y	121	79	01111001
z	122	7A	01111010

Converting

```
static void Main(string[] args)
{
    for (byte c = 0; c < 255; c++)
    {
```

```
            char ch = (char)c;
            string Dec=c.ToString().PadLeft(3,'0');
            string Hex=c.ToString("X");
            string binary=Convert.ToString(c,2).PadLeft(8, '0');
            Console.WriteLine($"{ch} : {Dec} : {Hex}  : {binary}");
        }
    }
```

Why can transferring files from Windows to Linux or macOS cause issues?

Transferring files from Windows to Linux or macOS can cause issues due to differences in how each operating system handles line endings (Windows uses CRLF `\r\n`, while Linux/macOS use LF `\n`), file permissions, and file system formats. These differences may lead to problems with reading or executing files, incorrect formatting in text files, or loss of file permissions during the transfer. Tools like `dos2unix` and setting proper file permissions can help resolve these issues.

What is a Deference Between Ascii Code and Unicode?

ASCII:

- **What it is**: A character encoding standard that represents text using 7 bits.
- **Characters it supports**: Only 128 characters, which includes English letters (A-Z, a-z), numbers (0-9), punctuation marks, and some control characters (like newline).
- **Limitations**: It only supports characters for the English language and lacks support for characters from other languages, symbols, or special characters.

Example of Encoding Data at Ascii

```
static async Task GetData_At_AsciiCode(string Location,string Link)
{
    using(HttpClient client = new HttpClient())
    {
        Uri uri = new Uri(Link);
        using(HttpResponseMessage response = await client.GetAsync(uri))
        {
            var bytearray=await response.Content.ReadAsByteArrayAsync();
            var data=Encoding.ASCII.GetString(bytearray);
            File.WriteAllText(Location, data);
        }
    }
```

```
}
```

Unicode:

- **What it is**: A universal character encoding standard that represents text using a much wider range of characters.
- **Characters it supports**: Unicode can represent over **143,000 characters**, including characters from **virtually all writing systems** (like Chinese, Arabic, Cyrillic, etc.), emojis, symbols, and even ancient languages.
- **Benefits**: It supports a vast number of characters, making it more suitable for global applications and diverse languages.

Example of Encoding Data at Unicode

```
static async Task GetData_At_UniCode(string Location, string Link)
{
   using (HttpClient client = new HttpClient())
   {
      Uri uri = new Uri(Link);
      using (HttpResponseMessage response = await client.GetAsync(uri))
      {
         var bytearray = await response.Content.ReadAsByteArrayAsync();
         var data = Encoding.UTF8.GetString(bytearray);
         File.WriteAllText(Location, data);
      }
   }
}
```

How can Create and use String Instantiation (string object)?

```
//Quated String
string val = "data";

//string constractor
char[] chars = { 'd', 'a', 't', 'a' };

string val2=new string(chars);//data

//Repeated Characters
string str = new string('A', 5);//AAAAA
```

What a Deference Between Using safe Code Vs Unsafe Code?

Safe Code:

- **Memory Management**: the runtime managed automatically memory with garbage collector

that free the usage allocation memory blocks

- **No Pointers**: You can't directly use pointers or manipulate memory addresses.
- **Safety**: The compiler ensures no access to invalid memory, preventing issues like buffer overflows or memory corruption.
- **Performance**: there area performance overhead by use GC

Unsafe Code:

- **Memory Management**: You manage memory directly using pointers, and the garbage collector doesn't handle it.
- **Pointers**: You can use pointers to directly access and manipulate memory, similar to languages like C or C++.
- **Safety**: There's no memory protection, so you must be careful. Mismanagement can lead to crashes, memory corruption, or un security .
- **Performance**: Unsafe code can offer better performance for specific use cases because it avoids the overhead of garbage collection and allows low-level optimizations.

Here a simple code of write a **Unsafe Code in C#**

1. first you must enable it by add

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

2. simple code

```
static void Main()
{
   pointer();//10 20 30
}

static unsafe void pointer()
{
   int[] nums = { 1, 2, 3 };

   fixed (int* ptr = nums)
```

```
        {
            for (int i = 0; i < nums.Length; i++)
            {
                ptr[i] *= 10;
            }

            foreach (var n in nums)
            {
                Console.Write(n+" ");
            }
        }
    }
```

3. Row String

first enable

```
    <LangVersion>preview</LangVersion>
```

```
    static void Main()
    {
        string str = """
            <h1>
            Hello in my web

            </h1>

            """;

        Console.WriteLine(str);

        /*
         <h1>
        Hello in my web

          </h1>
         */
    }
```

How can Reduce overhead in Memory?

we know that a memory contain

1. stack
2. heap

to reduce overhead

3. intern pool

here we store a redundancy objects in keys(value of heap) and address (address of stack)

How Intern pool store data?

Example1

```
string str1 = "ahmed";
string str2=new string("ahmed");

Console.WriteLine(str1.Equals(str2));//true ⇒ compare values

Console.WriteLine(ReferenceEquals(str1,str2));//false  // two in deference locations
```

stack
str1(0x0001)
str2(0x0002)

heap
ahmed(0x0001)
ahmed(0x0002)

intern pool
key ⇒ “ahmed” , Address ⇒ 0x0001

Here same values but Deference Location(Addresses)

Example2

```
string s1 = "BackEnd";
string s2 = "Back" + "End";
```

```
        Console.WriteLine(s1.Equals(s2));//true
        Console.WriteLine(ReferenceEquals(s1,s2));//true
```

stack
s1(0x0001)
s2(0x0001)

heap
“BackEnd” ⇒ (0x0001)

intern pool
key ⇒ “BackEnd”, Address ⇒ 0x0001

Here same values and same Location(Addresses)

Example3:

```
    static void Main()
    {

        string s1 = "omar";
        string s2=new string("omar");

        string s3 = string.Intern(s2);

        Console.WriteLine(s1.Equals(s2));//true
        Console.WriteLine(ReferenceEquals(s1,s2));//false

        Console.WriteLine(s2.Equals(s3));//true
        Console.WriteLine(ReferenceEquals(s2, s3));//false



        Console.WriteLine(s1.Equals(s3));//true
        Console.WriteLine(ReferenceEquals(s1, s3));//true


    }
```

stack

s1(0x0001)
s2(0x0002)
s3(0x0001)

heap
“omar” ⇒ (0x0001)
“omar” ⇒ (0x0002)

intern pool
key ⇒ “omar”, Address ⇒ 0x0001

What is Deference between String and String Builder?

Feature	string	StringBuilder	
Mutability	Immutable (cannot be changed after creation),to change in it must create new object	Mutable (can be modified directly) without create new object	
Use case	Small, fixed text, or when the string doesn’t change often	Frequently modified strings (loops, string concatenation)	
Performance	Not efficient for frequent changes	More efficient for frequent modifications	

```
static void Main()
{
    // Declaration
    string str = "Hello";

    // Append (creates a new string)
    str += " World";
    Console.WriteLine("Appended: " + str);  // "Hello World"

    // Insert (creates a new string)
    str = str.Insert(6, "Beautiful ");
    Console.WriteLine("Inserted: " + str);  // "Hello Beautiful World"

    // Replace (creates a new string)
    str = str.Replace("Beautiful", "Amazing");
    Console.WriteLine("Replaced: " + str);  // "Hello Amazing World"

    // Remove (creates a new string)
```

```
        str = str.Remove(6, 7);  // Removes 7 characters starting from index 6
        Console.WriteLine("Removed: " + str);  // "Hello World"

        // here all edites in same object
          // Declaration
        StringBuilder sb = new StringBuilder("Hello");

        // Append
        sb.Append(" World");//concating with the default
        Console.WriteLine("Appended: " + sb);  // "Hello World"

        // Insert
        sb.Insert(6, "Beautiful ");
        Console.WriteLine("Inserted: " + sb);  // "Hello Beautiful World"

        // Replace
        sb.Replace("Beautiful", "Amazing");
        Console.WriteLine("Replaced: " + sb);  // "Hello Amazing World"

        // Remove
        sb.Remove(6, 7);  // Removes 7 characters starting from index 6
        Console.WriteLine("Removed: " + sb);  // "Hello World"

        // if you want to create new object
        sb=new new StringBuilder("ahmed")
          Console.WriteLine("Removed: " + str);  // "ahmed
    }
```

How String Builder Work?

by default buffer capacity =16 after it expanded

the capacity doubleted to 32

the edit in new object that will be parent , this object is a parent of

previous object that there is a pointer from new object in previous object

data store in chunks

```
        static void Main(string[] args)
        {
            StringBuilder strb = new StringBuilder();
            strb.Append("Welcom Iam ahmed");
            Console.WriteLine(strb.Length);//16
```

```
            Console.WriteLine(strb.Capacity);//16

            strb.Append("Welcom I ahmed");
            Console.WriteLine(strb.Length);//30
            Console.WriteLine(strb.Capacity);//32
            Console.WriteLine("---------");
            foreach (var chunk in strb.GetChunks())
            {
                Console.WriteLine(chunk); //Welcom Iam ahmed
                                          //Welcom I ahmed
            }

            Console.ReadKey();
        }
```

💡 بمعني ان هيبقي في مؤشر يؤشر علي محتويات الobject القديم

What is Deference Between Process And Thread?

A **process** is an **independent program** in execution, containing its own memory space and system resources.

A **thread** is a **smaller unit of execution** within a process. It is often called a **lightweight process**.

How can Run Threads in Parallel ?

```
static void Main(string[] args)
{
    Thread t1 = new Thread(Print1);
    Thread t2 = new Thread(Print2);
```

```
    t1.Name = "Thraed1";
    t2.Name = "Thraed2";

    t1.Start();
    t2.Start();

    Console.ReadKey();
}

static void Print1()
{
    DateTime starttime = DateTime.Now;

    while ((DateTime.Now-starttime).TotalMilliseconds<5000)
    {
        Console.WriteLine($"Thread Name : {Thread.CurrentThread.Name} / Id = {Thread.CurrentThre
        Thread.Sleep(1000);
    }
}

static void Print2()
{
    DateTime starttime = DateTime.Now;
    while ((DateTime.Now - starttime).TotalMilliseconds < 5000)
    {
        Console.WriteLine($"Thread Name : {Thread.CurrentThread.Name} / Id = {Thread.CurrentThre
        Thread.Sleep(1000);

    }
}
```

> Output
>
> Thread Name : Thraed1 / Id = 10
> Thread Name : Thraed2 / Id = 11
> Thread Name : Thraed1 / Id = 10
> Thread Name : Thraed2 / Id = 11
> Thread Name : Thraed2 / Id = 11
> Thread Name : Thraed1 / Id = 10
> Thread Name : Thraed1 / Id = 10

> Thread Name : Thraed2 / Id = 11
> Thread Name : Thraed1 / Id = 10
> Thread Name : Thraed2 / Id = 11

How can Synchronize Threads ?

Here an Example of Producer Consumer Problem:

```
static Queue<int> buffer = new Queue<int>();

static int bufferCapacity = 5;

static object lockObject = new object();  // Lock object for synchronization
                                //here every thread can not enter after
                                //existed thread in buffer completed
static void Producer()
{
   int item = 0;
   while (true)
   {

      item++;
      Thread.Sleep(1000);

      lock (lockObject)
      {

         while (buffer.Count >= bufferCapacity)
         {
            Console.WriteLine("Buffer is full, Producer is waiting...");
            Monitor.Wait(lockObject);
         }


         buffer.Enqueue(item);
         Console.WriteLine($"Produced: {item}");
```

```
            Monitor.Pulse(lockObject);
        }
    }
}


static void Consumer()
{
    while (true)
    {
        Thread.Sleep(1500);

        lock (lockObject)
        {

            while (buffer.Count == 0)
            {
                Console.WriteLine("Buffer is empty, Consumer is waiting...");
                Monitor.Wait(lockObject);
            }


            int item = buffer.Dequeue();
            Console.WriteLine($"Consumed: {item}");


            Monitor.Pulse(lockObject);
        }
    }
}

static void Main()
{

    Thread producerThread = new Thread(Producer);
    Thread consumerThread = new Thread(Consumer);

    producerThread.Start();
    consumerThread.Start();

   //here every thread wait another
    producerThread.Join();
    consumerThread.Join();
```

```
    }
}
```

> Output
>
> Produced: 1
> Produced: 2
> Produced: 3
> Produced: 4
> Produced: 5
> Buffer is full, Producer is waiting...
> Consumed: 1
> Produced: 6
> Consumed: 2
> Produced: 7
> Consumed: 3
> Buffer is full, Producer is waiting...
> Consumed: 4

What is a Thread Pool?

The
Thread Pool is a collection of worker threads that are managed by the system and used to execute tasks without needing to manually create and manage threads. It is part of the system's infrastructure that allows efficient handling of multiple tasks by reusing existing threads.

```
static void Main()
 {
    // Queue a task to run on a thread from the Thread Pool
    ThreadPool.QueueUserWorkItem(Task1);



   //the same
   Task T1=Task.Run(async()⇒await Task1());
```

```
    // Wait for user input before exiting
    Console.ReadLine();
}

static void Task1(object state)
{
    Console.WriteLine("Task started on thread: " +
    Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(2000);  // Simulate work
    Console.WriteLine("Task completed on thread: " +
     Thread.CurrentThread.ManagedThreadId);
}
```

What is Deference Between Synchronization and Asynchronization?

Synchronization:

- Synchronous operations are ones that are executed in a specific order, one after the other. In a synchronous context, a process or task will wait for another process or task to complete before it can proceed. This is often referred to as a blocking operation because the subsequent task is blocked until the previous one is done.
- Synchronous programming is best for straightforward, sequential tasks where the order of execution is important, and the tasks are relatively short-lived. It's easier to understand and debug because the code flows in a predictable, linear manner.
- Synchronous programming ensures that each operation completes fully before the next one begins, which can be beneficial for tasks that require strict sequential execution, such as financial transaction systems or batch data processing.

```
static void Main(string[] args)
{
    Method1();
    Method2();
    Method3();
}

static void Method1()
{
```

```
    Console.WriteLine("hello from method1");
}
static void Method2()
{
    Console.WriteLine("hello from method2");
}
static void Method3()
{
    Console.WriteLine("hello from method3");
}
```

Asynchronization:

- Asynchronous operations, on the other hand, allow multiple processes or tasks to run independently and concurrently without waiting for each other to complete. This can lead to more efficient use of system resources and can help improve the responsiveness of applications, especially those that involve I/O operations or network requests.
- Asynchronous programming excels in scenarios demanding responsiveness and concurrency, such as real-time systems or applications that need to handle many simultaneous tasks without delay.
- In asynchronous programming, tasks can run independently, and the system doesn't need to wait for one task to complete before starting another. This can be beneficial for improving efficiency and responsiveness, especially in I/O-heavy operations or real-time data handling.

1. Example of un using waiting

```
static async Task DownloadDataAsync()
{

    Console.WriteLine("Starting download...");
    await Task.Delay(3000);
    Console.WriteLine("Download finished");
}

static async Task Main()
{
    Console.WriteLine("Asynchronous :");


    Task downloadTask = DownloadDataAsync();
```

```
        Console.WriteLine("Doing other work while download is in progress...");




        Console.WriteLine("End of main program.");
    }
}
//output
/*
 Asynchronous :
Starting download...
Doing other work while download is in progress...
End of main program.
*/
```

2. Example of using waiting

```
    static async Task DownloadDataAsync()
    {

        Console.WriteLine("Starting download...");
        await Task.Delay(3000);
        Console.WriteLine("Download finished");
    }

    static async Task Main()
    {
        Console.WriteLine("Asynchronous :");


        Task downloadTask = DownloadDataAsync();


        Console.WriteLine("Doing other work while download is in progress...");


        await downloadTask;

        Console.WriteLine("End of main program.");
    }
}
```

```
//output
/*
Asynchronous :
Starting download...
Doing other work while download is in progress...
Download finished
End of main program..
*/
```

What is Deference between Concurrently and Parallelism?

Concurrency:

Running multiple tasks at the same time, but not necessarily simultaneously. Tasks may be interleaved, with the system switching between them (often on a single core).

```
static async Task Main()
{
    Task T1 =  task1();
    Task T2 = task2();
//here T1 and T2 will run in same thread ⇒ (main Thraed)
    await Task.WhenAll(T1, T2);
    Console.WriteLine("all tasks completed");


}

static async Task task1()
{
    Console.WriteLine("task1 started...");
    await Task.Delay(2000);
    Console.WriteLine("task1 completed...");
}


static async Task task2()
{
    Console.WriteLine("task2 started...");
    await Task.Delay(3000);
```

```
        Console.WriteLine("task2 completed...");
    }
```

Parallelism:

Running multiple tasks simultaneously on multiple CPU cores, where tasks truly execute at the same time.

```
  static async Task Main()
  {
  Task T1 =  Task.Run(async ()⇒await task1());
Task T2 = Task.Run(async () ⇒ await task2());
//here T1 and T2 Run in Deference Thraeds From Thraed Pool
      await Task.WhenAll(T1, T2);
      Console.WriteLine("all tasks completed");


  }

  static async Task task1()
  {
      Console.WriteLine("task1 started...");
      await Task.Delay(2000);
      Console.WriteLine("task1 completed...");
  }


  static async Task task2()
  {
      Console.WriteLine("task2 started...");
      await Task.Delay(3000);
      Console.WriteLine("task2 completed...");
  }
```

```
using System;
using System.Threading.Tasks;

class Program
{
```

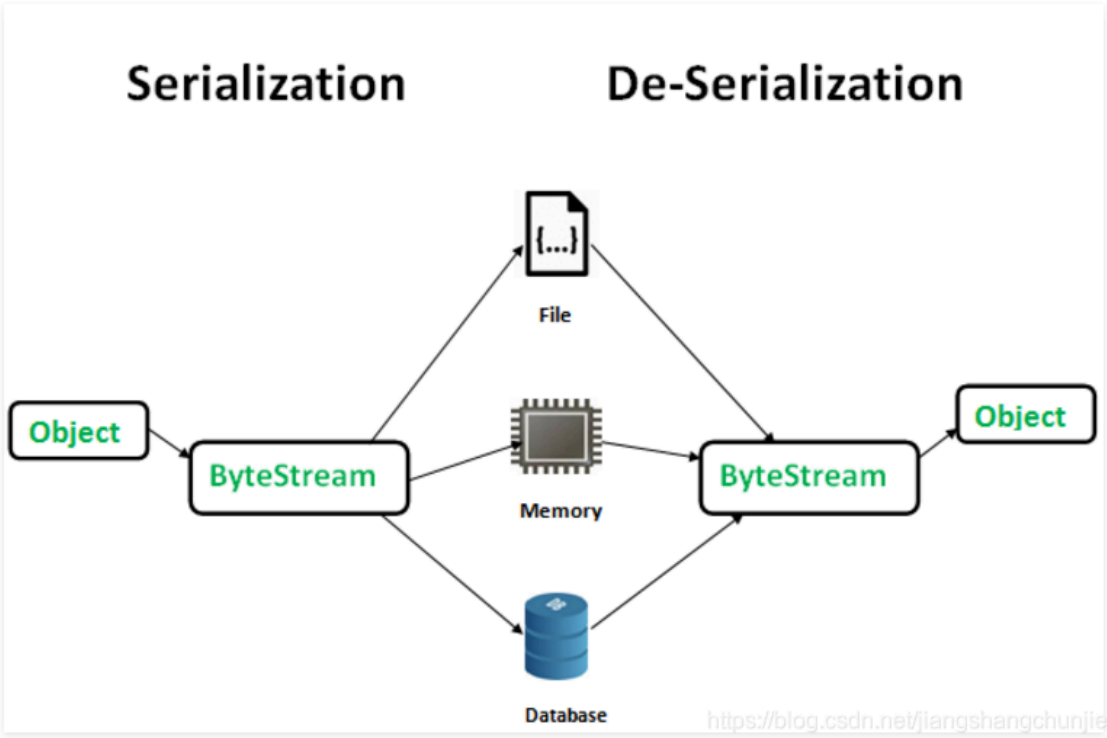
```
    // Simulated task that runs in parallel
    static void TaskInParallel(int i)
    {
        Console.WriteLine($"Task {i} started.");
        Task.Delay(1000).Wait();  // Simulate some work
        Console.WriteLine($"Task {i} completed.");
    }

    static void Main()
    {
        // Run tasks in parallel
        Console.WriteLine("Starting parallel tasks...");

        // Execute 5 tasks in parallel
        Parallel.For(0, 5, i =>
        {
            TaskInParallel(i);
        });

        Console.WriteLine("All tasks completed in parallel.");
    }
}
```

Serialization & Deserialization

What is Serialization & Deserialization ?

Serialization is the process of converting an object or data structure into a format that can be easily stored, transmitted, or persisted. It allows complex data structures to be translated into a form that can be efficiently stored or sent over a network, and later reconstructed through a process called deserialization. Serialization is crucial in various applications, such as network communication, data storage, and cross-platform compatibility. For example, in Java, serialization involves converting an object into a byte stream, which can then be saved to a file or sent over a network.

Deserialization is the process of converting a serialized data stream back into its original object or data structure form. This is the reverse operation of serialization. During deserialization, the data that was previously serialized (converted into a format suitable for storage or transmission) is read and reconstructed into an object or data structure that can be used in a program.



What is a Types of Serializations?

1️⃣ JSON (JavaScript Object Notation)

- **Type**: Text-based, human-readable
- **Structure**: Uses key-value pairs (like a dictionary)
- **Usage**: Common for APIs, web services, and configurations
- **Pros**:✅ Easy to read and write for humans✅ Lightweight and fast for parsing✅ Great for web applications and REST APIs
- **Cons**:❌ Cannot represent complex data types (like images) as easily❌ Larger in size than binary formats

Example:

```
{
  "name": "ahmed",
  "age": 20,
  "city": "Monofia"
}
```

2️⃣ XML (Extensible Markup Language)

- **Type**: Text-based, human-readable
- **Structure**: Uses tags and attributes to represent data
- **Usage**: Often used in document storage, older APIs, and web services
- **Pros**:✅ Highly flexible with custom tags and attributes✅ Supports complex data structures (e.g., nested elements)✅ Can define data schemas (XSD)
- **Cons**:❌ Larger and more verbose than JSON❌ Slower to parse❌ Less efficient for simple data compared to JSON

Example:

```
xml
<person>
  <name>ahmed</name>
  <age>20</age>
  <city>Monofia </city>
</person>
```

3️⃣ Binary Format

- **Type**: Non-text-based, machine-readable
- **Structure**: Data is encoded as binary (not human-readable)
- **Usage**: Used for performance-critical applications, custom protocols, or file storage
- **Pros**:✅ Extremely compact and efficient✅ Fast for serialization and deserialization✅ Can handle complex data types like images, audio, or files
- **Cons**:❌ Not human-readable❌ Requires special software or code to read/write❌ Less flexible when dealing with schema changes
- Not supported now ⇒ because it dangerous

Example

The binary data is not human-readable and appears as bytes:

```
1010011 10101000 01101100 11010010 ...
```

How Can Serialize& Deserialize with xml?

```
using System;
using System.Collections;
using System.Collections.ObjectModel;
using System.Runtime.Serialization;
using System.Xml;
using System.Xml.Serialization;

namespace serilization
{

   class Program
   {
      static void Main()
      {
         Employee employee1 = new Employee
         {
            Name="ahmed",
            Id= 1,
            Skills = {"C#","DB","Linq","EFCORE","OS","ASP.NetCore","API"}
         };

        var myxml=serilized_xml(employee1);
         Console.WriteLine(myxml);
         File.WriteAllText("myxmldata.xml", myxml);
         Console.WriteLine("----------------------------");
         Employee employee2 = Deserilized_xml(myxml);

         Console.WriteLine(employee2);

         Console.ReadKey();
      }

      private static Employee Deserilized_xml(string xml)
      {
         Employee? e = null;

         var xmlserilizer=new XmlSerializer(typeof(Employee));
         using(TextReader reader = new StringReader(xml))//using TextReader here for more flexibl
         {
         e=   xmlserilizer.Deserialize(reader)as Employee;
         }

         return e;
```

```
        }

        private static string serilized_xml(Employee employee)
        {
            var data = "";
            var xmlserilizer=new XmlSerializer(typeof(Employee));
            using(var sw=new StringWriter())
            {
                using(var writer= XmlWriter.Create(sw,new XmlWriterSettings { Indent=true}))
                {
                    xmlserilizer.Serialize(writer, employee);
                    data=sw.ToString();
                }
            }
            return data;
        }
    }

    public class Employee
    {


        public int Id { get; set; }
        public string Name { get; set; }

        public List<string> Skills { get; set; }= new List<string>();

        public override string ToString()
        {
            return $"{Name} / {Id} ";
        }
    }

}
```

How can Serialize and Deserialize with Json ?

```
using Newtonsoft.Json;
using System;
using System.Collections;
using System.Collections.ObjectModel;
using System.Runtime.Serialization;
```

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Text.Json;
using System.Xml;
using System.Xml.Serialization;
using Formatting = Newtonsoft.Json.Formatting;

namespace serilization
{

   class Program
   {
      static void Main()
      {
         Employee employee1 = new Employee
         {
            Name="ahmed",
            Id= 1,
            Skills = {"C#","DB","Linq","EFCORE","OS","ASP.NetCore","API"}
         };

         //    var myxml=serilized_xml(employee1);
         //Console.WriteLine(myxml);
         //File.WriteAllText("myxmldata.xml", myxml);
         //Console.WriteLine("----------------------------");
         //Employee employee2 = Deserilized_xml(myxml);

         //Console.WriteLine(employee2);

         //Json
         //can do by Newtonsoft_package
         var myjson = jsonserialzition_Newtonsoft_package(employee1);
         Console.WriteLine(myjson);

         var e2 = jsondeserialzition_Newtonsoft_package(myjson);
         Console.WriteLine(e2);

         Console.WriteLine("--------------------");

         //can do by json .net

         var myjson_net = jsonserialzition_Newtonsoft_package(employee1);
         Console.WriteLine(myjson_net);
```

```
        var e2_net = jsondeserialzition_Newtonsoft_package(myjson_net);
        Console.WriteLine(e2_net);
        Console.ReadKey();
    }


    private static string jsonserialzition_Net(Employee employee1)
    {
        string data = "";
        data = System.Text.Json.JsonSerializer.
            Serialize(employee1,new JsonSerializerOptions() { WriteIndented=true});
        return data;

    }

    private static Employee jsondeserialzition_Net(string jsondata)
    {
        Employee? e = null;
        e = System.Text.Json.JsonSerializer.Deserialize<Employee>(jsondata);
        return e;

    }

    private static string jsonserialzition_Newtonsoft_package(Employee employee1)
    {
        string data = "";
        data=JsonConvert.SerializeObject(employee1,Formatting.Indented);
        return data;

    }

    private static Employee jsondeserialzition_Newtonsoft_package(string jsondata)
    {
        Employee? e =null;
        e = JsonConvert.DeserializeObject<Employee>(jsondata) ;
        return e;

    }


}

public class Employee
```

```
    {


        public int Id { get; set; }
        public string Name { get; set; }

        public List<string> Skills { get; set; }= new List<string>();

        public override string ToString()
        {
            return $"{Name} / {Id} ";
        }
    }

}
```

How can Get Data from Json File ?

```
        class Program
    {
        private readonly static HttpClient httpClient = new HttpClient();
        static async Task Main()
        {
            string jsonlink = "*****";
            var data = await httpClient.GetStringAsync(jsonlink);

            var Emps=JsonSerializer.Deserialize<Employee>(data,new JsonSerializerOptions()
            {
                PropertyNameCaseInsensitive = true,
            });
        }
    }
    class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
```

What is `init` Keyword ?

The `init` keyword in C# is used to create **read-only properties** that can only be set during **object initialization**. Once the object is created, the property cannot be changed.

In simple terms, `init` allows you to set the value of a property when you create an object, but after that, the property becomes **immutable** (can't be changed).

Example:

```
class Program
{

   static  void Main( string[] args)
   {
      Employee employee = new Employee
      {
         Name="ahmed",
         Id=1
      };

      //employee.Name="" Compile Time Error

      Console.WriteLine($"{employee.Id} ⇒ {employee.Name}");
   }

}

class Employee
{
   public int Id { get; init; }
   public string Name { get; init; }
}
```

What is a record ?

A **record** in C# is a special kind of reference type introduced in C# 9.0. It is designed to be an immutable data container that holds values and provides built-in functionality for value-based equality, meaning that two records with the same data are considered equal

but by default record is mutable

By default, a record in C# overrides:

- `Equals()`: For value-based equality comparison.
- `GetHashCode()`: To ensure hash codes reflect the values of the properties.
- `ToString()`: To provide a readable string representation of the record.

- `Deconstruct()`: To deconstruct a record into its properties.
- Supports the `with` **expression** for easy object copying with property modification.

here example of default mutable record

```
class Program
{

   static  void Main( string[] args)
   {
      Test test = new Test(5);
      test.X = 55;// here by default record is Mutable
      //edit in same object

      Console.WriteLine(test.X);
   }

}

record Test
{
   public int X;
   public Test(int x)
   {
      X = x;
   }
}
```

here Example of immutable record

```
class Program
{

   static  void Main( string[] args)
   {

      //immutable
      //added init
      Test T1 = new Test(4, 5);

      Console.WriteLine(T1.X);
      Console.WriteLine(T1.Y);
```

```
        //T1.X = 55; here error
    }

}

record Test(int X,int Y);//positional record
```

What is a record struct ?

is a value type Positional record of it is Mutable by default

for keep it Immutable keep it a read only record struct

```
    static  void Main( string[] args)
    {
        Employee employee = new Employee();
        //Mutable
        employee.Name = "Ahmed";
        employee.Id= 1;

        //Immutable
        Manger manger = new Manger();
        // manger.Name=""  CompileTime Error
    }

}

record struct Employee( int Id,string Name);
readonly record   struct Manger(int Id, string Name);
```

What is `With` Keyword ?

In C#, the
`with` keyword is primarily used with **records** to create a **new instance** of a record with some **modified properties** while keeping the original properties unchanged. This is useful for creating immutable data structures where you want to change only certain fields without affecting the original instance.

```
class Program
{

    static  void Main( string[] args)
    {
       var e1 = new Employee { Id=1,Name="Ahmed"};
        var e2 = new Employee { Id=2,Name=e1.Name};
        var e3 = e1 with { Id = 3 };

        Console.WriteLine(e1);
        Console.WriteLine(e2);
        Console.WriteLine(e3);

        /*
         Output
         Employee { Id = 1, Name = Ahmed }
         Employee { Id = 2, Name = Ahmed }
         Employee { Id = 3, Name = Ahmed }
         */

    }

}

record Employee
{
    public int Id { get; set; }
    public string Name { get; set; }

}
```

What about Inheritance at records ?

Record can only inherit from another record

Data Structure (Collections)

First we will know a some methods that used with all collections

Method	Description
Add()	Adds an item to the collection.
Remove()	Removes an item from the collection.
Contains()	Checks if the collection contains a specified element.
Clear()	Removes all elements from the collection.
Count	Returns the number of elements in the collection.
IndexOf()	Returns the index of the first occurrence of an item in the collection.
Sort()	Sorts the elements in the collection.
AddRange()	Adds multiple elements to the collection.
ToArray()	Converts the collection to an array.
GetEnumerator()	Returns an enumerator to iterate over the collection.
ForEach()	Performs a specified action on each element in the collection.
Find()	Finds the first element that matches a condition.
Exists()	Checks if any element in the collection matches a given condition.
RemoveAt()	Removes an element from the collection at a specified index.
FindIndex()	Finds the index of the first element that matches a given condition.
ToList()	Converts the collection into a List.

Array

1. Definition

An **array** is a fixed-size, sequential collection of elements of the same type. It provides fast, indexed access to its elements and is stored in a contiguous block of memory. Arrays are one of the most fundamental data structures in C#.

---

2. When to Use

- **Best to use when**:
  - You know the number of elements in collection that you want.
  - You need fast, random access to elements by index.
  - Memory efficiency is important (arrays have minimal overhead).
  - You are working with low-level algorithms or performance-critical code.

- **Avoid using when**:
  - The size of the collection is dynamic or unknown.
  - You need frequent insertions or deletions (arrays are not efficient for this).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of arrays in their best and worst cases:

Operation	Best Case	Worst Case	Explanation
Access by index	`O(1)`	`O(1)`	Accessing an element by index is always constant time.
Update by index	`O(1)`	`O(1)`	Updating an element by index is always constant time.
Iteration	`O(n)`	`O(n)`	Iterating through all elements is linear time.
Insertion/Deletion	N/A	`O(n)`	Inserting or deleting requires shifting elements, which is linear time.
Search (unsorted)	`O(1)` (if lucky)	`O(n)`	In the worst case, you must check every element to find a value.
Search (sorted)	`O(log n)`	`O(log n)`	Binary search on a sorted array is logarithmic time.
Resizing	N/A	`O(n)`	Resizing requires creating a new array and copying all elements.

5. How to Increase Performance When Using Arrays

1. **Avoid Frequent Resizing**:
   - If you know the maximum size of the array in advance, initialize it with that size to avoid costly resizing operations.

```
 static void Main()
 {
  //To store {1,2,3}

     int[] ints = new int[3];//High Performance 🚀
//if you store 4,5 again
Array.Resize(ref ints, 5);//here created new array and coped in it all elements
//and add 0,0 ⇒ 1,2,3,0,0
ints[3]=4;
ints[4]=5;
//here 1,2,3,4,5
 }
```

2. **Use `Array.Copy` for Bulk Operations**:
   - Instead of manually copying elements, use `Array.Copy` for better performance.

```
class Program
{
   static void Main(string[] args)
   {
      int[] nums1 = { 1, 2, 3, 4, 5 };

      int[] nums2 = nums1;//low performance ,here there are a pointer in num1
                          //and any updates in num1 will update in num2 🐢

      int[] nums3 = new int[nums1.Length];//here created new location in memory

      Array.Copy(nums1, nums3, nums1.Length);//High Performance 🚀

      Console.WriteLine("before Edit");
      Console.WriteLine("native native⇒ " + string.Join(',', nums1));
      Console.WriteLine("copy array ⇒ " + string.Join(',', nums3));

      nums1[2] = 300;
      Console.WriteLine("after edit");
      Console.WriteLine("native native⇒ "+string.Join(',',nums1));
      Console.WriteLine("copy array ⇒ " + string.Join(',', nums3));


      /*
       * Output
       before Edit
       native native⇒ 1,2,3,4,5
       copy array ⇒ 1,2,3,4,5
       after edit
       native native⇒ 1,2,300,4,5
       copy array ⇒ 1,2,3,4,5
       */

      Console.ReadKey();
   }
}
```

here a code to show how pointer work :

```
    static void Main(string[] args)
    {
        int[] num1 = { 1, 2, 3 };
        int[] num2 = num1;//🐢

    for (int i = 0; i < 3; i++)
    {
            Console.Write(num1[i] + " ");
    }
    Console.WriteLine();

    for (int i = 0; i < 3; i++)
    {
        Console.Write(num1[i] + " ");
    }

    Console.WriteLine("\nafter update");

    num1[0] = 100;

    for (int i = 0; i < 3; i++)
    {
        Console.Write(num1[i] + " ");
    }
    Console.WriteLine();

    for (int i = 0; i < 3; i++)
    {
        Console.Write(num1[i] + " ");
    }

    Console.ReadKey();

    /*
     * output
        1 2 3
      1 2 3
    after update
      100 2 3
       100 2 3
     */
}
```

3. **Prefer `for` Over `foreach` for Performance-Critical Code**:
   - `for` loops are slightly faster than `foreach` because they avoid the overhead of the enumerator.

```
static void Main()
{
   int[] nums = { 1, 2, 3, 4, 5 };
   foreach (var item in nums)//Low Performance 🐢
   {
      Console.Write(item + " ");
   }
   Console.WriteLine();
   for (int i = 0; i < nums.Length; i++) //High Performance 🚀
   {
      Console.Write(nums[i] + " ");
   }
}
```

4. **Use Multidimensional Arrays for Fixed Grids**:
   - For matrices or grids, use multidimensional arrays (`int[,]`) instead of jagged arrays (`int[][]`) for better memory locality.

Multidimensional Array:

A **multidimensional array** is an array that contains more than one dimension. It can be thought of as an array of arrays, where each element can be another array

```
   //high performance 🚀
   int[,] nums =
   {
      {1,2,3 },
      {4,5,6},
      {5,6,7},
      {6,7,8}
   };
   Console.WriteLine(nums[0,1]);//2
```

Jagged Array:

A **jagged array** is an array of arrays, where each inner array can have a different length. Unlike multidimensional arrays, jagged arrays don't have a fixed structure for the number of elements in each "row".

```
//low performance 🐢
int[][] ints =
      {
         new int[] {1,2,3,4},
         new int[] {11,12,13,14},
         new int[] {10,20,30,40},

      };
      Console.WriteLine(ints[2][3]);//40
```

5. **Leverage `Array.Sort` and `Array.BinarySearch`:**
   - Use these built-in methods for sorting and searching, as they are highly optimized.
   - `Array.BinarySearch`: Complexity = **O(log n)**,
   - `IndexOf`:Complixty =O(n)
   - `Array.Sort` **: using a best sorting algorithm for the use case**

```
int[] nums = { 1, 2, 3, 4, 5 };

//sorting
Array.Sort(nums);

//searching


      char[] chars = { 'a', 'b', 'c' };

int index1= Array.IndexOf(chars, 'b');//🐢 low speed O(n)
  int index2= Array.BinarySearch(chars, 'b'); //🚀 high speed O(log(n))
```

6. **Consider `Span<T>` or `Memory<T>` for Slicing:**
   - If you need to work with slices of an array without copying data, use `Span<T>` or `Memory<T>` for better performance.

✅**No extra memory allocation**– it doesn't copy data.

✅**Works directly on the original array**– changes reflect immediately.

Span<T>

```
static void Main(string[] args)
{
```

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7 };

Span<int> slice = nums.AsSpan(2, 3);// pointer in slice of native array {3,4,5}

Console.WriteLine("Before Edit");
Console.WriteLine("native array ⇒ " + string.Join(',', nums));
Console.Write("slice array ⇒ ");
for (int i = 0; i < slice.Length; i++)
{
   Console.Write(slice[i]+" ");//3,4,5
}

nums[3] = 404;
Console.WriteLine("\nafter Edit");
Console.WriteLine("native array ⇒ " + string.Join(',', nums));
Console.Write("slice array ⇒ ");
for (int i = 0; i < slice.Length; i++)
{
   Console.Write(slice[i]+" ");//3,4,5
}
/*
 output

Before Edit
      native array ⇒ 1,2,3,4,5,6,7
      slice array ⇒ 3 4 5
after Edit
      native array ⇒ 1,2,3,404,5,6,7
      slice array ⇒ 3 404 5

 */

Console.ReadKey();
      }
```

Memory<T> the same of span<T> but Memory<T> can use with asyncrounization

```
static async Task Main()
{
   int[] nums = { 1, 2, 3, 4, 5, 6, 7 };

   Memory<int> slice = nums.AsMemory(2, 3);// pointer in slice of native array {3,4,5}
```

```
    Console.WriteLine("native array ⇒ " + string.Join(',', nums));
    for (int i = 0; i < slice.Length; i++)
    {
        Console.WriteLine(slice.Span[i]);//3,4,5
    }
}
```

7. **Use `ArrayPool<T>` for Reusable Arrays**:
    - If you frequently create and discard arrays, use `ArrayPool<T>` to reduce memory allocations and garbage collection overhead.
    - `ArrayPool<T>` is a class in C# that allows you to **rent** and **reuse arrays** from a shared pool, improving memory efficiency and performance by avoiding frequent allocations.

```
static void Main()
{
  ArrayPool<int> pool=ArrayPool<int>.Shared;
   int[] nums=pool.Rent(5);


    try
    {
        for (int i = 0; i < nums.Length; i++)
        {
            nums[i] = i;
        }

        Console.WriteLine(string.Join(',', nums).Take(5));
    }
    finally
    {
        pool.Return(nums);
    }

}
```

Here a simple Example of use Array:

```
using System;

class Program
{
```

```
static void Main()
{
    // Create an array of integers with a fixed size of 5
    int[] numbers = new int[5];

    // Initialize the array with values
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

    // Access elements by index (O(1) time complexity)
    Console.WriteLine("Element at index 2: " + numbers[2]); // Output: 30

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

    // Use foreach to iterate
    Console.WriteLine("Using foreach:");
    foreach (int num in numbers)
    {
        Console.WriteLine(num);
    }

    // Resize the array (creates a new array and copies elements)
    Array.Resize(ref numbers, 7);
    numbers[5] = 60;
    numbers[6] = 70;

    Console.WriteLine("After resizing:");
    foreach (int num in numbers)
    {
        Console.WriteLine(num);
    }

    // Sort the array
    Array.Sort(numbers);
    Console.WriteLine("Sorted array:");
```

```
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }

        // Search for an element using binary search (requires sorted array)
        int index = Array.BinarySearch(numbers, 40);
        Console.WriteLine("Index of 40: " + index); // Output: 4
    }
}
```

In the provided code, which loop is faster:

1. The first loop where you iterate through columns first (`matrix[row, col]`),
2. Or the second loop where you iterate through rows first (`matrix[row, col]`)?

```
    for (int col = 0; col <cols ; col++)
    {
        for (int row = 0; row < rows; row++)
        {
            sum += matrix[row, col];
        }
    }

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            sum += matrix[row, col];
        }
    }
```

👨🏽‍💻:the second loop is faster

Why?

👨🏽‍💻:

The speed difference often comes down to how data is stored in memory (specifically in multi-dimensional arrays). In .NET, arrays are stored in **row-major order**, meaning that elements are stored row by row.

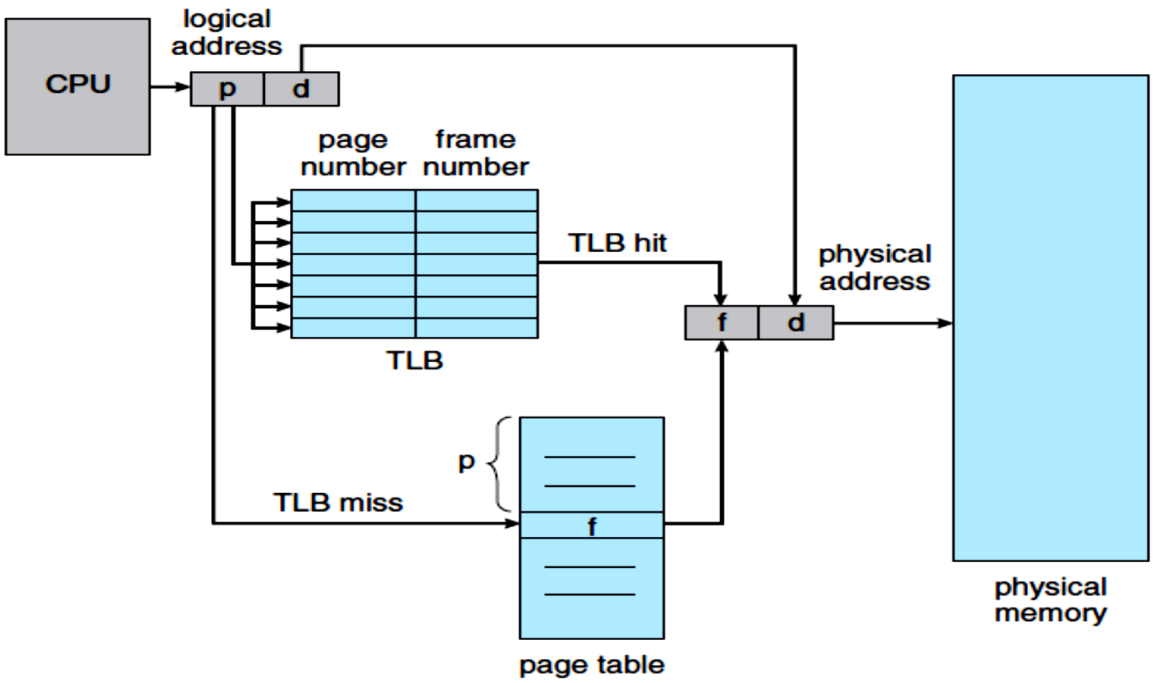
- **First loop (columns first)**:

The loop iterates over columns first, then rows. Since the memory is stored row by row, this iteration pattern may result in cache misses and less efficient memory access because the CPU cache is not utilized as effectively.

- **Second loop (rows first)**:

  The loop iterates over rows first. This matches the natural memory storage order (row-major), leading to better **cache locality** (cache hit) and likely **faster access** to elements.

Here you can show how cache hit and miss act



Arraylist

1. Definition

An **ArrayList** is a non-generic, dynamically resizable collection that can store elements of any type. It is part of the `System.Collections` namespace and provides features like adding, removing, and accessing elements by index. Unlike arrays, `ArrayList` can grow or shrink in size automatically.

2. When to Use

- **Best to use when**:
  - You need a dynamically resizable collection.
  - You want to store elements of different types integers, strings, objects).

- You don’t know the size of the collection in advance.
- You need features like adding, removing, or inserting elements at any position.

- **Avoid using when**:
  - You need type safety (use `List<T>` instead).
  - You are working with performance-critical code (ArrayList is slower due to boxing/unboxing).
  - You are storing value types (boxing/unboxing overhead can degrade performance).

---

3. Best and Worst Cases

Here’s a table explaining the **time and space complexity** of `ArrayList` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Access by index	`O(1)`	`O(1)`	Accessing an element by index is always constant time.
Update by index	`O(1)`	`O(1)`	Updating an element by index is always constant time.
Add to end	`O(1)` (amortized)	`O(n)`	Adding to the end is usually `O(1)`, but resizing the internal array is `O(n)`.
Insert at index	`O(n)`	`O(n)`	Inserting requires shifting elements, which is linear time.
Remove by index	`O(n)`	`O(n)`	Removing requires shifting elements, which is linear time.
Search (unsorted)	`O(1)` (if lucky)	`O(n)`	In the worst case, you must check every element to find a value.
Search (sorted)	`O(log n)`	`O(log n)`	Binary search on a sorted ArrayList is logarithmic time.
Space Complexity	`O(n)`	`O(n)`	ArrayList uses an internal array, so space complexity is linear.

---

4. Code Example

Here’s a simple, complete code example demonstrating the features of `ArrayList` in C#:

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        // Create an ArrayList
        ArrayList list = new ArrayList();
```

```
// Add elements of different types
list.Add(10); // Integer
list.Add("Hello"); // String
list.Add(3.14); // Double
list.Add(true); // Boolean

// Access elements by index (O(1) time complexity)
Console.WriteLine("Element at index 1: " + list[1]); // Output: Hello

// Iterate through the ArrayList
Console.WriteLine("All elements:");
foreach (var item in list)
{
    Console.WriteLine(item);
}

// Insert an element at a specific index (O(n) time complexity)
list.Insert(2, "Inserted");
Console.WriteLine("After insertion:");
foreach (var item in list)
{
    Console.WriteLine(item);
}

// Remove an element by value (O(n) time complexity)
list.Remove("Hello");
Console.WriteLine("After removal:");
foreach (var item in list)
{
    Console.WriteLine(item);
}

// Remove an element by index (O(n) time complexity)
list.RemoveAt(0);
Console.WriteLine("After removal at index 0:");
foreach (var item in list)
{
    Console.WriteLine(item);
}

// Check if an element exists (O(n) time complexity)
bool contains = list.Contains(3.14);
```

```
        Console.WriteLine("Contains 3.14: " + contains); // Output: True

        // Sort the ArrayList (O(n log n) time complexity)
        list.Sort(); // Note: This will throw an exception if elements are not comparable
        Console.WriteLine("Sorted ArrayList:");
        foreach (var item in list)
        {
            Console.WriteLine(item);
        }
    }
}
```

5. How to Increase Performance When Using Arraylist

1. **Use `List<T>` Instead**:
   - `List<T>` is a generic alternative to `ArrayList` that provides type safety and avoids boxing/unboxing overhead. Use it whenever possible.
2. **Avoid Frequent Insertions/Deletions**:
   - Inserting or deleting elements in the middle of an `ArrayList` is expensive (`O(n)`). Use `List<T>` or other collections like `LinkedList<T>` if you need frequent modifications.
3. **Preallocate Capacity**:
   - If you know the approximate size of the collection, use the `Capacity` property to preallocate memory and avoid frequent resizing.
4. **Avoid Sorting Mixed Types**:
   - Sorting an `ArrayList` with mixed types integers and strings) will throw an exception. Ensure all elements are of the same type before sorting.
5. **Use `ArrayList.TrimToSize()`**:
   - If the collection size is fixed and won't grow further, use `TrimToSize()` to reduce memory usage by removing unused capacity.

```
 static void Main()
 {
   ArrayList arrayList = new ArrayList();
     arrayList.Add(1);
     arrayList.Add(2);
     arrayList.Add(3);
     arrayList.Add("ahmed");
     arrayList.Add(true);

     Console.WriteLine(arrayList.Capacity);//8
```

```
    arrayList.TrimToSize();

    Console.WriteLine(arrayList.Capacity);//5
}
```

6. **Avoid Boxing/Unboxing**:
   - If you are storing value types (`int`, `double`), use `List<T>` instead to avoid the performance penalty of boxing/unboxing.
7. **Use `ArrayList.BinarySearch()` for Sorted Data**:
   - If the `ArrayList` is sorted, use `BinarySearch()` for faster searching (`O(log n)`)

BitArray

1. Definition

A **BitArray** is a specialized collection in the `System.Collections` namespace that stores bits (Boolean values) compactly. Each element in a `BitArray` represents a single bit (`true` or `false`), and the collection is optimized for memory efficiency when working with large sets of binary data.

---

2. When to Use

- **Best to use when**:
  - You need to store and manipulate a large number of Boolean values (bits) efficiently.
  - Memory efficiency is critical (when working with large bitmaps or flags).
  - You need to perform bitwise operations like AND, OR, NOT, and XOR on collections of bits.
- **Avoid using when**:
  - You need to store non-Boolean data.
  - You require fast access to individual bits by index (BitArray is slower than arrays for random access due to internal bit packing).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `BitArray` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Access by index	`O(1)`	`O(1)`	Accessing a bit by index is constant time.
Update by index	`O(1)`	`O(1)`	Updating a bit by index is constant time.
Add/Remove bits	N/A	N/A	`BitArray` has a fixed size; you cannot add or remove bits after creation.

Bitwise operations	`O(n)`	`O(n)`	Bitwise operations (AND, OR, NOT, XOR) are linear time, worked long circuit
Initialization	`O(n)`	`O(n)`	Initializing a `BitArray` with `n` bits is linear time.
Space Complexity	`O(n)`	`O(n)`	`BitArray` uses approximately `n/8` bytes of memory for `n` bits.

4. Code Example

Here's a simple, complete code example demonstrating the features of `BitArray` in C#:

```
using System;
using System.Collections;

class Program
{
   static void Main()
   {
      // Create a BitArray with 8 bits (1 byte)
      BitArray bits = new BitArray(8);
// by default bit = false
      // Set individual bits
      bits[0] = true;  // 00000001
      bits[3] = true;  // 00001001
      bits[7] = true;  // 10001001

      // Access bits by index
      Console.WriteLine("Bit at index 3: " + bits[3]); // Output: True

      // Display all bits
      Console.WriteLine("All bits:");
      for (int i = 0; i < bits.Length; i++)
      {
         Console.WriteLine($"Bit {i}: {bits[i]}");
      }

      // Perform bitwise NOT operation
      BitArray notBits = bits.Not();
      Console.WriteLine("After NOT operation:");
      for (int i = 0; i < notBits.Length; i++)
      {
         Console.WriteLine($"Bit {i}: {notBits[i]}");
      }

      // Perform bitwise AND operation
```

```
        BitArray otherBits = new BitArray(8);
        otherBits[2] = true; // 00000100
        BitArray andBits = bits.And(otherBits);
        Console.WriteLine("After AND operation:");
        for (int i = 0; i < andBits.Length; i++)
        {
            Console.WriteLine($"Bit {i}: {andBits[i]}");
        }

        // Perform bitwise OR operation
        BitArray orBits = bits.Or(otherBits);
        Console.WriteLine("After OR operation:");
        for (int i = 0; i < orBits.Length; i++)
        {
            Console.WriteLine($"Bit {i}: {orBits[i]}");
        }

        // Perform bitwise XOR operation
        BitArray xorBits = bits.Xor(otherBits);
        Console.WriteLine("After XOR operation:");
        for (int i = 0; i < xorBits.Length; i++)
        {
            Console.WriteLine($"Bit {i}: {xorBits[i]}");
        }
    }
}
//XOR
// trur ^ true =false
//false ^ false =false
//true ^ false=true
```

5. How to Increase Performance When Using BitArray

1. **Preallocate Size**:
   - If you know the number of bits in advance, initialize the `BitArray` with the correct size to avoid resizing (though `BitArray` itself cannot resize).
2. **Use Bitwise Operations Efficiently**:
   - Use built-in methods like `And`, `Or`, `Not`, and `Xor` for bulk bitwise operations instead of manually iterating and updating bits.
3. **Avoid Frequent Access by Index**:

- Accessing individual bits by index is slower than working with entire bytes or words. If possible, perform operations on the entire `BitArray` at once.

4. **Use `BitArray.CopyTo` for Bulk Operations**:
   - If you need to convert a `BitArray` to a byte array or other format, use `CopyTo` for better performance.

```
byte[] bytes = new byte[(bits.Length + 7) / 8];
bits.CopyTo(bytes, 0);
```

5. **Combine with Other Data Structures**:
   - For more complex scenarios, combine `BitArray` with other collections like `List<T>` or arrays to optimize performance.
6. **Avoid Unnecessary Conversions**:
   - Converting between `BitArray` and other formats (byte arrays) can be expensive. Minimize such conversions if possible.

List

1. Definition

A **List<T>** is a generic, dynamically resizable collection that stores elements of a specified type `T`. It is part of the `System.Collections.Generic` namespace and provides features like adding, removing, and accessing elements by index. Unlike arrays, `List<T>` can grow or shrink in size automatically.

---

2. When to Use

- **Best to use when**:
  - You need a dynamically resizable collection.
  - You want type safety (unlike `ArrayList`).
  - You need fast access to elements by index.
  - You need features like adding, removing, or inserting elements at any position.
  - You are working with a collection of objects or value types.
- **Avoid using when**:
  - You need a fixed-size collection (use arrays instead).
  - You need a collection optimized for frequent insertions/deletions at the beginning or middle (consider `LinkedList<T>`).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `List<T>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Access by index	`O(1)`	`O(1)`	Accessing an element by index is always constant time.
Update by index	`O(1)`	`O(1)`	Updating an element by index is always constant time.
Add to end	`O(1)` (amortized)	`O(n)`	Adding to the end is usually `O(1)`, but resizing the internal array is `O(n)`.
Insert at index	`O(n)`	`O(n)`	Inserting requires shifting elements, which is linear time.
Remove by index	`O(n)`	`O(n)`	Removing requires shifting elements, which is linear time.
Remove by value	`O(n)`	`O(n)`	Removing by value requires searching and shifting elements.
Search (unsorted)	`O(n)`	`O(n)`	Searching requires checking each element in the worst case.
Search (sorted)	`O(log n)`	`O(log n)`	Binary search on a sorted list is logarithmic time.
Space Complexity	`O(n)`	`O(n)`	`List<T>` uses an internal array, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `List<T>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a List of integers
        List<int> numbers = new List<int>();

        // Add elements to the list
        numbers.Add(10);
        numbers.Add(20);
        numbers.Add(30);
        numbers.Add(40);
        numbers.Add(50);

        // Access elements by index (O(1) time complexity)
```

```
Console.WriteLine("Element at index 2: " + numbers[2]); // Output: 30

// Iterate through the list
Console.WriteLine("All elements:");
for (int i = 0; i < numbers.Count; i++)
{
    Console.WriteLine(numbers[i]);
}

// Use foreach to iterate
Console.WriteLine("Using foreach:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Insert an element at a specific index (O(n) time complexity)
numbers.Insert(2, 25);
Console.WriteLine("After insertion:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Remove an element by value (O(n) time complexity)
numbers.Remove(30);
Console.WriteLine("After removal:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Remove an element by index (O(n) time complexity)
numbers.RemoveAt(0);
Console.WriteLine("After removal at index 0:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Check if an element exists (O(n) time complexity)
bool contains = numbers.Contains(40);
Console.WriteLine("Contains 40: " + contains); // Output: True
```

```
        // Sort the list (O(n log n) time complexity)
        numbers.Sort();
        Console.WriteLine("Sorted list:");
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }

        // Binary search on a sorted list (O(log n) time complexity)
        int index = numbers.BinarySearch(40);
        Console.WriteLine("Index of 40: " + index); // Output: 3
    }
}
```

5. How to Increase Performance When Using List<T>

1. **Preallocate Capacity**:
   - If you know the approximate size of the collection, use the `Capacity` property to preallocate memory and avoid frequent resizing.

   ```
   List<int> numbers = new List<int>(100); // Preallocate capacity
   ```

2. **Use `AddRange` for Bulk Operations**:
   - When adding multiple elements, use `AddRange` instead of multiple `Add` calls to minimize resizing.

   ```
   numbers.AddRange(new int[] { 60, 70, 80 });
   ```

3. **Avoid Frequent Insertions/Deletions in the Middle**:
   - Inserting or deleting elements in the middle of a `List<T>` is expensive (`O(n)`). Use `LinkedList<T>` if you need frequent modifications.

4. **Use `BinarySearch` for Searching of Data**:
   - If the list is sorted, use `BinarySearch` for faster searching (`O(log n)`).

5. **Use `TrimExcess` to Reduce Memory Usage**:
   - If the list size is fixed and won't grow further, use `TrimExcess` to reduce memory usage by removing unused capacity like `TrimSize` in `ArrayList`.

   ```
   numbers.TrimExcess();
   ```

6. **Avoid Unnecessary Copies**:
   - Use `List<T>.GetRange` or `List<T>.CopyTo` for bulk operations instead of manually copying elements.

Method	Performance	Memory Usage	Returns	Best For
`GetRange`	Slower	❌ Allocates new memory	`List<T>` (new instance)	When you need a new list
`CopyTo`	Faster	✅ No extra allocation	Copies to an existing array	When working with arrays efficiently

🔷 **Use `GetRange()` if you need a new `List<T>`.**

🔷 **Use `CopyTo()` for better performance and lower memory usage.** ✅

```
List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7 };

// GetRange: Creates a new List 🐢
List<int> rangeList = nums.GetRange(2, 3); // {3, 4, 5}

// CopyTo: Copies elements into an existing array 🚀
int[] copyArray = new int[3];
nums.CopyTo(2, copyArray, 0, 3); // {3, 4, 5}
```

7. **Use `Span<T>` or `Memory<T>` for Slicing**:
   - If you need to work with slices of a list without copying data, use `Span<T>` or `Memory<T>` for better performance.

Linked List

1. Definition

A **LinkedList<T>** is a generic collection in the `System.Collections.Generic` namespace that represents a doubly linked list. Each element in a `LinkedList<T>` is stored in a `LinkedListNode<T>`, which contains references to the previous and next nodes. This allows for efficient insertions and deletions at both ends and in the middle of the list.

---

2. When to Use

- **Best to use when**:
  - You need frequent insertions or deletions at the beginning, middle, or end of the list.
  - You don’t need random access to elements by index.

- You are working with a collection where the order of elements is important, and you need to maintain that order efficiently.

- **Avoid using when**:
  - You need fast, random access to elements by index (use `List<T>` or arrays instead).
  - Memory overhead is a concern (each node in a `LinkedList<T>` has additional overhead for storing references).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `LinkedList<T>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Access by index	`O(n)`	`O(n)`	Accessing an element by index requires traversing the list.
Add to beginning	`O(1)`	`O(1)`	Adding to the beginning is constant time.
Add to end	`O(1)`	`O(1)`	Adding to the end is constant time.
Insert at position	`O(1)`	`O(n)`	Inserting at a known node is `O(1)`, but finding the node is `O(n)`.
Remove by node	`O(1)`	`O(1)`	Removing a node is constant time if you have a reference to it.
Remove by value	`O(n)`	`O(n)`	Removing by value requires searching and then removing, which is linear time.
Search	`O(n)`	`O(n)`	Searching requires traversing the list in the worst case.
Space Complexity	`O(n)`	`O(n)`	Each node stores the element and two references (previous and next).

---

4. Code Example

Here's a simple, complete code example demonstrating the features of `LinkedList<T>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a LinkedList of integers
        LinkedList<int> numbers = new LinkedList<int>();

        // Add elements to the end of the list
        numbers.AddLast(10);
```

```
numbers.AddLast(20);
numbers.AddLast(30);

// Add elements to the beginning of the list
numbers.AddFirst(5);

// Insert an element after a specific node
LinkedListNode<int>? node = numbers.Find(20);
numbers.AddAfter(node!, 25);

// Insert an element before a specific node
numbers.AddBefore(node, 15);

// Access elements by iterating
Console.WriteLine("All elements:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Remove an element by value
numbers.Remove(15);
Console.WriteLine("After removing 15:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Remove the first element
numbers.RemoveFirst();
Console.WriteLine("After removing the first element:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}

// Remove the last element
numbers.RemoveLast();
Console.WriteLine("After removing the last element:");
foreach (int num in numbers)
{
    Console.WriteLine(num);
}
```

```
        // Check if an element exists
        bool contains = numbers.Contains(25);
        Console.WriteLine("Contains 25: " + contains); // Output: True

        // Access the first and last elements
        Console.WriteLine("First element: " + numbers.First.Value); // Output: 10
        Console.WriteLine("Last element: " + numbers.Last.Value);   // Output: 25
    }
}
```

5. How to Increase Performance When Using LinkedList<T>

1. **Use for Frequent Insertions/Deletions**:
   - `LinkedList<T>` excels when you need to frequently insert or delete elements at the beginning, middle, or end of the list. Use it in such scenarios.
2. **Avoid Random Access by Index**:
   - Accessing elements by index is slow (`O(n)`). If you need random access, consider using `List<T>` or arrays.
3. **Use `LinkedListNode<T>` for Efficient Operations**:
   - If you have a reference to a `LinkedListNode<T>`, you can perform insertions and deletions in `O(1)` time. Store node references if you need to perform frequent operations at specific positions.
4. **Avoid Frequent Searches**:
   - Searching for an element in a `LinkedList<T>` is `O(n)`. If you need frequent searches, consider using a `HashSet<T>` or `Dictionary<TKey, TValue>` for faster lookups.
5. **Use `AddFirst` and `AddLast` for Efficient Additions**:
   - Adding elements to the beginning or end of the list is `O(1)`. Use these methods when possible.
6. **Minimize Memory Overhead**:
   - Each node in a `LinkedList<T>` stores additional references (previous and next), which increases memory usage. Use `List<T>` if memory overhead is a concern.
7. **Combine with Other Data Structures**:
   - For complex scenarios, combine `LinkedList<T>` with other collections like `Dictionary<TKey, TValue>` to optimize performance.

Hashset

1. Definition

A `HashSet<T>` is a generic collection in the `System.Collections.Generic` namespace that stores unique elements. It is implemented using a hash table, which provides fast lookups, insertions, and deletions. Unlike `List<T>`, `HashSet<T>` does not allow duplicate elements and does not maintain the order of elements.

---

2. When to Use

- **Best to use when**:
  - You need to store a collection of unique elements.
  - You need fast lookups, insertions, and deletions.
  - You don't care about the order of elements.
  - You need to perform set operations like union, intersection, or difference.
- **Avoid using when**:
  - You need to store duplicate elements (use `List<T>` instead).
  - You need to maintain the order of elements (use `List<T>` or `LinkedList<T>` instead).
  - You need to access elements by index (use `List<T>` or arrays instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `HashSet<T>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	`O(1)`	`O(n)`	Adding an element is usually `O(1)`, but can degrade to `O(n)` due to resizing.
Remove	`O(1)`	`O(n)`	Removing an element is usually `O(1)`, but can degrade to `O(n)` due to resizing.
Contains	`O(1)`	`O(n)`	Checking if an element exists is usually `O(1)`, but can degrade to `O(n)`.
UnionWith	`O(m + n)`	`O(m + n)`	Union of two sets is linear in the size of both sets.
IntersectWith	`O(m + n)`	`O(m + n)`	Intersection of two sets is linear in the size of both sets.
ExceptWith	`O(m + n)`	`O(m + n)`	Difference of two sets is linear in the size of both sets.
Space Complexity	`O(n)`	`O(n)`	`HashSet<T>` uses a hash table, so space complexity is linear.

---

4. Code Example

Here's a simple, complete code example demonstrating the features of `HashSet<T>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a HashSet of integers
        HashSet<int> numbers = new HashSet<int>();

        // Add elements to the HashSet
        numbers.Add(10);
        numbers.Add(20);
        numbers.Add(30);
        numbers.Add(40);
        numbers.Add(50);

        // Try to add a duplicate element (will be ignored)
        bool added = numbers.Add(10);
        Console.WriteLine("Was 10 added again? " + added); // Output: False

        // Check if an element exists
        bool contains = numbers.Contains(30);
        Console.WriteLine("Contains 30? " + contains); // Output: True

        // Remove an element
        numbers.Remove(20);
        Console.WriteLine("After removing 20:");
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }

        // Create another HashSet
        HashSet<int> otherNumbers = new HashSet<int> { 30, 40, 50, 60, 70 };

        // Perform a union of two sets
        numbers.UnionWith(otherNumbers);
        Console.WriteLine("After union with otherNumbers:");
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }
```

```
        // Perform an intersection of two sets
        numbers.IntersectWith(otherNumbers);
        Console.WriteLine("After intersection with otherNumbers:");
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }

        // Perform a difference of two sets
        numbers.ExceptWith(otherNumbers);
        Console.WriteLine("After difference with otherNumbers:");
        foreach (int num in numbers)
        {
            Console.WriteLine(num);
        }

        // Clear the HashSet
        numbers.Clear();
        Console.WriteLine("After clearing the HashSet, count: " + numbers.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using HashSet<T>

1. **Use for Fast Lookups**:
   - `HashSet<T>` is ideal for scenarios where you need to check for the existence of elements quickly (`O(1)` on average).
2. **Avoid Duplicates**:
   - Use `HashSet<T>` when you need to ensure that all elements in the collection are unique.
3. **Preallocate Capacity**:
   - If you know the approximate size of the collection, use the constructor that accepts an initial capacity to minimize resizing.

```
HashSet<int> numbers = new HashSet<int>(100); // Preallocate capacity
```

4. **Use Set Operations Efficiently**:
   - Use methods like `UnionWith`, `IntersectWith`, and `ExceptWith` for efficient set operations.
5. **Avoid Frequent Resizing**:

- Resizing a `HashSet<T>` can be expensive. Preallocate capacity or use a larger initial size if you expect the collection to grow.

6. **Use Custom Equality Comparer**:
   - If you are storing custom objects, override `GetHashCode` and `Equals` methods or provide a custom `IEqualityComparer<T>` to ensure proper hashing and equality checks.
7. **Minimize Boxing/Unboxing**:
   - If you are storing value types, ensure that the hash function is efficient to avoid performance degradation.

Dictionary

1. Definition

A **Dictionary<TKey, TValue>** is a generic collection in the `System.Collections.Generic` namespace that stores key-value pairs. It is implemented using a hash table, which provides fast lookups, insertions, and deletions based on the key. Each key in a dictionary must be unique.

---

2. When to Use

- **Best to use when**:
  - You need to store key-value pairs and retrieve values quickly by key.
  - You need fast lookups, insertions, and deletions.
  - You need to ensure that all keys are unique.
  - You don't care about the order of elements.
- **Avoid using when**:
  - You need to store duplicate keys (use `Lookup<TKey, TValue>` or a custom solution instead).
  - You need to maintain the order of elements (use `SortedDictionary<TKey, TValue>` or `SortedList<TKey, TValue>` instead).
  - You need to access elements by index (use `List<T>` or arrays instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `Dictionary<TKey, TValue>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	`O(1)`	`O(n)`	Adding a key-value pair is usually `O(1)`, but can degrade to `O(n)` due to resizing.
Remove	`O(1)`	`O(n)`	Removing a key-value pair is usually `O(1)`, but can degrade to `O(n)` due to resizing.

Access by key	`O(1)`	`O(n)`	Accessing a value by key is usually `O(1)`, but can degrade to `O(n)`.
ContainsKey	`O(1)`	`O(n)`	Checking if a key exists is usually `O(1)`, but can degrade to `O(n)`.
ContainsValue	`O(n)`	`O(n)`	Checking if a value exists requires iterating through all elements.
Space Complexity	`O(n)`	`O(n)`	`Dictionary<TKey, TValue>` uses a hash table, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `Dictionary<TKey, TValue>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a Dictionary with string keys and int values
        Dictionary<string, int> ages = new Dictionary<string, int>();

        // Add key-value pairs to the Dictionary
        ages.Add("Alice", 25);
        ages.Add("Bob", 30);
        ages.Add("Charlie", 35);

        // Access values by key (O(1) time complexity)
        Console.WriteLine("Alice's age: " + ages["Alice"]); // Output: 25

        // Update a value by key
        ages["Bob"] = 31;
        Console.WriteLine("Bob's updated age: " + ages["Bob"]); // Output: 31

        // Check if a key exists
        bool containsKey = ages.ContainsKey("Charlie");
        Console.WriteLine("Contains key 'Charlie'? " + containsKey); // Output: True

        // Check if a value exists (O(n) time complexity)
        bool containsValue = ages.ContainsValue(40);
        Console.WriteLine("Contains value 40? " + containsValue); // Output: False

        // Remove a key-value pair
```

```
        ages.Remove("Alice");
        Console.WriteLine("After removing Alice:");
        foreach (var kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }

        // Iterate through the Dictionary
        Console.WriteLine("All key-value pairs:");
        foreach (KeyValuePair<string, int> kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }

        // Clear the Dictionary
        ages.Clear();
        Console.WriteLine("After clearing the Dictionary, count: " + ages.Count);
        // Output: 0
    }
}
```

5. How to Increase Performance When Using Dictionary<TKey, TValue>

1. **Use for Fast Lookups**:
   - `Dictionary<TKey, TValue>` is ideal for scenarios where you need to retrieve values quickly by key (`O(1)` on average).
2. **Avoid Duplicate Keys**:
   - Ensure that all keys are unique. Use `TryAdd` or check with `ContainsKey` before adding a new key-value pair.
3. **Preallocate Capacity**:
   - If you know the approximate size of the collection, use the constructor that accepts an initial capacity to minimize resizing.

   ```
   Dictionary<string, int> ages = new Dictionary<string, int>(100); // Preallocate capacity
   ```
4. **Use Custom Equality Comparer**:
   - If you are using custom objects as keys, override `GetHashCode` and `Equals` methods or provide a custom `IEqualityComparer<TKey>` to ensure proper hashing and equality checks.
5. **Avoid Frequent Resizing**:

- Resizing a `Dictionary<TKey, TValue>` can be expensive. Preallocate capacity or use a larger initial size if you expect the collection to grow.

6. **Minimize Boxing/Unboxing**:
   - If you are using value types as keys, ensure that the hash function is efficient to avoid performance degradation.
7. **Use `TryGetValue` for Safe Lookups**:
   - Use `TryGetValue` to safely retrieve values without throwing exceptions if the key does not exist.

```
if (ages.TryGetValue("Alice", out int age))
{
    Console.WriteLine("Alice's age: " + age);
}
else
{
    Console.WriteLine("Alice not found.");
}
```

8. **Avoid Using `ContainsValue`**:
   - Checking for the existence of a value is `O(n)`. If you need frequent value lookups, consider maintaining a reverse dictionary or a separate collection.

ListDictionary

1. Definition

A **ListDictionary** is a non-generic collection in the `System.Collections.Specialized` namespace that stores key-value pairs. It is optimized for small collections (typically fewer than 10 items) and uses a singly linked list internally. Unlike `Hashtable` or `Dictionary<TKey, TValue>`, it does not provide fast lookups for large collections but is memory-efficient for small datasets.

---

2. When to Use

- **Best to use when**:
  - You need to store key-value pairs in a small collection (fewer than 10 items).
  - Memory efficiency is critical, and performance is not a primary concern.
  - You are working with a non-generic collection and do not need type safety.
- **Avoid using when**:
  - You need to store a large number of key-value pairs (use `Dictionary<TKey, TValue>` instead).
  - You need fast lookups, insertions, or deletions for large collections.

- You need type safety (use `Dictionary<TKey, TValue>` instead).

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `ListDictionary` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	`O(1)`	`O(n)`	Adding a key-value pair is `O(1)` for small collections but `O(n)` for large collections.
Remove	`O(1)`	`O(n)`	Removing a key-value pair is `O(1)` for small collections but `O(n)` for large collections.
Access by key	`O(n)`	`O(n)`	Accessing a value by key requires traversing the linked list.
ContainsKey	`O(n)`	`O(n)`	Checking if a key exists requires traversing the linked list.
Iteration	`O(n)`	`O(n)`	Iterating through all key-value pairs is linear time.
Space Complexity	`O(n)`	`O(n)`	`ListDictionary` uses a singly linked list, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `ListDictionary` in C#:

```
using System;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        // Create a ListDictionary
        ListDictionary listDict = new ListDictionary();

        // Add key-value pairs
        listDict.Add("Alice", 25);
        listDict.Add("Bob", 30);
        listDict.Add("Charlie", 35);

        // Access values by key
        Console.WriteLine("Alice's age: " + listDict["Alice"]); // Output: 25

        // Update a value by key
        listDict["Bob"] = 31;
        Console.WriteLine("Bob's updated age: " + listDict["Bob"]); // Output: 31
```

```
        // Check if a key exists
        bool containsKey = listDict.Contains("Charlie");
        Console.WriteLine("Contains key 'Charlie'? " + containsKey); // Output: True

        // Remove a key-value pair
        listDict.Remove("Alice");
        Console.WriteLine("After removing Alice:");
        foreach (DictionaryEntry entry in listDict)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }

        // Iterate through the ListDictionary
        Console.WriteLine("All key-value pairs:");
        foreach (DictionaryEntry entry in listDict)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }

        // Clear the ListDictionary
        listDict.Clear();
        Console.WriteLine("After clearing the ListDictionary, count: " + listDict.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using ListDictionary

1. **Use for Small Collections**:
   - `ListDictionary` is optimized for small collections (fewer than 10 items). Use it when memory efficiency is more important than performance.
2. **Avoid Frequent Searches**:
   - Searching for keys or values is `O(n)`. If you need frequent lookups, consider using `Dictionary<TKey, TValue>` or `Hashtable`.
3. **Avoid Frequent Insertions/Deletions**:
   - Insertions and deletions are `O(n)` for large collections. Use `Dictionary<TKey, TValue>` if you need frequent modifications.
4. **Use `Contains` for Key Existence Checks**:
   - Use the `Contains` method to check if a key exists in the collection.
5. **Prefer Generic Collections for Type Safety**:

- If you need type safety, use `Dictionary<TKey, TValue>` instead.

6. **Combine with Other Data Structures**:
   - For complex scenarios, combine `ListDictionary` with other collections like `Dictionary<TKey, TValue>` or `List<T>` to optimize performance.
7. **Avoid Using `ListDictionary` for Large Collections**:
   - For large collections, `ListDictionary` is not efficient. Use `Dictionary<TKey, TValue>` or `Hashtable` instead.

Hybrid Dictionary

1. Definition

A **HybridDictionary** is a collection in the `System.Collections.Specialized` namespace that optimizes memory usage by switching between a `ListDictionary` and a `Hashtable` based on the number of elements. It is designed to provide efficient performance for both small and large collections.

- For **small collections** (typically fewer than 10 items), it uses a `ListDictionary`, which is a singly linked list.
- For **larger collections**, it automatically switches to a `Hashtable`, which provides faster lookups.
- it converted directly From `ListDictionary` to `Hashtable` when items increased over 10

---

2. When to Use

- **Best to use when**:
  - You need a dictionary-like collection that is memory-efficient for small collections.
  - You expect the collection to grow over time and want automatic optimization for both small and large sizes.
  - You need to store key-value pairs with fast lookups.
- **Avoid using when**:
  - You need a generic collection (use `Dictionary<TKey, TValue>` instead).
  - You need to store value types without boxing/unboxing overhead.
  - You are working with a fixed-size collection.

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `HybridDictionary` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation

Add	O(1)	O(n)	Adding a key-value pair is O(1) for small collections and O(1) for large collections (amortized).
Remove	O(1)	O(n)	Removing a key-value pair is O(1) for small collections and O(1) for large collections.
Access by key	O(n)	O(1)	Accessing a value by key is O(n) for small collections and O(1) for large collections.
ContainsKey	O(n)	O(1)	Checking if a key exists is O(n) for small collections and O(1) for large collections.
Space Complexity	O(n)	O(n)	Space usage is linear, but it is optimized for small collections using ListDictionary.

4. Code Example

Here's a simple, complete code example demonstrating the features of HybridDictionary in C#:

```
using System;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        // Create a HybridDictionary
        HybridDictionary hybridDict = new HybridDictionary();

        // Add key-value pairs to the HybridDictionary
        hybridDict.Add("Alice", 25);
        hybridDict.Add("Bob", 30);
        hybridDict.Add("Charlie", 35);

        // Access values by key
        Console.WriteLine("Alice's age: " + hybridDict["Alice"]); // Output: 25

        // Update a value by key
        hybridDict["Bob"] = 31;
        Console.WriteLine("Bob's updated age: " + hybridDict["Bob"]); // Output: 31

        // Check if a key exists
        bool containsKey = hybridDict.Contains("Charlie");
        Console.WriteLine("Contains key 'Charlie'? " + containsKey); // Output: True

        // Remove a key-value pair
        hybridDict.Remove("Alice");
```

```
        Console.WriteLine("After removing Alice:");
        foreach (DictionaryEntry entry in hybridDict)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }

        // Iterate through the HybridDictionary
        Console.WriteLine("All key-value pairs:");
        foreach (DictionaryEntry entry in hybridDict)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }

        // Clear the HybridDictionary
        hybridDict.Clear();
        Console.WriteLine("After clearing the HybridDictionary, count: " + hybridDict.Count); // Out
put: 0
    }
}
```

5. How to Increase Performance When Using HybridDictionary

1. **Use for Small to Medium-Sized Collections**:
   - `HybridDictionary` is optimized for small collections. Use it when you expect the collection to start small and grow over time.
2. **Avoid Frequent Resizing**:
   - If you know the approximate size of the collection, use a `Dictionary<TKey, TValue>` instead to avoid the overhead of switching between `ListDictionary` and `Hashtable`.
3. **Use `Dictionary<TKey, TValue>` for Large Collections**:
   - For large collections, `Dictionary<TKey, TValue>` is more efficient and provides better performance.
4. **Avoid Boxing/Unboxing**:
   - `HybridDictionary` stores keys and values as `object`, which can lead to boxing/unboxing overhead for value types. Use `Dictionary<TKey, TValue>` for better performance with value types.
5. **Use `Contains` for Key Lookups**:
   - Use the `Contains` method to check if a key exists in the collection.
6. **Minimize Iterations**:
   - Iterating through a `HybridDictionary` can be slower for small collections due to the use of `ListDictionary`. Use `Dictionary<TKey, TValue>` if you need frequent iterations.

7. **Consider Thread Safety**:
    - `HybridDictionary` is not thread-safe. If you need thread safety, consider using `ConcurrentDictionary<TKey, TValue>`.

Sorted Dictionary

1. Definition

A **SortedDictionary<TKey, TValue>** is a generic collection in the `System.Collections.Generic` namespace that stores key-value pairs in sorted order based on the keys. It is implemented using a self-balancing binary search tree (typically a Red-Black Tree), which ensures that elements are always sorted and provides efficient lookups, insertions, and deletions.

---

2. When to Use

- **Best to use when**:
    - You need to maintain key-value pairs in sorted order.
    - You need fast lookups, insertions, and deletions while keeping the data sorted.
    - You need to perform range queries or iterate through elements in sorted order.
- **Avoid using when**:
    - You don't need the elements to be sorted (use `Dictionary<TKey, TValue>` instead for better performance).
    - You need to store duplicate keys (use `Lookup<TKey, TValue>` or a custom solution instead).
    - You need to access elements by index (use `List<T>` or arrays instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `SortedDictionary<TKey, TValue>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	`O(log n)`	`O(log n)`	Adding a key-value pair is logarithmic time due to the balanced tree structure.
Remove	`O(log n)`	`O(log n)`	Removing a key-value pair is logarithmic time.
Access by key	`O(log n)`	`O(log n)`	Accessing a value by key is logarithmic time.
ContainsKey	`O(log n)`	`O(log n)`	Checking if a key exists is logarithmic time.
ContainsValue	`O(n)`	`O(n)`	Checking if a value exists requires iterating through all elements.
Iteration	`O(n)`	`O(n)`	Iterating through all elements is linear time.

Space Complexity	`O(n)`	`O(n)`	`SortedDictionary<TKey, TValue>` uses a balanced tree, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `SortedDictionary<TKey, TValue>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a SortedDictionary with string keys and int values
        SortedDictionary<string, int> ages = new SortedDictionary<string, int>();

        // Add key-value pairs to the SortedDictionary
        ages.Add("Charlie", 35);
        ages.Add("Alice", 25);
        ages.Add("Bob", 30);

        // Access values by key (O(log n) time complexity)
        Console.WriteLine("Alice's age: " + ages["Alice"]); // Output: 25

        // Update a value by key
        ages["Bob"] = 31;
        Console.WriteLine("Bob's updated age: " + ages["Bob"]); // Output: 31

        // Check if a key exists
        bool containsKey = ages.ContainsKey("Charlie");
        Console.WriteLine("Contains key 'Charlie'? " + containsKey); // Output: True

        // Check if a value exists (O(n) time complexity)
        bool containsValue = ages.ContainsValue(40);
        Console.WriteLine("Contains value 40? " + containsValue); // Output: False

        // Remove a key-value pair
        ages.Remove("Alice");
        Console.WriteLine("After removing Alice:");
        foreach (var kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
```

```
        }

        // Iterate through the SortedDictionary (in sorted order)
        Console.WriteLine("All key-value pairs (sorted by key):");
        foreach (KeyValuePair<string, int> kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }

        // Clear the SortedDictionary
        ages.Clear();
        Console.WriteLine("After clearing the SortedDictionary, count: " + ages.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using SortedDictionary<TKey, TValue>

1. **Use for Sorted Data**:
   - `SortedDictionary<TKey, TValue>` is ideal for scenarios where you need to maintain key-value pairs in sorted order.
2. **Avoid Frequent Insertions/Deletions**:
   - While insertions and deletions are efficient (`O(log n)`), they are slower than `Dictionary<TKey, TValue>`. Use `Dictionary<TKey, TValue>` if you don't need sorted data.
3. **Use Custom Comparer**:
   - If you need custom sorting logic, provide a custom `IComparer<TKey>` to the constructor.

```
SortedDictionary<string, int> ages = new SortedDictionary<string, int>(StringComparer.OrdinalIgnoreCase);
```

4. **Avoid Using `ContainsValue`**:
   - Checking for the existence of a value is `O(n)`. If you need frequent value lookups, consider maintaining a reverse dictionary or a separate collection.
5. **Use `TryGetValue` for Safe Lookups**:
   - Use `TryGetValue` to safely retrieve values without throwing exceptions if the key does not exist.

```
if (ages.TryGetValue("Alice", out int age))
{
    Console.WriteLine("Alice's age: " + age);
}
else
```

```
{
    Console.WriteLine("Alice not found.");
}
```

6. **Minimize Iterations**:
    - Iterating through a `SortedDictionary<TKey, TValue>` is `O(n)`. If you need frequent iterations, ensure that the collection size is manageable.
7. **Prefer `Dictionary<TKey, TValue>` for Unsorted Data**:
    - If you don't need sorted data, use `Dictionary<TKey, TValue>` for faster lookups, insertions, and deletions (`O(1)` on average).

SortedList

1. Definition

A **SortedList<TKey, TValue>** is a generic collection in the `System.Collections.Generic` namespace that stores key-value pairs in sorted order based on the keys. It is implemented using two internal arrays: one for keys and one for values. The keys are always sorted, which allows for efficient lookups and range queries.

---

2. When to Use

- **Best to use when**:
    - You need to maintain key-value pairs in sorted order.
    - You need fast lookups by key and efficient iteration in sorted order.
    - You need to perform range queries or access elements by index.
    - Memory usage is a concern (it uses less memory than `SortedDictionary<TKey, TValue>` for small to medium-sized collections).
- **Avoid using when**:
    - You need frequent insertions or deletions (use `SortedDictionary<TKey, TValue>` instead for better performance).
    - You don't need the elements to be sorted (use `Dictionary<TKey, TValue>` instead for better performance).
    - You need to store duplicate keys (use `Lookup<TKey, TValue>` or a custom solution instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `SortedList<TKey, TValue>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	O(n)	O(n)	Adding a key-value pair is linear time due to the need to maintain sorted order.
Remove	O(n)	O(n)	Removing a key-value pair is linear time due to the need to maintain sorted order.
Access by key	O(log n)	O(log n)	Accessing a value by key is logarithmic time due to binary search.
Access by index	O(1)	O(1)	Accessing a value by index is constant time.
ContainsKey	O(log n)	O(log n)	Checking if a key exists is logarithmic time.
ContainsValue	O(n)	O(n)	Checking if a value exists requires iterating through all elements.
Iteration	O(n)	O(n)	Iterating through all elements is linear time.
Space Complexity	O(n)	O(n)	`SortedList<TKey, TValue>` uses two arrays, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `SortedList<TKey, TValue>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a SortedList with string keys and int values
        SortedList<string, int> ages = new SortedList<string, int>();

        // Add key-value pairs to the SortedList
        ages.Add("Charlie", 35);
        ages.Add("Alice", 25);
        ages.Add("Bob", 30);

        // Access values by key (O(log n) time complexity)
        Console.WriteLine("Alice's age: " + ages["Alice"]); // Output: 25

        // Update a value by key
        ages["Bob"] = 31;
        Console.WriteLine("Bob's updated age: " + ages["Bob"]); // Output: 31

        // Check if a key exists
        bool containsKey = ages.ContainsKey("Charlie");
```

```
        Console.WriteLine("Contains key 'Charlie'? " + containsKey); // Output: True

        // Check if a value exists (O(n) time complexity)
        bool containsValue = ages.ContainsValue(40);
        Console.WriteLine("Contains value 40? " + containsValue); // Output: False

        // Remove a key-value pair
        ages.Remove("Alice");
        Console.WriteLine("After removing Alice:");
        foreach (var kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }

        // Iterate through the SortedList (in sorted order)
        Console.WriteLine("All key-value pairs (sorted by key):");
        foreach (KeyValuePair<string, int> kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }

        // Access elements by index
        Console.WriteLine("Element at index 0: " + ages.Values[0]); // Output: 31 (Bob's age)
        Console.WriteLine("Key at index 0: " + ages.Keys[0]);       // Output: Bob

        // Clear the SortedList
        ages.Clear();
        Console.WriteLine("After clearing the SortedList, count: " + ages.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using SortedList<TKey, TValue>

1. **Use for Sorted Data with Fast Lookups**:
   - `SortedList<TKey, TValue>` is ideal for maintaining key-value pairs in sorted order and performing efficient lookups by key.
2. **Avoid Frequent Insertions/Deletions**:
   - Insertions and deletions are `O(n)` due to the need to maintain sorted order. Use `SortedDictionary<TKey, TValue>` if you need frequent modifications.
3. **Use Custom Comparer**:
   - If you need custom sorting logic, provide a custom `IComparer<TKey>` to the constructor.

```
SortedList<string, int> ages = new SortedList<string, int>(StringComparer.OrdinalIgnoreCase);
```

4. **Avoid Using `ContainsValue`**:
   - Checking for the existence of a value is `O(n)`. If you need frequent value lookups, consider maintaining a reverse dictionary or a separate collection.
5. **Use `TryGetValue` for Safe Lookups**:
   - Use `TryGetValue` to safely retrieve values without throwing exceptions if the key does not exist.

```
if (ages.TryGetValue("Alice", out int age))
{
    Console.WriteLine("Alice's age: " + age);
}
else
{
    Console.WriteLine("Alice not found.");
}
```

6. **Leverage Index-Based Access**:
   - Use the `Keys` and `Values` properties to access elements by index in `O(1)` time.
7. **Prefer `Dictionary<TKey, TValue>` for Unsorted Data**:
   - If you don't need sorted data, use `Dictionary<TKey, TValue>` for faster lookups, insertions, and deletions (`O(1)` on average).

Sortedset

Definition:

`SortedSet<T>` is a generic collection in C# that stores unique elements in a sorted order. It is part of the `System.Collections.Generic` namespace. `SortedSet<T>` uses a red-black tree internally to maintain the sorted order of elements.

When to Use:

`SortedSet<T>` is best used when you need a collection that:

1. Maintains elements in sorted order.
2. Does not allow duplicate elements.
3. Requires efficient set operations like union, intersection, and difference.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Search	O(log n)	O(log n)	Searching for an element in the sorted set.
Addition	O(log n)	O(log n)	Adding a new element to the sorted set.
Removal	O(log n)	O(log n)	Removing an element from the sorted set.
Iteration	O(n)	O(n)	Iterating through all elements in the sorted set.

Simple Complete Code Example:

Here's an example demonstrating various features of `SortedSet<T>`:

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

class Program
{
    static void Main()
    {
        SortedSet<int> numbers = new SortedSet<int>();

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

        // The elements are automatically sorted
        Console.WriteLine("Sorted elements:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        // Searching for an element
        bool contains = numbers.Contains(20);
        Console.WriteLine("Contains 20: " + contains);

        // Removing an element
        numbers.Remove(20);
```

```
        Console.WriteLine("Elements after removal:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        // Set operations
        SortedSet<int> anotherSet = new SortedSet<int> { 20, 30, 40 };
        numbers.UnionWith(anotherSet);
        Console.WriteLine("Elements after union:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        numbers.IntersectWith(anotherSet);
        Console.WriteLine("Elements after intersection:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        numbers.ExceptWith(anotherSet);
        Console.WriteLine("Elements after difference:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

Output:

```
Sorted elements:
10
20
30
Contains 20: True
Elements after removal:
10
30
Elements after union:
10
```

```
20
30
40
Elements after intersection:
20
30
Elements after difference:
10
40
```

How to Increase Performance When Using SortedSet:

1. **Pre-sorting**: If you're adding a large number of elements at once, consider pre-sorting them before adding them to the `SortedSet` to reduce the overhead of maintaining order.
2. **Batch Operations**: Use batch operations like `UnionWith`, `IntersectWith`, and `ExceptWith` to perform set operations efficiently.
3. **Custom Comparer**: If you have specific sorting requirements, provide a custom `IComparer<T>` to control the sorting order, which can be more efficient than the default comparer.
4. **Avoid Unnecessary Searches**: Since searches are O(log n), they can become costly in a large set. If possible, structure your program to minimize the number of searches needed.

Type	**Time Complexity** (for common operations)	**Use Case**	**Key Differences**
SortedDictionary 📜	- Insert: O(log n) - Search: O(log n) - Remove: O(log n)	- Key-value pairs - Requires fast access and sorting by key	- Stores key-value pairs - Internally implemented as a red-black tree - Ordered by key
SortedSet 🔑	- Insert: O(log n) - Search: O(log n) - Remove: O(log n)	- Unique elements only - When order and uniqueness matter	- Stores only unique elements - Internally implemented as a red-black tree - Ordered by element value
SortedList 📄	- Insert: O(n) - Search: O(log n) - Remove: O(n)	- Key-value pairs - When memory efficiency and fast search are important	- Stores key-value pairs - Internally implemented as an array - Ordered by key

When to Choose Each?

- `SortedDictionary` is great for fast searches and maintaining both keys and values 🔑➡️🔢.
- `SortedSet` is ideal if you just need **unique sorted keys** 🔑.

- `SortedList` is useful when you have **few elements** and need fast access with minimal changes 📋.

.

Stack

1. Definition

A **Stack<T>** is a generic collection in the `System.Collections.Generic` namespace that represents a last-in, first-out (LIFO) collection of elements. It provides fast push (add) and pop (remove) operations for the top element, making it ideal for scenarios where you need to process elements in reverse order.

---

2. When to Use

- **Best to use when**:
  - You need to process elements in a last-in, first-out (LIFO) order.
  - You need fast push and pop operations.
  - You are implementing algorithms like depth-first search (DFS), expression evaluation, or undo/redo functionality.
- **Avoid using when**:
  - You need to access elements by index (use `List<T>` or arrays instead).
  - You need to process elements in a first-in, first-out (FIFO) order (use `Queue<T>` instead).
  - You need to frequently search for elements (use `HashSet<T>` or `Dictionary<TKey, TValue>` instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `Stack<T>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Push	`O(1)`	`O(n)`	Adding an element to the top of the stack is usually `O(1)`, but can degrade to `O(n)` due to resizing.
Pop	`O(1)`	`O(1)`	Removing the top element is constant time.
Peek	`O(1)`	`O(1)`	Accessing the top element without removing it is constant time.
Contains	`O(n)`	`O(n)`	Checking if an element exists requires iterating through the stack.
Space Complexity	`O(n)`	`O(n)`	`Stack<T>` uses an internal array, so space complexity is linear.

---

4. Code Example

Here's a simple, complete code example demonstrating the features of `Stack<T>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a Stack of integers
        Stack<int> stack = new Stack<int>();

        // Push elements onto the stack
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        // Peek at the top element without removing it
        Console.WriteLine("Top element: " + stack.Peek()); // Output: 30

        // Pop elements from the stack
        Console.WriteLine("Popped element: " + stack.Pop()); // Output: 30
        Console.WriteLine("Popped element: " + stack.Pop()); // Output: 20

        // Check if the stack contains an element
        bool contains = stack.Contains(10);
        Console.WriteLine("Contains 10? " + contains); // Output: True

        // Iterate through the stack (note: this does not change the stack)
        Console.WriteLine("All elements in the stack:");
        foreach (int item in stack)
        {
            Console.WriteLine(item); // Output: 10
        }

        // Clear the stack
        stack.Clear();
        Console.WriteLine("After clearing the stack, count: " + stack.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using Stack<T>

1. **Use for LIFO Scenarios**:
   - `Stack<T>` is ideal for scenarios where you need to process elements in reverse order, such as undo/redo functionality or depth-first search (DFS).
2. **Avoid Frequent Resizing**:
   - If you know the approximate size of the stack, use the constructor that accepts an initial capacity to minimize resizing.

```
Stack<int> stack = new Stack<int>(100); // Preallocate capacity
```

3. **Use `Peek` Instead of `Pop` for Inspection**:
   - Use `Peek` to inspect the top element without removing it. This avoids unnecessary modifications to the stack.
4. **Avoid Using `Contains`**:
   - Checking if an element exists is `O(n)`. If you need frequent lookups, consider using a `HashSet<T>` or `Dictionary<TKey, TValue>`.
5. **Use `TryPop` for Safe Removal**:
   - Use `TryPop` to safely remove the top element without throwing an exception if the stack is empty.

```
if (stack.TryPop(out int result))
{
    Console.WriteLine("Popped element: " + result);
}
else
{
    Console.WriteLine("Stack is empty.");
}
```

6. **Minimize Iterations**:
   - Iterating through a `Stack<T>` is `O(n)`. If you need frequent iterations, ensure that the stack size is manageable.
7. **Combine with Other Data Structures**:
   - For complex scenarios, combine `Stack<T>` with other collections like `Queue<T>` or `List<T>` to optimize performance.

Queue

Definition:
A

`Queue<T>` in C# is a generic collection that follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed. It is part of the `System.Collections.Generic` namespace.

When to Use:
Use a
`Queue<T>` when you need to maintain the order in which elements were added and you want to perform operations in that order. It's ideal for scenarios such as:

- Managing tasks in a processing queue.
- Handling requests in a server.
- Implementing breadth-first search algorithms.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Enqueue	O(1)	O(n)	Adding an element to the end of the queue. In the worst case, this may require resizing the internal array.
Dequeue	O(n)	O(n)	Removing an element from the front of the queue. This operation may require shifting all elements.
Peek	O(1)	O(1)	Getting the value at the front of the queue without removing it.
Count	O(1)	O(1)	Getting the number of elements in the queue.

Simple Complete Code Example:
Here's an example demonstrating various features of
`Queue<T>`:

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

class Program
{
    static void Main()
    {
        Queue<int> queue = new Queue<int>();

        // Enqueue elements
        queue.Enqueue(10);
        queue.Enqueue(20);
        queue.Enqueue(30);
```

```
        // Peek at the front element
        Console.WriteLine("Front element: " + queue.Peek());

        // Dequeue elements
        Console.WriteLine("Dequeued: " + queue.Dequeue());
        Console.WriteLine("Dequeued: " + queue.Dequeue());

        // Check the count of elements
        Console.WriteLine("Count of elements: " + queue.Count);

        // Iterate through the queue
        foreach (int number in queue)
        {
            Console.WriteLine(number);
        }
    }
}
```

Output:

```
Front element: 10
Dequeued: 10
Dequeued: 20
Count of elements: 1
30
```

How to Increase Performance When Using Queue:

1. **Pre-allocate Capacity**: If you know the approximate number of elements in advance, you can pre-allocate the capacity using `Queue<T>.Capacity` to avoid multiple reallocations.
2. **Batch Operations**: Use batch operations like `EnqueueRange` to add multiple elements at once instead of adding them one by one.
3. **Avoid Frequent Resizing**: Be mindful of the resizing behavior of `Queue<T>`. If you're adding a large number of elements, consider pre-allocating the necessary space to minimize the number of resize operations.
4. **Use ConcurrentQueue for Multi-threading**: If you're working in a multi-threaded environment and need thread-safe operations, consider using `ConcurrentQueue<T>` instead of `Queue<T>`. It provides thread-safe methods for enqueuing and dequeuing elements.

PriorityQueue

1. **Definition**

A **PriorityQueue<TElement, TPriority>** is a generic collection introduced in .NET 6 that represents a queue where each element has a priority. Elements are dequeued in order of their priority (highest priority first). It is implemented using a **min-heap** or **max-heap** data structure, which ensures efficient insertion and removal of elements based on priority.

2. When to Use

- **Best to use when**:
  - You need to process elements based on their priority.
  - You need efficient enqueue and dequeue operations for prioritized elements.
  - You are implementing algorithms like Dijkstra's shortest path, A* search, or task scheduling.
- **Avoid using when**:
  - You need a simple first-in, first-out (FIFO) queue (use `Queue<T>` instead).
  - You need to access elements by index (use `List<T>` or arrays instead).
  - You need to frequently search for elements (use `HashSet<T>` or `Dictionary<TKey, TValue>` instead).

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `PriorityQueue<TElement, TPriority>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Enqueue	`O(log n)`	`O(log n)`	Adding an element with a priority is logarithmic time due to heap maintenance.
Dequeue	`O(log n)`	`O(log n)`	Removing the highest-priority element is logarithmic time.
Peek	`O(1)`	`O(1)`	Accessing the highest-priority element without removing it is constant time.
Contains	`O(n)`	`O(n)`	Checking if an element exists requires iterating through the heap.
Space Complexity	`O(n)`	`O(n)`	`PriorityQueue<TElement, TPriority>` uses a heap, so space complexity is linear.

4. Code Example

Here's a simple, complete code example demonstrating the features of `PriorityQueue<TElement, TPriority>` in C#:

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

```
class Program
{
    static void Main()
    {
        // Create a PriorityQueue where elements are integers and priorities are integers
        PriorityQueue<int, int> priorityQueue = new PriorityQueue<int, int>();

        // Enqueue elements with priorities
        priorityQueue.Enqueue(10, 3); // Element: 10, Priority: 3
        priorityQueue.Enqueue(20, 1); // Element: 20, Priority: 1
        priorityQueue.Enqueue(30, 2); // Element: 30, Priority: 2

        // Peek at the highest-priority element without removing it
        if (priorityQueue.TryPeek(out int topElement, out int topPriority))
        {
            Console.WriteLine($"Top element: {topElement}, Priority: {topPriority}"); // Output: Top el
ement: 20, Priority: 1
        }

        // Dequeue elements in order of priority
        while (priorityQueue.TryDequeue(out int element, out int priority))
        {
            Console.WriteLine($"Dequeued element: {element}, Priority: {priority}");
        }
        // Output:
        // Dequeued element: 20, Priority: 1
        // Dequeued element: 30, Priority: 2
        // Dequeued element: 10, Priority: 3

        // Check if the priority queue is empty
        Console.WriteLine("Is priority queue empty? " + (priorityQueue.Count == 0)); // Output: Tru
e
    }
}
```

5. How to Increase Performance When Using PriorityQueue<TElement, TPriority>

1. **Use for Priority-Based Processing**:
    - `PriorityQueue<TElement, TPriority>` is ideal for scenarios where you need to process elements based on their priority, such as task scheduling or graph algorithms.

2. **Avoid Frequent Contains Checks**:
   - Checking if an element exists is `O(n)`. If you need frequent lookups, consider using a `HashSet<T>` or `Dictionary<TKey, TValue>` alongside the priority queue.
3. **Use Custom Comparer for Priorities**:
   - If you need custom priority logic, provide a custom `IComparer<TPriority>` to the constructor.

```
var priorityQueue = new PriorityQueue<int, int>(Comparer<int>.Create((x, y) => y.CompareTo(x))); // Max-heap
```

4. **Use `TryDequeue` and `TryPeek` for Safe Operations**:
   - Use `TryDequeue` and `TryPeek` to safely remove or inspect the highest-priority element without throwing exceptions if the queue is empty.
5. **Minimize Heap Resizing**:
   - If you know the approximate size of the queue, use the constructor that accepts an initial capacity to minimize resizing.

```
var priorityQueue = new PriorityQueue<int, int>(initialCapacity: 100);
```

6. **Avoid Frequent Enqueue/Dequeue Operations**:
   - While enqueue and dequeue operations are efficient (`O(log n)`), frequent operations can still impact performance. Optimize your algorithm to minimize unnecessary operations.
7. **Combine with Other Data Structures**:
   - For complex scenarios, combine `PriorityQueue<TElement, TPriority>` with other collections like `Dictionary<TKey, TValue>` or `HashSet<T>` to optimize performance.

ReadOnlyCollection

Definition:
A `ReadOnlyCollection<T>` in C# is a wrapper around an existing collection that provides read-only access to the collection. It is part of the `System.Collections.ObjectModel` namespace and is designed to prevent modifications to the underlying collection while still allowing read operations such as iteration and element retrieval.

When to Use:
You should use `ReadOnlyCollection<T>` when you want to:

- Expose a collection to users of your class without allowing them to modify the collection.
- Provide a thread-safe way to access a collection without allowing concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(1)	Accessing an element by index in the read-only collection is constant time.
Iteration	O(n)	O(n)	Iterating through the collection is linear in time relative to the number of elements.
Contains	O(n)	O(n)	Checking if the collection contains a certain element is linear in time.

Simple Complete Code Example:
Here's an example demonstrating various features of `ReadOnlyCollection<T>`:

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

class Program
{
    static void Main()
    {
        List<string> dinosaurs = new List<string>{
            "Tyrannosaurus",
            "Amargasaurus",
            "Deinonychus",
            "Compsognathus"
        };

        ReadOnlyCollection<string> readOnlyDinosaurs = new ReadOnlyCollection<string>(dinosaurs

        // Demonstrating read operations
        Console.WriteLine("Dinosaurs in the ReadOnlyCollection:");
        foreach (string dinosaur in readOnlyDinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine($"\nContains 'Deinonychus': {readOnlyDinosaurs.Contains("Deinonychus")}

        // Attempting to modify the ReadOnlyCollection will throw an exception
        // readOnlyDinosaurs.Add("Oviraptor"); // This line would cause a NotSupportedException
```

```
        // The underlying list can still be modified
        dinosaurs.Add("Oviraptor");
        Console.WriteLine("\nDinosaurs after adding to the underlying list:");
        foreach (string dinosaur in readOnlyDinosaurs)
        {
            Console.WriteLine(dinosaur);
        }
    }
}
```

Output:

```
Dinosaurs in the ReadOnlyCollection:
Tyrannosaurus
Amargasaurus
Deinonychus
Compsognathus

Contains 'Deinonychus': True

Dinosaurs after adding to the underlying list:
Tyrannosaurus
Amargasaurus
Deinonychus
Compsognathus
Oviraptor
```

How to Increase Performance When Using ReadOnlyCollection:

1. **Minimize Wrapper Creation**: Since `ReadOnlyCollection<T>` is just a wrapper, creating multiple wrappers can introduce unnecessary overhead. Create the wrapper only when necessary.
2. **Thread Safety**: `ReadOnlyCollection<T>` can support multiple readers concurrently as long as the collection is not modified. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration.
3. **Avoid Modification**: Any attempt to modify the `ReadOnlyCollection<T>` directly will result in a `NotSupportedException`. If you need to modify the collection, work with the underlying collection and then recreate the `ReadOnlyCollection<T>` if necessary.

ObservableCollection

Definition:
An `ObservableCollection<T>` is a dynamic data collection that provides notifications when items get added, removed, or when the entire list is refreshed. It implements the `INotifyCollectionChanged` interface, which

allows it to notify UI elements like `ListBox`, `DataGrid`, and other collections that they need to update themselves when the collection changes.

When to Use:

`ObservableCollection<T>` is best used when you need to bind a collection to a UI element that requires updates to be reflected automatically. It's commonly used in WPF and UWP applications where data binding is a key feature.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Add	O(1)	O(n)	Adding an element to the collection. In the worst case, this may require notifying the UI and updating the display.
Remove	O(1)	O(n)	Removing an element from the collection.
Iterate	O(n)	O(n)	Iterating through the collection to perform operations.

Simple Complete Code Example:
Here's an example demonstrating various features of `ObservableCollection<T>`:

```
using System;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        ObservableCollection<string> fruits = new ObservableCollection<string>();

        // Subscribe to CollectionChanged event
        fruits.CollectionChanged += (s, e) =>
        {
            Console.WriteLine("Collection Changed!");
            switch (e.Action)
            {
                case NotifyCollectionChangedAction.Add:
                    Console.WriteLine("Added: " + string.Join(", ", e.NewItems.Cast<string>()));
                    break;
                case NotifyCollectionChangedAction.Remove:
                    Console.WriteLine("Removed: " + string.Join(", ", e.OldItems.Cast<string>()));
                    break;
```

```
        }
    };

    // Add elements
    fruits.Add("Apple");
    fruits.Add("Banana");
    fruits.Add("Cherry");

    // Remove an element
    fruits.Remove("Banana");
  }
}
```

Output:

```
Collection Changed!
Added: Apple
Collection Changed!
Added: Banana
Collection Changed!
Added: Cherry
Collection Changed!
Removed: Banana
```

How to Increase Performance When Using ObservableCollection:

1. **Batch Updates**: If you need to perform multiple updates, consider using `ObservableCollection<T>`'s `BeginUpdate` and `EndUpdate` methods to minimize the number of notifications sent to the UI.
2. **Use Asynchronous Operations**: For operations that may take some time, consider performing them asynchronously to avoid blocking the UI thread.
3. **Optimize CollectionChanged Events**: Be mindful of the impact of the `CollectionChanged` event on performance. If you're only updating the UI based on specific changes, consider filtering the events or using a more efficient notification mechanism.

KeyedCollection

Definition:
A

`KeyedCollection<TKey, TItem>` is a collection that allows you to access elements by a key. It is a subclass of `Collection<TItem>` and provides an indexer that enables access to elements by key. Unlike a dictionary, a `KeyedCollection` allows duplicate keys, but it is up to the developer to manage these duplicates.

When to Use:

`KeyedCollection` is best used when you need a collection that can be accessed both by index and by a unique key, and when you want to avoid the overhead of a full `Dictionary<TKey, TValue>`. It is particularly useful when the key can be derived from the item itself, and you want to ensure that the key is automatically updated when the item changes.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access by Key	O(1)	O(n)	Accessing an element by key is generally constant time, but can take linear time in the worst case if the key is not found.
Access by Index	O(1)	O(1)	Accessing an element by index is constant time.
Insertion	O(1)	O(n)	Adding an element is generally constant time, but can take linear time if the key needs to be updated.
Deletion	O(1)	O(n)	Removing an element is constant time if the key is known, but can take linear time if the element needs to be found by iterating through the collection.

Simple Complete Code Example:
Here's an example demonstrating various features of `KeyedCollection<TKey, TItem>`:

```
using System.Collections.ObjectModel;

class Program
{
   static void Main() {


      Employees emps = new Employees()
      {
         new Employee{Id=1,Name="ahmed"},
         new Employee{Id=2,Name="omar"},
         new Employee{Id=3,Name="said"}
```

```
        };

        Console.WriteLine(emps[1].Name);//ahmed

        Console.ReadKey();
    }

    public class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }

    public class Employees : KeyedCollection<int, Employee>
    {
        protected override int GetKeyForItem(Employee item)
        {
            return item.Id;
        }
    }
}
```

Output:

```
Order items:
1 - Widget, Quantity: 100, Price: 2.5
2 - Sprocket, Quantity: 50, Price: 1.75
3 - Gear, Quantity: 25, Price: 5

Item with key 2: 2 - Sprocket, Quantity: 50, Price: 1.75

After removing item with key 2:
1 - Widget, Quantity: 100, Price: 2.5
3 - Gear, Quantity: 25, Price: 5
```

How to Increase Performance When Using KeyedCollection:

1. **Override GetKeyForItem Efficiently**: Ensure that the `GetKeyForItem` method is implemented efficiently to avoid unnecessary overhead when deriving keys from items.
2. **Use Indexer for Random Access**: Use the indexer to access elements by index when you don't need to use the key, as this operation is generally faster.
3. **Manage Duplicate Keys**: Be cautious with duplicate keys, as they can lead to unexpected behavior. If duplicates are possible, ensure that your application logic can handle them correctly.

NameValueCollection

1. Definition

A **NameValueCollection** is a collection in the `System.Collections.Specialized` namespace that stores key-value pairs where each key can have multiple associated values. It is commonly used for handling HTTP headers, query strings, or form data, where a single key might correspond to multiple values.

---

2. When to Use

- **Best to use when**:
  - You need to store multiple values for a single key.
  - You are working with HTTP headers, query strings, or form data.
  - You need a simple, non-generic collection for key-value pairs.
- **Avoid using when**:
  - You need type safety (use `Dictionary<TKey, TValue>` or `Lookup<TKey, TElement>` instead).
  - You need to store single values per key (use `Dictionary<TKey, TValue>` instead).
  - You are working with performance-critical code (use generic collections instead).

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `NameValueCollection` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Add	`O(1)`	`O(n)`	Adding a key-value pair is usually `O(1)`, but can degrade to `O(n)` due to resizing.
Remove	`O(n)`	`O(n)`	Removing a key-value pair requires searching and shifting elements.
Access by key	`O(n)`	`O(n)`	Accessing values by key requires searching through the collection.
ContainsKey	`O(n)`	`O(n)`	Checking if a key exists requires searching through the collection.
Iteration	`O(n)`	`O(n)`	Iterating through all key-value pairs is linear time.
Space Complexity	`O(n)`	`O(n)`	`NameValueCollection` uses an internal data structure, so space complexity is linear.

---

4. Code Example

Here's a simple, complete code example demonstrating the features of `NameValueCollection` in C#:

```
using System;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        // Create a NameValueCollection
        NameValueCollection collection = new NameValueCollection();

        // Add key-value pairs (a key can have multiple values)
        collection.Add("fruit", "apple");
        collection.Add("fruit", "banana");
        collection.Add("color", "red");
        collection.Add("color", "blue");

        // Access values by key
        Console.WriteLine("Values for key 'fruit':");
        foreach (string value in collection.GetValues("fruit"))
        {
            Console.WriteLine(value); // Output: apple, banana
        }

        // Access a single value by key (returns the first value)
        Console.WriteLine("First value for key 'color': " + collection["color"]); // Output: red

        // Check if a key exists
        bool containsKey = collection.HasKeys();
        Console.WriteLine("Contains any keys? " + containsKey); // Output: True

        // Iterate through all keys and values
        Console.WriteLine("All key-value pairs:");
        foreach (string key in collection.AllKeys)
        {
            Console.WriteLine($"Key: {key}");
            foreach (string value in collection.GetValues(key))
            {
                Console.WriteLine($"  Value: {value}");
            }
        }

        // Remove a key-value pair
        collection.Remove("fruit");
```

```
        Console.WriteLine("After removing key 'fruit':");
        foreach (string key in collection.AllKeys)
        {
            Console.WriteLine($"Key: {key}");
            foreach (string value in collection.GetValues(key))
            {
                Console.WriteLine($"  Value: {value}");
            }
        }

        // Clear the collection
        collection.Clear();
        Console.WriteLine("After clearing the collection, count: " + collection.Count); // Output: 0
    }
}
```

5. How to Increase Performance When Using NameValueCollection

1. **Use for Multi-Valued Keys**:
   - `NameValueCollection` is ideal for scenarios where a single key can have multiple values, such as handling HTTP headers or query strings.
2. **Avoid Frequent Searches**:
   - Searching for keys or values is `O(n)`. If you need frequent lookups, consider using `Dictionary<TKey, TValue>` or `Lookup<TKey, TElement>`.
3. **Use `GetValues` for Multi-Valued Access**:
   - Use the `GetValues` method to retrieve all values associated with a key.
4. **Avoid Using `Remove` Frequently**:
   - Removing a key-value pair is `O(n)`. If you need frequent modifications, consider using a more efficient collection.
5. **Use `AllKeys` for Iteration**:
   - Use the `AllKeys` property to iterate through all keys in the collection.
6. **Prefer Generic Collections for Type Safety**:
   - If you need type safety, use `Dictionary<TKey, TValue>` or `Lookup<TKey, TElement>` instead.
7. **Combine with Other Data Structures**:
   - For complex scenarios, combine `NameValueCollection` with other collections like `Dictionary<TKey, TValue>` or `List<T>` to optimize performance.

Concurrently collections :

Concurrent collections are thread-safe collections designed to handle multiple threads accessing and modifying them simultaneously without causing data corruption or requiring external synchronization mechanisms.

1. ConcurrentStack

Definition:

`ConcurrentStack<T>` is a thread-safe stack collection from the `System.Collections.Concurrent` namespace in C#. It allows multiple threads to perform push and pop operations concurrently without the need for explicit locking, making it a great choice for concurrent programming scenarios.

When to Use:

`ConcurrentStack<T>` is best used when:

- You have a multi-threaded application where threads need to add (`Push`) or remove (`Pop`) items from a stack concurrently.
- You want to avoid the overhead of locking mechanisms that are typically required to synchronize access to a standard `Stack<T>`.
- You need a thread-safe collection that follows the Last-In-First-Out (LIFO) principle.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Push	O(1)	O(n)	Adding an element to the top of the stack is generally very fast but can take longer if the collection needs to be resized.
Pop	O(1)	O(n)	Removing an element from the top of the stack is generally very fast but can take longer if the stack needs to be traversed.
Peek	O(1)	O(1)	Retrieving the top element without removing it is constant time.

Simple Complete Code Example:
Here's an example demonstrating various features of `ConcurrentStack<T>`:

```
using System;
using System.Collections.Concurrent;
using System.Threading;
```

```
class Program
{
    static void Main()
    {
        ConcurrentStack<int> stack = new ConcurrentStack<int>();

        // Producer thread
        Thread producer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                stack.Push(i);
                Console.WriteLine($"Produced: {i}");
                Thread.Sleep(100); // Simulate work
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                if (stack.TryPop(out int item))
                {
                    Console.WriteLine($"Consumed: {item}");
                }
                Thread.Sleep(50); // Simulate work
            }
        });

        producer.Start();
        consumer.Start();

        producer.Join();
        consumer.Join();
    }
}
```

Output:

```
Produced: 0
Produced: 1
...
```

```
Produced: 9
Consumed: 0
Consumed: 1
...
Consumed: 9
```

How to Increase Performance When Using ConcurrentStack:

1. **Minimize Contention**: Ensure that access to the `ConcurrentStack<T>` is not highly contended among threads to reduce the chance of frequent context switching and lock contention.
2. **Batch Operations**: When possible, perform batch operations to reduce the number of individual `Push` and `Pop` operations.
3. **Order of Operations**: Be aware that the order of elements in a `ConcurrentStack<T>` is not guaranteed to be the same across different executions due to concurrent modifications.
4. **Use TryPop for Non-Blocking Operations**: Use `TryPop` for non-blocking operations. It attempts to pop an item from the stack without blocking and returns a boolean indicating success or failure.

2. ConcurrentQueue

Definition:

`ConcurrentQueue<T>` is a thread-safe queue collection from the `System.Collections.Concurrent` namespace in C#. It is designed to allow multiple threads to add (`Enqueue`) and remove (`Dequeue`) items concurrently without the need for explicit synchronization.

When to Use:

`ConcurrentQueue<T>` is best used when:

- You have a multi-threaded application where threads need to add or remove items from a queue concurrently.
- You want to avoid the overhead of locking mechanisms that are typically required to synchronize access to a standard `Queue<T>`.
- You need a thread-safe collection that follows the First-In-First-Out (FIFO) principle.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Enqueue	O(1)	O(n)	Adding an element to the end of the queue is generally very fast but can take longer if the collection needs to be resized.

Dequeue	O(1)	O(n)	Removing an element from the front of the queue is generally very fast but can take longer if the queue needs to be traversed.
Peek	O(1)	O(1)	Retrieving the front element without removing it is constant time.

Simple Complete Code Example:
Here's an example demonstrating various features of `ConcurrentQueue<T>`:

```
using System;
using System.Collections.Concurrent;
using System.Threading;

class Program
{
    static ConcurrentQueue<int> queue = new ConcurrentQueue<int>();

    static void Main()
    {
        // Producer thread
        Thread producer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                queue.Enqueue(i);
                Console.WriteLine($"Produced: {i}");
                Thread.Sleep(100); // Simulate work
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                if (queue.TryDequeue(out int item))
                {
                    Console.WriteLine($"Consumed: {item}");
                }
                Thread.Sleep(50); // Simulate work
            }
        });
```

```
        producer.Start();
        consumer.Start();

        producer.Join();
        consumer.Join();
    }
}
```

Output:

```
Produced: 0
Produced: 1
...
Produced: 9
Consumed: 0
Consumed: 1
...
Consumed: 9
```

How to Increase Performance When Using ConcurrentQueue:

1. **Minimize Contention**: Ensure that access to the `ConcurrentQueue<T>` is not highly contended among threads to reduce the chance of frequent context switching and lock contention.
2. **Batch Operations**: When possible, perform batch operations to reduce the number of individual `Enqueue` and `Dequeue` operations.
3. **Order of Operations**: Be aware that the order of elements in a `ConcurrentQueue<T>` is not guaranteed to be the same across different executions due to concurrent modifications.
4. **Use TryDequeue for Non-Blocking Operations**: Use `TryDequeue` for non-blocking operations. It attempts to dequeue an item from the queue without blocking and returns a boolean indicating success or failure.

3. ConcurrentDictionary

Definition:

`ConcurrentDictionary<TKey, TValue>` is a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently. It uses fine-grained locking to ensure thread safety for modifications and write operations, while read operations are performed in a lock-free manner.

When to Use:
This collection is best used in scenarios where:

- You need a dictionary that can be accessed by multiple threads without the need for explicit synchronization.
- You want to avoid the overhead of locking mechanisms typically required for concurrent access to a standard `Dictionary<TKey, TValue>`.
- You require atomic operations for add, update, and remove actions.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Add	O(1)	O(n)	Adding an element is generally constant time but can take longer if the dictionary needs resizing
Remove	O(1)	O(n)	Removing an element is constant time unless the dictionary needs to be searched
Update	O(1)	O(n)	Updating an element is constant time unless the element needs to be found
Access	O(1)	O(1)	Accessing an element by key is constant time

Simple Complete Code Example:
Here's an example demonstrating various features of `ConcurrentDictionary<TKey, TValue>`:

```
using System;
using System.Collections.Concurrent;

class Program
{
    static void Main()
    {
        ConcurrentDictionary<int, string> dictionary = new ConcurrentDictionary<int, string>();

        // Adding elements
        dictionary.TryAdd(1, "one");
        dictionary.TryAdd(2, "two");
        dictionary.TryAdd(3, "three");

        // Updating an element
        dictionary.AddOrUpdate(2, "TWO", (key, value) => "TWO");

        // Retrieving an element
        string value;
        if (dictionary.TryGetValue(2, out value))
```

```
        {
            Console.WriteLine($"Value for key 2: {value}");
        }

        // Iterating through the dictionary
        foreach (var kvp in dictionary)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }
    }
}
```

Output

> `Value for key 2: TWO`
> `Key: 1, Value: one`
> `Key: 2, Value: TWO`
> `Key: 3, Value: three`

How to Increase Performance When Using ConcurrentDictionary:

1. **Minimize Contention**: Ensure that access to the `ConcurrentDictionary<TKey, TValue>` is not highly contended among threads to reduce the chance of frequent context switching and lock contention.
2. **Batch Operations**: Use batch operations like `AddOrUpdate` to reduce the number of individual add and update operations.
3. **Use TryGetValue for Non-Blocking Operations**: Use `TryGetValue` for non-blocking retrieval of elements. It attempts to get the value associated with the specified key without blocking and returns a boolean indicating success or failure.
4. **Avoid Excessive Locking**: Since `ConcurrentDictionary<TKey, TValue>` handles locking internally, avoid additional locking mechanisms that can lead to deadlocks or performance bottlenecks.

4. BlockingCollection

Definition:

`BlockingCollection<T>` is a thread-safe collection class from the `System.Collections.Concurrent` namespace in C#. It provides a way to add and take items from the collection in a thread-safe manner, with the option to block operations when the collection is empty or full. This class is designed to be used in

producer-consumer scenarios where multiple threads are adding items (producers) and multiple threads are removing items (consumers).

When to Use:
You should use `BlockingCollection<T>` when:

- You have multiple threads that need to add items to a collection, and you want them to block when the collection is full.
- You have multiple threads that need to remove items from a collection, and you want them to block when the collection is empty.
- You want to avoid the complexity of implementing your own synchronization logic for thread-safe collections.
- You need a way to signal to consumer threads that no more items will be added to the collection.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Add	O(1)	O(n)	Adding an item to the collection is generally constant time unless the collection needs to grow.
Take	O(1)	O(n)	Removing an item from the collection is generally constant time unless the collection is empty and threads are blocked.
CompleteAdding	N/A	O(n)	This operation signals that no more items will be added to the collection, potentially unblocking blocked consumers.

Simple Complete Code Example:
Here's an example demonstrating various features of `BlockingCollection<T>`:

```
using System;
using System.Collections.Concurrent;
using System.Threading;

class Program
{
    static void Main()
    {
        BlockingCollection<int> collection = new BlockingCollection<int>(new ConcurrentQueue<int>

        // Producer thread
        Thread producer = new Thread(() =>
```

```
        {
            for (int i = 0; i < 10; i++)
            {
                collection.Add(i);
                Console.WriteLine($"Produced: {i}");
            }
            collection.CompleteAdding();
        });

        // Consumer thread
        Thread consumer = new Thread(() =>
        {
            foreach (var item in collection.GetConsumingEnumerable())
            {
                Console.WriteLine($"Consumed: {item}");
            }
        });

        producer.Start();
        consumer.Start();

        producer.Join();
        consumer.Join();
    }
}
```

Output

```
Produced: 0
Produced: 1
...
Produced: 9
Consumed: 0
Consumed: 1
...
Consumed: 9
```

How to Increase Performance When Using BlockingCollection:

1. **Batching**: Use batching to reduce the number of `Add` and `Take` operations, which can help reduce overhead.
2. **CompleteAdding**: Call `CompleteAdding` as soon as all items have been added to prevent consumers from waiting indefinitely for more items.

3. **CancellationToken**: Use `CancellationToken` with `Add` and `Take` to respond to cancellation requests and exit loops when necessary.
4. **Capacity Management**: If you know the maximum number of items that will be in the collection, set the `Bounds` property to control the capacity and prevent excessive resizing.

5. ConcurrentBag

Definition:

`ConcurrentBag<T>` is a concurrent collection from the `System.Collections.Concurrent` namespace in C#. It is designed to allow multiple threads to add (`Add`) and take (`Take`) elements concurrently without the need for explicit synchronization. Unlike other concurrent collections such as `ConcurrentQueue<T>` or `ConcurrentStack<T>`, `ConcurrentBag<T>` does not maintain any order among elements.

When to Use:

`ConcurrentBag<T>` is best used when:

- You need a collection that can be accessed by multiple threads concurrently.
- You don't care about the order of the elements in the collection.
- You want to avoid the overhead of locking mechanisms that are typically required to synchronize access to a standard collection.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Add	O(1)	O(n)	Adding an element to the bag is generally constant time but can take longer if the internal array needs to be resized.
Take	O(1)	O(n)	Removing an element from the bag is constant time unless the bag is empty, in which case it may take longer to find an element to remove.

Simple Complete Code Example:
Here's an example demonstrating various features of `ConcurrentBag<T>`:

```
using System;
using System.Collections.Concurrent;
using System.Threading;

class Program
{
    static ConcurrentBag<int> bag = new ConcurrentBag<int>();
```

```
    static void Main()
    {
        // Producer thread
        Thread producer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                bag.Add(i);
                Console.WriteLine($"Produced: {i}");
                Thread.Sleep(100); // Simulate work
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() =>
        {
            for (int i = 0; i < 10; i++)
            {
                if (bag.TryTake(out int item))
                {
                    Console.WriteLine($"Consumed: {item}");
                }
                Thread.Sleep(50); // Simulate work
            }
        });

        producer.Start();
        consumer.Start();

        producer.Join();
        consumer.Join();
    }
}
```

Output:

```
 Produced: 0
Produced: 1
...
Produced: 9
Consumed: 0
Consumed: 1
```

> ...
> Consumed: 9

The actual order of produced and consumed elements may vary due to the concurrent nature of the `ConcurrentBag<T>`.

How to Increase Performance When Using ConcurrentBag:

1. **Minimize Contention**: Ensure that access to the `ConcurrentBag<T>` is not highly contended among threads to reduce the chance of frequent context switching and lock contention.
2. **Batch Operations**: When possible, perform batch operations to reduce the number of individual `Add` and `Take` operations.
3. **Use TryTake for Non-Blocking Operations**: Use `TryTake` for non-blocking operations. It attempts to take an item from the bag without blocking and returns a boolean indicating success or failure.
4. **Avoid Unnecessary Synchronization**: Since `ConcurrentBag<T>` is already thread-safe, avoid adding additional synchronization mechanisms that can lead to deadlocks or performance bottlenecks.

Feature ⚡	`ConcurrentBag<T>` 👜	`BlockingCollection<T>` ⛓️
Thread-Safe (`Thread-Safe`)	✅ Yes, lock-free	✅ Yes, with blocking
Ordering (`Ordering`)	❌ No order	✅ Depends on collection type (FIFO, LIFO, etc.)
Blocking (`Blocking`)	❌ No blocking	✅ Waits for data if none is available
Capacity Limiting (`Bounded Capacity`)	❌ Not possible	✅ Can set a limit
Performance on Add (`Add`)	🔥 Very fast	🐢 Slower due to waiting mechanisms
Performance on Remove (`Remove`)	✅ Fast but unordered	✅ Fast with proper sequencing

📌 When to Use Each?

✅ Use `ConcurrentBag<T>` if:

✔️ You need **fast concurrent adding and removing**.

✔️ **Order of retrieval doesn't matter**.

✔️ You need **high performance for adding items**.

✅ Use `BlockingCollection<T>` if:

✔️ You need a **Producer-Consumer pattern**.

✔️ You want **blocking behavior when no data is available**.

✔️ You need **to control memory usage with a bounded collection**.

Immutable collections

Immutable collections are collections that cannot be modified after they are created. Any operation that appears to modify the collection actually returns a new collection with the changes, leaving the original collection unchanged.

1. ImmutableArray<T>

Definition:

`ImmutableArray<T>` is a type from the `System.Collections.Immutable` namespace that represents an immutable array. It provides a thread-safe, efficient way to handle arrays that should not be modified after their creation. `ImmutableArray<T>` is commonly used in scenarios where data integrity and thread safety are crucial, and it is designed to work well with other immutable collections.

When to Use:

`ImmutableArray<T>` is best used when:

- You need to ensure that the contents of an array cannot be changed after its creation.
- You want to use an array in a multi-threaded environment without worrying about synchronization.
- You are working with data that should remain consistent and unaltered once it's set.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(1)	Accessing an element by index is constant time.
Creation	O(n)	O(n)	Creating an immutable array from an existing collection requires copying all elements.
Modification	N/A	N/A	Modifying an `ImmutableArray<T>` is not possible; you must create a new one.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableArray<T>`:

```
using System;
using System.Collections.Immutable;

class Program
{
    static void Main()
```

```
    {
        ImmutableArray<int> immutableArray = ImmutableArray.Create(1, 2, 3, 4, 5);

        // Access elements
        Console.WriteLine(immutableArray[2]); // Outputs: 3

        // Create a new ImmutableArray with an additional element
        ImmutableArray<int> newImmutableArray = immutableArray.Add(6);

        // Iterate through the array
        foreach (var item in newImmutableArray)
        {
            Console.WriteLine(item);
        }
    }
}
```

Output

```
3
1
2
3
4
5
6
```

How to Increase Performance When Using ImmutableArray:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable array. This approach minimizes the overhead of creating new immutable arrays for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableArray<T>` involves copying the contents of the existing array, try to minimize the number of times you create new instances. Instead, use methods like `Add`, `Remove`, or `Replace` to create a new immutable array based on an existing one.
3. **Use ImmutableArray in Multi-threaded Scenarios**: Take advantage of `ImmutableArray<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableDictionary

Definition:

`ImmutableDictionary<TKey, TValue>` is a type from the `System.Collections.Immutable` namespace that represents an immutable collection of key/value pairs. It provides thread-safe operations for creating, manipulating, and querying dictionaries without altering the original instance. It ensures that once an `ImmutableDictionary` is created, it cannot be changed, which helps in maintaining data integrity and thread safety.

When to Use:

`ImmutableDictionary<TKey, TValue>` is best used when:

- You need to ensure that the contents of a dictionary cannot be changed after its creation.
- You want to use a dictionary in a multi-threaded environment without worrying about synchronization.
- You need a dictionary that can be safely shared across threads without the risk of concurrent modifications.
- You require atomic operations for adding, updating, and removing key/value pairs.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(log n)	Accessing an element by key is generally constant time.
Creation	O(n)	O(n log n)	Creating an immutable dictionary from an existing collection requires copying all elements.
Modification	N/A	N/A	Modifying an `ImmutableDictionary<TKey, TValue>` is not possible; you must create a new one.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableDictionary<TKey, TValue>`:

```
using System;
using System.Collections.Immutable;

class Program
{
    static void Main()
    {
        var builder = ImmutableDictionary.CreateBuilder<string, int>();
        builder.Add("one", 1);
        builder.Add("two", 2);
```

```
        builder.Add("three", 3);

        var immutableDictionary = builder.ToImmutable();

        // Access elements
        Console.WriteLine(immutableDictionary["two"]); // Outputs: 2

        // Create a new ImmutableDictionary with an additional key/value pair
        var newImmutableDictionary = immutableDictionary.Add("four", 4);

        // Iterate through the dictionary
        foreach (var kvp in newImmutableDictionary)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }
    }
}
```

Output:

```
2
Key: one, Value: 1
Key: two, Value: 2
Key: three, Value: 3
Key: four, Value: 4
```

How to Increase Performance When Using ImmutableDictionary:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable dictionary. This approach minimizes the overhead of creating new immutable dictionaries for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableDictionary<TKey, TValue>` involves copying the contents of the existing dictionary, try to minimize the number of times you create new instances. Instead, use methods like `Add`, `Remove`, or `SetItem` to create a new immutable dictionary based on an existing one.
3. **Use Immutable Interchangeably**: Take advantage of `ImmutableDictionary<TKey, TValue>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableHashSet

Definition:

`ImmutableHashSet<T>` is a type from the `System.Collections.Immutable` namespace that represents an immutable collection of unique elements. It provides thread-safe operations for creating, manipulating, and querying sets without altering the original collection. It ensures that once an `ImmutableHashSet<T>` is created, it cannot be changed, which helps in maintaining data integrity and thread safety.

When to Use:

`ImmutableHashSet<T>` is best used when:

- You need a collection that can be accessed by multiple threads concurrently without the risk of concurrent modifications.
- You want to ensure that the contents of a set cannot be changed after its creation.
- You require a collection that maintains uniqueness of elements without any particular order.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Contains	O(1)	O(n)	Checking if the set contains a specific element is generally constant time due to the hash-based structure, but can take longer if there are hash collisions.
Add	O(n)	O(n)	Adding an element to the set is linear time relative to the number of elements because it may require creating a new set.
Remove	O(n)	O(n)	Removing an element is linear time relative to the number of elements because it may require creating a new set.
Creation	O(n)	O(n log n)	Creating an immutable set from an existing collection requires copying all elements and potentially sorting them.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableHashSet<T>`:

```
using System;
using System.Collections.Immutable;
```

```
class Program
{
    static void Main()
    {
        var builder = ImmutableHashSet.CreateBuilder<int>();
        builder.Add(1);
        builder.Add(2);
        builder.Add(3);

        var immutableHashSet = builder.ToImmutable();

        // Check if the set contains an element
        Console.WriteLine(immutableHashSet.Contains(2)); // Outputs: True

        // Create a new set with an additional element
        var newImmutableHashSet = immutableHashSet.Add(4);

        // Iterate through the set
        foreach (var item in newImmutableHashSet)
        {
            Console.WriteLine(item);
        }
    }
}
```

Output:

```
True
1
2
3
4
```

How to Increase Performance When Using ImmutableHashSet:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable set. This approach minimizes the overhead of creating new immutable sets for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableHashSet<T>` involves copying the contents of the existing set, try to minimize the number of times you create new instances.

Instead, use methods like `Add`, `Remove`, or `Union` to create a new immutable set based on an existing one.

3. **Use Immutable Interchangeably**: Take advantage of `ImmutableHashSet<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableList

Definition:

`ImmutableList<T>` is a type from the `System.Collections.Immutable` namespace that represents an immutable list of elements. It provides a thread-safe, efficient way to handle lists that should not be modified after their creation. This class is designed to ensure that the list contents cannot be altered, which helps maintain data integrity and thread safety without the need for additional synchronization.

When to Use:

`ImmutableList<T>` is best used when:

- You need a list that can be accessed by multiple threads concurrently without the risk of concurrent modifications.
- You want to ensure that the contents of a list cannot be changed after its creation.
- You require a list that can be safely shared across threads without the risk of concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(1)	Accessing an element by index is constant time.
Creation	O(n)	O(n log n)	Creating an immutable list from an existing collection requires copying all elements and potentially sorting them.
Modification	N/A	N/A	Modifying an `ImmutableList<T>` is not possible; you must create a new one.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableList<T>`:

```
using System;
using System.Collections.Immutable;
```

```
class Program
{
    static void Main()
    {
        ImmutableList<int> immutableList = ImmutableList.Create(1, 2, 3, 4, 5);

        // Access elements
        Console.WriteLine(immutableList[2]); // Outputs: 3

        // Create a new ImmutableList with an additional element
        ImmutableList<int> newImmutableList = immutableList.Add(6);

        // Iterate through the list
        foreach (var item in newImmutableList)
        {
            Console.WriteLine(item);
        }
    }
}
```

Output:

```
3
1
2
3
4
5
6
```

How to Increase Performance When Using ImmutableList:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable list. This approach minimizes the overhead of creating new immutable lists for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableList<T>` involves copying the contents of the existing list, try to minimize the number of times you create new instances. Instead, use methods like `Add`, `Remove`, or `Replace` to create a new immutable list based on an existing one.
3. **Use ImmutableList in Multi-threaded Scenarios**: Take advantage of `ImmutableList<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableQueue

Definition:

`ImmutableQueue<T>` is a type from the `System.Collections.Immutable` namespace that represents an immutable collection of elements that follow the First-In-First-Out (FIFO) principle. It provides thread-safe operations for creating, manipulating, and querying queues without altering the original instance. It ensures that once an `ImmutableQueue<T>` is created, it cannot be changed, which helps maintain data integrity and thread safety.

When to Use:

`ImmutableQueue<T>` is best used when:

- You need a queue that can be accessed by multiple threads concurrently without the risk of concurrent modifications.
- You want to ensure that the contents of a queue cannot be changed after its creation.
- You require a queue that can be safely shared across threads without the risk of concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Enqueue	O(log n)	O(log n)	Adding an element to the end of the queue is generally logarithmic time due to potential resizing.
Dequeue	O(log n)	O(log n)	Removing an element from the front of the queue is generally logarithmic time due to potential resizing.
Peek	O(1)	O(1)	Retrieving the front element without removing it is constant time.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableQueue<T>`:

```
using System;
using System.Collections.Immutable;

class Program
{
    static void Main()
    {
```

```
        ImmutableQueue<int> immutableQueue = ImmutableQueue.Create(1, 2, 3, 4, 5);

        // Enqueue an element
        var newQueue = immutableQueue.Enqueue(6);

        // Dequeue an element
        ImmutableQueue<int> dequeuedQueue;
        if (newQueue.TryDequeue(out int value))
        {
            dequeuedQueue = newQueue;
            Console.WriteLine($"Dequeued: {value}");
        }

        // Iterate through the queue
        foreach (var item in dequeuedQueue)
        {
            Console.WriteLine(item);
        }
    }
}
```

Output:

```
Dequeued: 1
2
3
4
5
6
```

How to Increase Performance When Using ImmutableQueue:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable queue. This approach minimizes the overhead of creating new immutable queues for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableQueue<T>` involves copying the contents of the existing queue, try to minimize the number of times you create new instances. Instead, use methods like `Enqueue` and `Dequeue` to create a new immutable queue based on an existing one.

3. **Use Immutable Interchangeably**: Take advantage of `ImmutableQueue<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableSortedDictionary

Definition:

`ImmutableSortedDictionary<TKey, TValue>` is a type from the `System.Collections.Immutable` namespace that represents a thread-safe, immutable collection of key/value pairs that are automatically sorted by keys. This collection cannot be modified after it is created, ensuring data integrity and thread safety without the need for additional synchronization mechanisms.

When to Use:

`ImmutableSortedDictionary<TKey, TValue>` is best used when:

- You require a dictionary that maintains a sorted order of keys.
- You need a collection that is safe to access from multiple threads without concurrent modification issues.
- You want to avoid the overhead of locking mechanisms typically required for concurrent access to a standard `Dictionary<TKey, TValue>`.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(log n)	O(log n)	Accessing an element by key is logarithmic time due to the sorted nature of the dictionary.
Creation	O(n log n)	O(n log n)	Creating an immutable sorted dictionary from an existing collection requires sorting and copying all elements.
Modification	N/A	N/A	Modifying an `ImmutableSortedDictionary<TKey, TValue>` is not possible; you must create a new one.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableSortedDictionary<TKey, TValue>`:

```
using System;
using System.Collections.Immutable;

class Program
{
```

```
    static void Main()
    {
        var builder = ImmutableSortedDictionary.CreateBuilder<string, int>();
        builder.Add("one", 1);
        builder.Add("two", 2);
        builder.Add("three", 3);

        var immutableSortedDictionary = builder.ToImmutable();

        // Access elements
        Console.WriteLine(immutableSortedDictionary["two"]); // Outputs: 2

        // Create a new ImmutableSortedDictionary with an additional key/value pair
        var newImmutableSortedDictionary = immutableSortedDictionary.Add("four", 4);

        // Iterate through the dictionary
        foreach (var kvp in newImmutableSortedDictionary)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }
    }
}
```

Output:

```
2
Key: four, Value: 4
Key: one, Value: 1
Key: three, Value: 3
Key: two, Value: 2
```

How to Increase Performance When Using ImmutableSortedDictionary:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable dictionary. This approach minimizes the overhead of creating new immutable dictionaries for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableSortedDictionary<TKey, TValue>` involves copying the contents of the existing dictionary, try to minimize the number of times you create new instances. Instead, use methods like `Add`, `Remove`, or `Replace` to create a new immutable dictionary based on an existing one.

3. **Use Immutable Interchangeably**: Take advantage of `ImmutableSortedDictionary<TKey, TValue>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableSortedSet

Definition:

`ImmutableSortedSet<T>` is a type from the `System.Collections.Immutable` namespace in C# that represents an immutable collection of unique elements that are automatically sorted. This collection is designed to provide thread-safe operations for creating, manipulating, and querying sets without altering the original instance. It ensures that once an `ImmutableSortedSet<T>` is created, it cannot be changed, which helps maintain data integrity and thread safety without the need for additional synchronization.

When to Use:

`ImmutableSortedSet<T>` is best used when:

- You need a collection that maintains its elements in sorted order.
- You want to ensure that the contents of a set cannot be changed after its creation.
- You require a set that can be safely shared across threads without the risk of concurrent modifications.
- You need to perform set operations like union, intersection, and difference in an immutable fashion.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Contains	O(log n)	O(log n)	Checking if the set contains a specific element is logarithmic time due to the sorted nature of the set.
Add	O(log n)	O(log n)	Adding an element to the set is logarithmic time because it may require maintaining the sorted order.
Remove	O(log n)	O(log n)	Removing an element is logarithmic time because it may require maintaining the sorted order.
Creation	O(n log n)	O(n log n)	Creating an immutable sorted set from an existing collection requires sorting and copying all elements.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableSortedSet<T>`:

```
using System;
using System.Collections.Immutable;

class Program
{
    static void Main()
    {
        var builder = ImmutableSortedSet.CreateBuilder<int>();
        builder.Add(3);
        builder.Add(1);
        builder.Add(2);

        var immutableSortedSet = builder.ToImmutable();

        // Check if the set contains an element
        Console.WriteLine(immutableSortedSet.Contains(2)); // Outputs: True

        // Create a new ImmutableSortedSet with an additional element
        var newImmutableSortedSet = immutableSortedSet.Add(4);

        // Iterate through the set
        foreach (var item in newImmutableSortedSet)
        {
            Console.WriteLine(item);
        }
    }
}
```

Output:

```
True
1
2
3
4
```

How to Increase Performance When Using ImmutableSortedSet:

1. **Reuse Builder Instances**: When performing multiple operations, use the `ToBuilder` method to create a mutable builder that can be efficiently mutated across multiple operations, then converted back to an immutable set. This approach minimizes the overhead of creating new immutable sets for each operation.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableSortedSet<T>` involves copying the contents of the existing set, try to minimize the number of times you create new instances. Instead, use methods like `Add`, `Remove`, or `Union` to create a new immutable set based on an existing one.
3. **Use Immutable Interchangeably**: Take advantage of `ImmutableSortedSet<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

ImmutableStack

Definition:

`ImmutableStack<T>` is a type from the `System.Collections.Immutable` namespace that represents an immutable collection of elements that follow the Last-In-First-Out (LIFO) principle. It provides thread-safe operations for creating, manipulating, and querying stacks without altering the original instance, ensuring data integrity and thread safety without the need for additional synchronization.

When to Use:

`ImmutableStack<T>` is best used when:

- You need a stack that can be accessed by multiple threads concurrently without the risk of concurrent modifications.
- You want to ensure that the contents of a stack cannot be changed after its creation.
- You require a stack that can be safely shared across threads without the risk of concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Push	O(log n)	O(n)	Adding an element to the top of the stack is generally logarithmic time but can take longer if the internal array needs to be resized.
Pop	O(log n)	O(n)	Removing an element from the top of the stack is generally logarithmic time but can take longer if the stack needs to be traversed.
Peek	O(1)	O(1)	Retrieving the top element without removing it is constant time.

Simple Complete Code Example:
Here's an example demonstrating various features of `ImmutableStack<T>`:

```
using System;
using System.Collections.Immutable;

class Program
{
   static void Main()
   {
      var stack = ImmutableStack.Create(1, 2, 3, 4, 5);

      // Push an element
      var newStack = stack.Push(6);

      // Pop an element
      var result = newStack.TryPop(out var item) ? item : -1;
      Console.WriteLine($"Popped: {result}");

      // Iterate through the stack
      foreach (var item in newStack)
      {
         Console.WriteLine(item);
      }
   }
}
}
```

Output:

```
Popped: 6
1
2
3
4
5
```

How to Increase Performance When Using ImmutableStack:

1. **Minimize Contention**: Ensure that access to the `ImmutableStack<T>` is not highly contended among threads to reduce the chance of frequent context switching and lock contention.
2. **Avoid Unnecessary Copies**: Since creating a new `ImmutableStack<T>` involves copying the contents of the existing stack, try to minimize the number of times you create new instances. Instead, use

methods like `Push` and `Pop` to create a new immutable stack based on an existing one.

3. **Use Immutable Interchangeably**: Take advantage of `ImmutableStack<T>`'s thread safety when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms.

summary of DS

Collection Type 📦	Time Complexity ⏱️	Description 💬	When to Use 📝
Array 📊	Access: O(1)	Fixed-size, fast access.	When you need fast access with a fixed size. ⚡
ArrayList 📋	Access: O(1), Add: O(1)	Dynamic array, resizes automatically.	When you need dynamic resizing with fast access. 🚀
BitArray 🧠	Access: O(1)	Collection of bits (True/False).	For handling Boolean data efficiently. 🔢
Dictionary<TKey, TValue> 🗝️	Add/Remove: O(1)	Key-value pair collection.	Quick lookups using keys. 🔍
HashSet 🧯	Add/Remove/Contains: O(1)	Unique, unordered collection.	When you need uniqueness with fast access. ✨
DictionaryList<TKey, TValue> 📖	Varies	A combo of Dictionary and List.	When you need to map keys to multiple values. 🔄
HybridDictionary 🔀	Add/Remove: O(log n), Lookup: O(1)	Combines Hashtable and Dictionary.	For features of both Hashtable and Dictionary. 🔁
List 📚	Add: O(1), Access: O(1), Remove: O(n)	Flexible, dynamic array.	For resizing lists with efficient access. 💻
LinkedList 🔗	Add/Remove: O(1), Search: O(n)	Doubly-linked list.	Fast operations at both ends of the list. ↕️
NameValueCollection 🏷️	Add/Remove: O(1)	Key-value pairs with multiple values.	When each key can map to multiple values. 🔢
Queue ⏳	Enqueue/Dequeue: O(1)	FIFO collection (First-In-First-Out).	Task scheduling or sequential processing. 📅

Stack 📚	Push/Pop: O(1)	LIFO collection (Last-In-First-Out).	Undo functionality, or function calls. ⏪
SortedDictionary<TKey, TValue> 🔢	Access/Insert/Remove: O(log n)	Sorted key-value pairs.	When you need ordered data and fast lookups. 🌐
SortedList<TKey, TValue> 📈	Access: O(log n), Insert: O(n)	Key-value pairs sorted by the key.	Sorted data, fast key access. 🌳
SortedSet 🔠	Add/Remove/Contains: O(log n)	Sorted, unique elements.	Sorted data with no duplicates. ✨
KeyedCollection<TKey, TItem> 🔑	Access: O(1)	Key-based collection.	Fast item retrieval by key. 🔍
PriorityQueue<TElement, TPriority> ⬆️ TOP	Enqueue/Dequeue: O(log n)	Elements based on priority.	Task scheduling or processing based on priority. 🏆
ConcurrentDictionary<TKey, TValue> 🔒	Add/Remove: O(1)	Thread-safe key-value pairs.	When thread safety is essential in multi-threading. 👷
ConcurrentQueue ⏳	Enqueue/Dequeue: O(1)	Thread-safe FIFO queue.	Thread-safe queue in concurrent tasks. 🔄
ConcurrentStack 🏗️	Push/Pop: O(1)	Thread-safe LIFO stack.	Thread-safe stack in multi-threaded environments. 🔒
ConcurrentBag 👜	Add/Take: O(1)	Thread-safe, unordered collection.	Thread-safe collection with no order. 🛠️
BlockingCollection 🚧	Add/Take: O(1), Wait: O(1)	Blocking, thread-safe collection.	Producer-consumer model with multi-threading. ⏳
ImmutableArray 🛑	Access: O(1)	Read-only array.	Immutable, fixed-size, read-only collection. 🛑
ImmutableDictionary<TKey, TValue> 🔐	Access: O(log n)	Read-only, immutable dictionary.	Immutable, thread-safe dictionary. 🔐
ImmutableHashSet 🛑	Add/Remove: O(1)	Read-only, immutable set.	Immutable set, no duplicates. 🛑
ImmutableList 📚	Access: O(1), Add/Remove: O(n)	Read-only, immutable list.	Immutable list with fixed content. 📜
ImmutableQueue 🏁	Enqueue/Dequeue: O(1)	Read-only, immutable FIFO queue.	Immutable, thread-safe queue. 🔒
ImmutableSortedDictionary<TKey, TValue> 📊	Access: O(log n), Insert: O(n)	Read-only, immutable sorted	Immutable, sorted key-value pairs. 🏷️

		dictionary.	
ImmutableSortedSet 🔐	Add/Remove: O(log n)	Read-only, immutable sorted set.	Immutable, sorted, unique elements. 🛑
ImmutableStack 🏋️	Push/Pop: O(1)	Read-only, immutable stack.	Immutable stack for fixed content. 📦

Keys 🧠

- **General Collections**: Best for basic use without thread-safety or immutability concerns. 📈
- **Concurrent Collections**: Use these when working with multi-threaded environments, ensuring thread safety. 🔒
- **Immutable Collections**: Ideal when you need data integrity, ensuring collections cannot be modified after creation. 🛑
- **Specialized Collections**: Use them for specific needs such as priority-based tasks, ordered data, or key-value lookups. 🎯

Collection Interfaces

IEnumerable
ICollection
IList
IDictionary
IReadOnlyCollection<T>
IReadOnlyList<T>
IReadOnlyDictionary<TKey, TValue>
IQueryable

What is Deference between using Collection Interfaces or Collections ?

1️⃣ Collection Interfaces (`ICollection<T>` , `IList<T>`)

✅ **More flexible** – You can change the collection type later (switch from `List<T>` to `HashSet<T>`).

✅ **Better for abstraction** – Allows writing reusable code that works with any collection type.

✅ **Useful for dependency injection** – Makes unit testing easier by allowing mock data.

❌ **Slightly slower in some cases** due to **Interface Dispatch** (extra lookup when calling methods).

Interface Dispatch is the process of determining and calling the correct method implementation at runtime when using an interface, instead of directly calling a method on a concrete class. This adds a small performance overhead.

2️⃣ Concrete Collections (`List<T>` , `Dictionary<TKey, TValue>`)

✅ **Faster** – Direct method calls without extra lookup.

✅ **More control** – Provides additional methods (`Sort()` in `List<T>`).

✅ **Better for performance-critical code** – Avoids interface overhead.

❌ **Less flexible** – If you use `List<T>` , you **must** always use `List<T>` .

When to Use Each?

🔷 **Use Collection Interfaces** when:

- You want flexibility (switching between `List<T>` and `HashSet<T>`).
- You're writing reusable or testable code.
- You don’t need specific features of `List<T>` , `Dictionary<T>`

🔷 **Use Concrete Collections** when:

- You need maximum performance.
- You need collection-specific features (sorting in `List<T>` , fast lookups in `Dictionary<T>`).
- You don’t need to change the collection type later.

💡 **Best practice:** Use interfaces (`ICollection<T>` , `IList<T>`) when writing APIs or reusable code. Use concrete collections (`List<T>` , `Dictionary<T>`) inside methods when performance matters. 🚀

1️⃣ IEnumerable

1. Definition

IEnumerable<T> is a generic interface in the `System.Collections.Generic` namespace that represents a sequence of elements that can be enumerated (iterated over). It is the foundation of all collections in .NET that support iteration using `foreach` or LINQ queries. The non-generic version, `IEnumerable` , is also available for working with non-generic collections.

---

2. When to Use

- **Best to use when**:
  - You need to iterate over a collection of elements.
  - You want to support LINQ queries on your collection.

- You are creating a custom collection that needs to be enumerable.
- You want to abstract the iteration logic from the underlying data structure.

- **Avoid using when**:
  - You need to modify the collection (use `ICollection<T>` or `IList<T>` instead).
  - You need direct access to elements by index (use `IList<T>` or arrays instead).
  - You need to perform non-iterative operations like adding or removing elements.

---

3. Best and Worst Cases

Here's a table explaining the **time and space complexity** of `IEnumerable<T>` in its best and worst cases:

Operation	Best Case	Worst Case	Explanation
Iteration	`O(n)`	`O(n)`	Iterating through all elements is linear time.
LINQ Operations	`O(n)`	`O(n)`	Most LINQ operations (e.g., `Where`, `Select`) are linear time.
Space Complexity	`O(1)`	`O(1)`	`IEnumerable<T>` itself does not store elements; it only provides iteration.

---

4. Code Example

Here's a simple, complete code example demonstrating the features of `IEnumerable<T>` in C#:

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

class Program
{
    static void Main()
    {
        // Create a list of integers
        List<int> numbers = new List<int> { 10, 20, 30, 40, 50 };

        // Use IEnumerable<T> to iterate over the list
        IEnumerable<int> enumerable = numbers;

        // Iterate using foreach
        Console.WriteLine("Iterating using foreach:");
        foreach (int number in enumerable)
        {
            Console.WriteLine(number);
        }
```

```
        // Use LINQ to filter and project the sequence
        IEnumerable<int> filtered = enumerable.Where(x => x > 20);
        IEnumerable<string> projected = filtered.Select(x => $"Number: {x}");

        // Iterate over the filtered and projected sequence
        Console.WriteLine("Filtered and projected sequence:");
        foreach (string item in projected)
        {
            Console.WriteLine(item);
        }

        // Create a custom collection that implements IEnumerable<T>
        MyCollection<int> myCollection = new MyCollection<int>(new int[] { 1, 2, 3, 4, 5 });

        // Iterate over the custom collection
        Console.WriteLine("Iterating over custom collection:");
        foreach (int item in myCollection)
        {
            Console.WriteLine(item);
        }
    }
}

// Custom collection that implements IEnumerable<T>
class MyCollection<T> : IEnumerable<T>
{
    private T[] _items;

    public MyCollection(T[] items)
    {
        _items = items;
    }

    public IEnumerator<T> GetEnumerator()
    {
        foreach (T item in _items)
        {
            yield return item;
        }
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
```

```
    {
        return GetEnumerator();
    }
}
```

5. How to Increase Performance When Using IEnumerable<T>

1. **Use for Abstraction**:
   - Use `IEnumerable<T>` to abstract the iteration logic from the underlying data structure. This allows you to change the implementation without affecting the code that uses it.
2. **Leverage LINQ for Queries**:
   - Use LINQ methods like `Where`, `Select`, and `OrderBy` to perform complex queries on `IEnumerable<T>` sequences.
3. **Avoid Multiple Iterations**:
   - Iterating over an `IEnumerable<T>` multiple times can be expensive. If you need to reuse the results, materialize the sequence into a list or array using `ToList()` or `ToArray()`.

   ```
   List<int> list = enumerable.ToList();
   ```
4. **Use `yield return` for Custom Iterators**:
   - When implementing `IEnumerable<T>`, use `yield return` to create custom iterators. This simplifies the implementation and improves performance by lazily generating elements.
5. **Avoid Modifying the Collection During Iteration**:
   - Modifying a collection while iterating over it can lead to unexpected behavior or exceptions. Use a separate collection for modifications.
6. **Use `IEnumerable<T>` for Lazy Evaluation**:
   - `IEnumerable<T>` supports lazy evaluation, meaning elements are generated or fetched only when needed. Use this feature to optimize performance for large or expensive sequences.
7. **Combine with Other Interfaces**:
   - If you need additional functionality like adding or removing elements, combine `IEnumerable<T>` with other interfaces like `ICollection<T>` or `IList<T>`.

2️⃣ IQueryable

Definition:

`IQueryable<T>` is an interface in C# that represents a collection of objects that can be queried using Language Integrated Query (LINQ) syntax or method call syntax. It is part of the `System.Linq`

namespace and is used to translate queries into expressions that can be executed by different data sources, such as databases or in-memory collections.

When to Use:

`IQueryable<T>` is best used when:

- You want to write queries that are translated into a different query language, such as SQL for databases.
- You need to work with data sources that support the `IQueryable<T>` interface, like Entity Framework or LINQ to SQL.
- You want to take advantage of deferred execution, where the query is not executed until the data is actually needed.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Query Execution	Varies by data source	Varies by data source	The time complexity depends on the underlying data source and the complexity of the query.
Expression Tree Creation	O(n)	O(n)	Creating an expression tree from a query is linear in the number of operations in the query.

Simple Complete Code Example:
Here's an example demonstrating various features of `IQueryable<T>`:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample data
        var products = new[]
        {
            new { Name = "Apple", Price = 0.99m, Category = "Fruit" },
            new { Name = "Bread", Price = 1.99m, Category = "Bakery" },
            new { Name = "Carrot", Price = 0.69m, Category = "Vegetable" }
        }.AsQueryable();

        // Query using LINQ
        var query = from product in products
```

```
                where product.Price < 1m
                select product;

        // Execute query and print results
        foreach (var product in query)
        {
            Console.WriteLine($"{product.Name}: {product.Price}");
        }
    }
}
```

Output:

> Apple: 0.99
> Carrot: 0.69

How to Increase Performance When Using IQueryable:

1. **Optimize Queries**: Write efficient queries that minimize the amount of data processed and transferred.
2. **Use Indexes**: If you're querying a database, make sure that the columns used in `WHERE` clauses are indexed.
3. **Batching**: When working with databases, consider batching operations to reduce the number of round trips to the database.
4. **AsNoTracking**: When using Entity Framework, consider using `AsNoTracking()` to avoid loading related entities and to improve performance.
5. **Deferred Execution**: Take advantage of deferred execution to delay query execution as much as possible.

3️⃣ ICollection

Definition:

`ICollection<T>` is a non-generic interface in the `System.Collections` namespace that represents a collection of objects that can be individually added, removed, and accessed by index. It is part of the .NET Framework's collection hierarchy and is the base interface for non-generic collections.

When to Use:

`ICollection<T>` is best used when:

- You need a collection that supports adding, removing, and iterating over elements.
- You are working in a context where you cannot or do not want to use generic collections, such as when targeting .NET Framework versions older than .NET 2.0, which introduced generics.
- You require a basic collection interface without the additional functionality provided by more derived interfaces like `IList<T>` or `ICollection<T>` (generic version).

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Add	O(1)	O(n)	Adding an element is generally constant time unless the internal array needs to be resized.
Remove	O(1)	O(n)	Removing an element is constant time unless the element needs to be searched.
Access by Index	O(1)	O(1)	Accessing an element by index is constant time.
Contains	O(1)	O(n)	Checking if the collection contains a certain element is constant time if the element is found early, or linear in time if a search is required.

Simple Complete Code Example:
Here's an example demonstrating various features of `ICollection<T>` using the `ArrayList` class, which implements `ICollection`:

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        ArrayList collection = new ArrayList();

        // Adding elements
        collection.Add("Apple");
        collection.Add("Banana");
        collection.Add("Cherry");

        // Accessing an element by index
        Console.WriteLine("Element at index 1: " + collection[1]);

        // Removing an element
        collection.Remove("Banana");
        Console.WriteLine("Collection after removal:");
```

```
        foreach (var item in collection)
        {
            Console.WriteLine(item);
        }

        // Checking if the collection contains an element
        bool containsCherry = collection.Contains("Cherry");
        Console.WriteLine($"Contains Cherry: {containsCherry}");
    }
}
```

Output:

```
Element at index 1: Banan
Collection after removal:
Apple
Cherry
Contains Cherry: True
```

How to Increase Performance When Using ICollection:

1. **Pre-allocate Capacity**: If you know the approximate number of elements in advance, you can pre-allocate the capacity to avoid multiple reallocations.
2. **Batch Operations**: Use batch operations to add or remove multiple elements at once instead of one by one.
3. **Avoid Unnecessary Searches**: If you can structure your program logic to minimize the use of search operations, you can improve performance by avoiding the linear search altogether.
4. **Use Generic Collections**: If you are using a version of .NET that supports generics and your collection will only contain one type of element, consider using `ICollection<T>` (generic version) for better type safety and performance.

4️⃣ IList

Definition:

`IList<T>` is a generic interface in C# that represents a list of objects. It is part of the `System.Collections.Generic` namespace and provides methods to access and manipulate elements in the list. `IList<T>` is used when you need a collection that can be accessed by index and supports dynamic resizing.

When to Use:

`IList<T>` is best used when:

- You need to access elements by their index.
- You want to add or remove elements from the list.
- You need a collection that can grow or shrink in size.
- You are working with a list of items of the same type.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access by Index	O(1)	O(1)	Accessing an element by index is constant time.
Insertion	O(1)	O(n)	Adding an element at the end of the list is constant time, but adding at the beginning or middle requires shifting elements.
Removal	O(1)	O(n)	Removing an element from the end of the list is constant time, but removing from the beginning or middle requires shifting elements.
Searching	O(n)	O(n)	Searching for an element requires iterating through the list.

Simple Complete Code Example:
Here's an example demonstrating various features of `IList<T>`:

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

class Program
{
    static void Main()
    {
        List<int> numbers = new List<int>();

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

        // Accessing an element by index
        Console.WriteLine("Element at index 1: " + numbers[1]);

        // Inserting an element at a specific index
        numbers.Insert(1, 25);
```

```
        // Removing an element
        numbers.Remove(20);

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

Output:

```
Element at index 1: 20
10
25
30
```

How to Increase Performance When Using IList:

1. **Pre-allocate Capacity**: If you know the approximate number of elements in advance, you can pre-allocate the capacity using `List<T>.Capacity` to avoid multiple reallocations.
2. **Batch Operations**: Use batch operations like `AddRange` to add multiple elements at once instead of adding them one by one.
3. **Avoid Frequent Resizing**: Be mindful of the resizing behavior of `List<T>`. If you're adding a large number of elements, consider pre-allocating the necessary space to minimize the number of resize operations.
4. **Use Indexer for Random Access**: Use the indexer to access elements by index when you don't need to use the key, as this operation is generally faster.

5️⃣ IDictionary

Definition:

`IDictionary<TKey, TValue>` is a generic interface in C# that represents a collection of key/value pairs. It is part of the `System.Collections.Generic` namespace and provides methods to access and manipulate the elements in the dictionary. The interface ensures that each key is unique and can be used to quickly retrieve the associated value.

When to Use:

`IDictionary<TKey, TValue>` is best used when:

- You need to associate keys with values and retrieve values based on their keys.
- You require fast lookups, additions, and deletions based on keys.
- You want a collection that maintains a specific ordering of elements, as some implementations (like `SortedDictionary<TKey, TValue>`) can preserve the order of keys.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access by Key	O(1)	O(n)	Accessing an element by key is generally constant time, but can take linear time if the key is not found.
Add	O(1)	O(n)	Adding an element is generally constant time unless the internal data structure needs to be resized or rehashed.
Remove	O(1)	O(n)	Removing an element is constant time if the key is found quickly; otherwise, it can take linear time to search for the key.
Contains	O(1)	O(n)	Checking if the dictionary contains a certain key is constant time on average, but can take linear time in the worst case.

Simple Complete Code Example:
Here's an example demonstrating various features of `IDictionary<TKey, TValue>` using the `Dictionary<TKey, TValue>` class, which implements `IDictionary<TKey, TValue>`:

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

class Program
{
    static void Main()
    {
        Dictionary<string, int> dictionary = new Dictionary<string, int>();

        // Adding elements
        dictionary.Add("apple", 1);
        dictionary.Add("banana", 2);
        dictionary.Add("cherry", 3);

        // Accessing an element by key
        Console.WriteLine("Value for 'apple': " + dictionary["apple"]);
```

```
        // Checking if the dictionary contains a key
        bool containsCherry = dictionary.ContainsKey("cherry");
        Console.WriteLine("Contains 'cherry': " + containsCherry);

        // Removing an element
        dictionary.Remove("banana");
        Console.WriteLine("Dictionary after removal:");
        foreach (var kvp in dictionary)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }
    }
}
```

Output:

```
Value for 'apple': 1
Contains 'cherry': True
Dictionary after removal:
Key: apple, Value: 1
Key: cherry, Value: 3
```

How to Increase Performance When Using IDictionary:

1. **Pre-allocate Capacity**: If you know the approximate number of elements in advance, you can pre-allocate the capacity using `Dictionary<TKey, TValue>.Capacity` to avoid multiple reallocations.
2. **Use Efficient Hash Codes**: Ensure that the keys used have good hash codes to distribute them evenly across the hash table, reducing collisions and improving lookup performance.
3. **Avoid Resizing**: Be mindful of the resizing behavior of `Dictionary<TKey, TValue>`. If you're adding a large number of elements, consider pre-allocating the necessary space to minimize the number of resize operations.
4. **Use Concurrent Collections**: If you're working in a multi-threaded environment and need thread-safe operations, consider using `ConcurrentDictionary<TKey, TValue>` instead of `Dictionary<TKey, TValue>`.

6 IReadOnlyCollection<T>

Definition:

`IReadOnlyCollection<T>` is a generic interface in C# that represents a collection of elements that can be read but not modified. This interface provides a way to expose a collection to users of your class without allowing them to change the collection. It is part of the `System.Collections.ObjectModel` namespace.

When to Use:

`IReadOnlyCollection<T>` is best used when:

- You want to provide read-only access to a collection.
- You need to prevent modifications to a collection after it has been created.
- You want to wrap an existing collection to provide read-only access without creating a new copy of the collection.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(1)	Accessing an element by index is constant time.
Iteration	O(n)	O(n)	Iterating through the collection is linear time relative to the number of elements.
Contains	O(1)	O(n)	Checking if the collection contains a certain element is constant time if the element is found early, or linear in time if a search is required.

Simple Complete Code Example:
Here's an example demonstrating various features of `IReadOnlyCollection<T>`:

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

class Program
{
    static void Main()
    {
        List<int> list = new List<int> { 1, 2, 3, 4, 5 };
        IReadOnlyCollection<int> readOnlyList = new ReadOnlyCollection<int>(list);

        // Accessing an element by index
        Console.WriteLine("Element at index 2: " + readOnlyList[2]);

        // Iterating through the collection
        foreach (int number in readOnlyList)
        {
```

```
            Console.WriteLine(number);
        }

        // Checking if the collection contains an element
        bool containsThree = readOnlyList.Contains(3);
        Console.WriteLine("Contains 3: " + containsThree);
    }
}
```

Output:

```
Element at index 2: 3
1
2
3
4
5
Contains 3: True
```

How to Increase Performance When Using IReadOnlyCollection:

1. **Minimize Wrapper Creation**: Since `IReadOnlyCollection<T>` is just a wrapper, creating multiple wrappers can introduce unnecessary overhead. Create the wrapper only when necessary.
2. **Use Efficient Underlying Collections**: Choose an efficient underlying collection that provides fast read operations.
3. **Avoid Unnecessary Copies**: If you need to provide read-only access to a collection, consider using `IReadOnlyCollection<T>` instead of creating a copy of the collection.
4. **Use Parallel Processing**: If you need to perform read-only operations in parallel, consider using PLINQ or the `Parallel` class to take advantage of multi-core processors.

7️⃣ IReadOnlyDictionary

Definition:

`IReadOnlyDictionary<TKey, TValue>` is a generic interface in C# that represents a read-only collection of key/value pairs. It is part of the `System.Collections.Generic` namespace and provides a way to access elements by key without allowing modifications to the collection. This interface is useful when you want to ensure that a dictionary cannot be changed after it has been created, providing a layer of safety when exposing data to parts of your application that shouldn't need write access.

When to Use:

`IReadOnlyDictionary<TKey, TValue>` is best used when:

- You need to provide read-only access to a dictionary.
- You want to prevent modifications to a dictionary after it has been populated.
- You require a dictionary that can be safely shared across multiple threads without concern for concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access by Key	O(1)	O(log n)	Accessing an element by key is generally constant time, but can take longer if a hash collision occurs and the dictionary needs to probe for the correct slot.
Iteration	O(n)	O(n)	Iterating through the collection is linear time relative to the number of elements.
ContainsKey	O(1)	O(log n)	Checking if the dictionary contains a certain key is constant time on average, but can take longer due to hash collisions.

Simple Complete Code Example:
Here's an example demonstrating various features of `IReadOnlyDictionary<TKey, TValue>`:

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

class Program
{
   static void Main()
   {
      var dictionary = new Dictionary<string, int>{
        {"apple", 1},
        {"banana", 2},
        {"cherry", 3}
      };

      IReadOnlyDictionary<string, int> readOnlyDictionary = new ReadOnlyDictionary<string, int>(d

      // Accessing an element by key
      Console.WriteLine(readOnlyDictionary["banana"]); // Outputs: 2
```

```
        // Checking if the dictionary contains a key
        bool containsCherry = readOnlyDictionary.ContainsKey("cherry");
        Console.WriteLine(containsCherry); // Outputs: True

        // Iterating through the dictionary
        foreach (var kvp in readOnlyDictionary)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }
    }
}
```

Output:

```
2
True
Key: apple, Value: 1
Key: banana, Value: 2
Key: cherry, Value: 3
```

How to Increase Performance When Using IReadOnlyDictionary:

1. **Minimize Accesses**: Since accessing elements by key can have varying performance based on hash collisions, ensure that your keys have good hash codes to distribute them evenly across the hash table.
2. **Avoid Unnecessary Copies**: When passing a dictionary to parts of your application that shouldn't modify it, wrap it in `ReadOnlyDictionary<TKey, TValue>` to prevent changes without creating a new copy of the data.
3. **Use in Multi-threaded Scenarios**: Take advantage of `IReadOnlyDictionary<TKey, TValue>`'s thread safety when working with data across multiple threads, as it ensures that the dictionary cannot be concurrently modified.

🎱IReadOnlyList

Definition:

`IReadOnlyList<T>` is an interface in C# that represents a read-only list of elements. It is part of the

`System.Collections.Generic` namespace. This interface is not built into the .NET Framework, but you can create your own implementation of `IReadOnlyList<T>` to provide read-only access to a list.

When to Use:
You might want to use
`IReadOnlyList<T>` when:

- You need to provide read-only access to a list without allowing modifications.
- You want to ensure that the list cannot be changed after it has been created.
- You require a list that can be safely shared across multiple threads without concern for concurrent modifications.

Best and Worst Case Scenarios:

Operation	Best Case Time Complexity	Worst Case Time Complexity	Description
Access	O(1)	O(1)	Accessing an element by index is constant time.
Iteration	O(n)	O(n)	Iterating through the collection is linear time relative to the number of elements.
Contains	O(n)	O(n)	Checking if the collection contains a certain element is linear time.

Simple Complete Code Example:
Here's an example demonstrating how to create a read-only list using an interface:

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

class Program
{
    static void Main()
    {
        // Creating a List and casting it to IReadOnlyList
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
        IReadOnlyList<int> readOnlyNumbers = numbers.AsReadOnly();

        // Accessing elements from IReadOnlyList
        Console.WriteLine("Elements in IReadOnlyList:");
        for (int i = 0; i < readOnlyNumbers.Count; i++)
        {
            Console.WriteLine(readOnlyNumbers[i]);
        }
```

```
        // Trying to modify the list will give an error
        // readOnlyNumbers.Add(6); // Error: 'IReadOnlyList<int>' does not contain a definition for 'Ad
    }
}
```

Output:

> Elements in IReadOnlyList:
> 1
> 2
> 3
> 4
> 5

How to Increase Performance When Using IReadOnlyList:

1. **Minimize Wrapper Creation**: Creating multiple wrappers can introduce unnecessary overhead. Create the wrapper only when necessary.
2. **Use Efficient Underlying Collections**: Choose an efficient underlying collection that provides fast read operations.
3. **Avoid Unnecessary Copies**: If you need to provide read-only access to a list, consider using a wrapper instead of creating a copy of the list.
4. **Use in Multi-threaded Scenarios**: Take advantage of the thread safety of read-only collections when working with data across multiple threads, as it eliminates the need for additional synchronization mechanisms

What is Tuple?

A **Tuple** is like a small basket 🧺 where you can put different types of things together—numbers, text, or other values—just like how you might put fruit 🍎🍌 together in a basket! But once you put them in, you can't change them 🌿.

Key Features:

- **Immutable** (Can't change it once set) 🔒
- **Mixed types** (Can hold different things, like text and numbers!) 🔢💬
- **Fixed size** (No adding or removing items) 🔪

Use Cases:

1. **Returning Multiple Values** 🚀

   Need to return more than one thing from a function? Use a Tuple!

   ```
   csharp
   CopyEdit
   (25, "John") // 🎂 age and name
   ```

2. **Grouping Data Together** 🤝

   Want to keep related info together? Use a Tuple!

   ```
   csharp
   CopyEdit
   (name: "Alice", age: 30) // 👩 and 🎉 age
   ```

3. **Dictionary Keys** 🗝️

   Need to create a unique pair of keys in a dictionary? A Tuple can help!

   ```
   csharp
   CopyEdit
   (("John", 1)) ⇒ "Doctor" // 🧑‍⚕️
   ```

4. **Swapping Values** 🔁

   Want to swap values easily? A Tuple to the rescue!

   ```
   csharp
   CopyEdit
   (1, 2) → (2, 1) // 🔄 switch
   ```

Examples:

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

class TupleExamples
{
    // 1. Returning Multiple Values
    public static (int, string) GetPersonInfo()
```

```
{
    return (25, "John"); // 🎂 age and name
}

// 2. Grouping Related Data
public static void GroupData()
{
    var person = (name: "Alice", age: 30); // 👩 and 🎉 age
    Console.WriteLine($"Name: {person.name}, Age: {person.age}");
}

// 3. Using Tuple as Dictionary Key
public static void DictionaryWithTuple()
{
    var dictionary = new Dictionary<(string, int), string>();
    dictionary.Add(("John", 1), "Doctor"); // 👨‍⚕️

    var role = dictionary[("John", 1)];
    Console.WriteLine($"Role of John: {role}"); // 🎭 Role
}

// 4. Swapping Values Using Tuple
public static void SwapValues()
{
    var tuple = (a: 1, b: 2); // 🔢 Values
    var swapped = (tuple.b, tuple.a); // 🔄 swap
    Console.WriteLine($"Swapped: {swapped.a}, {swapped.b}");
}

static void Main(string[] args)
{
    // 1. Calling function to return multiple values
    var personInfo = GetPersonInfo();
    Console.WriteLine($"Person Info: {personInfo.Item1} years old, {personInfo.Item2}");

    // 2. Group related data
    GroupData();

    // 3. Dictionary with tuple as key
    DictionaryWithTuple();

    // 4. Swapping values
    SwapValues();
```

```
    }
}
```

LINQ🐋

LINQ (Language Integrated Query) is a feature in C# that allows you to query, filter, and manipulate data directly in your code using a simple, readable syntax. It works with various data sources like arrays, lists, and databases. 🌟

Filtering Methods 🔍

🎯 Method Name: `Where()`

🔍 **Category**: **Filtering Method**

📚 **Description**:

The `Where()` method filters elements in a sequence based on a given predicate (condition). It returns a sequence that contains only the elements that satisfy the condition. It's one of the most commonly used LINQ methods.

⚡ **Performance Tip**:

- To improve performance, use `Where()` with simple, fast conditions (e.g., value comparisons).
- Avoid complex or nested conditions inside `Where()`, as they can slow down execution.
- Try to apply filtering as early as possible in your query to minimize the number of elements being processed.

🛠️ **Usage**:

- Use `Where()` when you need to filter a collection based on a condition.
- It's lazy-loaded, meaning it won't execute until the sequence is iterated (great for performance).

📄 **Example Code**:

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

```
class Program
{
    static void Main()
    {
        // Creating a collection of numbers
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // Using Where() to filter even numbers
        var evenNumbers = numbers.Where(n => n % 2 == 0);

        // Output filtered result
        Console.WriteLine("Even numbers: " + string.Join(", ", evenNumbers));

        // 👀 Result will be: Even numbers: 2, 4, 6, 8, 10
    }
}
```

🎯 Method Name: `OfType()`

🔍 **Category**: **Filtering Method**

📚 **Description**:

The `OfType()` method filters a collection and only returns elements of a specific type, ignoring elements that are not of the specified type. It is particularly useful when working with collections that contain objects of different types and you want to filter only a certain type.

⚡ **Performance Tip**:

- Use `OfType()` when you need to filter by type, especially in mixed-type collections (e.g., `IEnumerable<object>` containing both integers and strings).
- Be mindful that `OfType()` performs a runtime type check, so avoid calling it repeatedly on large collections inside loops or performance-sensitive sections of your code.

🛠️ **Usage**:

- It's ideal when you have a collection of a base type (like `object`) and you want to select only a specific subclass type.
- It's also useful when working with collections that may contain multiple types (like `List<object>`), and you want to extract only the elements of a certain type.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a mixed collection of objects
        List<object> mixedCollection =
      new List<object> { 1, "Hello", 3.14, 2, "World", true };

        // Using OfType() to filter only integers from the collection
        var integers = mixedCollection.OfType<int>();

        // Output filtered result
        Console.WriteLine("Integers: " + string.Join(", ", integers));

        // 👀 Result will be: Integers: 1, 2
    }
}
```

🎯 Method Name: `Distinct()`

🔍 **Category**: **Set Operations**

📚 **Description**:

The `Distinct()` method removes duplicate elements from a collection, ensuring that each element in the returned sequence is unique. It works based on the default equality comparer, meaning it compares elements using their `Equals()` method and `GetHashCode()`.

⚡ **Performance Tip**:

- `Distinct()` can be useful in scenarios where you need to remove duplicates, but be mindful that it internally uses a hash set for comparison, which has a time complexity of **O(n)**.
- It's most effective when used on collections where duplicate elements are likely and you want a simple way to get unique results.

🛠️ **Usage**:

- Best used when you want to ensure that a collection only contains unique elements, and you don't care about the order.
- If you want to remove duplicates from an array or list, `Distinct()` is a quick and easy solution.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list with duplicate elements
        List<int> numbers = new List<int> { 1, 2, 2, 3, 4, 4, 5 };

        // Using Distinct() to remove duplicates
        var uniqueNumbers = numbers.Distinct();

        // Output the unique result
        Console.WriteLine("Unique Numbers: " + string.Join(", ", uniqueNumbers));

        // 🎯 Result will be: Unique Numbers: 1, 2, 3, 4, 5
    }
}
```

🎯 Method Name: `TakeWhile()`

🔍 **Category**: **Partitioning Methods**

📚 **Description**:

The `TakeWhile()` method returns elements from the start of a collection as long as a specified condition is true. Once the condition is no longer met, it stops and returns the elements collected up to that point.

⚡ **Performance Tip**:

- `TakeWhile()` is great for situations where you want to process a sequence of elements up until a certain condition fails.
- It has **O(n)** complexity, where **n** is the number of elements in the collection, and it short-circuits (stops) as soon as the condition fails.

🛠️ **Usage**:

- Useful when you need to take elements from the start of a collection until a certain condition is met, like processing a list of numbers until you find a negative value or a specific threshold.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of numbers
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7 };

        // Using TakeWhile to take numbers until the first number > 4
        var result = numbers.TakeWhile(n ⇒ n <= 4);

        // Output the result
        Console.WriteLine("Result: " + string.Join(", ", result));

        // 🎯 Result will be: Result: 1, 2, 3, 4
    }
}
```

🎯 Method Name: `SkipWhile()`

🔍 **Category**: **Partitioning Methods**

📚 **Description**:

The `SkipWhile()` method skips elements from the beginning of a collection as long as a specified condition is true. Once the condition fails, it returns the rest of the collection starting from that point.

⚡ **Performance Tip**:

- `SkipWhile()` is perfect for situations where you need to "ignore" or "skip" elements at the start of a sequence until a certain condition is no longer met.
- It has **O(n)** complexity, where **n** is the number of elements in the collection, and it stops once the condition fails, meaning it can be more efficient than scanning the entire sequence.

🛠️ **Usage**:

- Ideal when you need to skip over elements based on a condition (e.g., skip all negative numbers until you hit the first positive number).

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of numbers
        List<int> numbers = new List<int> { -1, -2, 3, 4, 5 };

        // Using SkipWhile to skip all negative numbers
        var result = numbers.SkipWhile(n ⇒ n < 0);

        // Output the result
        Console.WriteLine("Result: " + string.Join(", ", result));

        // 🎯 Result will be: Result: 3, 4, 5
    }
}
```

🎯 Method Name: `Take()`

🔍 **Category**: **Partitioning Methods**

📚 **Description**:

The `Take()` method retrieves a specified number of elements from the start of a collection. If the collection contains fewer elements than specified, it returns as many elements as available.

⚡ **Performance Tip**:

- `Take()` is ideal for when you need only the first **n** elements from a collection.
- It has **O(n)** complexity, where **n** is the number of elements to take, and it stops once the required number of elements is retrieved, making it efficient for limited data retrieval.

🛠️ **Usage**:

- Great for scenarios like paging through a collection, fetching the first few records, or working with a finite number of elements at the start of a sequence.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of numbers
        List<int> numbers = new List<int> { 10, 20, 30, 40, 50 };

        // Using Take to get the first 3 elements
        var result = numbers.Take(3);

        // Output the result
        Console.WriteLine("Result: " + string.Join(", ", result));

        // 🎯 Result will be: Result: 10, 20, 30
    }
}
```

🎯 Method Name: `Skip()`

🔍 **Category**: **Partitioning Methods**

📚 **Description**:

The `Skip()` method skips a specified number of elements from the beginning of a collection and returns the remaining elements after that.

⚡ **Performance Tip**:

- `Skip()` is useful when you want to "ignore" the first few elements of a collection, for example, when paginating data or skipping header rows.
- It has **O(n)** complexity, where **n** is the number of elements to skip, and then returns the rest of the collection, making it efficient for selective data processing.

🛠️ **Usage**:

- Ideal for scenarios like pagination (skip X items and take the next Y), processing data from a specific point, or skipping a known fixed number of elements.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of numbers
        List<int> numbers = new List<int> { 10, 20, 30, 40, 50 };

        // Using Skip to skip the first 2 elements
        var result = numbers.Skip(2);

        // Output the result
        Console.WriteLine("Result: " + string.Join(", ", result));

        // 🎯 Result will be: Result: 30, 40, 50
    }
}
```

🎯 Method Name: `DefaultIfEmpty()`

🔍 **Category**: **Element Inspection Methods**

📚 **Description**:

The `DefaultIfEmpty()` method returns the elements of a collection or a default value (e.g., `null` or `0`) if the collection is empty.

⚡ **Performance Tip**:

- This method is particularly useful when you need to ensure that your collection never returns empty, but instead defaults to a predefined value. It's good for scenarios where you need a fallback value.
- It has **O(n)** complexity, where **n** is the number of elements in the collection, and it adds an extra check for an empty sequence, so it's best used when there's a possibility of an empty collection.

🛠️ **Usage**:

- Ideal for situations where you expect an empty collection and want to avoid `null` checks by providing a fallback value.

⌨️ **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating an empty list of numbers
        List<int> numbers = new List<int>();

        // Using DefaultIfEmpty to provide a default value if the list is empty
        var result = numbers.DefaultIfEmpty(100);

        // Output the result
        Console.WriteLine("Result: " + string.Join(", ", result));

        // 🎯 Result will be: Result: 100
    }
}
```

Projection Methods 🎳

🎯 Method Name: `Select()`

🔍 **Category**: **Projection Methods**

📚 **Description**:

The `Select()` method projects each element of a sequence into a new form. It's one of the most commonly used methods for transforming or mapping data.

⚡ **Performance Tip**:

- It is highly efficient when transforming elements in a collection.
- **Use it with a projection** to transform data only when needed rather than fetching and processing unnecessary fields.
- Be mindful of **large collections**—the transformation logic should be lightweight for performance optimization.

🛠️ **Usage**:

- Ideal for scenarios where you need to transform or map elements into a new form. Commonly used in LINQ queries to shape data (e.g., converting an object into a DTO or extracting specific properties).

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of numbers
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

        // Using Select to transform each number into its square
        var squares = numbers.Select(n ⇒ n * n);

        // Output the result
        Console.WriteLine("Squares: " + string.Join(", ", squares));

        // 🎯 Result will be: Squares: 1, 4, 9, 16, 25
    }
}
```

🎯 Method Name: `SelectMany()`

🔍 **Category**: **Projection Methods**

📚 **Description**:

The `SelectMany()` method flattens a collection of collections into a single collection. It is used when each element of the original collection is itself a collection, and you want to flatten them into one sequence.

⚡ **Performance Tip**:

- It's a great way to flatten nested collections, but be careful when using it with **large nested collections** as it can lead to performance bottlenecks.

- **Consider the structure** of the data. If you're working with hierarchical data, `SelectMany()` helps to simplify the data structure, but it should be used judiciously to avoid unnecessary memory consumption.

🛠️ **Usage**:

- Use `SelectMany()` when you have a collection of collections (like a list of lists) and want to combine them into a single flat collection.
- Useful in scenarios where you are querying multiple collections and need to merge or flatten them into one.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // Creating a list of lists of numbers
        List<List<int>> listOfLists = new List<List<int>>()
        {
            new List<int> { 1, 2 },
            new List<int> { 3, 4 },
            new List<int> { 5, 6 }
        };

        // Using SelectMany to flatten the list of lists into a single sequence
        var flattened = listOfLists.SelectMany(list ⇒ list);

        // Output the result
        Console.WriteLine("Flattened: " + string.Join(", ", flattened));

        // 🎯 Result will be: Flattened: 1, 2, 3, 4, 5, 6
    }
}
```

🎯 Method Name: `Cast<T>()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `Cast<T>()` method is used to convert a non-generic collection to a specific type (`T`). It converts elements in the collection to the specified type, throwing an exception if any element cannot be cast to `T`.

⚡ **Performance Tip**:

- Use `Cast<T>()` when you're sure that all elements in the collection can be safely cast to the desired type.
- It's not efficient if you need to check types before casting—use `OfType<T>()` for type-safe conversions where you want to ignore items that can't be cast.

🛠️ **Usage**:

- Ideal when you have a non-generic collection (like `ArrayList` or `IEnumerable`) and you want to convert it to a specific type without iterating manually.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // A non-generic collection (ArrayList)
        ArrayList list = new ArrayList() { 1, 2, 3, 4 };

        // Use Cast<int>() to convert it to an IEnumerable<int>
        var intList = list.Cast<int>();

        // Output the result
        Console.WriteLine("Casted List: " + string.Join(", ", intList));

        // 🎯 Result will be: Casted List: 1, 2, 3, 4
    }
}
```

🎯 Method Name: `ToArray()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `ToArray()` method converts an `IEnumerable<T>` collection (like a list, query result, or any sequence) into an array of type `T[]`. It allows you to take the contents of any collection and put them into a fixed-size array.

⚡ **Performance Tip**:

- `ToArray()` is best used when you need a snapshot of the data that doesn't change after the conversion. It allocates memory for the array, so avoid calling it repeatedly in a loop.
- Be cautious of memory usage when converting large collections—arrays are fixed in size, so ensure you're not exceeding memory limits.

🛠️ **Usage**:

- Useful when you need to work with data as an array (e.g., for passing data to methods that require arrays or for performance optimizations that benefit from array operations).

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        // A simple collection
        var numbers = new List<int> { 1, 2, 3, 4, 5 };

        // Convert to an array using ToArray()
        int[] numberArray = numbers.ToArray();

        // Output the result
        Console.WriteLine("Array: " + string.Join(", ", numberArray));

        // 🎯 Result will be: Array: 1, 2, 3, 4, 5
    }
}
```

🎯 Method Name: `ToList()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `ToList()` method converts an `IEnumerable<T>` collection into a `List<T>`. It creates a new list and copies all elements from the source collection into it.

⚡ **Performance Tip**:

- `ToList()` is great for creating a copy of a collection when you need to modify the data, because a `List<T>` is mutable and allows for adding, removing, and accessing elements by index.
- Avoid unnecessary calls to `ToList()` in performance-critical code, especially within loops. Instead, work directly with `IEnumerable<T>` if you don't need the full list.

🛠️ **Usage**:

- Use it when you need a modifiable collection (like adding or removing items) or if you need to store data in a list format.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // A collection of numbers
        var numbers = new[] { 1, 2, 3, 4, 5 };

        // Convert the array to a List using ToList()
        List<int> numberList = numbers.ToList();

        // Output the result
        Console.WriteLine("List: " + string.Join(", ", numberList));

        // 🎯 Result will be: List: 1, 2, 3, 4, 5
    }
}
```

🎯 Method Name: `ToDictionary()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `ToDictionary()` method converts an `IEnumerable<T>` collection into a `Dictionary<TKey, TValue>`, where you specify the key and value for each item in the collection. It requires a key selector function to decide how to generate the key for each element and a value selector function for the values.

⚡ **Performance Tip**:

- The `ToDictionary()` method is fast when creating a dictionary from a collection, but it can be less efficient if you're using it with large datasets and if the key selector function is complex.
- Make sure the key is unique, as the `ToDictionary()` method will throw an exception if there are duplicate keys.

🛠️ **Usage**:

- Use it when you need to quickly access elements by a specific key, such as for lookups, caching, or grouping data in key-value pairs.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // A collection of objects
        var people = new[]
        {
            new { Name = "Alice", Age = 30 },
            new { Name = "Bob", Age = 25 },
            new { Name = "Charlie", Age = 35 }
        };

        // Convert the collection to a Dictionary using ToDictionary()
        Dictionary<string, int> peopleDict = people.ToDictionary(p => p.Name, p => p.Age);

        // Output the result
        foreach (var person in peopleDict)
        {
            Console.WriteLine($"{person.Key}: {person.Value}");
        }

        // 🎯 Result will be:
        // Alice: 30
        // Bob: 25
```

```
        // Charlie: 35
    }
}
```

🎯Method Name: `ToLookup()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `ToLookup()` method in LINQ creates a `Lookup<TKey, TElement>` from an `IEnumerable<T>`. It is similar to `ToDictionary()`, but instead of a single value, each key is associated with a collection of values. This is particularly useful when you have multiple elements with the same key and want to group them together.

⚡ **Performance Tip**:

- `ToLookup()` is efficient for grouping elements by a key. It's slightly slower than `ToDictionary()` because it allows multiple values per key, so it needs to store the values in a collection (such as a `List<T>` or `IEnumerable<T>`).
- It’s ideal when you need to group data but don't necessarily need fast key lookups (like a dictionary).

🛠️ **Usage**:

- Use it when you need to group elements based on a key and maintain all associated elements in an iterable collection.

⌨️ **Example Code**:

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

class Program
{
    static void Main()
    {
        // A collection of objects
        var people = new[]
        {
            new { Name = "Alice", Age = 30 },
            new { Name = "Bob", Age = 25 },
            new { Name = "Alice", Age = 35 },
            new { Name = "Charlie", Age = 35 }
```

```
        };

        // Convert the collection to a Lookup using ToLookup()
        var peopleLookup = people.ToLookup(p => p.Name);

        // Output the result
        foreach (var group in peopleLookup)
        {
            Console.WriteLine($"Group: {group.Key}");
            foreach (var person in group)
            {
                Console.WriteLine($"  {person.Name}, {person.Age}");
            }
        }

        // 🎯 Result will be:
        // Group: Alice
        //   Alice, 30
        //   Alice, 35
        // Group: Bob
        //   Bob, 25
        // Group: Charlie
        //   Charlie, 35
    }
}
```

🎯 Method Name: `AsEnumerable()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `AsEnumerable()` method in LINQ returns the elements of a sequence as an `IEnumerable<T>`. It's often used to explicitly cast a collection to `IEnumerable<T>` when you're working with other LINQ methods or operations that require this interface. The main benefit is that it allows you to call LINQ methods on collections that implement other interfaces, such as `IQueryable<T>` or `ICollection<T>`, without losing their specific functionality.

⚡ **Performance Tip**:

- `AsEnumerable()` is essentially a no-op, as it just returns the same collection with a different interface. There's no performance penalty for calling this method, but it can be helpful when working with different collection types.

- It should be used if you need to apply LINQ operations on collections that don't directly expose them (e.g., `IQueryable`), but don’t want to execute the underlying query prematurely.

🛠️ **Usage**:

- Use `AsEnumerable()` when you need to treat a collection like an `IEnumerable<T>` for LINQ methods but don’t want to perform deferred execution until further steps.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // A list of numbers
        var numbers = new List<int> { 1, 2, 3, 4, 5 };

        // Use AsEnumerable() to treat the collection as IEnumerable
        var evenNumbers = numbers.AsEnumerable().Where(n => n % 2 == 0);

        // Output the result
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }

        // 🎯 Result will be:
        // 2
        // 4
    }
}
```

🎯 Method Name: `AsQueryable()`

🔍 **Category**: **Conversion Methods**

📚 **Description**:

The `AsQueryable()` method in LINQ is used to convert an `IEnumerable<T>` collection into an `IQueryable<T>`. This is particularly useful when you need to perform operations that are optimized for query

providers, such as database queries, or when you want to leverage LINQ's deferred execution with external query providers like Entity Framework.

⚡ **Performance Tip**:

- `AsQueryable()` should be used when you need to run LINQ queries that will be translated into SQL or handled by a query provider. It allows for optimization of the query by the provider.
- Don't use `AsQueryable()` on collections that are already `IQueryable<T>`. It's redundant and won't offer any performance benefit in that case.

🛠️ **Usage**:

- Use `AsQueryable()` when working with collections that are not inherently `IQueryable<T>`, but you want to apply LINQ methods that are designed for queryable collections.
- Common use cases include scenarios with Entity Framework or other LINQ to SQL implementations.

📓 **Example Code**:

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

class Program
{
    static void Main()
    {
        // A list of numbers
        var numbers = new List<int> { 1, 2, 3, 4, 5 };

        // Use AsQueryable() to treat the collection as IQueryable
        var evenNumbers = numbers.AsQueryable().Where(n ⇒ n % 2 == 0);

        // Output the result
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }

        // 🎯 Result will be:
        // 2
        // 4
    }
}
```

Ordering Methods🧩

🎯 Method Name: `OrderBy()`

🔍 **Category**: **Ordering Methods**

📚 **Description**:

The `OrderBy()` method is used to sort the elements of a sequence in ascending order based on a specified key. It is one of the most commonly used LINQ methods for arranging data.

⚡ **Performance Tip**:

- `OrderBy()` uses a stable sorting algorithm (like MergeSort) in most implementations.
- When sorting large collections, it is important to keep in mind that the method could impact performance due to the need for comparing and rearranging elements.
- If the dataset is already sorted or doesn't require complex ordering, it may be better to avoid using `OrderBy()` as it adds unnecessary overhead.

🛠️ **Usage**:

- **Sorting Collections**: Use `OrderBy()` to sort data in ascending order based on one or more properties.
- **Simple Key Sorting**: When you need to sort by one field of an object, `OrderBy()` is a great choice.

📝 **Example Code**:

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

class Program
{
    static void Main()
    {
        // List of students
        var students = new List<Student>
        {
            new Student { Name = "John", Age = 23 },
            new Student { Name = "Sarah", Age = 21 },
            new Student { Name = "Mike", Age = 25 }
        };

        // Sort students by age using OrderBy()
```

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

        // Output the result
        foreach (var student in sortedStudents)
        {
            Console.WriteLine($"{student.Name} - {student.Age}");
        }

        // 🎯 Result will be:
        // Sarah - 21
        // John - 23
        // Mike - 25
    }
}

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

🎯 Method Name: `OrderByDescending()`

🔍 **Category**: **Ordering Methods**

📚 **Description**:

The `OrderByDescending()` method sorts the elements of a sequence in descending order based on a specified key. This method is essentially the reverse of `OrderBy()`.

⚡ **Performance Tip**:

- Similar to `OrderBy()`, `OrderByDescending()` utilizes stable sorting algorithms.
- Sorting in descending order may take slightly more time than sorting in ascending order, especially if the comparison of elements is complex.
- To optimize performance, consider using `OrderByDescending()` only when necessary, especially on large datasets.

🛠️ **Usage**:

- **Descending Order**: Use `OrderByDescending()` when you need to sort data in descending order based on one or more properties.
- **Descending Numeric or Date Sort**: This method is ideal when sorting numeric or date values in descending order.

📄 **Example Code**:

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

class Program
{
   static void Main()
   {
      // List of students
      var students = new List<Student>
      {
         new Student { Name = "John", Age = 23 },
         new Student { Name = "Sarah", Age = 21 },
         new Student { Name = "Mike", Age = 25 }
      };

      // Sort students by age in descending order using OrderByDescending()
      var sortedStudents = students.OrderByDescending(s => s.Age);

      // Output the result
      foreach (var student in sortedStudents)
      {
         Console.WriteLine($"{student.Name} - {student.Age}");
      }

      // 🎯 Result will be:
      // Mike - 25
      // John - 23
      // Sarah - 21
   }
}

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

🎯 Method Name: `ThenBy()`

🔍 **Category**: **Ordering Methods**

📚 **Description**:

The `ThenBy()` method is used to perform a secondary sort on elements that are already sorted using `OrderBy()`. It allows sorting by an additional key in ascending order.

⚡ **Performance Tip**:

- `ThenBy()` is useful for multi-level sorting (e.g., first by one property and then by another).
- Ensure that the primary key used in `OrderBy()` is already sorted to avoid unnecessary re-sorting.
- Use `ThenBy()` after `OrderBy()` for stable, multi-level sorting to keep your data organized.

🛠️ **Usage**:

- **Multi-Level Sorting**: When sorting on multiple keys, use `OrderBy()` for the primary key, and `ThenBy()` for secondary, tertiary, etc.
- **Secondary Sort for Tied Elements**: When the first sorting criterion results in tied values (e.g., multiple students with the same age), use `ThenBy()` to break the ties based on another criterion (e.g., name).

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // List of students
        var students = new List<Student>
        {
            new Student { Name = "John", Age = 23 },
            new Student { Name = "Sarah", Age = 21 },
            new Student { Name = "Mike", Age = 23 },
            new Student { Name = "Anna", Age = 21 }
        };

        // Sort students by age first, then by name in alphabetical order
        var sortedStudents = students
            .OrderBy(s => s.Age)       // First sort by Age
            .ThenBy(s => s.Name);      // Then sort by Name
```

```
        // Output the result
        foreach (var student in sortedStudents)
        {
            Console.WriteLine($"{student.Name} - {student.Age}");
        }

        // 🎯 Result will be:
        // Sarah - 21
        // Anna - 21
        // Mike - 23
        // John - 23
    }
}

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

🎯 Method Name: `ThenByDescending()`

🔍 **Category**: **Ordering Methods**

📚 **Description**:

The `ThenByDescending()` method allows you to perform a secondary sort on a collection after you've already used `OrderBy()` or `OrderByDescending()`. Unlike `ThenBy()`, `ThenByDescending()` sorts in **descending order**.

⚡ **Performance Tip**:

- Like `ThenBy()`, this method is used for multi-level sorting but ensures the secondary sorting happens in descending order.
- Keep the primary sorting criteria sorted before applying `ThenByDescending()` to avoid unnecessary reordering.

🛠️ **Usage**:

- **Descending Secondary Sort**: Use `ThenByDescending()` when you want to sort by a secondary criterion in descending order after an initial ascending sort.
- **For Reverse Order Sorting**: It's perfect when you need to reverse the order of items based on a secondary key.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        // List of students
        var students = new List<Student>
        {
            new Student { Name = "John", Age = 23 },
            new Student { Name = "Sarah", Age = 21 },
            new Student { Name = "Mike", Age = 23 },
            new Student { Name = "Anna", Age = 21 }
        };

        // Sort students by age first (ascending), then by name (descending)
        var sortedStudents = students
            .OrderBy(s => s.Age)          // First sort by Age in ascending order
            .ThenByDescending(s => s.Name); // Then sort by Name in descending order

        // Output the result
        foreach (var student in sortedStudents)
        {
            Console.WriteLine($"{student.Name} - {student.Age}");
        }

        // 🎯 Result will be:
        // Sarah - 21
        // Anna - 21
        // Mike - 23
        // John - 23
    }
}

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

🎯 Method Name: `Reverse()`

🔍 **Category**: **Miscellaneous Methods**

📚 **Description**:

The `Reverse()` method in LINQ reverses the order of elements in a collection. It is a simple way to invert the sequence of elements, such as reversing a list, array, or any collection that implements `IEnumerable<T>`.

⚡ **Performance Tip**:

- The `Reverse()` method is an in-memory operation. It's fast for small collections but could be less efficient with large datasets since it creates a new collection with the reversed order.
- Avoid using `Reverse()` multiple times in complex queries as it might add unnecessary overhead.

🛠️ **Usage**:

- **Reverse the Order**: Use `Reverse()` when you need to reverse the order of elements in a sequence, such as when processing data from the end to the beginning.

📝 **Example Code**:

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

class Program
{
   static void Main()
   {
      // List of numbers
      var numbers = new List<int> { 1, 2, 3, 4, 5 };

      // Reverse the list
     numbers.Reverse();

      // Output the result
      foreach (var number in  numbers)
      {
         Console.WriteLine(number);
      }

      // 🎯 Result will be:
      // 5
      // 4
```

```
        // 3
        // 2
        // 1
    }
}
```

🎯 Method Name: `Zip()`

🔍 **Category**: **Miscellaneous Methods**

📚 **Description**:

The `Zip()` method combines two sequences into one, pairing elements from both sequences. It produces a new sequence where each element is a combination of corresponding elements from two collections. It's useful when you need to merge two sequences element by element.

⚡ **Performance Tip**:

- `Zip()` is optimized for cases where you want to combine two sequences with the same or similar lengths. Ensure that the sequences are not excessively large, as combining them may involve extra memory and processing overhead.

🛠️ **Usage**:

- **Merging Two Sequences**: Use `Zip()` when you need to combine two sequences in a pair-wise fashion, like combining two lists of items into a list of tuples.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 1, 2, 3 };
        var letters = new[] { "A", "B", "C" };

        var zipped = numbers.Zip(letters, (n, l) => new { Number = n, Letter = l });

        foreach (var item in zipped)
        {
            Console.WriteLine($"{item.Number} - {item.Letter}");
        }
        // 🎯 Output:
        // 1 - A
```

```
        // 2 - B
        // 3 - C
    }
}
```

Aggregation Methods🎲

🎯 Method Name: `Sum()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Sum()` method calculates the sum of numeric values in a collection. It is typically used for arrays, lists, or any collection that contains numeric data (e.g., integers, floats, decimals). If the collection is empty, it returns 0 by default.

⚡ **Performance Tip**:

- `Sum()` is an efficient method when working with collections of numbers, but keep in mind that for large collections, it iterates over all elements.
- You can improve performance when dealing with complex data structures by using a filtered collection (e.g., `Where()`) before calling `Sum()`.

🛠️ **Usage**:

- **Calculate Total**: Use `Sum()` to calculate the total value from a list of numbers, such as summing up expenses or scores.

📝 **Example Code**:

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

class Program
{
    static void Main()
    {
        // List of expenses
        var expenses = new List<int> { 100, 200, 300, 400, 500 };

        // Calculate the total sum of expenses
        var totalExpense = expenses.Sum();
```

```
        // Output the result
        Console.WriteLine($"Total Expense: {totalExpense}"); // 🎯 Total Expense: 1500
    }
}
```

🎯 Method Name: `Average()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Average()` method computes the average value of a numeric collection. It is commonly used when you need to find the mean of a set of numbers (e.g., the average age, salary, or score).

⚡ **Performance Tip**:

- `Average()` is fast for small collections, but for large datasets, consider using `AsEnumerable()` or filtering with `Where()` first to optimize performance.

🛠️ **Usage**:

- **Find Average**: Calculate the average value of a collection of integers or floats.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var scores = new List<int> { 85, 90, 78, 88, 92 };

        var averageScore = scores.Average();

        Console.WriteLine($"Average Score: {averageScore}"); // 🎯 Average Score: 86.6
    }
}
```

🎯 Method Name: `Min()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Min()` method returns the smallest value from a collection. It can be applied to any collection of comparable types (e.g., integers, decimals, strings).

⚡ **Performance Tip**:

- `Min()` efficiently iterates through the collection once, making it a great choice when you need the minimum value without extra operations.

🛠️ **Usage**:

- **Find Minimum**: Use `Min()` to find the lowest number in a list of values, like finding the minimum temperature or lowest price.

📃 **Example Code**:

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

class Program
{
    static void Main()
    {
        var numbers = new List<int> { 10, 20, 5, 15, 30 };

        var minNumber = numbers.Min();

        Console.WriteLine($"Min Number: {minNumber}"); // 🎯 Min Number: 5
    }
}
```

🎯 Method Name: `Max()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Max()` method returns the largest value from a collection. It works similarly to `Min()` but for the maximum value.

⚡ **Performance Tip**:

- `Max()` is efficient when you only need the highest value, but be mindful of the collection size for performance.

🛠️ **Usage**:

- **Find Maximum**: Use `Max()` to find the highest number in a set of data, such as the highest salary or temperature.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var temperatures = new List<int> { 32, 45, 50, 38, 60 };

        var maxTemperature = temperatures.Max();

        Console.WriteLine($"Max Temperature: {maxTemperature}"); // 🎯 Max Temperature: 60
    }
}
```

🎯 Method Name: `Count()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Count()` method returns the number of elements in a collection. It's commonly used to count the total items or the number of elements that satisfy a condition when used with `Where()`.

⚡ **Performance Tip**:

- `Count()` operates efficiently on collections. Use it after filtering or with LINQ to count specific elements.

🛠️ **Usage**:

- **Count Elements**: Use `Count()` to count the number of items, like counting users or products.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var products = new List<string> { "Apple", "Banana", "Orange", "Grapes" };
```

```
        var productCount = products.Count();

        Console.WriteLine($"Total Products: {productCount}"); // 🎯 Total Products: 4
    }
}
```

🎯 Method Name: `LongCount()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `LongCount()` method is similar to `Count()` but returns a `long` value, which is useful when you are dealing with large collections that exceed the `int` range.

⚡ **Performance Tip**:

- Use `LongCount()` when working with large datasets or collections, especially when you're expecting more than 2 billion elements.

🛠️ **Usage**:

- **Count Large Collections**: Use `LongCount()` for collections that might exceed the range of `int`.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var largeCollection = Enumerable.Range(1, int.MaxValue).ToList(); // Simulating a large colle
ction

        var count = largeCollection.LongCount();

        Console.WriteLine($"Number of elements: {count}");
        // 🎯 Number of elements: 2147483647
    }
}
```

🎯 Method Name: `Aggregate()`

🔍 **Category**: **Aggregation Methods**

📚 **Description**:

The `Aggregate()` method performs a custom aggregation operation on a collection. It allows you to define how to combine the elements of a sequence using a function, enabling more complex calculations.

⚡ **Performance Tip**:

- Use `Aggregate()` when you need custom operations, but for simple sums or averages, built-in methods like `Sum()` or `Average()` might be more performant.

🛠️ **Usage**:

- **Custom Aggregation**: Use `Aggregate()` when performing complex calculations like combining all strings or multiplying all numbers in a collection.

📃 **Example Code**:

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

class Program
{
    static void Main()
    {
        var numbers = new List<int> { 1, 2, 3, 4 };

        var product = numbers.Aggregate((total, next) => total * next);

        Console.WriteLine($"Product: {product}"); // 🎯 Product: 24 (1*2*3*4)
    }
}
```

🎯 Method Name: `Concat()`

🔍 **Category**: **Concatenation Methods**

📚 **Description**:

The `Concat()` method combines two sequences (collections) into one. The original collections remain unchanged, and the result is a new sequence that contains all the elements of the first collection followed by all the elements of the second.

⚡ **Performance Tip**:

- `Concat()` is useful for merging collections, but avoid using it with large collections repeatedly, as it creates a new collection each time.

🛠️ **Usage**:

- **Concatenate Collections**: Use `Concat()` when merging multiple sequences into one.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var numbers1 = new List<int> { 1, 2, 3 };
        var numbers2 = new List<int> { 4, 5, 6 };

        var mergedNumbers = numbers1.Concat(numbers2);

        Console.WriteLine(string.Join(", ", mergedNumbers)); // 🎯 1, 2, 3, 4, 5, 6
    }
}
```

Join Methods🌟

🎯 Method Name: `Join()`

🔍 **Category**: **Join Methods**

📚 **Description**:

The `Join()` method is used to combine elements from two sequences based on a matching key. It results in a new sequence that pairs matching elements from the two collections.

⚡ **Performance Tip**:

- `Join()` performs well with large datasets because it uses an efficient internal algorithm, but ensure the key you're joining on is indexed if possible for optimal performance.

🛠️ **Usage**:

- **Inner Join**: Use `Join()` when you want to combine two collections where each item in the first collection has a corresponding match in the second collection.

📄 **Example Code**:

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

class Program
{
    static void Main()
    {
        var customers = new List<Customer>
        {
            new Customer { Id = 1, Name = "John" },
            new Customer { Id = 2, Name = "Jane" }
        };

        var orders = new List<Order>
        {
            new Order { CustomerId = 1, Product = "Laptop" },
            new Order { CustomerId = 2, Product = "Tablet" }
        };

        var joinedData = customers.Join(orders,
                                customer => customer.Id,
                                order => order.CustomerId,
                                (customer, order) => new
                                {
                                    customer.Name,
                                    order.Product
                                });

        foreach (var item in joinedData)
        {
            Console.WriteLine($"Customer: {item.Name}, Product: {item.Product}");
        }
        // 🎯 Output:
        // Customer: John, Product: Laptop
        // Customer: Jane, Product: Tablet
    }
}

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

```
}

class Order
{
    public int CustomerId { get; set; }
    public string Product { get; set; }
}
```

🎯 Method Name: `GroupJoin()`

🔍 **Category**: **Join Methods**

📚 **Description**:

The `GroupJoin()` method is a variation of `Join()`, but it groups the results into collections of matching elements from the second sequence for each element in the first sequence. It's useful for one-to-many relationships.

⚡ **Performance Tip**:

- `GroupJoin()` is efficient for grouping related data. However, for large datasets, using `AsEnumerable()` or filtering with `Where()` beforehand can improve performance.

🛠️ **Usage**:

- **Left Join**: Use `GroupJoin()` to get a collection of items that may not have a corresponding match in the second collection (left join).

⌨️ **Example Code**:

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

class Program
{
    static void Main()
    {
        var customers = new List<Customer>
        {
            new Customer { Id = 1, Name = "John" },
            new Customer { Id = 2, Name = "Jane" }
        };

        var orders = new List<Order>
        {
            new Order { CustomerId = 1, Product = "Laptop" }
            // Jane has no orders
```

```
        };

        var groupedData = customers.GroupJoin(orders,
                                  customer ⇒ customer.Id,
                                  order ⇒ order.CustomerId,
                                  (customer, customerOrders) ⇒ new
                                  {
                                     customer.Name,
                                     Orders = customerOrders.Select(order ⇒ order.Product)
                                  });

        foreach (var item in groupedData)
        {
           Console.WriteLine($"Customer: {item.Name}");
           if (item.Orders.Any())
           {
              foreach (var product in item.Orders)
              {
                 Console.WriteLine($"  Product: {product}");
              }
           }
           else
           {
              Console.WriteLine("  No Orders");
           }
        }
        // 🎯 Output:
        // Customer: John
        //   Product: Laptop
        // Customer: Jane
        //   No Orders
    }
}

class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Order
{
    public int CustomerId { get; set; }
```

```
    public string Product { get; set; }
}
```

Generation Methods💫

🎯 Method Name: `Range()`

🔍 **Category**: **Generation Methods**

📚 **Description**:

The `Range()` method generates a sequence of integral numbers within a specified range. It's useful when you need to create a list of numbers for iteration or other operations.

⚡ **Performance Tip**:

- `Range()` is very efficient for creating number sequences and is highly optimized for performance. It's ideal when working with predictable ranges.

🛠️ **Usage**:

- **Generating a Range of Numbers**: Use `Range()` when you need a simple list of numbers, either for iterations or initialization of collections.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = Enumerable.Range(1, 5); // Generates numbers 1 to 5
        foreach (var number in numbers)
        {
            Console.WriteLine(number);
        }
        // 🎯 Output:
        // 1
        // 2
        // 3
        // 4
        // 5
```

```
    }
}
```

🎯 Method Name: `Repeat()`

🔍 **Category**: **Generation Methods**

📚 **Description**:

The `Repeat()` method generates a sequence where each element is the same, repeated a specified number of times. It's useful for creating collections with the same item.

⚡ **Performance Tip**:

- `Repeat()` is perfect for initializing a collection with repeated elements and is highly performant for cases where you need identical elements.

🛠️ **Usage**:

- **Repeating an Element**: Use `Repeat()` when you need a collection of repeated elements, like initializing arrays or lists where all items have the same value.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var repeatedItems = Enumerable.Repeat("Hello", 3); // Repeats "Hello" 3 times
        foreach (var item in repeatedItems)
        {
            Console.WriteLine(item);
        }
        // 🎯 Output:
        // Hello
        // Hello
        // Hello
    }
}
```

Element Inspection Methods

🎯 Method Name: `First()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `First()` method returns the first element of a sequence. It throws an exception if the sequence is empty.

⚡ **Performance Tip**:

- **Be careful** when using `First()` on empty sequences, as it can cause exceptions. Use `FirstOrDefault()` if there's a possibility the sequence may be empty.

🛠️ **Usage**:

- **Retrieving First Item**: Use `First()` when you're sure that the sequence contains at least one element.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 5, 10, 15 };

        var firstNumber = numbers.First();  // 🎯 Output: 5
        Console.WriteLine(firstNumber);
    }
}
```

🎯 Method Name: `FirstOrDefault()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `FirstOrDefault()` method returns the first element of a sequence, or a default value (`null` for reference types) if the sequence is empty.

⚡ **Performance Tip**:

- **Safer for Empty Sequences**: Use `FirstOrDefault()` when you expect that the sequence might be empty, as it won't throw exceptions.

🛠️ **Usage**:

- **Gracefully Handling Empty Sequences**: Use `FirstOrDefault()` to avoid exceptions when the sequence might be empty.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var emptyList = new int[] { };

        var firstNumber = emptyList.FirstOrDefault();  // 🎯 Output: 0 (default for int)
        Console.WriteLine(firstNumber);
    }
}
```

🎯 Method Name: `Last()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `Last()` method returns the last element of a sequence. It throws an exception if the sequence is empty.

⚡ **Performance Tip**:

- **Check Sequence**: Ensure the sequence is not empty before using `Last()` to avoid exceptions.

🛠️ **Usage**:

- **Retrieving Last Item**: Use `Last()` when you want the last element, and you know the sequence contains elements.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 5, 10, 15 };
```

```
        var lastNumber = numbers.Last();  // 🎯 Output: 15
        Console.WriteLine(lastNumber);
    }
}
```

🎯 Method Name: `LastOrDefault()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `LastOrDefault()` method returns the last element of a sequence, or a default value (`null` for reference types) if the sequence is empty.

⚡ **Performance Tip**:

- **Graceful Handling**: Use `LastOrDefault()` to avoid exceptions in case of empty sequences.

🛠️ **Usage**:

- **Safe Retrieval of Last Item**: Use `LastOrDefault()` to safely retrieve the last item or handle empty sequences.

📝 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var emptyList = new int[] { };

        var lastNumber = emptyList.LastOrDefault();  // 🎯 Output: 0 (default for int)
        Console.WriteLine(lastNumber);
    }
}
```

🎯 Method Name: `Single()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `Single()` method returns the only element in a sequence that contains exactly one element. If there is more than one element, it throws an exception.

⚡ **Performance Tip**:

- **Use for Unique Items**: `Single()` is ideal when you expect exactly one element in the sequence. Otherwise, use `SingleOrDefault()` or `FirstOrDefault()` if you're unsure about the sequence's size.

🛠️ **Usage**:

- **Retrieve Unique Element**: Use `Single()` when you expect a sequence to have only one element.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 5 };

        var singleNumber = numbers.Single();  // 🎯 Output: 5
        Console.WriteLine(singleNumber);
    }
}
```

🎯 Method Name: `SingleOrDefault()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `SingleOrDefault()` method returns the only element in a sequence or a default value if the sequence is empty. If there are multiple elements, it throws an exception.

⚡ **Performance Tip**:

- **Safe for Empty Sequences**: Use `SingleOrDefault()` to safely handle cases where the sequence might be empty.

🛠️ **Usage**:

- **Unique Item or Default**: Use `SingleOrDefault()` when you expect either a single element or none in the sequence.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
```

```
{
    static void Main()
    {
        var numbers = new[] { 5, 10, 15 };

        var singleNumber = numbers.SingleOrDefault();  // 🎯 Exception thrown because of multipl
e elements
        Console.WriteLine(singleNumber);
    }
}
```

🎯 Method Name: `ElementAt()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `ElementAt()` method returns the element at the specified index in a sequence. It throws an exception if the index is out of range.

⚡ **Performance Tip**:

- **Check Bounds**: Use `ElementAt()` carefully to avoid exceptions when accessing an invalid index. Consider `ElementAtOrDefault()` as a safer alternative.

🛠️ **Usage**:

- **Access by Index**: Use `ElementAt()` to retrieve an element at a specific index.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 1, 2, 3, 4, 5 };

        var elementAtIndex = numbers.ElementAt(2);  // 🎯 Output: 3
        Console.WriteLine(elementAtIndex);
    }
}
```

🎯 Method Name: `ElementAtOrDefault()`

🔍 **Category**: **Element Methods**

📚 **Description**:

The `ElementAtOrDefault()` method returns the element at the specified index or a default value if the index is out of range.

⚡ **Performance Tip**:

- **Safer Alternative**: Use `ElementAtOrDefault()` when you want to avoid exceptions due to out-of-range indices.

🛠️ **Usage**:

- **Safe Indexing**: Use `ElementAtOrDefault()` for safer indexing that doesn't throw exceptions on invalid indices.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 1, 2, 3 };

        var elementAtIndex = numbers.ElementAtOrDefault(5);  // 🎯 Output: 0 (default for int)
        Console.WriteLine(elementAtIndex);
    }
}
```

Quantifiers

🎯 Method Name: `Any()`

🔍 **Category**: **Query Methods**

📚 **Description**:

The `Any()` method checks if any elements in the sequence satisfy a condition or if the sequence contains any elements at all.

⚡ **Performance Tip**:

- **Quick Exit**: `Any()` will stop processing as soon as it finds a matching element. Ideal for early termination in large collections.

🛠️ **Usage**:

- **Check Existence**: Use `Any()` when you need to check if there are any elements in a sequence or if any elements meet a condition.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 1, 2, 3, 4, 5 };

        var hasEvenNumbers = numbers.Any(n => n % 2 == 0);  // 🎯 Output: True
        Console.WriteLine(hasEvenNumbers);
    }
}
```

🎯 Method Name: `All()`

🔍 **Category**: **Query Methods**

📚 **Description**:

The `All()` method checks if all elements in the sequence satisfy a specified condition.

⚡ **Performance Tip**:

- **Complete Traversal**: `All()` will traverse the entire sequence, so be cautious with large sequences if early termination is needed.

🛠️ **Usage**:

- **Ensure All Conditions Met**: Use `All()` when you need to confirm that every element in the collection satisfies a specific condition.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
```

```
{
    static void Main()
    {
        var numbers = new[] { 2, 4, 6, 8, 10 };

        var allEven = numbers.All(n => n % 2 == 0);  // 🎯 Output: True
        Console.WriteLine(allEven);
    }
}
```

🎯 Method Name: `Contains()`

🔍 **Category**: **Query Methods**

📚 **Description**:

The `Contains()` method checks if a sequence contains a specific element.

⚡ **Performance Tip**:

- **Efficient for Simple Checks**: `Contains()` is useful for checking the existence of an element, but be aware that it performs a linear search.

🛠️ **Usage**:

- **Check for Specific Element**: Use `Contains()` when you want to check if a sequence contains a particular element.

📝 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var numbers = new[] { 1, 2, 3, 4, 5 };

        var containsThree = numbers.Contains(3);  // 🎯 Output: True
        Console.WriteLine(containsThree);
    }
}
```

🎯 Method Name: `SequenceEqual()`

🔍 **Category**: **Query Methods**

📚 **Description**:

The `SequenceEqual()` method checks if two sequences are equal by comparing their elements in order.

⚡ **Performance Tip**:

- **Exact Match Comparison**: `SequenceEqual()` compares both the values and the order of elements, so it's ideal for checking whether two sequences are identical.

🛠️ **Usage**:

- **Compare Sequences**: Use `SequenceEqual()` when you need to check if two sequences are exactly the same.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var sequence1 = new[] { 1, 2, 3 };
        var sequence2 = new[] { 1, 2, 3 };

        var areEqual = sequence1.SequenceEqual(sequence2);  // 🎯 Output: True
        Console.WriteLine(areEqual);
    }
}
```

Grouping Methods

🎯 Method Name: `GroupBy()`

🔍 **Category**: **Grouping Methods**

📚 **Description**:

The `GroupBy()` method groups elements of a sequence based on a specified key. This allows you to categorize data into collections based on shared properties.

⚡ **Performance Tip**:

- **Efficiency for Categorization**: `GroupBy()` is highly efficient for organizing elements into categories, but be cautious with large data sets as it involves sorting.

🛠️ **Usage**:

- **Group Elements by a Key**: Use `GroupBy()` when you need to group data based on specific criteria like age, category, etc.

📄 **Example Code**:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var people = new[]
        {
            new { Name = "Alice", Age = 30 },
            new { Name = "Bob", Age = 25 },
            new { Name = "Charlie", Age = 30 },
            new { Name = "David", Age = 25 }
        };

        var groupedByAge = people.GroupBy(p => p.Age);  // 🎯 Grouping by Age

        foreach (var group in groupedByAge)
        {
            Console.WriteLine($"Age: {group.Key}");
            foreach (var person in group)
            {
                Console.WriteLine($" - {person.Name}");
            }
        }
    }
}
```

🎯 **Output**:

```
Age: 30
 - Alice
 - Charlie
Age: 25
```

- Bob
- David

🔥 See you soon in Database 💉🧠