Pytania na rozmowę kwalifikacyjną

1. **Czym jest C#?**

- Język kompilowany wysokopoziomowy zorientowany obiektowo stworzony poprzez Microsoft,

- na podstawie C, C++,

- jest ogólnego przeznaczenia,

- kompilowany poprzez CLR (Common Language Runtime – maszyna wirtualna, środowisko uruchoieniowe) do Intermediate Language (IL) a potem do kodu natywnego.

**2. Czym jest .NET Framework?**

- Platforma programistyczna opracowana przez Microsoft, 2002,

- Obejmuje CLR, czyli środowisko uruchomieniowe, na którym działają napisane programy, a także biblioteki klas (zbiór funkcjonalności i metod),

- tylko aplikacje na Windows

**3. Jakie są różnice pomiędzy.NET Core, a .NET Framework?**

- .NET Core open source, 2016,

- .NET Core: Windows, Linux i macOS,

- .NET Core ma bardziej zoptymalizowane biblioteki i środowisko uruchomieniowe, bardziej wydajne aplikacje,

**4) Czym są namespace'y w C#?**  
Dzięki namespace, czyli przestrzeni nazw w C# może lepiej poukładać kod w C# i wprowadzić separację kodu. Przestrzenie nazw można traktować jako takie pudełko, które składa się z innych przestrzeni nazw, klas, metod, właściwości itd. W celu zdefiniowania przestrzeni nazw używamy słowa kluczowego namespace: Przykład:  
  
namespace App; //for the whole cs file

class Program

{

static void Main(string[] args)

{

Console.ReadLine();

}

}

OR

namespace App

{

class Program

{

static void Main(string[] args)

{

Console.ReadLine();

}

}

}

Czyli nasza klasa Program jest dostępna wewnątrz przestrzeni nazw App. Jeżeli natomiast chcielibyśmy użyć tej klasy w innym namespace'ie, to musimy podać pełną nazwę klasy wraz z przestrzenią nazw, lub dodać odwołanie za pomocą słowa kluczowego using.

using app;

**5. TypeScript jest kompilowany do JavaScript**

### ****10) Jaka jest różnica pomiędzy deklaracją a inicjalizacją?****

Sama deklaracja to tylko stworzenie miejsca w pamięci gdzie będzie przechowywana wartość danej zmiennej.

int a;

Natomiast inicjalizacja, to przypisanie jakiejś wartości do wcześniej już zadeklarowanej na przykład zmiennej.

a = 1;

### ****11) Czym jest konstruktor?****

Konstruktor jest to specjalna metoda, o takiej samej nazwie co klasa. Nie zawiera w sygnaturze typu zwracanego. Jest wywoływany przy utworzeniu obiektu. Konstruktorów może być wiele, jeżeli mają inne sygnatury, ale może też nie być żadnego, w takim przypadku uruchomiony zostaje domyślnie pusty konstruktor.

Default Constructor:

If a class does not explicitly define any constructors, C# provides a default constructor implicitly. The default constructor takes no parameters and initializes the object's members to their default values (e.g., numeric values to 0, object references to null, etc.).

### ****12) GUID (Global Unique Identifier)****

Genercja prawie unikalnego ciągu z 32 znaków:

Guid.NewGuid().ToString(); //7497599a-d2fc-43bc-84b1-ed67156f7672

### ****16) Wyjaśnij słowo kluczowe static?****

Jeśli klasa jest static:

- wszystkie elementy powinni zawierać modyfikator static,

- nie można stworzyć instancji klasy

Zwykła klasa z elementem static - dostęp do elementu poprzez klasę bądź instancję.

### ****17) Do czego służy słowo kluczowe this?****

1) Odwoływanie do bieżącej instancji klasy

2) W metodach rozszerzających:

public static class StringExt

{

   public static string ToAlternatingCase(this string s)

   {

   return String.Concat(s.Select(ch => char.IsUpper(ch) ? char.ToLower(ch) : char.ToUpper(ch)));

   }

}

Metoda rozszerzająca:

- klasa ma być statyczna (więc wszystkie elementy też)

- pierwszy parametr ma być z „this”

### ****20) Jak nazywa się klasa bazowa dla wszystkich wyjątków?****

Bazową klasą dla wszystkich wyjątków jest klasa Exception. Jeżeli chcesz stworzyć własny wyjątek, to Twoja klasa powinna właśnie dziedziczyć po klasie Exception.

### ****9. How does managed code execute in the .NET framework?****

There are four main steps that include in the execution of the managed code. They are as follows:

1. Choosing a compiler that can execute the code written by a user
2. Conversion of the code into Intermediate Language (IL) using a compiler
3. IL gets pushed to CLR, which converts it into native code using JIT
4. Native code is now executed using the .NET runtime

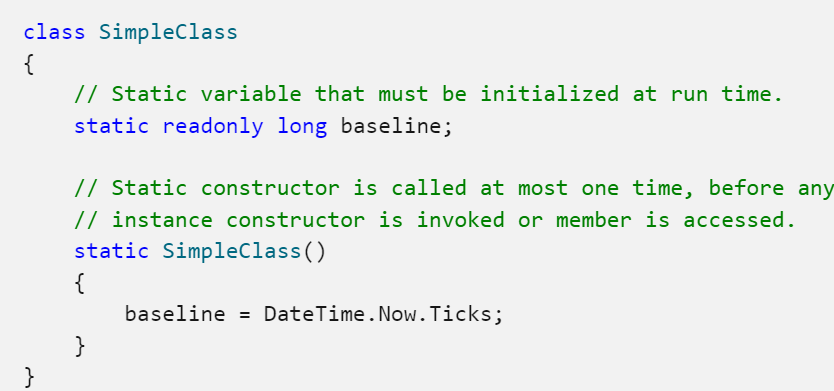
|  |  |
| --- | --- |
| Abstract class | Interface |
| Abstract classes can contain common methods that will use every derived class. | Interfaces also can have the methods with body, but you can call them only if you treat your class as this interface. |
| Can have a constructor | No constructor |
| Abstract methods can’t have a body and they are the same as empty methods in interface |  |

The **Common Language Infrastructure** (CLI) is a standardized framework and runtime environment that forms the foundation of the .NET framework. It is a set of specifications and standards developed by Microsoft to facilitate language interoperability and runtime execution of programs written in various programming languages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Common Language Infrastructure (CLI)** | | | | | |
| **Common Language Runtime (CLR):**  - memory management, - garbage collection,  - Thread management, - security,  - Just-In-Time compiler (JIT) | **Common Type System (CTS):**  It specifies how data types are  - declared,  - used,  - managed | **Common Language Specification (CLS):**  It defines rules for  - method signatures,  - exception handling,  - inheritance,  - naming conventions | **Common Intermediate Language (CIL/IL)** | **Base Class Library (BCL)**  is a collection of class libraries that provide fundamental functionality to .NET applications. | Feature:  **Language Interoperability**  - It allows developers to write code in different languages (e.g., C#, VB.NET, F#) and have them work together within the same application |

**Sealed** classes are used to restrict the users from inheriting the class. A class can be sealed by using the ***sealed*** keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

public sealed class Person

A **static** **constructor** is used to initialize any [static](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) data, or to perform a particular action that needs to be performed only once. It is called automatically before the first instance is created or any static members are referenced. A static constructor will be called at most once.

There are several actions that are part of static initialization. Those actions take place in the following order:

1. *Static fields are set to 0*. This is typically done by the runtime.
2. *Static field initializers run*. The static field initializers in the most derived type run.
3. *Base type static field initializers run*. Static field initializers starting with the direct base through each base type to [System.Object](https://learn.microsoft.com/en-us/dotnet/api/system.object).
4. *Base static constructors run*. Any static constructors, starting with [Object.Object](https://learn.microsoft.com/en-us/dotnet/api/system.object.-ctor) through each base class to the direct base class.
5. *The static constructor runs*. The static constructor for the type runs.

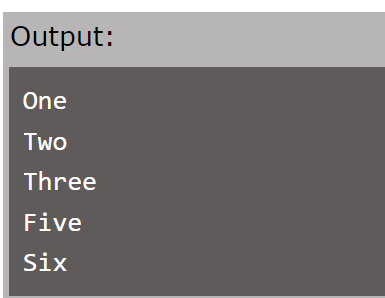
LINQ

**Concatenation**

The Concat() method appends two sequences of the same type and returns a new sequence (collection).

IList<string> collection1 = new List<string>() { "One", "Two", "Three" };

IList<string> collection2 = new List<string>() { "Five", "Six"};

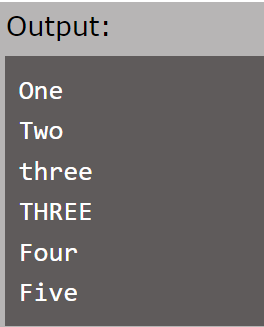
var collection3 = collection1.Concat(collection2);

**Union**

Union() requires two collections and returns a new collection that includes distinct elements from both the collections.

IList<string> strList1 = new List<string>() { "One", "Two", "three", "Four" };

IList<string> strList2 = new List<string>() { "Two", "THREE"};

var result = strList1.Union(strList2);

**Distinct**

Distinct() method is used to obtain the distinct elements from a single collection.

To obtain distinct objects there must be a comparing class

CamparingClass : IEqualityComparer<Item\_mast>

with bool Equals(Item? X, Item? Y) and int GetHashCode([DisallowNull] Item x)

Instead of [DisallowNull] I can just write `string[` if the project is nullable](#_Nullable_(starts_from) – since C# 8.0 (Core 3.x)

[DisallowNull] also provided since C# 8 for non-nullable projects

**Sting.Join()**

String.Join(“, “, collection) concats collection to a single string.

**SelectMany**

Cartesian product of two sets:

char[] charset1 = { 'X', 'Y', 'Z' };

int[] numset1 = { 1, 2, 3, 4 };

charset1.SelectMany(ch => numset1.Select(num => new {ch, num}));

Concatenation into one collection:

   List<List<string>> listOfLists = new List<List<string>>

      {

            new List<string> { "apple", "banana", "cherry" },

            new List<string> { "dog", "cat", "elephant" },

            new List<string> { "red", "green", "blue" }

      };

   var resLOL = listOfLists.SelectMany(list => list);

C# Collections

1. **List<T>:** Represents a dynamic array that can grow or shrink in size. It provides indexed access and supports various methods for adding, removing, and manipulating elements.
2. **Dictionary<TKey, TValue>:** Represents a collection of key-value pairs. It provides fast lookups by key and is commonly used for storing and retrieving data based on unique keys.
3. **HashSet<T>:** Represents an unordered collection of unique elements. It ensures that no duplicate elements are present in the collection.
4. **Queue<T>:** Represents a first-in-first-out (FIFO) collection of objects. It is often used for implementing scenarios where items are processed in the order they were added.
5. **Stack<T>:** Represents a last-in-first-out (LIFO) collection of objects. It is used for scenarios where items are added and removed in a reverse order.
6. **LinkedList<T>:** Represents a doubly linked list of objects. It provides efficient insertion and removal of elements in the middle of the list.
7. **ArrayList:** Represents a dynamic array similar to List<T>, but it is not type-safe and is less efficient.
8. **SortedSet<T>:** Represents a sorted collection of unique elements. It maintains elements in sorted order, which is useful for scenarios that require sorted data.
9. **SortedDictionary<TKey, TValue>:** Represents a collection of key-value pairs sorted by keys. It combines the features of a Dictionary and a SortedList.
10. **Queue and Stack (System.Collections namespace):** These are non-generic collections with the same names as the generic versions mentioned earlier. They are part of the older System.Collections namespace.
11. **Hashtable:** Represents a collection of key-value pairs that are organized based on the hash code of the key. It is not type-safe and is less efficient compared to Dictionary<TKey, TValue>.

**List<T>**

1. Dynamic Sizing

2. Indexed Access, Count() – O(1)

3. Insertion and Removal in any place O(n) (at the end O(1), rewrite on index O(1))

4. Ordered Collection

5. Duplicates Allowed

6. Slower compared to a **Dictionary** or **HashSet** because **List<T>** requires linear search.

**List**, **Stack**, **Queue** are using arrays and double their size every time the array is full, 4 -> 8 -> 16… and do not automatically shrink (if the size is 16, it won’t shrink in case the number of elements decrease to 7) Every resizing is O(n) but its so rarely.

**Use Cases**

* **List<T>** is a good choice when you need an ordered collection of elements that can grow or shrink in size dynamically.
* Use **List<T>** when you need to maintain the order of elements and need to access them by index.
* If you don't need key-value pairs (like in a **Dictionary**) and you need a simple container for elements that maintains order, **List<T>** is suitable.
* **List<T>** is widely used for general-purpose collections, such as managing a list of items in a shopping cart, holding records in a database query result, or managing a collection of objects for processing.

**Dictionary<TKey, TValue>**

1. Key-Value Pairs, unique Key (otherwise - exception)

2. Efficient insertion, removal and lookup by keys O(1) – same time on average,

lookup by value is O(n)

3. Value Types and Reference Types (both keys and values) (do not change Key mutable objects)

4. Unordered // can use **SortedDictionary<TKey, TValue>** to sort by Keys

5. Flexible Keys: Keys can be of any type that implements GetHashCode and Equals methods. Equals – checks are 2 keys equal due to custom parameters (e.g., memory address). GetHashCode – get unique custom hashCode (based on any key-object/struct/.. fields).

6. Elements are stored as KeyValuePair<TKey, TValue> objects.

7. Values can be accessed by passing associated key in the indexer e.g. myDictionary[key]

**Use Cases**

* **Dictionary<TKey, TValue>** is ideal when you need to associate values with unique keys and require fast lookup based on those keys.
* Use a dictionary when you want to implement a mapping or association between keys and values, such as a phone book (names and phone numbers) or a cache (URLs and their corresponding web page contents).
* When you need to count occurrences of items (e.g., word frequency in a text) or perform grouping of items based on a key (e.g., grouping data by category), a dictionary can be helpful.
* It's also useful when dealing with configurations, settings, or mappings between identifiers and objects.

**LinkedList<T>**

1. Dynamic Size
2. Constant-Time Insertions and Deletions O(1) (especially at the beginning or end of the list)
3. No indexes, double reference instead. Can be slower for large lists

**Use Cases**

* **Insertions and Deletions**: When frequent insertions or deletions are required, especially at the beginning or end of the list, a **LinkedList<T>** can provide better performance compared to other collections.
* **Building Complex Data Structures**: Linked lists are used as building blocks for more complex data structures like queues, stacks, and other advanced algorithms.
* **Implementing Undo/Redo functionality**: A linked list can be used to maintain a history of changes and provide undo/redo functionality in applications.
* **Circular Lists**: Linked lists can be used to implement circular structures like a round-robin scheduler.

**SortedDictionary<TKey, TValue>**

1. Uses a balanced binary tree to maintain the sorted order (instead of HashTable in Dictionary)
2. O(log n) time complexity for insertion, removal, and search/lookup operations (instead O(1) and O(n), only Count() is O(1))
3. Sorted by keys

**Use Cases**

Use when you need fast insertion, removal, search operations, but order is important.

1. **Maintaining Leaderboards:** When building applications that require leaderboards, rankings, or top scores, **SortedDictionary** can help keep the scores in order.
2. **Event Scheduling:** In scheduling applications, **SortedDictionary** can help maintain a sorted list of events based on their scheduled time.
3. **Log Entries:** If you need to store log entries with timestamps, a **SortedDictionary** can keep the log entries sorted chronologically.

**SortedList<TKey, Tvalue>**

1. O(n) insertion, removal, and search operations.
2. O(1) Accessing by index SL.GetKey( myIndex ), SL.GetByIndex( myIndex )
3. Internally has 2 arrays: for keys and values.

|  |  |
| --- | --- |
| Sorted data (insertion, removal, search) | Unsorted data |
| Slist faster than Sdictionary | Sdictionary – O(log n) Slist – O(n) |
|  |  |

**ArrayList**

1. Old implementation of List from System.Collections
2. Similar and worst than **List<Object>**, because **List<Object>** more type safe – elements do not need boxing/unboxing ( (int)list[0] )
3. Use **List<T>** instead of **ArrayList**.

**HashSet<T>**

1. Uniqueness, when you try to add an element that already exists, the addition is ignored.
2. Ultrafast insertion, removal, and containment checks(check if the element is in set) O(1), but if there are n elements with the same hashcode, the complexity is O(n) because there are inner linked lists for every hashcode that relative to more than 1 element.
3. No order.

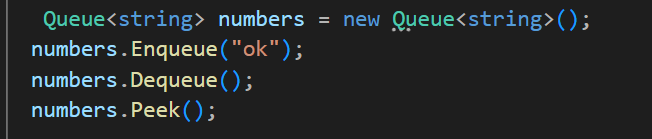
**Use Cases**

Every time you need a collection of distinct elements – HashSet is the best choice!

**SortedSet<T>**

1. Maintains elements in a sorted order based on the natural order of elements or a custom comparer. (balanced tree structure)
2. Insertions, deletions, and lookups – O(log n), always

**Queue<T>**

1. FIFO (First in first out) – as real queue
2. You can only **Enqueue**(add), **Dequeue**(get and remove) and **Peek**(get the oldest element but do not remove it), ContainsI
3. Use only if you want to implement a strict FIFO, otherwise use LinkedList<T>
4. Better adding (at end) and removing (from start) than LinkedList<T>, also O(1)
5. You can’t add to the middle or remove from the middle
6. ContainsI – O(n), Clear() – O(1)

**Stack<T>**

1. LIFO last-in-first-out
2. Push(add), Pop(get and remove), Peek(), Count(), Clear(), ContainsI, foreach
3. Pop, Peek, Push, Count(), Clear(), O(1).
4. ContainsI O(n).
5. Access to any another element except the top one is not a common scenario and so inefficient.

**Use Cases**

1. **Undo/Redo Functionality:** Stacks are commonly used to implement undo and redo functionality in applications. Each action is pushed onto the stack, and the most recent action can be quickly undone or redone by popping elements off the stack.
2. **Expression Evaluation:** Stacks are used in evaluating arithmetic expressions, especially those involving parentheses. Operators and operands are pushed onto the stack, and when an operator is encountered, the necessary operands are popped and the result is pushed back onto the stack.
3. **Memory Management:** Stacks are used in memory management for function calls and local variables in programming languages. When a function is called, a new frame is added to the call stack, and when the function completes, the frame is removed.
4. **Parsing and Syntax Checking:** Stacks are used in parsing and syntax checking to keep track of open and close symbols, such as brackets, parentheses, and tags.

**SortedSet<T>**

**HashSet<T>**

**ArrayList**

**SortedList<Tkey, Tvalue>**

**,SortedDictionary<Tkey, Tvalue>**

**Modifiers**

Internal – public in same assembly, but no access from another assembly (every project has its own assembly)

Protected – access only within base glass or in derived classes.

Protected Internal – Internal, but with access in derived classes in another assemblies.

Default modifier of classes – **internal**

Default modifier of class components – **private**

Default modifier of interface and interface components – **public**

**Variable number of parameters:**

public void AddMarks(params int[] numbers)

{

foreach (var n in numbers)

{

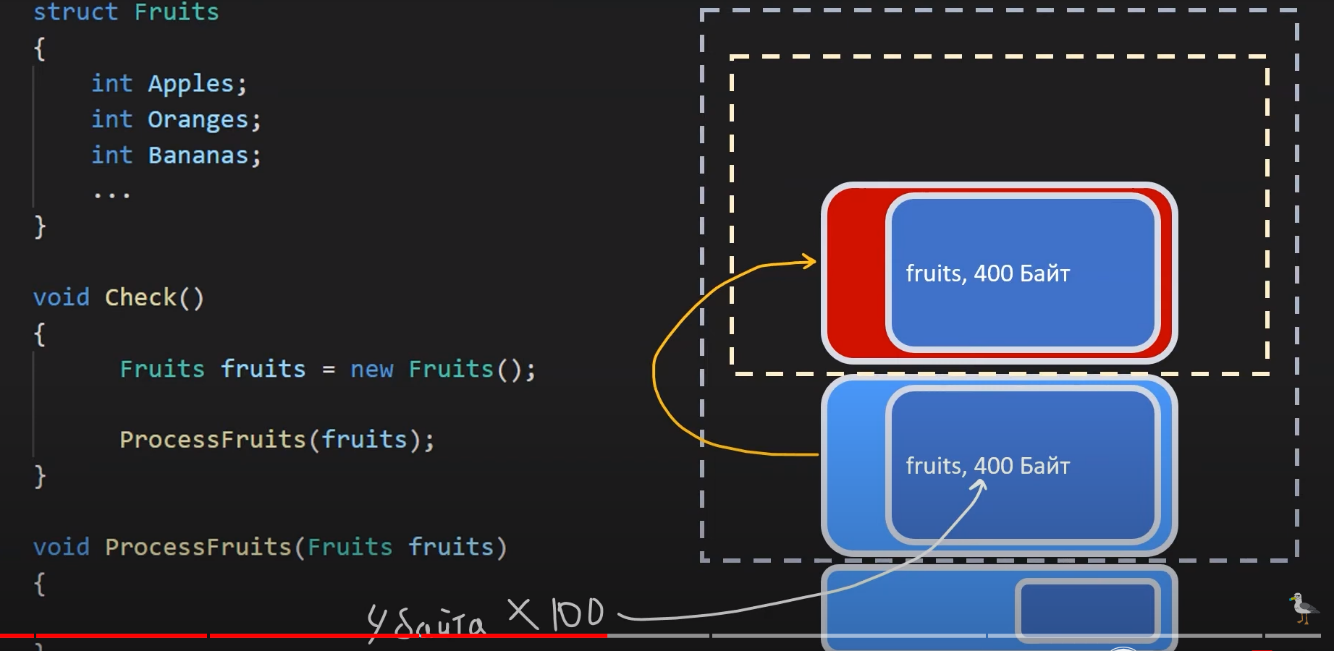
\_marks.Add(n);

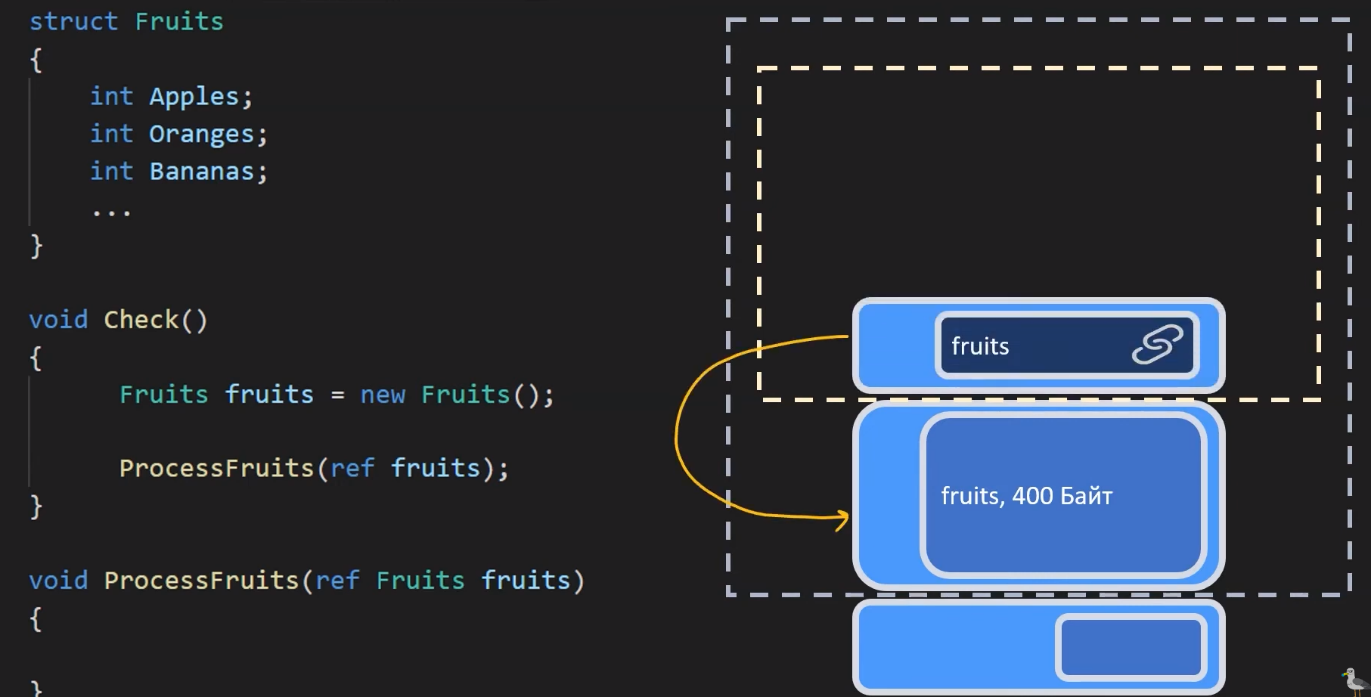
}

}

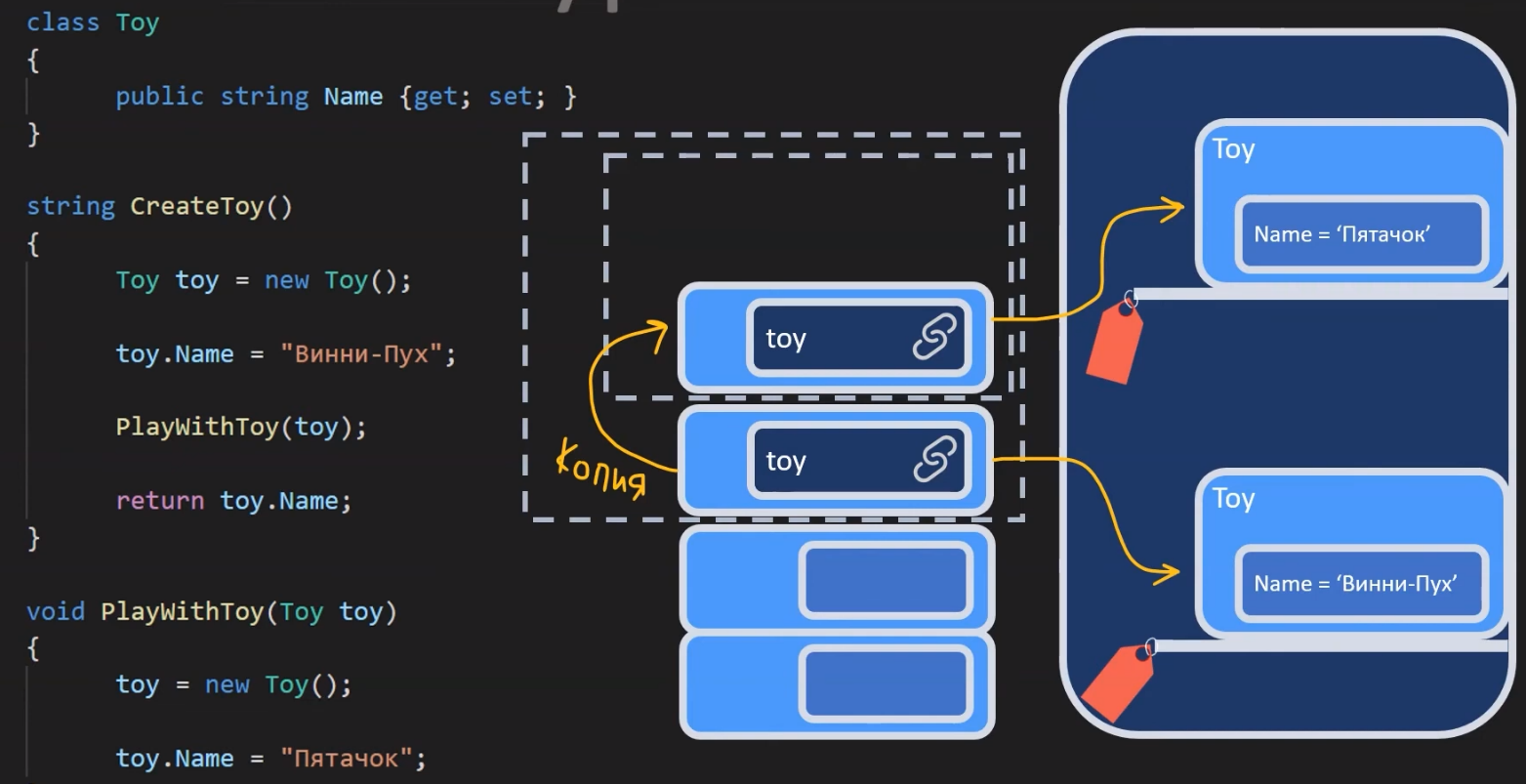
The collection after params must be single dimensional array.

**Heap and Stack**

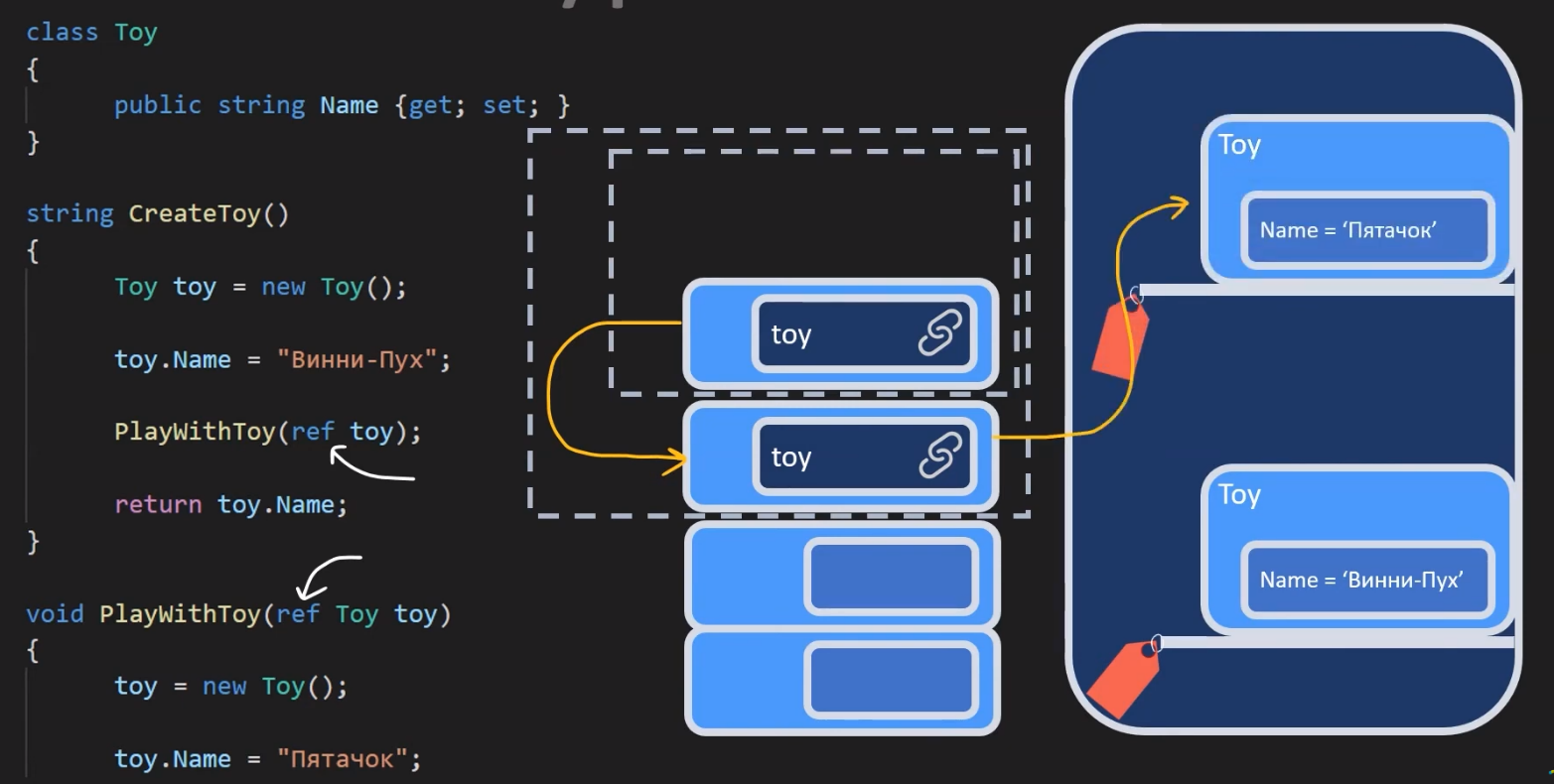


Pasting struct (value type) into new stack frame (method/function) copies the whole struct!

Sending struct reference using ‘ref’ creates in new frame a new reference to the same struct, no copy.



Sending object reference to new frame will copy this reference, if copy reference will change point it won’t affect to first reference and first object.

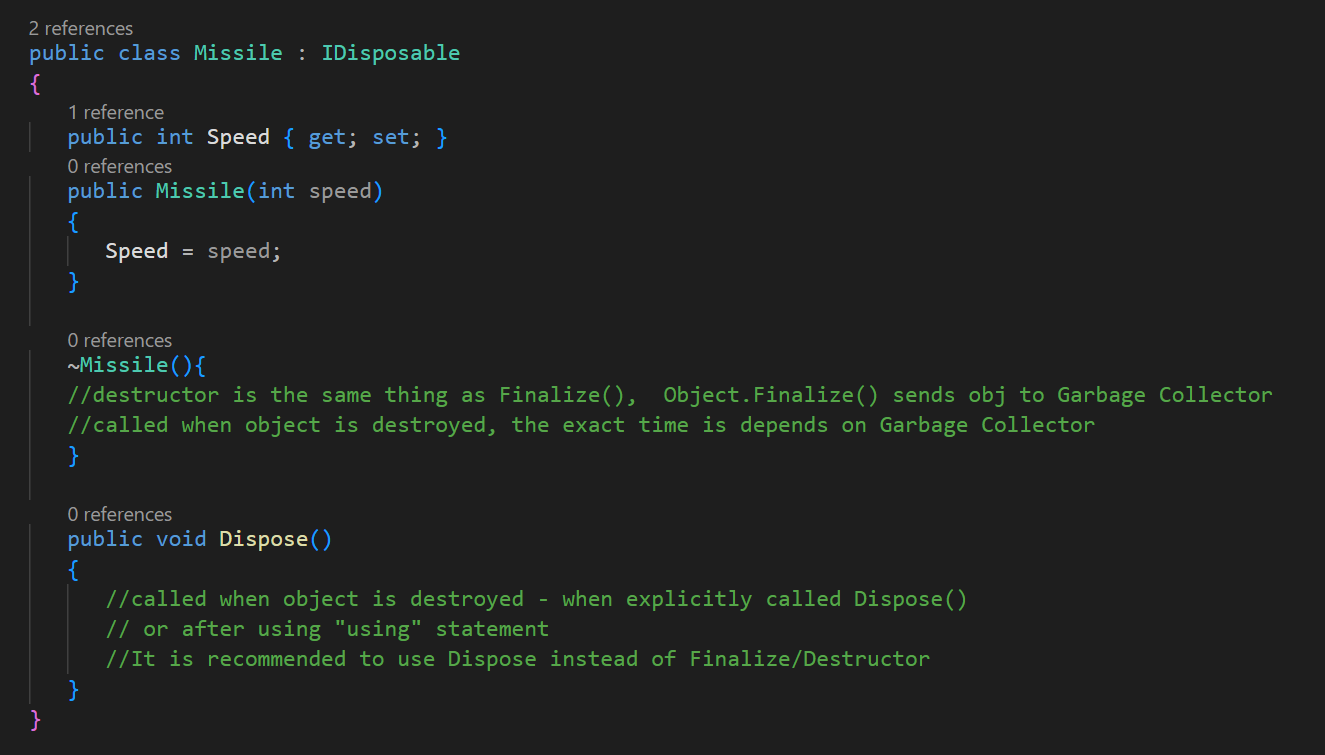
If I want to work with first reference, I need to send it with ‘ref’ keyword:

Program data is stored in Stack and Heap, that represents RAM memory.

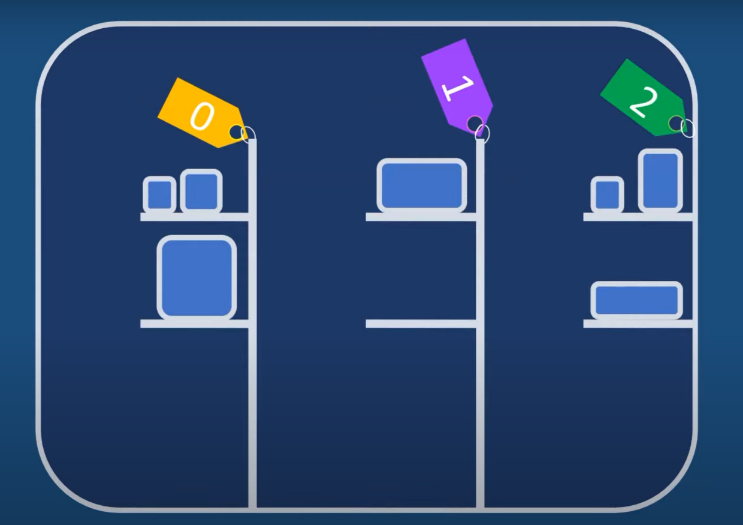
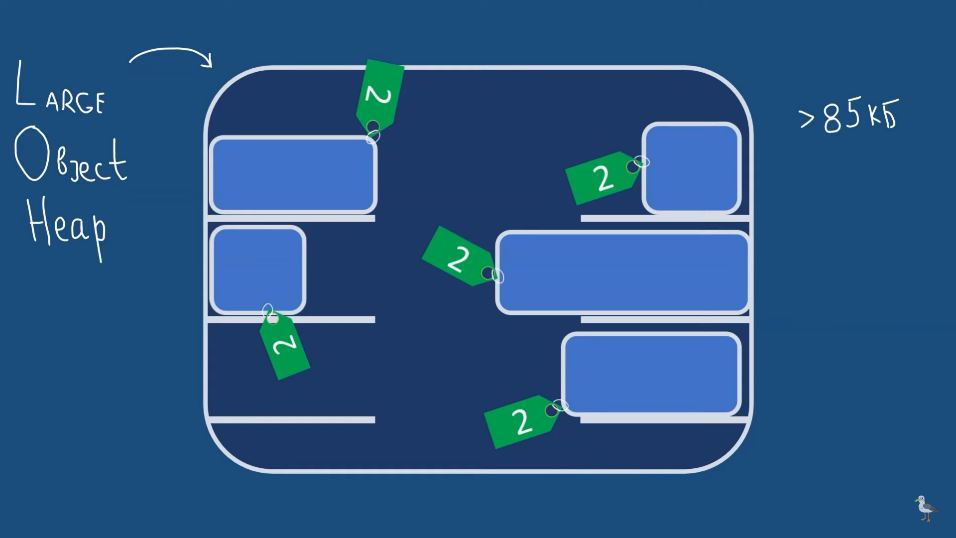
* + Stack size is 1 Mb, but I can configure another size. Too many elements -> StackOverflowException.
  + Every thread has his own stack (with 1 Mb memory by default).
  + Value types are stored in Stack (float, int, struct, enum, bool …)
  + Stack is faster than Heap, but be careful with coping large values.

+ Heap is limited only by physical RAM size.

+ Every thread has access to one global Heap.



Garbage collector: 3 generations



Regex

In C#, the System.Text.RegularExpressions namespace includes the Regex Class that provides several methods for working with regular expressions.

| Regex Method | Description |
| --- | --- |
| IsMatch(string input) | Returns true if the pattern matches the input string. |
| Match(string input) | Returns the first occurrence of the pattern in the input string. |
| Matches(string input) | Returns all occurrences of the pattern in the input string. |
| Replace(string input, string replacement) | Replaces all occurrences of the pattern in the input string with the replacement string. |
| Split(String) | Splits an input string into an array of substrings at the positions defined by a regular expression pattern. |

* ^: Indicates the start of a line.
* $: Indicates the end of a line.
* .: Matches any character except a newline.
* \*: Matches the preceding element 0 or more times.
* +: Matches the preceding element 1 or more times.
* {n}: Matches exactly n occurrences of the preceding element.
* [abc]: Matches any one of the characters a, b, or c.
* (abc): Groups the expression inside the parentheses and treats them as a single element.

**Iqueryable**

        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

        // Create an Iqueryable from the list

        Iqueryable<int> queryableData = numbers.AsQueryable();

        // Define a query using Iqueryable

        Iqueryable<int> query = queryableData.Where(n => n % 2 == 0);

        // Enumeration happens here, and the query is executed.

        Foreach (int result in query)

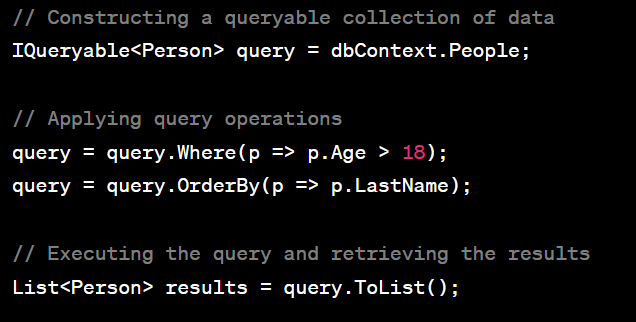
        {

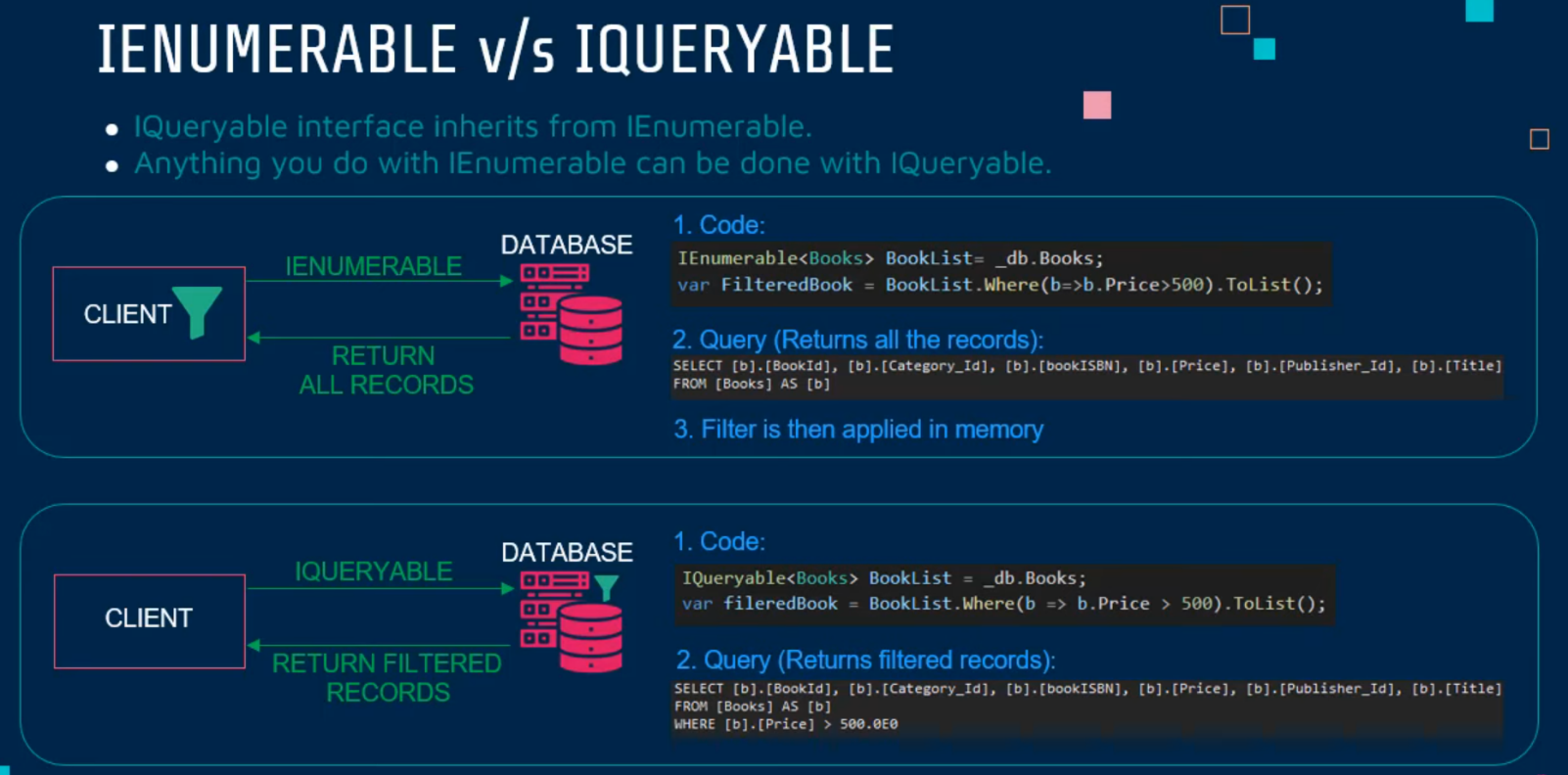
            Console.WriteLine(result);

        }

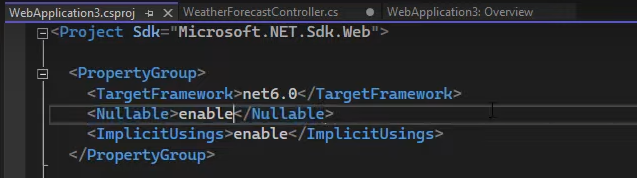
LINQ returns Ienumerable after operation over collections like List, you can proceed .AsQueryable to convert into Iqueryable

Entity Framework returns Iqueryable. Iqueryable is executed only when you begin to iterate over objects e.g. use .ToList() method or foreach.

You can assemble you Iqueryable from parts. During the execution the EF will optimize and convert the Iqueryable to SQL statement and execute it within database.



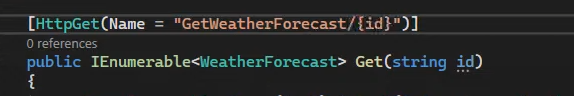
## Nullable (starts from c# 8.0 – core 3.x)

Starts from .net 6 you can turn on nullable mode in .csproj

Or disable(remove/disable) it.

If it enabled, you should explicitly mark what fields may be null (even strings)

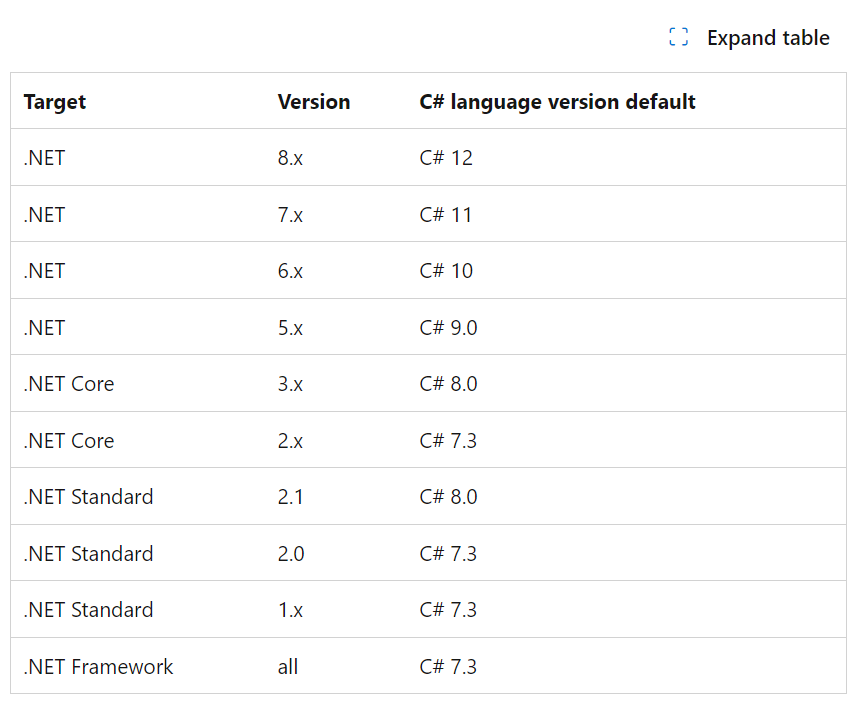
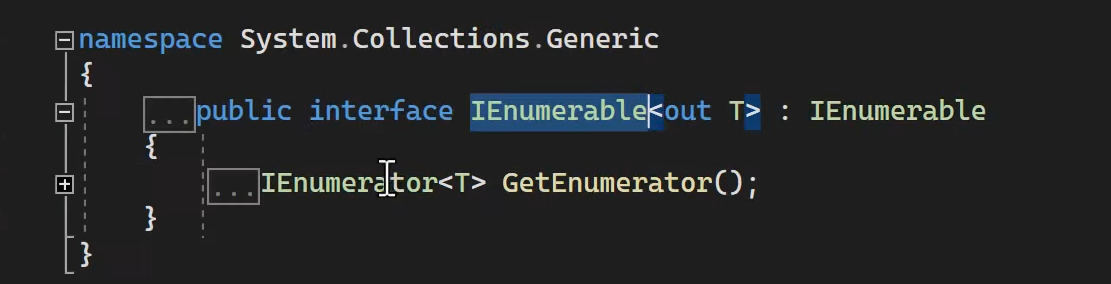
If project is nullable, the not nullable string requires not nullable string, otherwise – error:

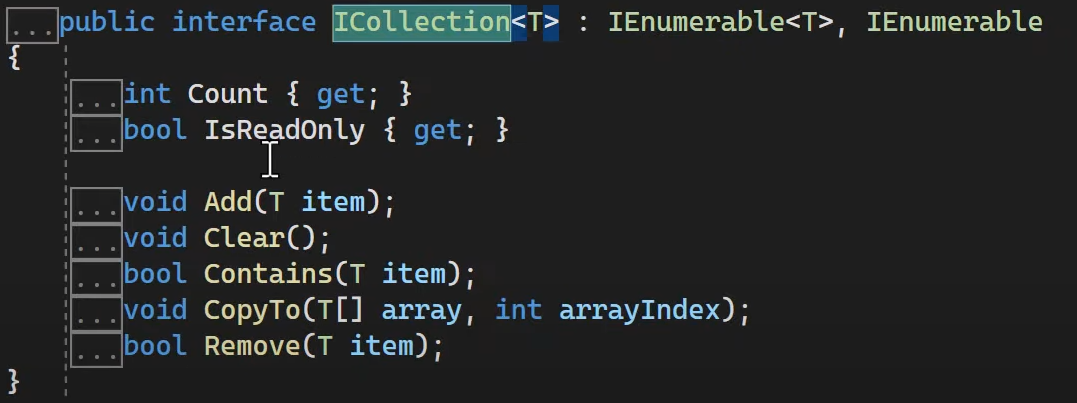
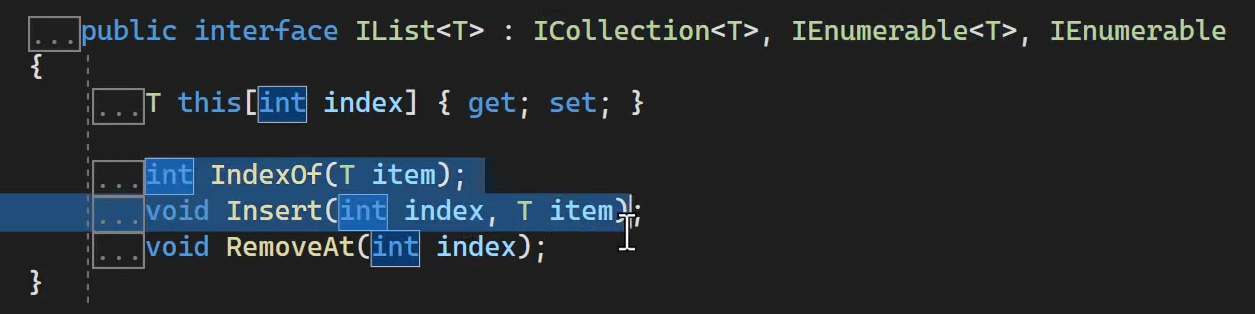
{id} is obligatory

But in case of non-nullable project it can be empty:

/api/GetWeatherForcast/



IEnumerable is only for iterating(reading). Deferred Execution – executes when you e.g. call ToList().

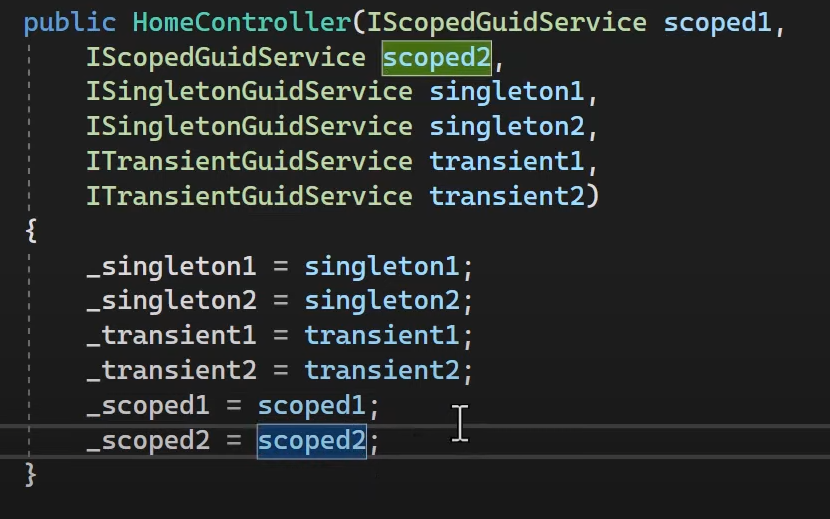


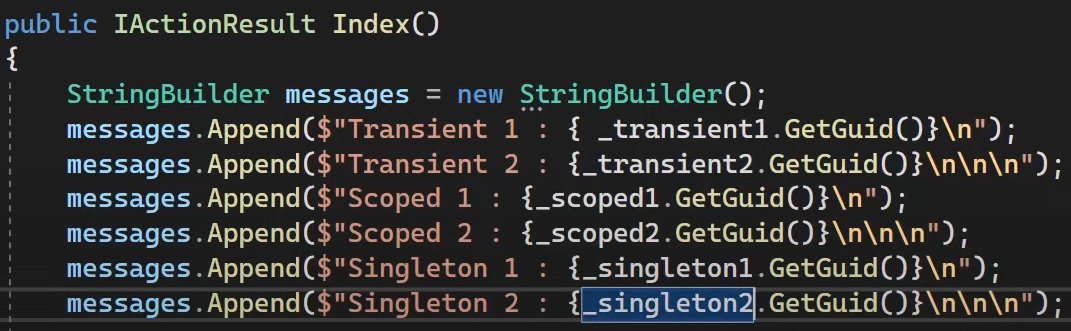
# Dependency injection

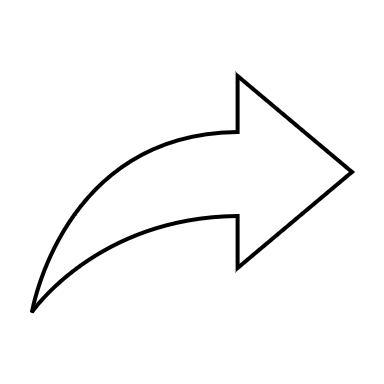
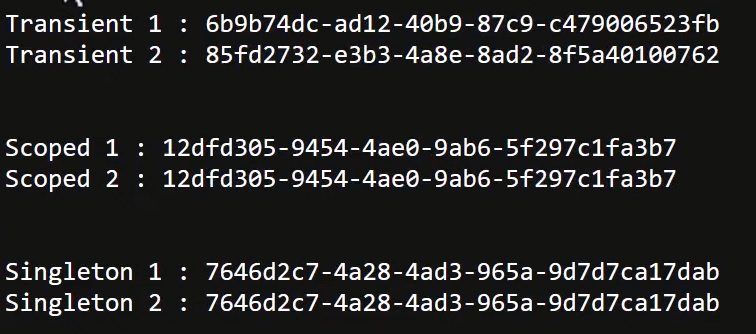
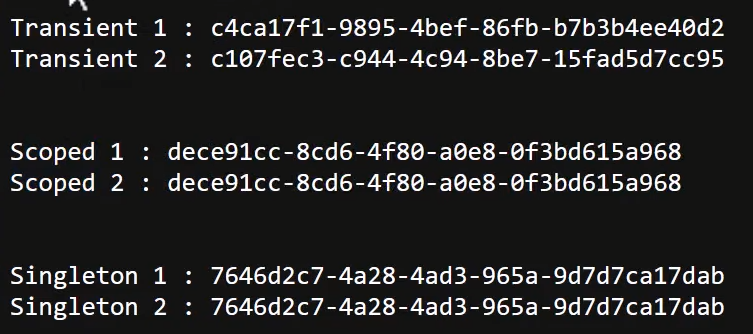
Service lifetimes

**Transient** – new instance for every service call. The simplest and most safety service lifetime.

**Scoped** – new instance for every http request. One instance within one http request.

**Singleton** – one instance for the lifetime of application. (One instance for every start of app)





Page refresh (new http request)

# School Register

