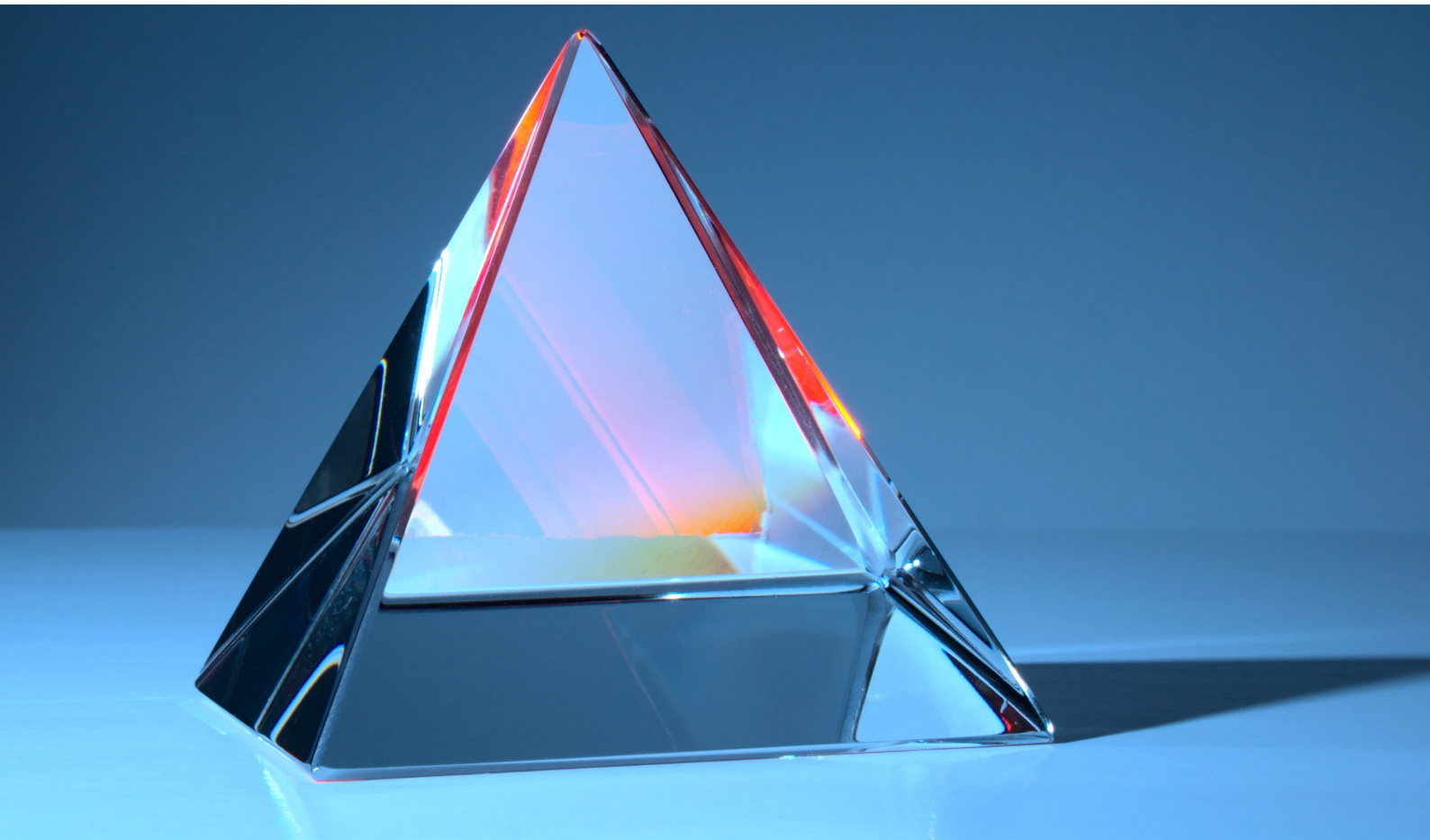
Krystyna Ślusarczyk

C#/.NET

50 ESSENTIAL INTERVIEW QUESTIONS

Mid Level



HELLO!

This e-book is a part of my course "C#/.NET - 50 Essential Interview Questions (**Mid Level**)".

https://bit.ly/3sC7FsW

You can find every lecture from the course here.

Remember that an e-book with 15 essential **Junior**-Level lectures is also provided with the course.

You can also check out full course "C#/.NET - 50 Essential Interview Questions (**Junior Level**)" which you can find under this link:

https://bit.ly/3hSRpOq

INTRODUCTION

Hello, I'm Krystyna! I'm a programmer who loves to write elegant code.
I've been working as a software developer since 2013. About half of this time I've been engaged in teaching programming.
I believe that with a proper explanation, everyone can understand even the most advanced topics related to programming.
I hope I can show you how much fun programming can be, and that you will enjoy it as much as I do!

CONTENTS

16.What is reflection?
17.What are attributes?
18.What is serialization?
19.What is pattern matching?
20.How does the binary number system work?
21.What is the purpose of the “checked” keyword?
22.What is the difference between double and decimal?
23.What is an Array?
24.What is a List?
25.What is an ArrayList?
26.What is the purpose of the GetHashCode method?
27.What is a Dictionary?
28.What are indexers?
29.What is caching?
30.What are immutable types and what’s their purpose?
31.What are records and record structs?
32.Why does string behave like a value type even though it is a reference type?
33.What is the difference between string and StringBuilder?
34.What is operator overloading?
35.What are anonymous types?
36.What is cohesion?
37. What is coupling?

38.What is the Strategy design pattern?
39.What is the Dependency Injection design pattern?
40.What is the Template Method design pattern?
41.What is the Decorator design pattern?
42.What is the Observer design pattern?
43.What are events?
44.What is Inversion of Control?
45.What is the “composition over inheritance” principle?
46.What are mocks?
47.What are NuGet packages?
48.What is the difference between Debug and Release builds?
49.What are preprocessor directives?
50.What are nullable reference types?

1. What is the difference between Tuples and ValueTuples?

Brief summary: The differences between tuples and ValueTuples are that tuples are reference types and ValueTuples are value types. Also, ValueTuples fields can be named, while with tuples we are stuck with properties named Item1, Item2, etc. Also, tuples are immutable while ValueTuples are mutable.

Before we dive into understanding the difference between System.Tuple and System.ValueTuple, let's make sure we understand what tuples are on a conceptual level. Tuples are small data structures used to bundle a couple of pieces of information together.

This can be useful when, for example, I want to create a method that needs to return two pieces of information:

```
int GetSummaryAboutCollection(IEnumerable<int> values)
{
    var sum = values.Sum();
    var count = values.Count();
    return sum; return count;
}
```

The problem is, I can't return two values from a function. If I want to do it, I can declare a special type that will bundle sum and count together:

```
SumAndCount GetSummaryAboutCollection(IEnumerable<int> values)
{
    var sum = values.Sum();
    var count = values.Count();
    return new SumAndCount { Sum = sum, Count = count };
}

2 references
struct SumAndCount
{
    public int Sum;
    public int Count;
}
```

I defined a dedicated struct to represent the data I need to return from the method, but this is a bit awkward. I will probably never use this type again, and it only exists so I can return two values from some method.

For such situations, tuples are the perfect solution. Tuple is a data structure that bundles some data together. Let's use it to make our code better:

```
Tuple<int,int> GetSummaryAboutCollection(IEnumerable<int> values)
{
    var sum = values.Sum();
    var count = values.Count();
    return new Tuple<int, int>(sum, count);
}
```

This way I can remove this awkward SumAndCount type from my program.

Please note that tuples are not exclusive to C# and many programming languages provide a similar mechanism.

In C#, we have two kinds of tuples: regular Tuples and ValueTuples. Before we move on, let me clarify one thing. From now on, when I will be using the word "tuple" I will be meaning the System.Tuple type. When I will have ValueTuple in mind, I will use its full name.

All right. We already have seen a simple tuple in action. Let's see another example, but this time we will create the tuple object with the Tuple.Create method:

```
var tuple1 = Tuple.Create(1, "aaa");
```

This is a tuple object holding two pieces of information in it - an int and a string. I can access them by using the Item1 and Item2 properties:

```
var one = tuple1.Item1;
var aaa = tuple1.Item2;
```

The Tuple.Create method is an alternative for using the tuple's constructor. It's more convenient because, unlike the constructor, it doesn't require providing the type parameters, as it infers them from the context.

```
var tuple3 = new Tuple<int, string>(3, "ccc");
```

Please be aware that we can hold more than 2 elements in a tuple. The maximum is 8, so calling the Create method or the constructor with more than 8 parameters will not compile:

```
var hugeTuple = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

At the end of this lecture, we will talk about how to bypass this limitation.

All right. Moving on to the ValueTuples. On the conceptual level, they serve the same purpose, so they bundle a couple of values together. I could create a ValueTuple with a constructor call…

```
var valueTuple = new ValueTuple<int, string>(3, "ccc");
```

…but there is a much simpler way to do so:

```
var valueTuple1 = (2, "bbb");
```

As you can see the construction of a ValueTuple looks much nicer than the creation of a tuple.

Let's discuss other differences between tuples and ValueTuples.

First of all, tuple is a reference type while ValueTuple is a value type. This has a lot of implications, for example, tuples are compared by reference while ValueTuples

are compared by value. This code will print False because both tuples are different objects with different references. The == operator compares the references and sees that they are not equal:

```
var tuple1 = Tuple.Create(1, "aaa");
var tuple2 = Tuple.Create(1, "aaa");
Console.WriteLine("tuple1 == tuple2: " + (tuple1 == tuple2));
```

On the other hand, this will print True because ValueTuple is a value type. There are no references to be compared, and the == operator simply checks if the fields in both tuples have the same value:

```
var valueTuple1 = (2, "bbb");
var valueTuple2 = (2, "bbb");
Console.WriteLine("valueTuple1 == valueTuple2: " +
    (valueTuple1 == valueTuple2));
```

Please note that we can compare tuples by value if we need to. To do so, we can use the Equals method which Tuple overrides. That's why this will print True:

```
Console.WriteLine("tuple1.Equals(tuple2) " + tuple1.Equals(tuple2));
```

Because tuple overrides both Equals and GetHashCode, we can safely use them as Dictionary keys. We will talk more about Dictionaries later in the course.

Another implication of the fact that tuples are reference types and ValueTuples are value types is that when used as parameters, are passed by copy while tuples are passed by reference.

The fact that tuples are reference types can have negative performance implications. Tuples are usually short-lived objects, and if we create a lot of them, the process of allocating and freeing the memory might take considerable time. This was one of the reasons why ValueTuples were created.

The next difference is that tuples are immutable, and ValueTuples are mutable. If an object is immutable it means, it cannot be modified once it has been created. We will learn more about immutable types later in the course.

Tuples are immutable, so this code will not compile:

```
tuple1.Item1 = 5;
```

But this will work fine:

```
valueTuple1.Item1 = 5;
```

The difference that probably matters most for us as the developers are that ValueTuples provide a couple of interesting features that make our work much easier. The first one is that we can name the fields of the ValueTuples as we like, and we don't need to use those awkward "Item1", "Item2" names. However, we still can, if we want to:

```
var valueTuple3 = (number: 2, text: "ccc");
var numberFromValueTuple = valueTuple3.number;
//...but Item1 and Item2 still works:
var numberFromValueTuple1 = valueTuple1.Item1;
```

This matters most when the ValueTuple is a result of some calculation, and we don't see what the values are in the current scope. For example, this code is pretty confusing, and to understand it we would need to look into the SumCollectionTuple method.

```
var result1 = SumCollectionTuple(
    new int[] { 1, 2, 3, 1, 4, 1 });
Console.WriteLine(
    $"Calculation result is: {result1.Item1}, {result1.Item2}");
```

If you are curious, this is what this method does:

```
Tuple<int, int> SumCollectionTuple(IEnumerable<int> values)
{
    return Tuple.Create(values.Sum(), values.Count());
}
```

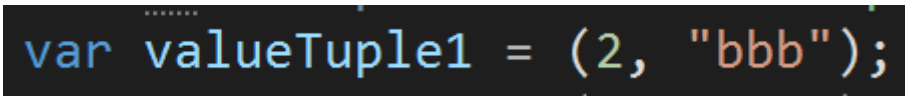
With ValueTuples, it is quite clear without even looking into the method:

```
var result2 = SumCollectionValueTuple(
    new int[] { 1, 2, 3, 1, 4, 1 });
Console.WriteLine(
    $"Calculation result is: {result2.sum}, {result2.count}");
```

And this is the method using ValueTuples:

```
(int sum, int count) SumCollectionValueTuple(IEnumerable<int> values)
{
    return (values.Sum(), values.Count());
}
```

Also, as we have already seen before, ValueTuples have special syntax for construction, which is much more convenient than the Create method or constructor call for tuples:



Another difference is that data members of ValueTuple are fields. Data members of tuple types are properties.

The last difference is that we can create ValueTuples with more than 8 elements:

```
var hugeValueTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
var last = hugeValueTuple.Item12;
```

This is quite interesting. If we looked at ValueTuple source code (you can see it here https://github.com/dotnet/roslyn/blob/main/src/Compilers/Test/Resources/Core/NetFX/ValueTuple/ValueTuple.cs, scroll down to line 1929) we will see that the ValueTuple that was created with more than 8 constructor parameters still has 8 fields only. The last one, called **Rest**, will hold the eighth, ninth, and all other elements. So it's basically a nested ValueTuple, in which the last element is also a ValueTuple. What is interesting is that in the example above we can use the field Item12. How is it possible if no such field is present in the source code? Well, it's a trick of the compiler. When calling "hugeValueTuple.Item12" the compiler actually calls "hugeValueTuple.Rest.Item5".

We can also create a regular tuple with this Rest property, but we must do it by hand - no compiler magic happens there for us.

All right. Let’s summarize the differences between tuples and ValueTuples:

- tuples are reference types, ValueTuples are value types. When a lot of short-lived tuples are created, it may decrease the performance of the application as the memory management for reference types is more demanding than for value types
- tuples are immutable, ValueTuples are mutable
- ValueTuples provide a convenient syntax for the creation
- In tuples, all properties are named Item1, Item2, etc. ValueTuples can have named fields.
- tuples have properties, ValueTuples have fields
- ValueTuples can easily be declared with more than 8 elements, and the compiler will translate them to ValueTuples with 8 elements with the last element being the “Rest” field, holding the excess elements. We can do the same with tuples but we must set the Rest property by hand

Bonus questions:

- "**Is it possible to have a tuple with more than 8 elements?**"
  Tuples are limited to hold up to 8 elements, however, we can bypass this limitation by storing the excessive data in the last property called (for example) “Rest” which is also a tuple, making our tuple nested. This is quite awkward for tuples, but for ValueTuples we get some help from the compiler - it allows us to use the tuple like it really contained more than 8 elements, for example by using Item12 field. Behind the scenes, the compiler will change this to the usage of tuple.Rest.Item5.

2. What is the difference between "is" and "as" keywords?

Brief summary: The "is" keyword checks if the object is of a given type. It returns a boolean result. The "as" keyword casts an object to a given type (it's applicable only to casting to reference types or nullable types).

The "is" keyword is used to check if an object is of a given type. It returns a boolean as a result.

```
object text = "Hello!";
bool isString = text is string;
```

In this case, the result will be true, because the **text** variable is a string.

We can use the "is" with value types and reference types as well:

```
object text = "Hello!";
bool isInt = text is int;
```

In this case, the result will be false, because the **text** variable is not an int.

The **is** keyword is most often used to ensure that a type can be safely cast to some other type. We can also have some business logic driven by the type of some variable - in the case of type A we want to execute different logic than in the case of type B.

The "as" keyword is used to cast a variable to a given type:

```
object text = "Hello!";
string textAsString = text as string;
```

Here I'm casting an object to a string. The cast will be successful, and the result will be the string "Hello!".

If the cast would not be successful, the result would be **null**.

```
List<int> list = text as List<int>;
```

In this case, the **list** variable will be null, because it's not possible to cast the **text** variable to a List<int>.

The fact that in the case of invalid casting the result will be null is the crucial difference between casting with "as" and regular casting with braces. Regular casting **throws an exception** when the cast fails. Let's consider the following code:

```
List<int> list2 = (List<int>)text;
```

In this case, an InvalidCastException will be thrown.

Because casting with "as" can return null, it can only be used with nullable types - so any reference types plus nullable value types. It's not valid with non-nullable value types:

```
object text = "Hello!";
int textAsInt = text as int;
```

This doesn't work, because, in case of invalid casting, we would try to assign null to non-nullable value type.

Let's summarize:

- The "**is**" keyword checks if the object is of a given type. It returns a boolean result.
- The "**as**" keyword casts an object to a given type. It can only be used for casting to a reference type or nullable value type.

What is the difference between classic cast and casting with "as"?

- Casting with "as" can be only used for casting to reference types or nullable types. It is because **when the cast will not succeed the result will be null** - so the type we cast to must be nullable. For example, an integer can't be null, so you can't use casting with "as" to cast to an int.
- Regular casting with parenthesis can be used to cast to all types. If the cast **will not succeed the InvalidCastException will be thrown**.

Bonus questions:

- "**What is the difference between regular casting and casting with "as" keyword?**"
  When casting with "as" fails, it will return null. When regular casting fails, an InvalidCastException will be thrown.

- "**Why can we only use the "as" keyword to cast objects to nullable types?**"
  Because if casting with "as" fails, null will be returned. Null can only be assigned to nullable types.

3. What is the use of the “using” keyword?

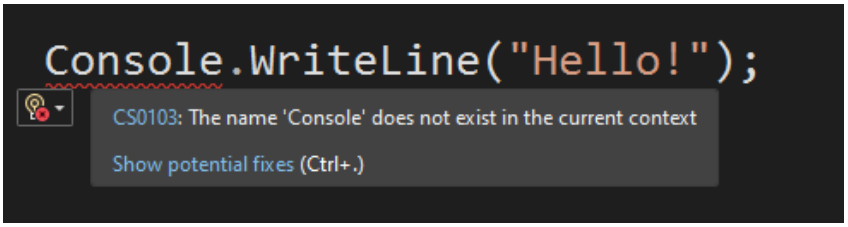
Brief summary: The “using” keyword has two main uses: the using directive, which allows using types from other namespaces and to create aliases for namespaces, and the using statement that defines the scope in which the IDisposable object will be used, and that will be disposed at the scope's end.

The “using” keyword has two main uses:

1) **The using directive**
   You are probably very familiar with code like this:

```
using System.IO;
using System.Threading;
using System;
```

   Those are **using directives**. They allow us to use types from the listed namespaces. For example, if I don’t import the System namespace, I won’t be able to use Console.WriteLine method:



   Please note that I’m still able to use this method if I specify the full type name:

```
System.Console.WriteLine("Hello!");
```

   Nevertheless, usually, the full type names are quite long and they obscure the true meaning of the code, so it’s usually better to import the namespaces we intend to use with the using directive.

The other use of the using directive is to create **aliases** for some types names. This is particularly useful when we have conflicting type names and we want to use them both in a single file. Let's say we have two classes named Person:

```
namespace DTOs
{
    2 references
    internal class Person
    {
        [JsonPropertyName("name")]
        0 references
        public string Name { get; set;  }
    }
}
```

```
namespace DomainObjects
{
    2 references
    internal class Person
    {
        0 references
        public string Name { get; }
    }
}
```

They look very similar, but nevertheless, they are two different types. If I simply imported both the namespaces in a single file and tried to create an object of the Person class, the compiler wouldn't know which one I mean:

```
using DTOs;
using DomainObjects;
var person = new Person();
```

CS0104: 'Person' is an ambiguous reference between 'DTOs.Person' and 'DomainObjects.Person'

Show potential fixes (Ctrl+.)

To solve this, I can create type aliases with the using directive:

```
using PersonDTO = DTOs.Person;
using PersonDomain = DomainObjects.Person;
var personDTO = new PersonDTO();
var domainPerson = new PersonDomain();
```

Now I can refer to the Person type from DTOs namespace by its alias PersonDTO.

There is also something called **the using static directive**. It is particularly useful when in a file we use a lot of static methods from a particular type. For example, in this code I use the Console type a lot:

```
Console.WriteLine("Hello!");
Console.WriteLine("How");
Console.WriteLine("are");
Console.WriteLine("you?");
Console.ReadLine();
```

I could shorten this code by importing all static methods from the Console class with the using static directive:

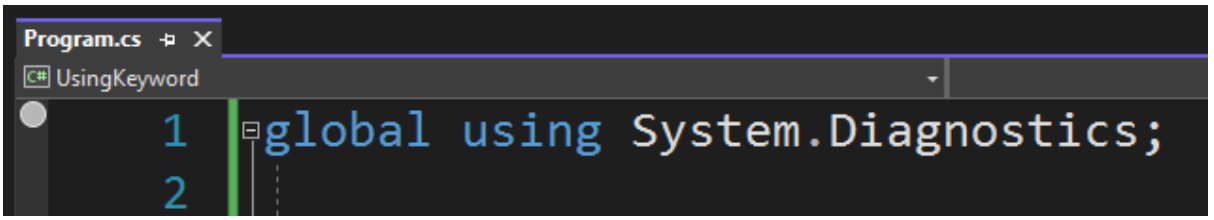
```
using static System.Console;
```

Now, I can skip the “Console.” in my code:

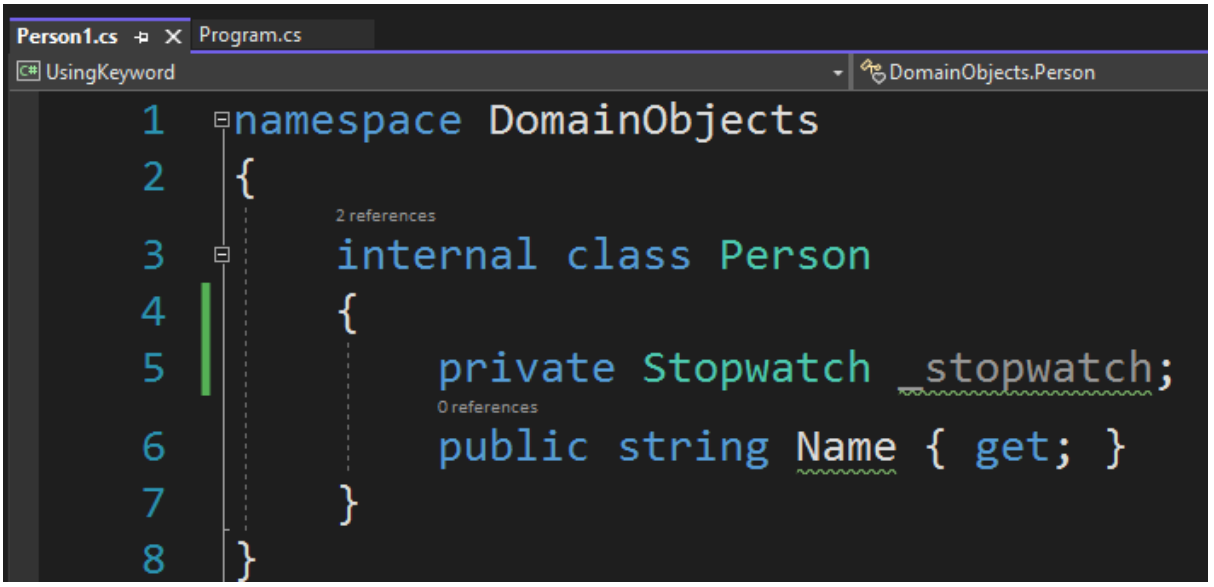
```
WriteLine("Hello!");
WriteLine("How");
WriteLine("are");
WriteLine("you?");
ReadLine();
```

The last thing worth mentioning is **the global using directive**. This feature is available starting with C# 10. When a type is imported in any file with this directive, it is like it was imported in all files in the project.

I'm going to globally import the System.Diagnostics namespace in the Program file:

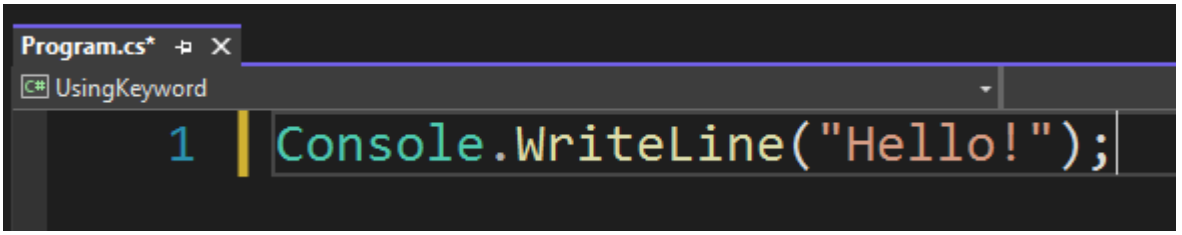


Now, in some other files, I can use types from this namespace as they were imported there too.

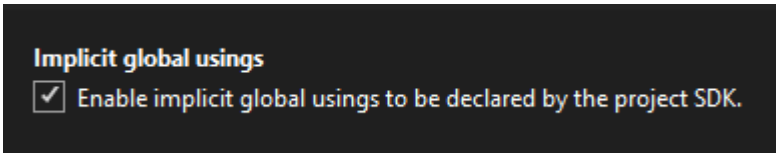


As you can see the System.Diagnostics namespace seems to not be imported here, but I can still declare a field of type Stopwatch, coming from this namespace. This is because it was globally imported in the Program file.

When creating new projects in Visual Studio 2022, you will notice something interesting:

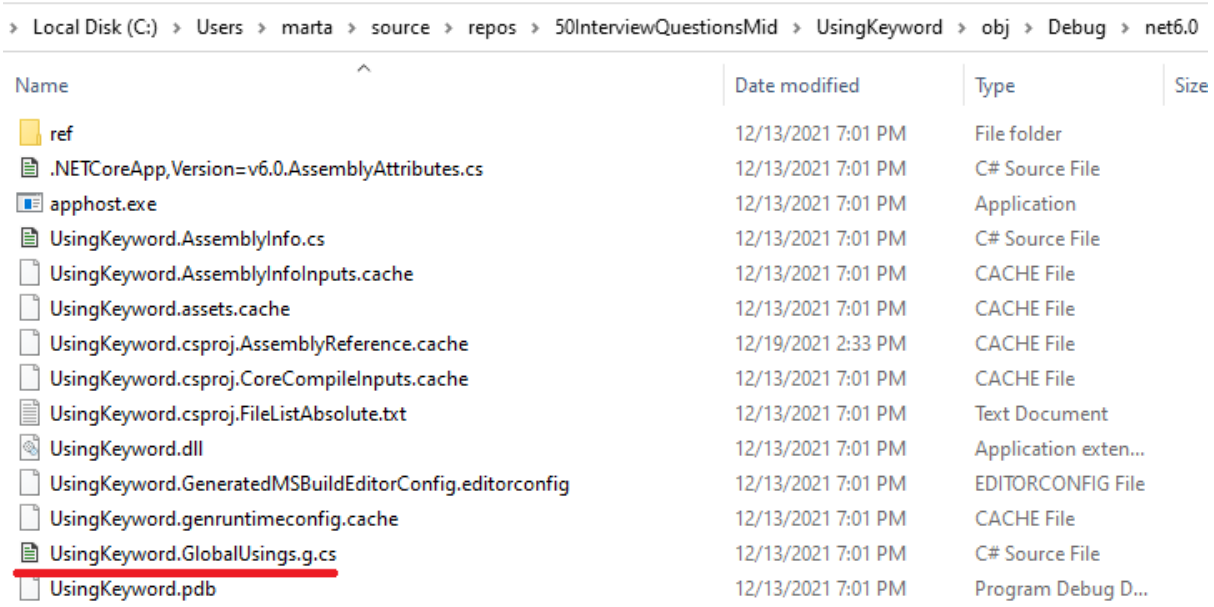


In this file, I can use the Console class even if the System namespace is not explicitly imported - I don't have "using System;" at the top of the file. How is it possible?
Let's take a look at the project settings. There is an interesting entry there:



This setting means that global usings will be defined for a couple of the most commonly used namespaces from C#, for example System or System.Linq.

To find where those usings are actually defined, we must build the project and go to the directory where it exists, and then to obj/Debug/net6.0 folder. There we will find an auto-generated file:



If we open this file, this is what we will see:

UsingKeyword.GlobalUsings.g.cs

```
1  // <auto-generated/>
2  global using global::System;
3  global using global::System.Collections.Generic;
4  global using global::System.IO;
5  global using global::System.Linq;
6  global using global::System.Net.Http;
7  global using global::System.Threading;
8  global using global::System.Threading.Tasks;
9
```

Here are the global usings generated when building the project. Remember, this is only available starting with C# 10.

2) **The using statement**

The second use of the "using" keyword is the using statement. It is used to define the scope in which the IDisposable object will be used, and that will be disposed of at the scope's end. We will learn more about the Dispose method in the "What is the difference between Dispose and Finalize methods?"

In simple terms, instead of writing this:

```
var fileStream = File.Open("some path", FileMode.Open);
try
{
    //some operations
}
finally
{
    fileStream.Dispose();
}
```

We can write this:

```
using(var fileStream = File.Open("some path", FileMode.Open))
{
    //some operations
}
```

Logically this code is the same, but we don't need to remember about calling the Dispose method and making sure the "finally" clause is there, and

because of that the code is much shorter and there is a smaller chance of making any mistake.

Starting with C# 8, we can write this code like this to reduce nesting:

```
using var fileStream3 = File.Open("some path", FileMode.Open);
//some operations
```

Bonus questions:

- "**What are the global using directives?**"
  When a type is imported in any file with the global using directive, it is like it was imported in all files in the project. This is convenient when some namespace (like, for example, System.Linq) is used in almost every file in the project.

4. What is the purpose of the “dynamic” keyword?

Brief summary: The “dynamic” keyword allows us to bypass static type checking that is done by default by the C# compiler. We can call any operations on dynamic variables and the code will still compile. Whether the operation is available in this object or not will only be checked at runtime. The “dynamic” keyword is most useful when working with types unknown in our codebase, like types being the result of dynamically-typed languages scripts or COM objects.

Before we understand the meaning of the “dynamic” keyword, we must first understand the difference between static and dynamic typing.

C# is a **statically-typed** and **strongly-typed** programming language.

The opposite of statically-typed is dynamically-typed. Let’s see the difference.

In Python, which is a **dynamically-typed** programming language, I can do something like this:

main.py

```
1  def toUppercase(text):
2      return "Text to uppercase: " + text.upper();
3
4  hello = "Hello"
5  print(toUppercase(hello))
6
7  hello = 10
8  print(toUppercase(hello))
```

In this short script, two things happen that could never happen in C#:

1) We change the type of the **hello** variable during the program execution - at first, it is a string, then it becomes an int.
2) The parameter of the **toUppercase** method does not have the type defined, even if the body of the method strongly indicates that it should be a string. From the compiler point of view, it is perfectly fine to first call this method with the string parameter, and then with the int parameter.

Let's see the result of this program:

```
Text to uppercase: HELLO
Traceback (most recent call last):
  File "<string>", line 8, in <module>
File "<string>", line 2, in toUppercase
AttributeError: 'int' object has no attribute 'upper'
>
```

First of all, the program compiled correctly, which would not happen in C#. In C# we would have to declare the type of the parameter of the **toUppercase** method. We would set it to string, and then, if we tried to call this method with an int parameter, the program would not compile.

As we can see in the first line of the output, the method call with the "Hello" parameter worked correctly and the result was printed to the console. But the second call caused a **runtime error**.

And this is the essence of the difference between statically-typed and dynamically-typed programming languages. In **statically-typed** languages like C#, all type checks happen at **compilation time**. In **dynamically-typed** languages like Python, type checks happen at **runtime**.

There is no universal answer to which is better. Let's see some advantages and disadvantages of both of those typing methods:

Static typing	Dynamic typing
Fewer runtime errors	Risk of runtime errors
No code to handle type mismatch required (also: no tests are needed to test this code)	The necessity to handle runtime errors (and adding tests to test this handling)
Ease of understanding what needs to be passed to a method so it works correctly	Confusion about what object exactly needs to be used as parameter
The necessity to always be aware of the variable type	Ability to pass variables around without worrying about the type
Language is more rigid	Language is elastic

Type declarations take space and clutter the code	No declarations of types required
Longer compilation	Faster compilation/interpretation
Methods are tied to specific types	Methods are more reusable, as they are less likely to be tied to specific types

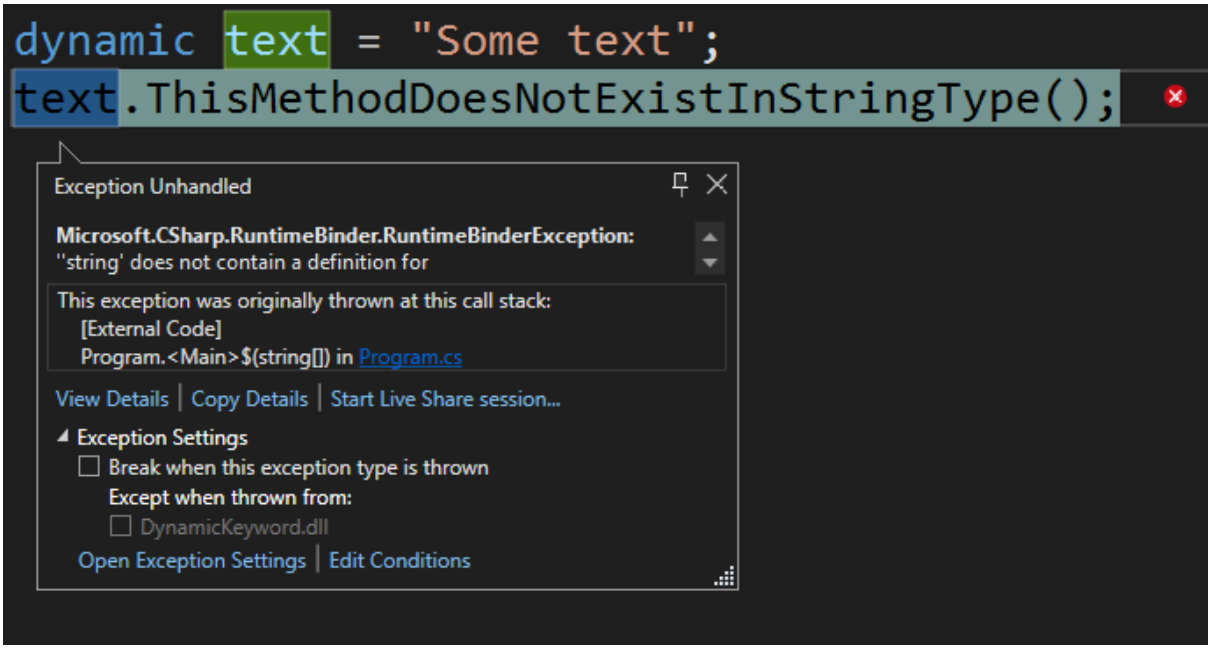
All right. We now know that C# is a statically-typed language and what are the consequences of this fact. If you are used to working in C# you probably don't ever suffer from the lack of dynamic typing. But what if we really, really wanted to use dynamic typing in C# for some reason? Well, in this case, the "dynamic" keyword comes in handy.

A variable declared with the dynamic type will bypass the static type checking. In other words, it will behave as it belonged to a dynamically-typed language. Let's see this in practice:

```
dynamic text = "Some text";
text.ThisMethodDoesNotExistInStringType();
```

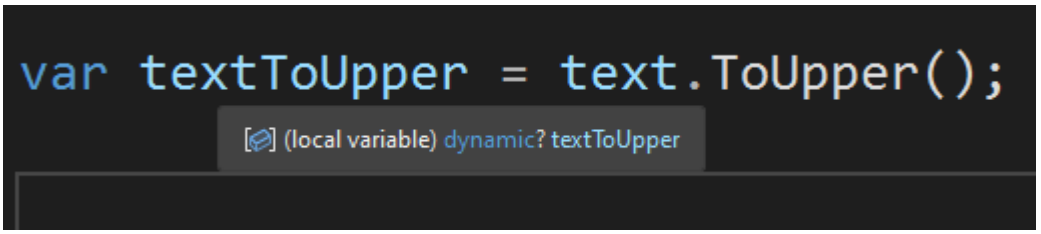
As you can see, I declared a variable of dynamic type and assigned it a string. Then, I called a method on this object. This method does not exist in string type. If the text variable was declared as a string, this code would not compile. But it's declared as dynamic, so the compiler doesn't execute type checking.

I will be able to run this program, but naturally, it will fail during the runtime:



This is what we talked about when defining a difference between static and dynamic typing. With dynamic typing, we exchange compile-time errors for runtime errors.

The result of most of the operations involving dynamic type is also the dynamic type:



In this case, the code will not fail in the runtime, because the ToUpper method exists in the string type. As you can see, the **textToUpper** variable is also of dynamic type.

There are only two cases when operations involving types do not result in dynamic types:

1) Casting from dynamic type to another type. We can cast it either explicitly…

```
string actualTextExplicitCast = (string)text;
```

…or implicitly (so without specifying the type in the braces).

```
string actualText = text;
```

This would naturally fail if the dynamic text variable did not hold a string.

2) The constructor calls using dynamic types as parameters:

```
var someClass = new SomeClass(text);
```

In this case, the **someClass** variable will not be dynamic, but rather SomeClass type.

All right. Now we know how the dynamic types work. So what can be the use case for them? Well, if you want to use them just to make C# more Python-y… please don't. Unless you do it for fun or out of curiosity in non-production code. Mixing static typing which is used in C# with dynamic typing is a risky business, and it will most likely result in code you can't maintain for long. Unless you are 100% sure what you are doing, and you have a rock-solid suite of unit tests, this is most likely a bad idea.

But there are cases when a type we are given is simply unknown in our project, but we as programmers actually know what it is. Sound weird, so let me give you an example. As you probably know, C# can cooperate with other .NET-compatible languages. And what if such a language is dynamically typed? An example can be IronPython, which is a .NET-compatible implementation of Python. We can actually execute IronPython code from C# code, but the result of this call will not be something C# can understand. This way, we can assign such a result to a dynamic variable. Let's see this in practice:

```
dynamic RunPython()
{
    const string pythonScript =
    @"class PythonClass:
    def toUpper(self, input):
        return input.upper()";

    var engine = Python.CreateEngine();
    var scope = engine.CreateScope();
    var operations = engine.Operations;

    engine.Execute(pythonScript, scope);
    var pythonClassObj = scope.GetVariable("PythonClass");
    dynamic instance = operations.CreateInstance(pythonClassObj);

    return instance.toUpper("Hello!");
}
```

In this code, we define an IronPython script, in which we define class **PythonClass** with one method **toUpper**. This method simply takes a variable and makes it

uppercase. Please note that in the last line of this method we call the **toUpper** method on the instance object. This is possible because the instance is dynamic. C# doesn't understand that the toUpper method exists in this object, but since it's dynamic the compiler doesn't complain. We as programmers know this method is there, and we take the risk of runtime error if we make a mistake and, for example, make a typo and call "toUppper" instead.

The result of the RunPython method is also dynamic, because again - the C# compiler doesn't know what will be the result of a call of any Python method, since Python is dynamically typed.

Another example of using the "dynamic" keyword to work with unknown types is using COM objects. COM stands for "Component Object Model" and it's a binary-interface standard for Windows software components. In simple terms, a COM object is something that can be understood by different Windows programs, and for example, it can allow communication between Excel and C# programs. But then, if we execute some method on a COM object from the C# code, its result's type will be unknown to the C# compiler. It will again be a case when declaring the result as dynamic will be useful. I don't want to get into too much detail on this topic, because most of the most useful COM objects are related to the Microsoft Office software, which is not free to use. If you are curious, I encourage you to read this article:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic#com-interop

Before we wrap up, I want to explain one thing I mentioned in this lecture, but did not go into details. I said that C# is a "statically-typed, strongly-typed programming language". I explained the difference between static and dynamic typing, but what is strong and weak typing?

Let's consider a simple operation like this (this is pseudocode, not any real language):

result = "2" + 8

In strongly-typed languages, this would raise an error. For example, C# and Python are strongly-typed languages. In weakly-typed languages like Perl, the result would be 10.

In short: in weakly-typed languages, variables are automatically converted from one type to another. In strongly-typed languages, this will not be the case.

Let's summarize. The "dynamic" keyword allows us to bypass static type checking that is done by default by the C# compiler. We can call any operations on dynamic

variables and the code will still compile. Whether the operation is available in this object or not will only be checked at runtime. The “dynamic” keyword is most useful when working with types unknown in our codebase, like types being the result of dynamically-typed languages scripts or operations called on COM objects.

Bonus questions:

- "**What is the difference between strongly-typed and weakly-typed programming languages?**"
  In weakly-typed languages, variables are automatically converted from one type to another. In strongly-typed languages, this will not be the case. For example, in C#, which is a strongly-typed language, the “2”+8 expression will not compile, while in weakly-typed Perl it will give 10 as a result.

- "**What is the difference between statically-typed and dynamically-typed programming languages?**"
  In statically-typed languages, the type checks are done at the compile time, while in dynamically-typed languages they are done at runtime. For example, in C# we can’t pass an integer to a method expecting a string. In Python, which is dynamically typed, we can, but the execution would result in a runtime error if in this method I would call some operation that is not available in int type.

- "**What are COM objects?**"
  COM stands for “Component Object Model” and it’s a binary-interface standard for Windows software components. In simple terms, a COM object is something that can be understood by different Windows programs, and for example, it can allow communication between Excel and C# programs.

5. What are expression-bodied members?

Brief summary: Expression-bodied members of a type are members defined with expression body instead of the regular body with braces. Using them allows us to shorten the code significantly.

Expression-bodied members of a type are members defined with expression body instead of the regular body with braces. That sounds very cryptic, but in practice, it's quite simple, and it allows us to write very concise and readable code.

Let's see expression-bodied members in practice. First, let's take a look at the Person class.

```
public class Person
{
    2 references
    public string Name { get; }
    2 references
    public int YearOfBirth { get; }

    0 references
    public Person(string name, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }

    1 reference
    public override string ToString()
    {
        return $"{Name} who was born on {YearOfBirth}";
    }
}
```

Let’s focus on the ToString method. This is a regular method and its body is contained in braces. Simple as it is, it takes quite a lot of space, while actually, only this part is really defining the logic of this method:

```
$"{Name} who was born on {YearOfBirth}";
```

This part contains a single **expression**. But first, what is an expression? An expression is a piece of code that evaluates to some value. Here we can see a single expression, that for the Person with name “John” and YearOfBirth 1972 it will evaluate to ”John who was born in 1972”.

If a method only contains only a single expression it can be defined as an expression-bodied **method**. Let's do this for the ToString method.

```
~references
public override string ToString() => $"{Name} who was born on {YearOfBirth}";
```

Let's see what happened. First of all, I removed the braces that typically surround the body of the method. I also removed the "return" keyword and simply placed what this method returns to the right side of the arrow operator. And that's it! This method does exactly the same thing as before. And it only takes a single line of code.

So the general blueprint for expression-bodied methods is this:

We define the return type and the name of the method on the left side of the arrow. On the right side, we define an expression whose value will be returned. Expression-bodied methods are short and readable. Their limitation is that they must only contain a **single expression**. So, for example, this method could not be changed to an expression-bodied method:

```
0 references
public bool IsNameValid()
{
    if (Name == null)
    {
        Console.WriteLine("Name is null.");
        return false;
    }
    if (Name.Length == 0)
    {
        Console.WriteLine("Name is too short.");
        return false;
    }
    if (Name.Length > 25)
    {
        Console.WriteLine("Name is too long.");
        return false;
    }
    return true;
}
```

This method contains more than one expression, as well as several **statements**. A statement is a piece of code that does something but does not evaluate to a value. For example, the Console.Writeline calls are statements.

If a method contains a single statement, it can also be changed to a void expression-body method. Let's see this in practice:

```
0 references
public void Print() => Console.WriteLine(ToString());
```

This method simply prints something to the console. It doesn't return anything, as its return type is void.

All right. We learned about the expression-bodied methods, but there are more expression-bodied members we can have. One of the most common use cases for them is a read-only **property**. Let's define an Age property, which will return the current year minus the year of birth of the person.

```
//this takes 7 lines
0 references
public int Age
{
    get
    {
        return DateTime.Now.Year - YearOfBirth;
    }
}
```

Again, it's extremely simple code, but it takes 7 lines! Let's change it to an expression-bodied read-only property.

```
//and this takes 1!
0 references
public int Age => DateTime.Now.Year - YearOfBirth;
```

Great! Now it only takes a single line of code.

The property doesn't need to be read-only to be defined with expression body. Let's add the LastName property to the Person class. Last names can be changed, most often when someone gets married, so let's enable the modification of this property.

```
private string _lastName;
0 references
public string LastName
{
    get
    {
        return _lastName;
    }
    set
    {
        _lastName = value;
    }
}
```

This is how the properties looked like before C# 4.0 (FYI, we are using C# 10 in this course, so you can guess it was quite some time ago). Starting with C# 4, the auto-implemented properties were introduced, allowing us to write this code like this.

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

This was a huge improvement and it still works great in 99% of the cases. But sometimes we want to execute some additional operations during getting and setting of the backing field - for example, we may want to trim the white-space characters on setting the last name. This way, we won't be able to use the auto-implemented property, because it simply assigns the value on **set**, without any additional operations. In this case, we are stuck with implementing the getter and setter by ourselves. Let's do it.

```
private string _lastName;
0 references
public string LastName
{
    get
    {
        return _lastName;
    }
    set
    {
        _lastName = value.Trim();
    }
}
```

Great. This will work, but it still takes a tremendous amount of space. Let's change this property to an expression-bodied property.

```
public string LastName
{
    get => _lastName;
    set => _lastName = value.Trim();
}
```

And here it is. Only 5 lines instead of 11.

Another thing I want to talk about is an expression-bodied **constructor**. The problem with it is that it allows only a single operation to be executed, which in our case is not the case - we assign both name and last name. But for the sake of the example, let me define a second constructor, which only assigns the name.

```
public Person(string name) => Name = name;
```

As you can see it's also a nice one-liner, so it may still be a good idea to use it if we only execute a single operation in the constructor.

We can also define expression-bodied **destructors**. Let's add a destructor to this class.

```
0 references
~Person() => Console.WriteLine("Destructing the Person object");
```

The last members that can be defined with an expression body are indexers, but we will talk more about them later in the course.

Bonus questions:

- "**What is an expression?**"
  An expression is a piece of code that evaluates to some value. For example "2 + 5" evaluates to 7.

- "**What is a statement?**"
  A statement is a piece of code that does something but does not evaluate to a value. For example, Console.Writeline("abc") is a statement. It does not evaluate to any value, as the Console.Writeline is a void method.

6. What are Funcs and lambda expressions?

Brief summary: In C#, we can treat functions like any other types - assign them to variables or pass them as parameters to other functions. The Func and Action types allow us to represent functions. Lambda expressions are a special way of declaring anonymous functions. They allow us to define functions in a concise way and are most useful when those functions will not be used in a different context.

When thinking about objects, we usually think about things that carry some data as the payload, as well as operations that can be executed on this data. An integer holds the value of 5, a Person object holds Name, LastName, and YearOfBirth, and a method calculating the Age.

We can easily understand that we can have variables of such types, or that we can pass objects of those types as parameters.

But could we have variables holding functions? Can we pass a function as a parameter?

As it turns out, we can. And it is extremely useful. Let me show you an example. I want to write a method that checks if any number in a collection is larger than 10. This is how I could do it:

```
bool IsAnyLargerThan10(IEnumerable<int> numbers)
{
    foreach(var number in numbers)
    {
        if(number > 10)
        {
            return true;
        }
    }
    return false;
}
```

All right. This was pretty simple. But soon after I am asked to add another method, that checks if any number in the collection is even. Let's create such a method:

```
bool IsAnyEven(IEnumerable<int> numbers)
{
    foreach (var number in numbers)
    {
        if (number % 2 == 0)
        {
            return true;
        }
    }
    return false;
}
```

The problem is that this method is almost the same as the previous one. The only place in which they differ is this:

```
if (number > 10)
if (number % 2 == 0)
```

This is the part that differs, so if we wanted to refactor those two methods into one, we would need to make this part a parameter.

But what is this part exactly? At first glance, you may think it's a boolean. But that is not correct. I couldn't pass this boolean to the IsAny method, as it may differ for each of the elements of the numbers collection. So, it's not a boolean. It's a function that takes a number and returns a boolean. If I want to refactor those methods and make them a single method, it will need to take a function as a parameter. And for this, we can use the Func type. Let me show you how it looks:

```
Func<int, bool> predicate
```

This Func can be assigned any function, that is taking a number and returning a bool. As you can see, Func is generic, and by type parameters, we define what is the return type of the function and what parameters it takes. The last type parameter is always the return type, everything preceding it are the types of parameters. For example, this variable can be assigned any function that takes an int, a DateTime, and a string parameters, and returns a decimal.

```
Func<int, DateTime, string, decimal> someFunc;
```

For void functions, we must use the Action type. This variable can represent any void function taking two strings and a bool:

```
Action<string, string, bool> someAction;
```

All right. Let's go back to refactoring. I want to have a single IsAny method, taking a collection of numbers and a function that defines the predicate that will be checked:

```
bool IsAny(
    IEnumerable<int> numbers,
    Func<int, bool> predicate)
```

Now, instead of using "number > 10" or "number % 2 == 0" I will simply use the predicate:

```
bool IsAny(
    IEnumerable<int> numbers,
    Func<int, bool> predicate)
{
    foreach (var number in numbers)
    {
        if (predicate(number))
        {
            return true;
        }
    }
    return false;
}
```

As you can see, we can call the Func object like any other method. Let's use this method. First of all, I will define IsLargerThan10 and IsEven methods:

```
bool IsLargerThan10(int number)
{
    return number > 10;
}

bool IsEven(int number)
{
    return number % 2 == 0;
}
```

Now I can use those methods as parameters of the IsAny method:

```
Func<int, bool> predicate1 = IsLargerThan10;
Console.WriteLine(
    "IsAnyLargerThan10? " + IsAny(numbers, predicate1));

Func<int, bool> predicate2 = IsEven;
Console.WriteLine(
    "IsAnyEven? " + IsAny(numbers, predicate2));
```

As you can see, we can simply assign methods to the variables of Func type. Naturally, it wouldn't compile if the signatures of the methods would not match the signatures of Funcs.

I can of course skip declaring variables and simply pass the methods as parameters:

```
Console.WriteLine(
    "IsAnyLargerThan10? " + IsAny(numbers, IsLargerThan10));

Console.WriteLine(
    "IsAnyEven? " + IsAny(numbers, IsEven));
```

Great. We learned that in C# we can treat functions like any other objects - we can assign them to variables or pass them as parameters. Func and Action are the types that allow us to do this.

Now, let's move on to the lambda expressions. Before I explain what they are exactly, let me show you again those two methods:

```
bool IsLargerThan10(int number)
{
    return number > 10;
}

bool IsEven(int number)
{
    return number % 2 == 0;
}
```

Those methods are very specific. I'm not likely to reuse them anywhere else, yet I needed to declare them in this class so I can pass them as parameters. Imagine you have a class that does plenty of very specific checks on collections: like checking if any number is divisible by 5, if any string is longer than 10 letters or if any Person is named John. We would clutter our class with tiny, simple methods that check those things, and are not used anywhere else. That's not a good approach. It would be better if we could have an easy, short way of defining small, simple, and specific functions.

And that's exactly what lambda expressions are for. First, let me show you how they look, and then I will explain in detail:

```
Console.WriteLine(
    "IsAnyLargerThan10? " + IsAny(numbers, n => n > 10));

Console.WriteLine(
    "IsAnyEven? " + IsAny(numbers, n => n % 2 == 0));
```

In this code, I did not use the **IsLargerThan10** and **IsEven** methods - I can now safely remove them. I defined those methods in form of lambda expressions. Lambda expression is a special way of defining anonymous function. This is the general pattern of any lambda expression:

param => expression

On the left side of the arrow, we declare a parameter or parameters of the function. On the right side, we declare the expression whose result will be returned from the function.

We can also have lambda expressions with more than one parameter. In this case, we must put parameters in parenthesis:

This is much shorter than the traditional way of declaring functions. There is no "return" keyword, as it is simply assumed that the result calculated on the right side is returned. The result type is not declared because it is inferred from what the expression evaluates to. For example, in this lambda expression, it is obvious that the return type of the function is string:

The types of the parameters are also not formally defined, so how does the compiler know what they are? Well, it also infers them from the context. Let's see this code again:

```
IsAny(numbers, IsLargerThan10)
```

The IsAny method expects a very particular function as the parameter - a function that takes a number and returns a bool:

```
bool IsAny(
    IEnumerable<int> numbers,
    Func<int, bool> predicate)
```

Because we use the lambda expression as the parameter of the IsAny method, the compiler infers, that the parameter of the lambda expression must be an integer, because the IsAny method expects such a function as the parameter.

The fact that the type of parameters is inferred from context is the reason why we can't assign lambda expressions to implicitly-typed variables:

```
var someFunc = n => n % 2 == 0;
```

In this case, there is no context from which the compiler can infer the type of the parameter. To fix it, we can make the variable explicitly typed:

```
Func<int,bool> someFunc = n => n % 2 == 0;
```

All right. You may say that there is a lot of assumptions and inferring, but see how little code we needed to write to declare a function checking if a number is even:

```
n => n % 2 == 0
```

And this is how this function looks like when declared in the traditional way:

```
bool IsEven(int number)
{
    return number % 2 == 0;
}
```

Lambda expressions are great when declaring short, specific functions that will most likely not be used in any other context. For example, they are extremely often used when working with LINQ. Actually, the IsAny method we declared is almost identical to the Any method from LINQ.

Let's summarize. In C#, we can treat functions like any other types - assign them to variables or pass them as parameters to other functions. The Func and Action types allow us to represent functions. Lambda expressions are a special way of declaring anonymous functions. They allow us to define functions in a concise way and are most useful when those functions will not be used in a different context.

Bonus questions:

- "**What is the signature of a function that could be assigned to the variable of type Func<int, int, bool>?**"
  It would be a function that takes two integers as parameters and returns a bool.

- "**What is an Action?**"
  Action is a type used to represent void functions. It works similarly to Func, but Func can only represent non-void functions.

7. What are delegates?

Brief summary: A delegate is a type whose instances hold a reference to a method with a particular parameter list and return type.

In C#, we can declare several kinds of types. We can declare a class or a struct to represent data and methods. We can declare enums to represent some predefined, discrete values. We can also declare delegates. A delegate is a type whose instances hold a reference to a method with a particular parameter list and return type.

Let's define a simple delegate:

```
delegate string ProcessString(string input);
```

We declare delegates with the "delegate" keyword (similarly as we define classes with the "class" keyword or structs with the "struct" keyword). Besides the "delegate" keyword, a delegate declaration looks the same as a method declaration - first the returned type, then the name, and finally the list of parameters.

All right. We defined a delegate type, and any variable of this type can be assigned a method that takes a string as a parameter, and also returns a string.

```
string TrimTo5Letters(string input)
{
    return input.Substring(0, 5);
}

string ToUpper(string input)
{
    return input.ToUpper();
}
```

Both those functions can be assigned to the variable of the ProcessString delegate type:

```
ProcessString processString1 = TrimTo5Letters;
ProcessString processString2 = ToUpper;
```

We can now call the delegate, as a result, execute the function that is stored in it:

```
Console.WriteLine(processString1("Hellooooooooooooo"));
Console.WriteLine(processString2("Hellooooooooooooo"));
```

```
Hello
HELLOOOOOOOOOOOO
```

As you can see, the methods stored in the delegate variables have been invoked, as expected.

The interesting feature of delegates is that we can store more than one function in one variable of delegate type:

```
delegate void Print(string input);
```

```
Print print1 = text => Console.WriteLine(text.ToLower());
Print print2 = text => Console.WriteLine(text.ToUpper());
Print print3 = print1 + print2;
print3("Crocodile");
```

As you can see, the **print3** variable holds references to two functions, not one. And that's why the result of the execution of the **print3** delegate variable will be this:

```
crocodile
CROCODILE
```

We can also use the "+=" operator to chain another method to a delegate variable:

```
Print print4 = text => Console.WriteLine(text.Substring(0,3));
print3 += print4;
print3("Giraffe");
```

Now the result will be:

```
giraffe
GIRAFFE
Gir
```

A delegate variable that holds references to more than one function is called a **multicast delegate**.

All right. In the previous lecture, we learned about the **Func and Action** types. What is interesting, both are simply generic delegates. For example, this is the definition of the Func taking two parameters:

```
delegate TReturn Func<TInput1, TInput2, TReturn>(
    TInput1 input1, TInput2 input2);
```

This delegate represents a function taking two parameters of any kind and returning some result, so exactly what a Func with two parameters can represent:

```
Func<string, string, int> sumLengths =
    (text1, text2) => text1.Length + text2.Length;
```

Of course, instead of a lambda expression, we can also assign a pre-existing function to a variable of Func type, as we can with any delegates:

```
Func<string, string, int> sumLengths =
    (text1, text2) => text1.Length + text2.Length;

Func<string, string, int> sumLengths2 = SumLenghts;
int SumLenghts(string text1, string text2)
{
    return text1.Length + text2.Length;
}
```

During the interview, you might be asked a question “What is the difference between a Func and a delegate?”. Well, **Func is a delegate**, simply defined by

Microsoft, not us. To be more precise, Func is a **generic** delegate used to represent any function with given parameters and returned type. A delegate is a broader concept than Func - we can define any delegate we want, and it doesn't need to be generic at all.

All right. So we know that Funcs and Actions are generic delegates. Because of that, we can represent any function we want with Func or Actions, right? Let's see again the delegate variables we declared at the beginning of this lecture:

```
delegate string ProcessString(string input);
```

```
ProcessString processString1 = TrimTo5Letters;
ProcessString processString2 = ToUpper;
```

We don't really need to define ProcessString delegate, we could use the Func instead.

```
Func<string, string> processString1Func = TrimTo5Letters;
Func<string, string> processString2Func = ToUpper;
```

So, why bother declaring delegates, if we can use Funcs instead?

Well, first of all - delegates existed in C# before Funcs, Actions, and lambda expressions. There was a time when they were simply the only way to represent functions as objects, so you can still see some old code full of custom delegates instead of Funcs and Actions for the simple reason that it was created a long time ago.

Nowadays, from my practice, I would say that in 99% of the cases there is no point in declaring customs delegates, and Funcs can be used instead. Personally, I very rarely use other delegates than Funcs and Actions.

But there are scenarios where custom, non-generic delegates can be useful.

First of all, some people prefer well-named delegates to slightly cryptic Funcs:

```
interface ICommandExecutor
{
    0 references
    void Execute(Func<CommandType, bool> command);
    0 references
    void Execute(RunCommand command);
}

2 references
enum CommandType { Start, Stop, Reset }
```

```
delegate bool RunCommand(CommandType commandType);
```

In this interface, we have two methods doing the same thing, but one takes a Func and the other a custom delegate. “RunCommand” is definitely more human-readable than “Func<CommandType, bool>”. Some people think such a named delegate is better.

Personally, I disagree, for a simple reason: when I look at the Func<CommandType, bool> I know that the input to this function is CommandType, and I also know that it returns a bool (and I can make an educated guess that this is probably a boolean telling me if the command execution was successful or not). With “RunCommand” I don’t have that information, and I need to go to RunCommand type to see what actually are the parameters and returned type. Also, with “Func” I can see right off the bat this is an executable method. “RunCommand” could be some ordinary object as well, and I won’t know unless I go to this type and see the “delegate” keyword.

But I want to underline that this is a matter of personal taste. If for you and your team custom delegates are more readable, you should definitely use them everywhere where you want.

Another use case where custom delegates can have an advantage over Funcs or Actions is when a specific delegate is used all around the application. This way, instead of declaring a (sometimes complex) Func everywhere, I can have a single declaration of a custom delegate:

```
Func<int, string, bool, DateTime, float> verySpecificFunc;
Calculate superComplexFuncionUsedEverywhereInTheProject;
```

```
delegate float Calculate(
    int number, string text, bool flag, DateTime date);
```

Also, there are some use cases when we simply **must** use custom delegates instead of Funcs or Actions. One of the examples is when we want to use them with methods with **ref** or **out** parameters. It's not possible using Funcs:

```
Func<out string, ref int, double> funcWithRefOut;
```

But it works fine with custom delegates:

```
delegate double DelegateWithRefOut(
    out string text, ref int number);
```

Similarly, Funcs can't have **optional parameters**, while custom delegates can:

```
Func<string, int, bool> funcWithOptionalParameters =
    (text, number = 0) => true;
```

This doesn't work for Funcs, but works fine for delegates:

```
delegate bool DelegateWithOptionalParam(
    string text, int number = 0);
```

Also, we can't have Funcs with the **params** parameters, but we can have custom delegates:

```
Func<params string[], int> funcWithParams;
```

```
delegate int DelegateWithParams(params string[] words);
```

All right. We will revisit the topic of delegates in the lecture about events, as using delegates is crucial when working with them.

Bonus questions:

- "**What is the difference between a Func and a delegate?**"
  Func is a delegate, simply defined by Microsoft, not us. To be more precise, Func is a generic delegate used to represent any function with given parameters and returned type. A delegate is a broader concept than Func - we can define any delegate we want, and it doesn't need to be generic at all.

- "**What is a multicast delegate?**"
  It's a delegate holding references to more than one function.

8. How does the Garbage Collector decide which objects can be removed from memory?

Brief summary: Garbage collector removes those objects, to which no references point. To decide whether a reference pointing to some object exists, the Garbage Collector builds a graph of all objects reachable from root objects of the application, which are things like references currently stored on the stack or in static fields of classes. If an object will not be included in this graph, it means it's not needed and can be removed from memory. After the graph of reachability is built, the Garbage Collector can continue its work and remove the unreachable objects.

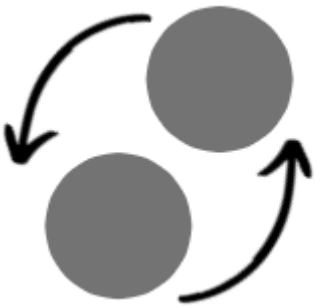
As you (hopefully) know, the Garbage Collector is the Common Language Runtime's mechanism that is responsible for **memory management** in .NET applications. Its main role is to remove from the memory the objects that are no longer needed. The algorithm it implements is called "**mark-and-sweep**" because it first **marks** objects that can be removed, and then **sweeps** them from memory.

But, how can we tell if an object can be removed? Well, in simplest terms, when there are no existing references that point to this object. But how can the Garbage Collector know it?

Well, in the family of tools similar to the Garbage Collector (remember, this concept is not exclusive to .NET alone) there are two most commonly used algorithms.

The first is called **reference counting,** which, for example, is used in Swift. According to this algorithm, the garbage collector keeps track of the count of references pointing to some object. When the count reaches zero, it considers this object unreferenced by any other object, and thus a candidate to be removed from memory.

There is a problem with this algorithm, though. Imagine there are two objects, A and B. There is a reference from A to B, and from B to A (such reference is called a **circular** reference), but no other object in the application references A nor B.



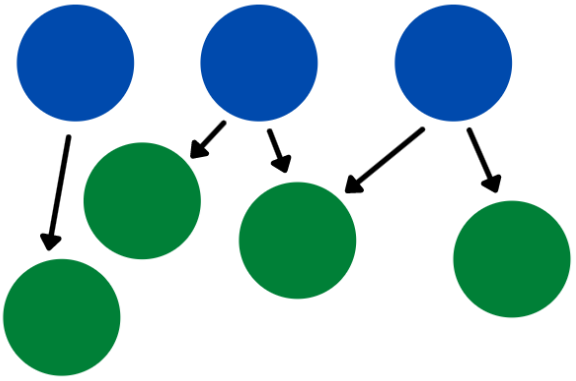
A and B objects could be removed from memory because they cannot be reached from any point in the application. But since for both of them, the reference count is 1, not 0, a garbage collector using the reference counting algorithm would not remove them.

Because of that, reference counting is not used in .NET. Instead, **tracing** is used.

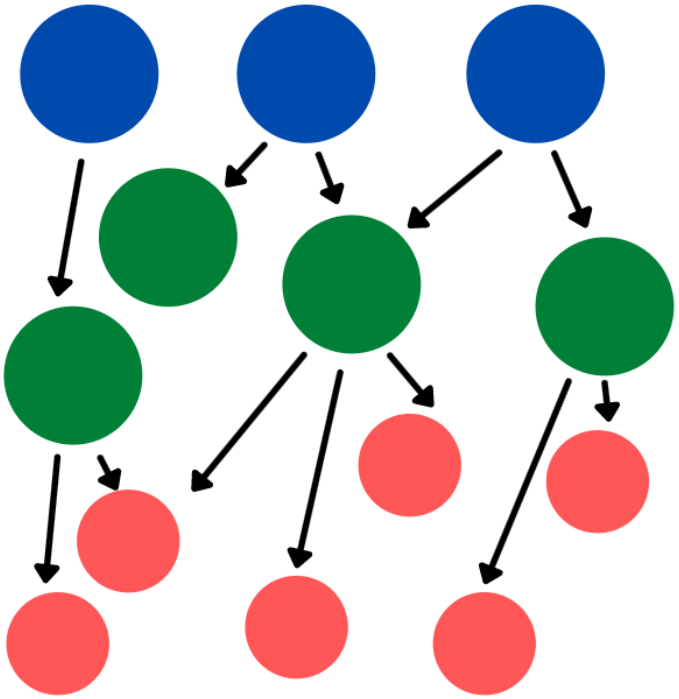
Tracing will determine whether an object is reachable or not by tracing it from a set of **application roots**. We will learn what application roots are in a minute, but for now, let's just say they are the objects that the Garbage Collector starts building the graph of reachability from. If an object is reachable from a root object, either directly or indirectly, then it will be considered alive. If it's not reachable, it means it can be removed from memory.

Let's say that the Garbage Collector identifies 3 application roots. They, by definition, are considered reachable:

Then, for each of them, the Garbage Collector checks what references they hold, and it adds the objects pointed by those references to the graph of reachable objects:

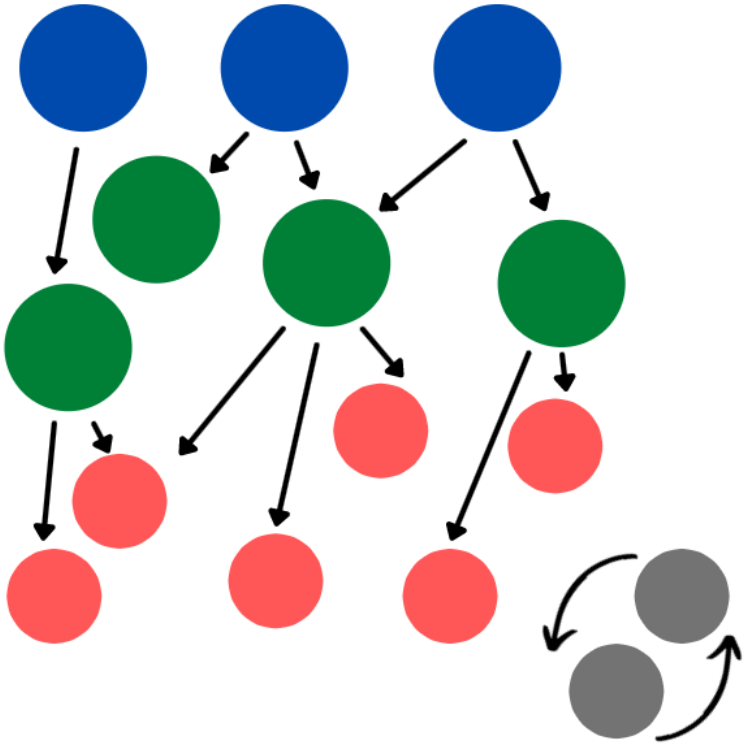


It repeats this step for each of the new objects:



It continues its work until there are no more objects having references to objects that are not included in the graph.

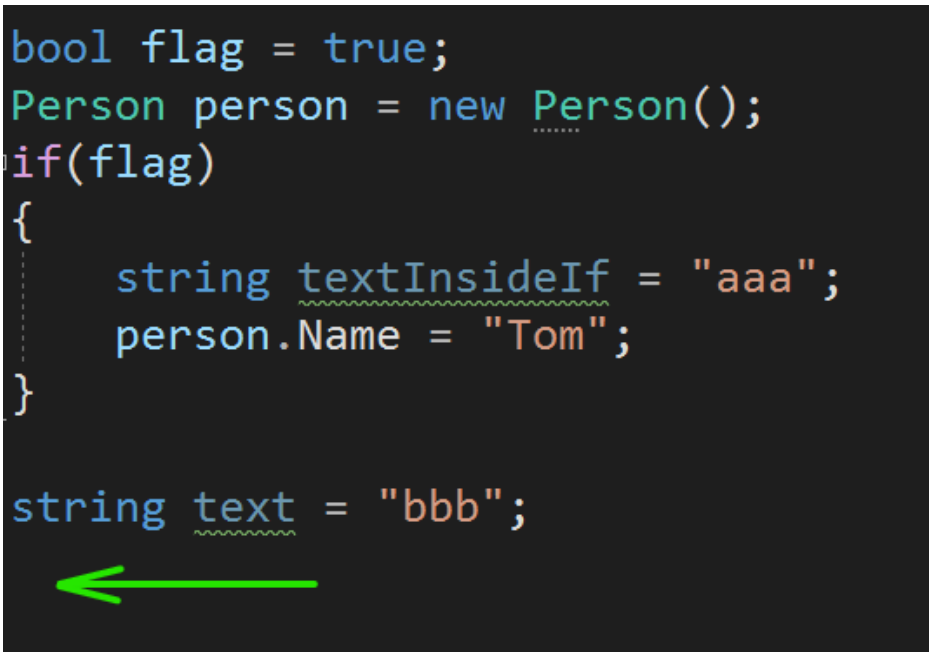
The power of this algorithm is that the circular references will not be included in the graph of reachable objects, because, well - they are not reachable:



All right. Let's now see what application roots are exactly. Roots include static fields and local variables on a thread's stack[1]. For us, the most important part is "local variables on a thread's stack". The first thing we need to clarify: each thread has its own stack, so in multi-threaded applications, we will have more than one stack. For simplicity, let's focus on single-threaded applications.

The Garbage Collector will look at the stack and see what references are currently stored on it. Remember - the reference itself is stored on the stack, and it points to an object stored on the heap.

Let's consider this code:



At the execution point marked by the green arrow, the following references are stored on the **stack**:

1) the reference to the **Person** object stored in the **person** variable
2) the reference to the "bbb" string stored in the **text** variable

And the following references are **not** stored on the stack:

1) the reference to the "Tom" string - it's stored in the **person** object

[1] There are also couple more things that will be included to the application roots collection, but they come from low-level mechanisms of C# which are beyond this course's level. If you are curious, I recommend reading this document: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals#:~:text=An%20application's%20roots%20include%20static,the%20runtime%20for%20these%20roots.

2) the reference to the “aaa” string stored in the **textInsideIf** variable, as it was removed from the stack once the code execution reached the end of the **if** statement

If the Garbage Collector gets triggered once code execution reaches the point marked by the green arrow, it will identify two references stored on the stack and will include them in the application roots: the reference to the **person** object, and the reference to the “**bbb**” string. It will start building the reachability graph from those roots. The **person** object holds a reference to the “**Tom**” string, so it will also get included in the graph. On the other hand, no object in the graph holds the reference to the “aaa” string, so it will be considered a candidate for removal by the Garbage Collector.

Let’s summarize. To decide whether a reference pointing to some object exists, the Garbage Collector builds a graph of all objects reachable from root objects of the application, which are things like references currently stored on the stack or in static fields of classes. If an object will not be included in this graph, it means it’s not needed and can be removed from memory. After the graph of reachability is built, the Garbage Collector can continue its work and remove the unreachable objects.

Bonus questions:

- "**What is the Mark-and-sweep algorithm?**"
  It’s the algorithm that the Garbage Collector implements. According to this algorithm, the GC first marks objects that can be removed (mark phase) and then actually removes them (sweep phase).

- "**How many stacks are there in a running .NET application?**"
  As many as threads. Each thread has its own stack.

- "**What two main algorithms of identifying used and unused objects are implemented by tools similar to .NET Garbage Collector?**"
  First is reference counting, which associates a count of references pointing to an object with each object. An example of a language using it is Swift. Another algorithm is tracing (this one is used in .NET) which builds a graph of reachability starting from the application roots.

9. What are generations?

Brief summary: The Garbage Collector divides objects into three generations - 0, 1, and 2 - depending on their longevity. Short-lived objects belong to generation 0, and if they survive their first collection, they are moved to generation 1, and after that - to generation 2. The Garbage Collector collects objects from generation 0 most often, and from generation 2 least often. This feature is introduced in order to improve Garbage Collector's performance. Objects that survived a couple of cycles of the GC's work tend to be long-lived and they don't need to be checked upon so often. This way, the Garbage Collector has less work to do, so it can do it faster.

Garbage Collector has a lot of work. It needs to identify objects to be removed, which we learned about in the previous lecture. Then, it must actually remove them. Finally, it must defragment the memory of the application.

As we know the Garbage Collector marks objects as reachable or unreachable and removes the latter from the memory. Imagine there is some object - let's call it object A. During the first execution of the Garbage Collector, this object is marked as reachable, and so are all objects reachable from object A. After some time the Garbage Collector gets to work again, and again, it marks object A and its friends as reachable. Then again, some time passes, and Garbage Collector gets triggered again… and again it marks object A as reachable.

If I were the Garbage Collector, I would probably get a bit frustrated. This object is obviously long-lived, and I don't want to check if it's still needed every time I get to work! I have other things to do!

Well, that's exactly the optimization the Garbage Collectors creators come up with - to not make the Garbage Collector check each object every time. If some object "survived" a couple of cycles of the Garbage Collector's work, it's most likely long-lived, and we should check it only every once in a while.

This optimization introduced the concept of **generations** of objects. Once an object is first created, it is assigned to generation 0. If it survives its first collection, it advances to generation 1. If it survives the second, it advances to generation 2.

The Garbage Collector checks objects from Generation 0 most frequently. Less frequently it checks objects from generation 1, and least frequently - from generation 2. It makes sense. If the object survived two collections, it's most likely

long-lived, and there is a big chance it will survive collections number 10, 20, or 100. We don't need to check on it that often.

For example, think of some logger objects. Often there is one logger object created at the start of the program execution, and it is passed around to any class that needs it. Its life cycle is basically as long as the time the application runs.

As the opposite, we often create objects that last only for a very short time, like anonymous objects created to temporarily carry some data between LINQ queries. In this case, it is reasonable that the Garbage Collector will quickly remove them, so they don't occupy memory for too long.

In a well-tuned application, most objects die in generation 0.

Please note that during a collection of a generation, all previous generations are also collected. So when generation 0 is collected, no other one is, but when generation 2 is collected, generations 0 and 1 are too (that's why collecting objects from generation 2 is sometimes called "a full garbage collection").

One more thing that needs to be mentioned is the LOH - the Large Objects Heap. When a very large object (larger than 85 000 bytes) is initially created, it is stored in a special area of memory called the **Large Objects Heap**, and it is assigned to generation 2 right away. It gets this special treatment because it rarely happens that very large objects are short-lived. Also, the objects in the Large Objects Heap have one more special feature - they are **pinned**. It means, they will not be moved in memory during the defragmentation step of the Garbage Collector's work, which happens after removing unreferenced objects from memory. This is because the larger the object is, the more expensive is the operation of moving it (and the harder it is to find the chunk of memory large enough to fit it). It's better to "pin" such large objects, and move smaller objects around them. (FYI, starting with .NET 4.5.1 we can change this default and "unpin" the pinned objects living in the LOH).

Let's summarize. The Garbage Collector divides objects into three generations - 0, 1, and 2 - depending on their longevity. Short-lived objects belong to generation 0, and if they survive their first collection, they are moved to generation 1, and after that - to generation 2. The Garbage Collector collects objects from generation 0 most often, and from generation 2 least often. This feature is introduced in order to improve Garbage Collector's performance. Objects that survived a couple of cycles of the GC's work tend to be long-lived and they don't need to be checked upon so often. This way, the Garbage Collector has less work to do, so it can do it faster.

Bonus questions:

- "**What is the Large Objects Heap?**"
  It's a special area of the heap reserved for objects larger than 85 000 bytes. Such objects logically belong to generation 2 from the very beginning of their existence and are pinned.

- "**What does it mean that the object is pinned?**"
  It means it will not be moved during the memory defragmentation that the Garbage Collector is executing. It is an optimization, as large objects are expensive to move, and it's hard to find a chunk of memory large enough for them.

10. What is the difference between Dispose and Finalize methods?

Brief summary: The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

First, let's focus on the **Dispose** method. This method comes from the IDisposable interface and it is used to free up any unmanaged resources used by an object when its work is finished.

First of all, let's understand what managed and unmanaged resources are.

Managed resources, as their name suggests, are managed by the Common Language Runtime. Any objects we create with C# are managed resources. The Garbage Collector is aware of their existence, and once they are no longer needed it will free up the memory they occupy. That means we don't need to worry about managed resources cleanup as it is done automatically for us.

Unmanaged resources are beyond the realm of the CLR. The Garbage Collector doesn't know about them, so it will not perform any cleanup on them. Examples of unmanaged resources are database connections, file handlers, COM objects, opened network connections, etc. We as developers are responsible to perform the cleanup after we are done with those objects. If we don't, bad things may happen. For example, if we open a file to read it and we don't close it, the next attempt to open the same file will fail with an error saying that the file is currently in use.

If we have a class that uses some unmanaged resources, it should implement the IDisposable interface and provide an implementation of the Dispose method. The Dispose method should contain the code that cleans the unmanaged resource, for example, closes a file. Let's see a simple class that does so:

```
class FileReader : IDisposable
{
    private StreamReader? _streamReader;
    private readonly string _path;

    1 reference
    public FileReader(string path)
    {
        _path = path;
    }

    2 references
    public string ReadLine()
    {
        _streamReader ??= new StreamReader(_path);
        return _streamReader.ReadLine();
    }

    0 references
    public void Dispose()
    {
        _streamReader?.Dispose();
    }
}
```

This class is simply providing a way to read a file line-by-line. Please note that this implementation is simplified for example's sake so it doesn't contain any error handling. This class implements the IDisposable interface, and because of that, it is forced to provide the implementation of the Dispose method. In this method, we clean up any unmanaged resources. In this case, we simply call the Dispose method of the StreamReader. This is a very common practice when implementing the Dispose method because it rarely happens that we need to access unmanaged resources that do not have any C# class meant to use them provided. Calling Dispose method from a dependency of a class (or methods, if we have more IDisposable dependencies) in the Dispose method o this class is called a **cascade Dispose**.

All right. Let's use the FileReader class:

```
var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
```

At the first glance, it may seem ok - line1 and line2 are set to values coming from the input.txt file. The problem here is that we do not actually call the Dispose method, and the StreamReader is never closed. Let's fix that. We could call the Dispose method manually:

```
var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
fileReader.Dispose();
```

…but this is a bit awkward and easy to forget, not to mention that if the exception will be thrown in this code, the Dispose method may never be called. It's better to use the **using statement**:

```
using (var fileReader = new FileReader("input.txt"))
{
    var line1 = fileReader.ReadLine();
    var line2 = fileReader.ReadLine();
}
```

Starting with C# 8 we can use the following syntax without braces:

```
using var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
```

Remember, the using statement is just syntactic sugar for this:

```
FileReader fileReader = null;
try
{
    fileReader = new FileReader("input.txt");
    {
        var line1 = fileReader.ReadLine();
        var line2 = fileReader.ReadLine();
    }
}
finally
{
    fileReader?.Dispose();
}
```

The “finally” block is used to ensure that the Dispose method will be called no matter if the exception will be thrown or not.

All right. One more thing before we move on to the Finalize method. Remember that **the Garbage Collector does not call the Dispose method**. We must call it ourselves, and the best way to do it is by using the **using statement**, which ensures that the Dispose method will be called.

Now, let’s move to the **Finalize** method. This method is called on an object when it is being cleaned up by the Garbage Collector. That means it can only be added to reference types, so a struct or a record struct can’t have a finalizer defined. Please notice that in C# the destructor, the finalizer, and the Finalize method are the same things. We can’t even define the Finalize method in a class - we must do so by defining a destructor. Let’s see this in practice.

```
class Person
{
    2 references
    string Name { get; }
    1 reference
    public Person(string name) => Name = name;
    0 references
    ~Person()
    {
        Console.WriteLine($"Person {Name} is being destructed");
    }
}
```

I defined a Person class that contains a destructor. When an object of this class will be cleaned up by the Garbage Collector this method will be executed. In my Main method, I run the following code:

```
SomeMethod();

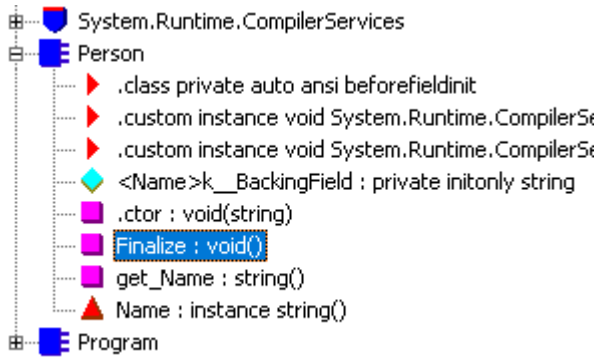
GC.Collect();
Console.ReadKey();

void SomeMethod()
{
    var john = new Person("John");
}
```

john object lives in the scope of SomeMethod, and after this method finishes it is no longer needed. By running **GC.Collect** command I ask the Garbage Collector to do its work. And this is the result of the program:

```
Person John is being destructed
```

We said before that the Finalize method and the destructor are the same things. To prove it, let me show you how the Person class looks after being compiled into the Common Intermediate Language. I will use **ildasm** to read the dll.

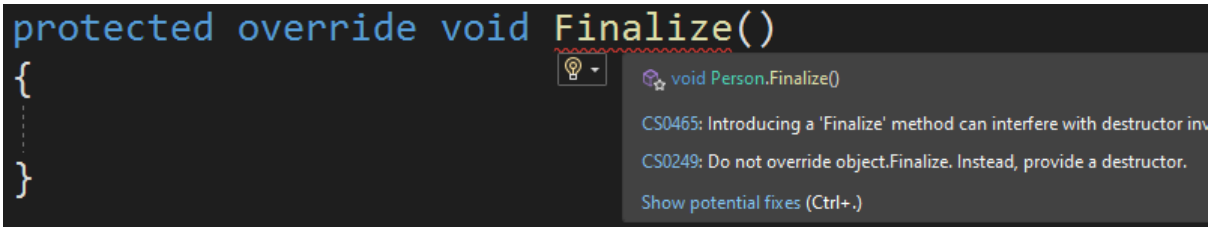


As you can see the Finalize method is added. And this is how it looks in CIL:

```
.method family hidebysig virtual instance void
        Finalize() cil managed
{
  .override [System.Runtime]System.Object::Finalize
  // Code size       40 (0x28)
  .maxstack  3
  IL_0000:  nop
  .try
  {
    IL_0001:  nop
    IL_0002:  ldstr      "Person "
    IL_0007:  ldarg.0
    IL_0008:  call       instance string Person::get_Name()
    IL_000d:  ldstr      " is being destructed"
```

As you can see it prints the same message as we defined in the destructor. In other words, the destructor is changed into the Finalize method during the compilation.

If I tried to add the Finalize method manually, I would get an error "Do not override object.Finalize. Instead, provide a destructor":



All right. We now know how to define destructors, so it's time to learn **when to use them**.

Well, the answer is "**almost never**" and I'm quoting Eric Lippert, one of the designers of C#. As we learned before, if a class is using some unmanaged resources that must be cleaned up, it should implement the IDisposable interface. Some people think that having a destructor that calls the Dispose method can be an assurance that those resources will be cleaned up if someone forgets to call the Dispose method manually or with the using statement. But this is solving an issue that shouldn't happen at all if developers know what they are doing, and in the process, we may cause much more problems. Let me quote Mr. Lippert again:

"If you make a destructor *be extremely careful* and *understand how the garbage collector works*. Destructors are *really weird*:

- They don't run on your thread; they run on their own thread. Don't cause deadlocks!
- An unhandled exception thrown from a destructor is bad news. It's on its own thread; who is going to catch it?
- A destructor may be called on an object *after* the constructor starts but *before* the constructor finishes. A properly written

destructor will not rely on invariants established in the constructor.
- A destructor can "resurrect" an object, making a dead object alive again. That's really weird. Don't do it.
- **A destructor might never run**; you can't rely on the object ever being scheduled for finalization. It *probably* will be, but that's not a guarantee.

Almost nothing that is normally true is true in a destructor. Be really, really careful.

Writing a correct destructor is very difficult."

Then, he also states that this was the only scenario when he needed to actually write destructors:

"When testing the part of the compiler that handles destructors. I've never needed to do so in production code."

The quote comes from this thread on Stack Overflow:
https://stackoverflow.com/questions/4898733/when-should-i-create-a-destructor/4899622

If you want to learn more about the tricky beasts that destructors are, make sure to read this article by Eric Lippert:
https://ericlippert.com/2015/05/18/when-everything-you-know-is-wrong-part-one/

So the bottom line here is: do not write destructors. If your objects must clean up some resources after they finish their work, make them implement the IDisposable interface.

Let's summarize. The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

Bonus questions:

- "**What is the difference between a destructor, a finalizer, and the Finalize method?**"
  There is no difference, as they are the same thing. During the compilation, the destructor gets changed to the Finalize method which is commonly called a finalizer.

- "**Does the Garbage Collector call the Dispose method?**"
  No. The Garbage Collector is not aware of this method. We must call it ourselves, usually by using the using statement.

- "**When should we write our own destructors?**"
  The safest answer is "almost never". Destructors are very tricky and we don't even have a guarantee that they will run. Use IDisposable instead.

- "**What are managed and unmanaged resources?**"
  *The **managed resources** are managed by the Common Language Runtime. Any objects we create with C# are managed resources. The Garbage Collector is aware of their existence, and once they are no longer needed it will free up the memory they occupy. That means we don't need to worry about managed resources cleanup as it is done automatically for us. **Unmanaged resources** are beyond the realm of the CLR. The Garbage Collector doesn't know about them, so it will not perform any cleanup on them. Examples of unmanaged resources are database connections, file handlers, COM objects, opened network connections, etc. We as developers are responsible to perform the cleanup after we are done with those objects.*

11. What are default implementations in interfaces?

Brief summary: Starting with C# 8, we can provide methods implementations in interfaces. This feature was mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code.

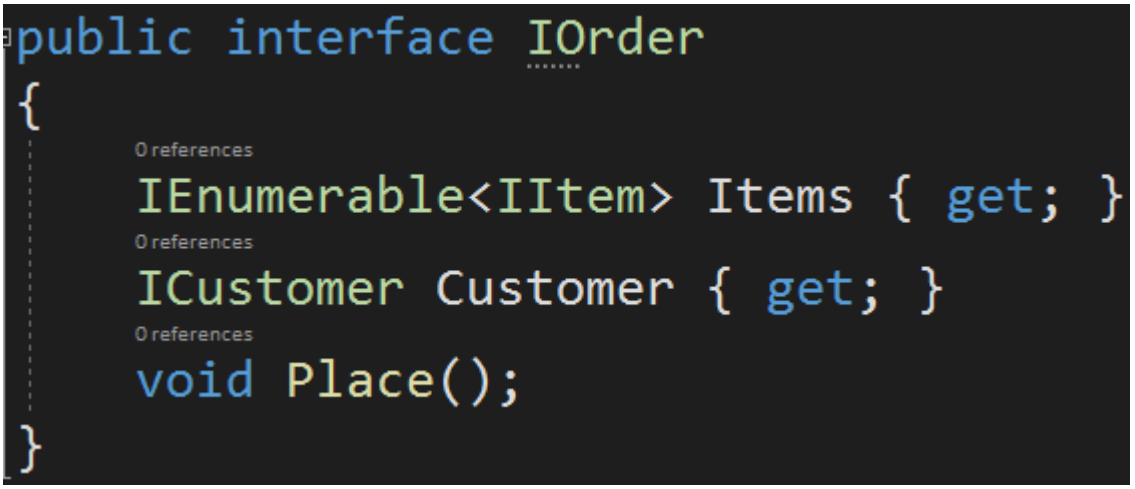
If I asked you "What are the characteristics of interfaces in C#?" you would probably be able to list them pretty easily:

- they can only contain methods, properties, indexers, and events declarations. They can't have fields.
- the methods declared in the interface can't have implementations. In other words, they are methods with no bodies.
- the methods can't be declared abstract or virtual (because they are implicitly virtual).
- all its members are by default public and using any other access modifier leads to a compilation error.
- they can't have static methods.

This is all perfectly correct… or rather was, until C# 8 was introduced in 2019. With this version of C# new feature was introduced: **default implementations in interfaces**. In short, it means that interfaces can now contain methods with bodies. Because of that a couple of other changes must have been introduced too - for example, methods in the interface can be private now, it can contain fields, etc.

If you (like me) are used to the "old" interfaces, you are probably quite surprised now. This change goes against everything we knew about them - that they are an abstraction over behavior, or that they only define a contract a class must fulfill. When I learned about this change my first thought was "so what will be the difference between an interface and abstract class now?". It seemed to me (correctly) that they will be very similar concepts from now on, so I was asking myself why did Microsoft decide to introduce this change.

So, before we dive into details, let's understand **why**. I will explain it by showing you an example. Let's say we develop some library that other people or companies will use. We will publish it as a NuGet package. Let's say this is one of the interfaces we define in our library, and we expect our customers to provide their own implementations:

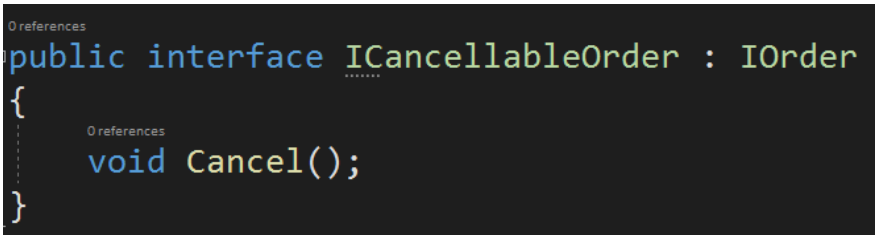


Our library is meant for e-commerce, and this interface defines what functionality should the classes representing orders contain.

We finish our work and we release our library. The release is a roaring success, and more and more people download the package with NuGet. It is widely used and highly rated.

One year later we are almost ready to release version 2.0. There is one problem, though. We would like to add “void Cancel();” method to the IOrder interface. But if we do so, and our customers upgrade the version of the library, they all will suddenly see compilation errors all-around their codebases. The classes they defined to implement the IOrder interface do not provide the implementation of the Cancel method. We will force our customers to adjust to this breaking change, and this may not be easy. Some of them may be stuck with development for days, and their business may be impacted by it.

One solution could be to extend the existing interface like this:



We will create a new interface extending the old one. Our customers can gradually start using it in their codebase, and everyone is happy.

But there is a problem with this solution. As our library evolves, we may want to add more and more methods to the IOrder interface. This will lead to creating new interfaces, and soon it will become hard to maintain. No one will wrap their heads

around what is what in the application, as everywhere we will see things like IOrder, ICancellableOrder, ICancellableOrderWithDeliveryDelay, ICancellableOrderWithDeliveryDelayAndDiscout, etc.

And this is where default implementations in interfaces can help. We can add new methods to an existing interface, and provide default implementations, so we won't break our customers' code. If they want to, they can provide their own implementation, but until then the default implementation will be used. Let's see this in code:

```
public interface IOrder
{
    1 reference
    IEnumerable<IItem> Items { get; }
    1 reference
    ICustomer Customer { get; }
    1 reference
    void Place();

    1 reference
    public void DelayDeliveryByDays(int days)
    {
        Console.WriteLine("DelayDeliveryByDays from interface");
    }
}
```

As you can see the DelayDeliveryByDays method has a body, which was impossible before C# 8.

Now, let's define a class implementing this interface:

```
class CustomOrder : IOrder
{
    1 reference
    public IEnumerable<IItem> Items { get; } = new List<IItem>();

    1 reference
    public ICustomer Customer {get;}

    1 reference
    public void Place()
    {
        Console.WriteLine("Placing an order");
    }
}
```

And let's see if we can call the DelayDeliveryByDays method on an object of this class:

```
CustomOrder order = new CustomOrder();
order.DelayDeliveryByDays(1);
```

Well, that doesn't compile. We can't use the default interface implementations on variables of a concrete type. We must use it via the interface:

```
IOrder orderByInterface = new CustomOrder();
orderByInterface.DelayDeliveryByDays(1);
```

Of course, if we provide the implementation of this method in the concrete class, it will be used instead of the implementation from the interface. Let's add another class implementing the IOrder interface:

```
class CustomOrderWithDelay : IOrder
{
    1 reference
    public IEnumerable<IItem> Items { get; } = new List<IItem>();

    1 reference
    public ICustomer Customer { get; }

    1 reference
    public void Place()
    {
        Console.WriteLine("Placing an order");
    }

    2 references
    public void DelayDeliveryByDays(int days)
    {
        Console.WriteLine("DelayDeliveryByDays from CustomOrderWithDelay");
    }
}
```

And now, let's see what this code will print:

```
IOrder order = new CustomOrder();
order.DelayDeliveryByDays(1);

IOrder orderWithDelay = new CustomOrderWithDelay();
orderWithDelay.DelayDeliveryByDays(1);
```

And the result is:

```
DelayDeliveryByDays from interface
DelayDeliveryByDays from CustomOrderWithDelay
```

As you can see, if the non-default implementation is provided, it will be used.

All right. Since we now can define methods in interfaces, we may need some other things that typically are used in methods, like:

- other, private methods that can enclose some piece of logic.
- static methods to do the same, if they don't use any non-static members of the interface.
- private fields.

Also, if an interface is derived from another interface, we may want to use the members of the parent. This means we can declare interface methods or fields as protected. The protected methods or fields are only available in derived interfaces, not classes implementing the interface.

We can also have virtual methods in the interface, but they can only be overridden by derived interfaces, not the classes implementing the interface. Also, if we declare a virtual method in the interface, it must contain a body. So for example, this method is defined in the base interface:

```
public interface IOrder
{
    2 references
    public virtual void VirtualMethodFromInterface()
    {
        Console.WriteLine("IOrder interface");
    }
```

And we can override it in the derived interface like this:

```
public interface ICancellableOrder : IOrder
{
    2 references
    void IOrder.VirtualMethodFromInterface()
    {
        Console.WriteLine("ICancellableOrder interface");
    }
```

All right. Let's summarize the topic of default implementations in interfaces.

This feature was added in C# 8. It's mostly designed to make it easier to add new methods to interfaces without breaking the existing code. Also, they make it possible for C# to work with APIs targeting Android (written in Java) and iOS (written in Swift) as those languages support similar features. They also enable using something called Traits, which is beyond this course's level; if you are curious, you can read about it here:
https://en.wikipedia.org/wiki/Trait_(computer_programming).
https://dlang.org/spec/traits.html
https://stackoverflow.com/questions/59547812/c-sharp-interface-with-default-method-vs-traits

As you can see this was a huge change and it completely changed what was true about interfaces in C#. This feature received a lot of criticism, and to be honest I can see why. The line between interfaces and abstract classes is very blurry now. In practice, it's hard to provide a default implementation that brings any value. I recommend this extensive article pointing out some problems with the default implementations in interfaces:
https://jeremybytes.blogspot.com/2019/09/interfaces-in-c-8-are-bit-of-mess.html

My recommendation is as follows: be aware that something like the default implementation in interfaces exists. Still, I think it's best if you use interfaces as they were meant to be used before C# 8 unless you are 100% sure you know what you are doing and equally sure that this will bring value to your application.

Bonus questions:

- "**What can be the reason for using default implementations in interfaces?**"
  Default implementations in interfaces are mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code. Without it, if we add a method to an interface we release it as a public library, we will force everyone who updates this library to provide the implementation immediately - otherwise, their code will not build.

12. What is deconstruction?

Brief summary: Deconstruction is a mechanism that allows breaking a tuple or a positional record into individual variables. It is also possible to define how deconstruction should work for user-defined types by implementing the Deconstruct method.

Deconstruction is a mechanism that allows breaking a tuple or a positional record into individual variables. It is also possible to define how deconstruction should work for user-defined types by implementing the Deconstruct method.

Deconstruction was first introduced with C# 7.

First, let's see some code.

```
(int sum, int count, double average) AnalyzeNumbers(
    IEnumerable<int> numbers)
{
    var sum = 0;
    var count = 0;
    foreach(var number in numbers)
    {
        sum += number;
        count++;
    }
    var average = (double)sum / count;
    return (sum, count, average);
}
```

This method takes a collection of integers and returns the **sum**, **count,** and **average** as a three-element tuple. For simplicity, I skipped handling empty collections. Now let's see how this method could be used:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var analysisResult = AnalyzeNumbers(numbers);

if (analysisResult.count == 0)
{
    Console.WriteLine("The collection is empty");
}
else
{
    Console.WriteLine($"The collection has {analysisResult.count}" +
      $" elements, with total sum of {analysisResult.sum} " +
      $"and average of {analysisResult.average}");
}
var numbersAverageSize = analysisResult.average > 100 ?
    "large" :
    "small";

Console.WriteLine(
    $"The numbers in the collection are " +
    $"relatively {numbersAverageSize}");
```

Since we use each of the tuple's elements quite often, let's store them in variables:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var analysisResult = AnalyzeNumbers(numbers);
var count = analysisResult.count;
var sum = analysisResult.sum;
var average = analysisResult.average;

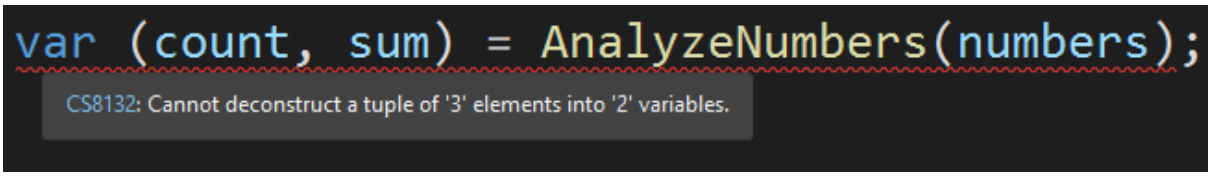
if (count == 0)
{
    Console.WriteLine("The collection is empty");
}
else
{
    Console.WriteLine($"The collection has {count}" +
      $" elements, with total sum of {sum} " +
      $"and average of {average}");
}
var numbersAverageSize = average > 100 ?
    "large" :
    "small";

Console.WriteLine(
    $"The numbers in the collection are " +
    $"relatively {numbersAverageSize}");
```

This works, but it's a bit cumbersome. It would be better if we could create those three variables in the same line the AnalyzeNumbers method is executed. And that's exactly what **deconstruction** is for. Let's see this in code:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var (count, sum, average) = AnalyzeNumbers(numbers);
```

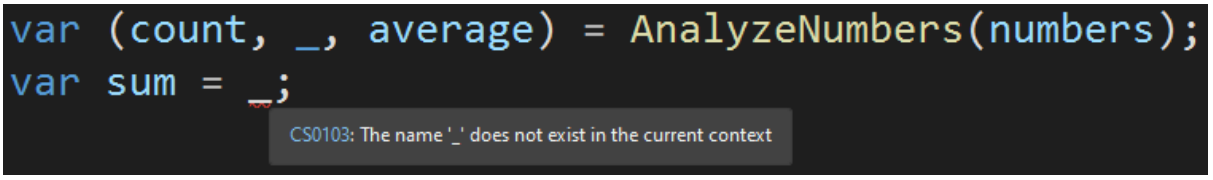
In the second line, we declared three variables and assigned the first element of the tuple to the first one, the second to the second one, and the third to the third one. The count of variables must be equal to the count of tuple elements. Because of that, the following code will not compile:



But we don't need to declare every variable if we don't want to. Let's say that for some reason I don't care about the second tuple's element, which is the sum. I can skip it by using the **discard**:

```
var (count, _, average) = AnalyzeNumbers(numbers);
```

Discard is a special, write-only variable, and we can't use it after it's assigned. Its only purpose is to be a placeholder for ignored elements of a tuple:



It is also possible to deconstruct tuples into variables that we already have. In this case, we just need to skip the "var" keyword:

```
int sum;
double average;
(sum, _ , average) = AnalyzeNumbers(numbers);
```

We can also mix using the existing variables with declaring new ones:

```
int sum;
double average;
(sum, var count, average) = AnalyzeNumbers(numbers);
```

All right. So far we've been deconstructing ValueTuples. We can also deconstruct ordinary tuples…

```
var tuple = new Tuple<string, bool, int>("abc", true, 10);
var (text, boolean, number) = tuple;
```

…as well as positional records:

```
var bob = new Person("Bob", 1950, "USA");
var (name, _, country) = bob;

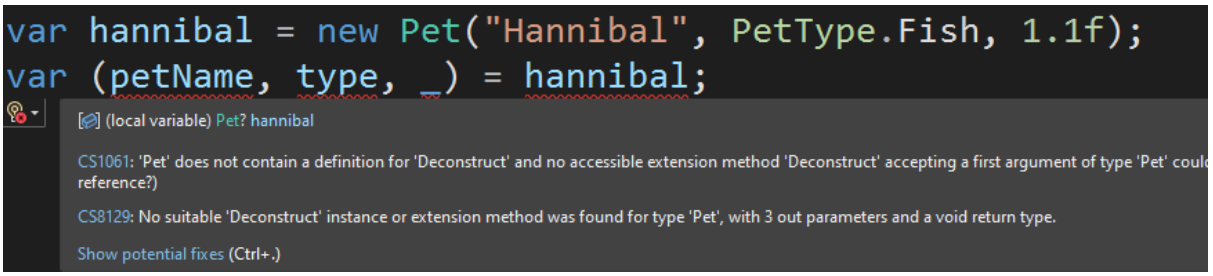
1 reference
record Person(string Name, int YearOfBirth, string Country);
```

Let's define a new class:

```
class Pet
{
    1 reference
    public string Name { get; }
    1 reference
    public PetType PetType { get; }
    1 reference
    public float Weight { get; }

    0 references
    public Pet(string name, PetType petType, float weight)
    {
        Name = name;
        PetType = petType;
        Weight = weight;
    }
}
```

Classes, by default, do not support being deconstructed:



But we can provide our own Deconstruct method to enable it. Such a method must be void, and it must have one **out** parameter for each variable that will be created as the result. Let's add the Deconstruct method to the Pet class:

```
0 references
public void Deconstruct(
    out string name,
    out PetType petType,
    out float weight)
{
    name = Name;
    petType = PetType;
    weight = Weight;
}
```

Now we can deconstruct the Pet object into three variables:

```
var taiga = new Pet("Taiga", PetType.Dog, 30f);
var (petName, petType, weight) = taiga;
```

We can define as many Deconstruct methods in a class as we want. We can also add the Deconstruct method to structs, records, and interfaces.

Even if we did not create some class and we don't have access to its source code, we can still "add" the Deconstruct method to it using **extension methods**. Let's see this in practice. Let's say I wished I could deconstruct a DateTime object:

```
var date = new DateTime(2020, 1, 8);
var (year, month, day) = date;
```

Unfortunately, this doesn't work, because DateTime does not have the Deconstruct method implemented. Let's fix it by defining the Deconstruct extension method:

```
static class DateTimeExtensions
{
    1 reference
    public static void Deconstruct(
        this DateTime date,
        out int year,
        out int month,
        out int day)
    {
        year = date.Year;
        month = date.Month;
        day = date.Day;
    }
}
```

Now the deconstruction works as expected:

```
var date = new DateTime(2020, 1, 8);
var (year, month, day) = date;
```

Bonus questions:

- "**What is the difference between the destructor and the Deconstruct method?**"
  The destructor is a method that's called on an object when this object is being removed from memory by the Garbage Collector. The Deconstruct method allows the object to be deconstructed into single variables. It is by default generated for tuples, ValueTuples, and positional records, but we can also define it in custom types.

- "**How can we define deconstruction for types that we did not create and we don't have access to their source code?**"
  We can define the Deconstruct method as an extension method for this type.

13. Why is “catch(Exception)” almost always a bad idea (and when it is not?)?

Brief summary: Using “catch(Exception)” should be avoided, because it catches every kind of exception. When we decide to catch an exception, we should know how to handle it, and it’s not feasible if the exception’s type is unknown. The acceptable use cases for catching any type of exceptions are:

- The global catch block that is catching all exceptions not handled elsewhere and shows them to the user.
- Any catch block in which we rethrow an exception without handling it.

Catching an object of System.Exception type, so the most general type of exceptions in C#, is almost always considered a bad idea. This is because when you catch an exception, you should handle it appropriately. But if you don’t know what the exception type is exactly, how can you know how to handle it?

Let’s consider the following code:

```
private static T? GetFirstOrDefault<T>(IEnumerable<T> items)
{
    try
    {
        return items.First();
    }
    catch (Exception)
    {
        Console.WriteLine("The collection is empty!");
        return default(T);
    }
}
```

This method works similarly to the parameterless FirstOrDefault method from LINQ - it returns the first element from the collection, but if the collection is empty, it returns the default value.

In the catch clause, we catch any exception and we print the information that the collection is empty. But other exceptions, not related to the emptiness of the

collection, can be thrown in the try clause, like OutOfMemoryException or StackOverflowException. For those two types, it is extremely hard to predict when they will happen (of course, in a healthy application they shouldn't happen at all, but we don't know how the rest of the application looks like). It may be the case that the catch clause will catch OutOfMemoryException, and it will handle it with the false message, saying that the collection is empty, even if it wasn't. The true problem - the lack of memory in the application - will be swept under the rug.

Because of that, we should always catch as specific exceptions as possible:

```
private static T? GetFirstOrDefault<T>(IEnumerable<T> items)
{
    try
    {
        return items.First();
    }
    catch (InvalidOperationException)
    {
        Console.WriteLine("The collection is empty!");
        return default(T);
    }
}
```

The First method throws InvalidOperationException, so handling it here is most appropriate.

For the same reasons, we should throw as specific exceptions as possible from our own code. Let's consider this method:

```
private static double Average(IEnumerable<int> numbers)
{
    if(numbers == null)
    {
        throw new Exception("The collection is null!");
    }
    double sum = 0;
    int count = 0;
    foreach (var number in numbers)
    {
        sum += number;
        count++;
    }
    if(count > 0)
    {
        return sum / count;
    }
    throw new Exception("The collection is empty!");
}
```

Throwing an Exception here is not a good idea. We should be more precise when choosing an exception type, so it fits the situation. When the first exception is thrown, the problem lies in the null collection passed to the Average method, so ArgumentNullException is a perfect fit. In the second case, when the collection is not null, but empty, InvalidOperationException or ArgumentException seem most appropriate:

```
private static double Average(IEnumerable<int> numbers)
{
    if(numbers == null)
    {
        throw new ArgumentNullException(
            "The numbers collection is null!");
    }
    double sum = 0;
    int count = 0;
    foreach (var number in numbers)
    {
        sum += number;
        count++;
    }
    if(count > 0)
    {
        return sum / count;
    }
    throw new InvalidOperationException(
        "The collection is empty!");
}
```

Exceptions give us priceless insight into what problems do we have in our application, and we should never mask them with some generic handling that will hide the detailed information they carry.

So is it ever appropriate to have the catch(Exception) clause?

Well, it is, in two specific scenarios.

The first one is **the global catch block** in our application. This is the last resort of exception handling. If something totally unexpected happens and there is no way of continuing the application's work after that, we should simply show the

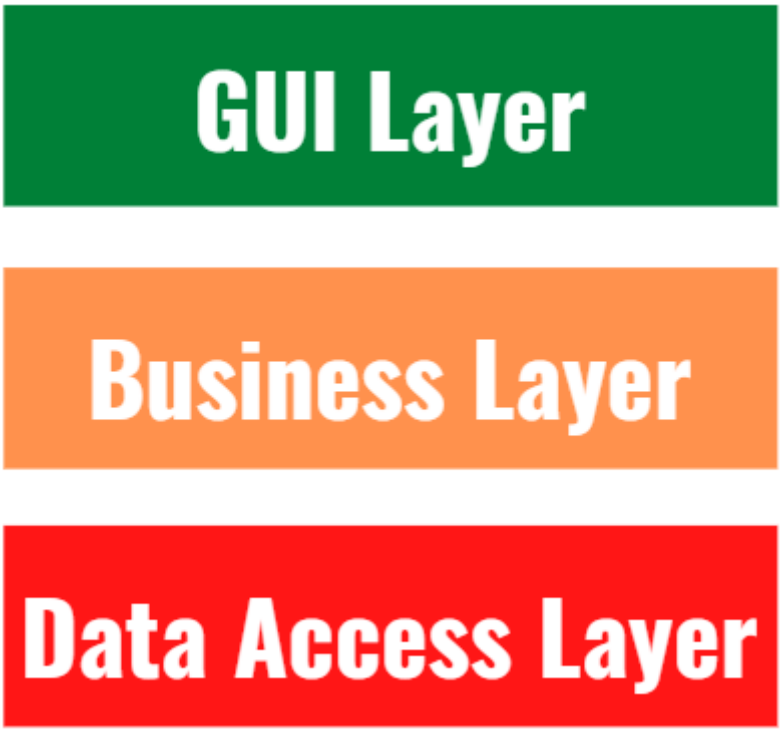
exception to the user and/or store it in some logs. After the user reads the error, the application will be stopped. Let's add a global catch block to a console application:

```
public static void Main(string[] args)
{
    try
    {
        var numbers = Enumerable.Empty<int>();
        var first = GetFirstOrDefault(numbers);

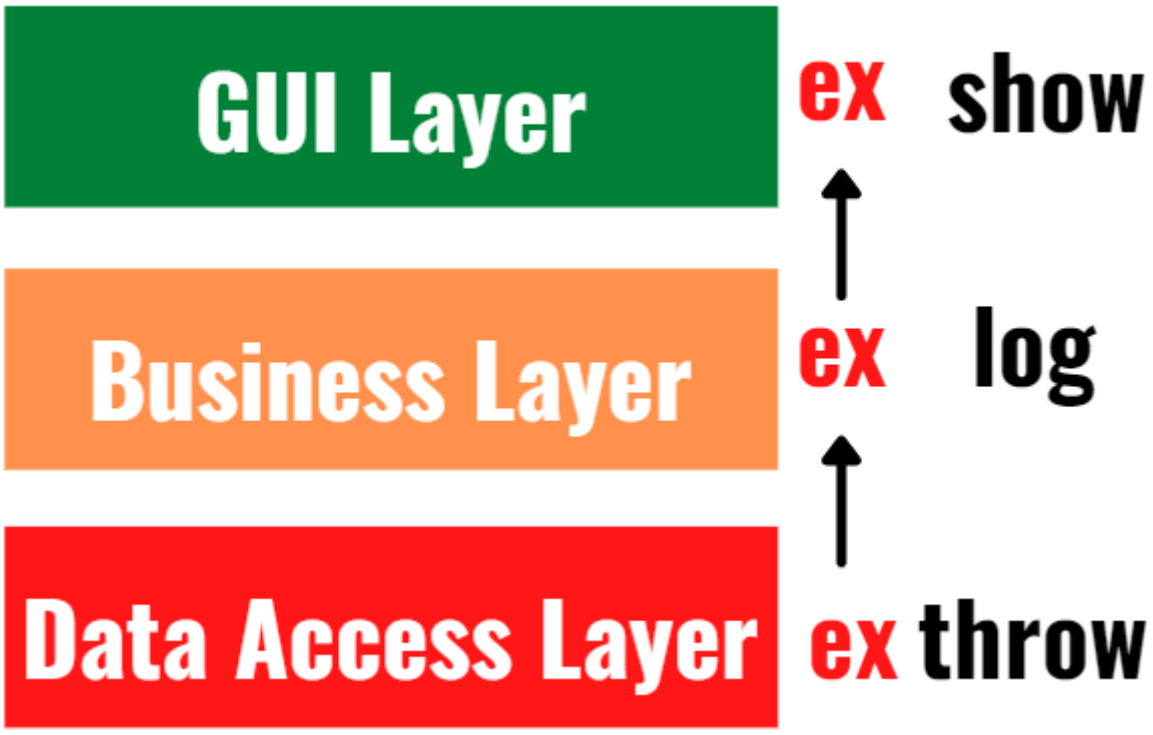
        //let's make the application crash on purpose:
        throw new Exception("Unknown exception");
    }
    catch (Exception ex)
    {
        //writing to some application logs would be appropriate here
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"An unexpected error happened and " +
            $"the application can't continue. The error message is" +
            $" '{ex.Message}', stack trace is : {ex.StackTrace}");
    }
    Console.ReadKey();
}
```

In applications with some proper GUI, such global catch block usually shows some error popup.

Another case when catching any type of exception could be OK is when we don't intend to handle it - we only rethrow it, possibly log it, or add some additional information to it. In such a tiny application as ours, it doesn't make much sense, but in big projects, it's often the case that applications are multi-layered, and each layer has its own way of reporting and organizing errors. This way, an exception thrown in the lower layer will be logged in each layer it crosses, but finally, it will be handled in some of the upper layers. Let's consider the following architecture:



If the exception is thrown at the Data Access Layer, it can be intercepted in the Business Layer, which logs it and then rethrows it. It is then handled in the GUI Layer by showing some popup with an error message to the user:



We will talk more about how to rethrow exceptions in the next lecture. In a simplified way, such code could look like this:

```
public class DataAccessLayer
{
    private readonly ILogger _logger;

    0 references
    public DataAccessLayer(ILogger logger)
    {
        _logger = logger;
    }

    1 reference
    public string GetRawData()
    {
        try
        {
            return " some raw data ";
        }
        catch (Exception ex)
        {
            _logger.LogError("Error in DataAccessLayer: " + ex);
            throw; //rethrowing the exception
        }
    }
}
```

This is the lowest-level layer. If an exception is thrown on data access, it is logged and rethrown. Later, it will be handled by the next layer - the Business Layer:

```
public class BusinessLayer
{
    private ILogger _logger;
    private DataAccessLayer _dataAccessLayer;

    0 references
    public BusinessLayer(
        DataAccessLayer dataAccessLayer,
        ILogger logger)
    {
        _dataAccessLayer = dataAccessLayer;
        _logger = logger;
    }

    1 reference
    public string GetProcessedData()
    {
        try
        {
            var rawData = _dataAccessLayer.GetRawData();
            return rawData.Trim();
        }
        catch(Exception ex)
        {
            _logger.LogError("Error in BusinessLayer: " + ex);
            throw; //rethrowing the exception
        }
    }
}
```

If Data Access Layer throws an exception here, or if the processing of the raw data does, the exception will be logged and rethrown. It will be truly handled (by showing some error to the user) in the upper-most layer - the GUI Layer:

```
public class GuiLayer
{
    private BusinessLayer _businessLayer;
    private ILogger _logger;

    0 references
    public GuiLayer(
        BusinessLayer businessLayer,
        ILogger logger)
    {
        _businessLayer = businessLayer;
        _logger = logger;
    }

    0 references
    public void ShowToUser()
    {
        try
        {
            var data = _businessLayer.GetProcessedData();
            Console.WriteLine($"Showing to user: {data}");
        }
        catch(Exception ex)
        {
            _logger.LogError("Error in GUI Layer: " + ex);
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"An unexpected error: '{ex.Message}'");
        }
    }
}
```

This way, the exception thrown at the lowest layer of the application will be logged at each layer, but it will only be handled at the GUI Layer. As you can see, at this point it is not rethrown, as this is the last place where it can be handled. Of course, if any of the lower layers could actually handle the exception and continue working without problem, it should not rethrow the exception, and it would never be shown in the error window.

Using “catch(Exception)” should be avoided, because it catches every kind of exception. When we decide to catch an exception, we should know how to handle it, and it’s not feasible if the exception’s type is unknown. We should be precise in both catching exceptions, as well as in throwing them. The acceptable use cases for catching any type of exceptions are:

- The global catch block that catches all exceptions not handled elsewhere, and shows them to the user.
- Any catch block in which we rethrow an exception without handling it.

Bonus questions:

- "**What are the acceptable cases of catching any type of exception?**"
  The acceptable use cases for catching any type of exceptions are:
  - *The global catch block that is catching all exceptions not handled elsewhere and shows them to the user.*
  - *Any catch block in which we rethrow an exception without handling it.*

- "**What is the global catch block?**"
  The global catch block is the catch block defined at the upper-most level of the application, that is supposed to catch any exceptions that hadn't been handled elsewhere. It usually logs the exception and shows some information to the user, before stopping the application.

14. What is the difference between “throw” and “throw ex”?

Brief summary: The difference between “throw” and “throw ex” is that “throw” preserves the stack trace (the stack trace will point to the method that caused the exception in the first place) while ”throw ex” does not preserve the stack trace (we will lose the information about the method that caused the exception in the first place. It will seem like the exception was thrown from the place of its catching and re-throwing)

When catching an exception we don’t always want to handle it and then let the program execution continue. Sometimes we want the exception to move on and perhaps be caught in the next catch clause, but we want to log something or do any other action related to this exception occurrence. Such a thing is called **exception re-throwing**. We can do it by either using “throw” or “throw ex”.

The difference between them is that:

- “throw” **preserves the stack trace** (the stack trace will point to the method that caused the exception in the first place)
- “throw ex” **does not preserve the stack trace** (we will lose the information about the method that caused the exception in the first place. It will seem like the exception was thrown from the place of its catching and re-throwing)

We will see this in the code in a second, but first, let’s be sure we understand what the **stack trace** is. The stack trace is a trace of all methods that have been called, that lead to a particular moment in code. Let’s consider the following code:

```
static void MethodA()
{
    Console.WriteLine("In method A");
    MethodB();
}

static void MethodB()
{
    Console.WriteLine("In method B");
    MethodC();
}

static void MethodC()
{
    Console.WriteLine("In method C");
    Console.WriteLine(Environment.StackTrace);
    Console.WriteLine();
}
```

As you can see MethodA calls MethodB which calls MethodC. In the MethodC we log the current state of the stack trace. Let's see what will be printed to the console if I call MethodA from the Main method of the program:

```
In method A
In method B
In method C
   at System.Environment.get_StackTrace()
   at Program.<<Main>$>g__MethodC|0_4() in C:\Users\marta\source\
repos\50InterviewQuestionsMid\ThrowVsThrowEx\Program.cs:line 70
   at Program.<<Main>$>g__MethodB|0_3() in C:\Users\marta\source\
repos\50InterviewQuestionsMid\ThrowVsThrowEx\Program.cs:line 64
   at Program.<<Main>$>g__MethodA|0_2() in C:\Users\marta\source\
repos\50InterviewQuestionsMid\ThrowVsThrowEx\Program.cs:line 58
   at Program.<Main>$(String[] args) in C:\Users\marta\source\rep
os\50InterviewQuestionsMid\ThrowVsThrowEx\Program.cs:line 1
```

All right. At the top of the stack trace we have the method that has been called most recently, and at the bottom - the one that has been called first. The stack trace stores the information about all method calls that lead to the current moment in the program execution. As you can see the getter of the

Environment.StackTrace property is at the very top because we read the stack trace with this method exactly, so it's the latest to be called. Please note that also the numbers of lines of code where those methods have been called are stored in the stack trace.

Stack trace has many uses, but for us as developers the most important value it brings is that it helps us track where some exceptions happened. Imagine that you have a huge app, and when it throws an exception all it says is "object is null!". That wouldn't be very helpful. We want to know the exact method that caused the problem, and even more - the exact line. This will help us to take a look at the right place, place a breakpoint there, and overall solve the problem easier and faster.

All right. We said that "throw" preserves the stack trace while "throw ex" does not - it's sometimes called "resetting the stack trace". Let's see the code that will show us the difference:

```
void MethodThrow()
{
    try
    {
        var collection = Enumerable.Empty<int>();
        var first = collection.First();
    }
    catch (Exception ex)
    {
        throw;
    }
}
```

Here is the first method. As you can see it's designed to throw an exception (it will try to access the first number of an empty collection). And here is the second - almost the same, but doing "throw ex" instead of throw:

```
void MethodThrowEx()
{
    try
    {
        var collection = Enumerable.Empty<int>();
        var first = collection.First();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

CA2200: Re-throwing caught exception changes stack information

Show potential fixes (Ctrl+.)

As you can see Visual Studio underlines “throw ex” and it has good reasons to do so. We will go back to it in a while. First, let’s see those methods in use. I will call both similarly, so below I’m showing the code for MethodThrow only for brevity.

```
try
{
    MethodThrow();
}
catch (Exception ex)
{
    Console.WriteLine("Throw");
    Console.WriteLine(
        "Exception caught, logging some" +
        " information. Stack trace:\n");
    Console.WriteLine(ex.StackTrace);
    Console.WriteLine();
}
```

All right. Let’s see the output of the program.

```
Throw
Exception caught, logging some information. Stack trace:

   at System.Linq.ThrowHelper.ThrowNoElementsException()
   at System.Linq.Enumerable.First[TSource](IEnumerable`1
 source)
   at Program.<<Main>$>g__MethodThrow|0_0() in C:\Users\m
arta\source\repos\50InterviewQuestionsMid\ThrowVsThrowEx\
Program.cs:line 37
   at Program.<Main>$(String[] args) in C:\Users\marta\so
urce\repos\50InterviewQuestionsMid\ThrowVsThrowEx\Program
.cs:line 5

Throw ex
Exception caught, logging some information. Stack trace:

   at Program.<<Main>$>g__MethodThrowEx|0_1() in C:\Users
\marta\source\repos\50InterviewQuestionsMid\ThrowVsThrowE
x\Program.cs:line 54
   at Program.<Main>$(String[] args) in C:\Users\marta\so
urce\repos\50InterviewQuestionsMid\ThrowVsThrowEx\Program
.cs:line 19
```

For throw, the stack trace ends at Linq.ThrowHelper.ThrowNoElementsException. The previous entry says about the First method, which already gives us some information about the nature of the problem.

For throw ex, the stack trace ends at MethodThrowEx line 54, which is exactly this line:

```
45  void MethodThrowEx()
46  {
47      try
48      {
49          var collection = Enumerable.Empty<int>();
50          var first = collection.First();
51      }
52      catch (Exception ex)
53      {
54          throw ex;
55      }
56  }
```

This means that all information that has been stored in the stack trace before reaching the “throw ex” command is lost. This is not good for us, as we lose

valuable data about the origins of the exception. This is why earlier we saw that Visual Studio suggested to us that using “throw ex” is not a very good idea. **We should stick to using “throw”, not “throw ex”.**

One may wonder “So why is there a possibility to do it in C# at all if we should not use it?”. Well, remember that “ex” is just an object of the Exception class. We throw objects belonging to this class all the time and it’s perfectly fine, only that we throw brand-new exceptions, not the ones that have been already thrown:

```
decimal Divide(decimal a, decimal b)
{
    if(b == 0)
    {
        throw new ArgumentException("B can't be zero!");
    }
    return a / b;
}
```

If the compiler allows us to use “throw (some exception here, should be brand-new)” it can’t really prevent us from throwing an already-thrown exception with “throw ex” even if it’s not a good idea.

Let’s summarize. The difference between “throw” and “throw ex” is that “throw” preserves the stack trace (the stack trace will point to the method that caused the exception in the first place) while ”throw ex” does not preserve the stack trace (we will lose the information about the method that caused the exception in the first place. It will seem like the exception was thrown from the place of its catching and re-throwing).

Bonus questions:

- "**What is the stack trace?**"
  The stack trace is a trace of all methods that have been called, that lead to the current moment of the execution. At the top of the stack trace we have the method that has been called most recently, and at the bottom - the one that has been called first. Stack trace allows us to locate the exact line in code that was the source of an exception.

- "**Should we use “throw” or “throw ex”, and why?**"
  We should use “throw” as it preserves the stack trace and helps us find the original source of the problem.

15. What is the difference between typeof and GetType?

Brief summary: Both **typeof** keyword and the **GetType** method are used to get the information about some type. The differences between them are:

- **typeof** takes the name of the type we want to inspect, so we must know the type before. **typeof** is resolved at compile time.
- **GetType** is a method that must be executed on an object. Because of that, it is resolved at runtime. This method comes from the System.Object base class, so it is available in any object in C#

Both **typeof** keyword and the **GetType** method are used to get the information about some type. The differences between them are:

- **typeof** takes the name of the type we want to inspect, so we must know the type before. **typeof** is resolved at compile time.
- **GetType** is a method that must be executed on an object. Because of that, it is resolved at runtime. This method comes from the System.Object base class, so it is available in any object in C#

Let's see this in practice. If I know the type already, and I only want to get the Type object for some reason, typeof and the GetType method will give me the same result:

```
var type1 = typeof(Base);
Console.WriteLine(type1.FullName);

var baseObj = new Base();
var type2 = baseObj.GetType();
Console.WriteLine(type2.FullName);
```

In the first case, I use the **typeof** with the Base type. In the second, I execute the GetType method on an object of this type. In both cases, the result is the Type object containing full information about the Base type.

```
type1: Base
type2: Base
```

But I don't always know the type at compile time. Let's consider this code:

```
void PrintTypeName(object obj)
{
    var type = obj.GetType();
    Console.WriteLine("type name is: " + type.FullName);
}
```

The obj may be anything, and it will only be known at runtime what it is exactly. That's why we can't use **typeof** here. We should rather use **GetType**. Let's see this method in action:

```
Derived derived = new Derived();
PrintTypeName(derived);

string text = "abc";
PrintTypeName(text);
```

```
type name is: Derived
type name is: System.String
```

The important thing to understand is that GetType always returns the actual type of an object. Let's consider this code:

```
Base derivedAsBase = new Derived();
var type4 = derivedAsBase.GetType();
Console.WriteLine("type4: " + type4.FullName);
```

Even if the variable **derivedAsBase** is of type Base, we assign an object of type Derived to it. It is possible because Derived inherits from Base. The GetType method will print the actual type:

```
type4: Derived
```

We actually have seen this before, in the PrintTypeName method. Even if it took a parameter of type System.Object, it printed the actual type of the given object. Remember that the GetType method belongs to System.Object type, so it can be called for any object in C#.

Let's summarize. Both **typeof** and the **GetType** method return a Type object, which holds the information about a type. **typeof** takes the name of the type, and it gets resolved at compile time. The GetType method is called upon an object, so it is resolved at runtime.

Both **typeof** and the **GetType** method are parts of the reflection mechanism, which we will learn about in the next lecture.

Bonus questions:

- "**What is the purpose of the GetType method?**"
  This method returns the Type object which holds all information about the type of the object it was called on. For example, it contains the type name, list of the constructors, attributes, the base type, etc.

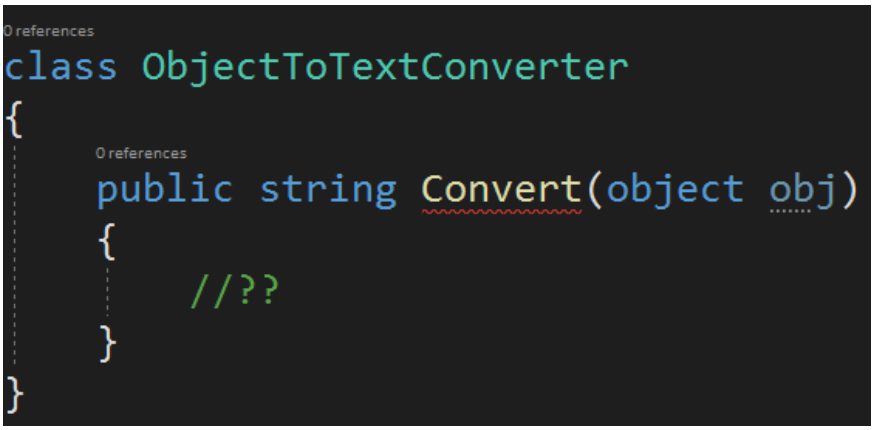
- "**Where is the GetType method defined?**"
  It is defined in the System.Object type, which is a base type for all types in C#. This is why we can call the GetType method on objects of any type.

16. What is reflection?

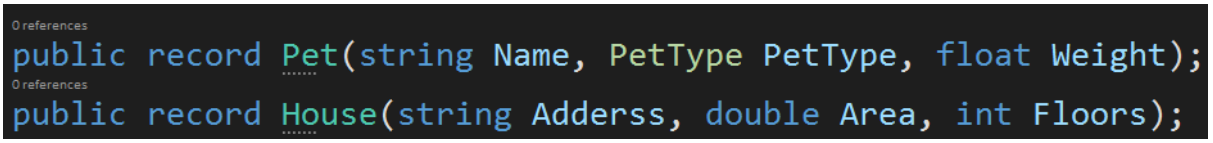
Brief summary: Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example, using reflection, we can list all fields and their values belonging to a given object, even if at compile time we don't know what type it is exactly.

Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example, using reflection, we can list all fields and their values belonging to a given object, even if at compile time we don't know what type it is exactly.

This all probably sounds a bit mysterious to you, so let's consider the following use case: we want to write a class that can take various objects and save them to a text file. There are already mechanisms that do it, and store objects are JSON or XMLs, but let's say we want some custom format so we must implement it ourselves. We want this class to be completely generic, so it can take any type of object.



Let's consider two sample types this class could convert to text. For brevity, defined them as records, which we will learn about later in the course.



If a Pet object is being converted, I would like the result to be for example:
"Name is Taiga, PetType is Dog, Weight is 30.0"
Similarly, for a House I would like to have:

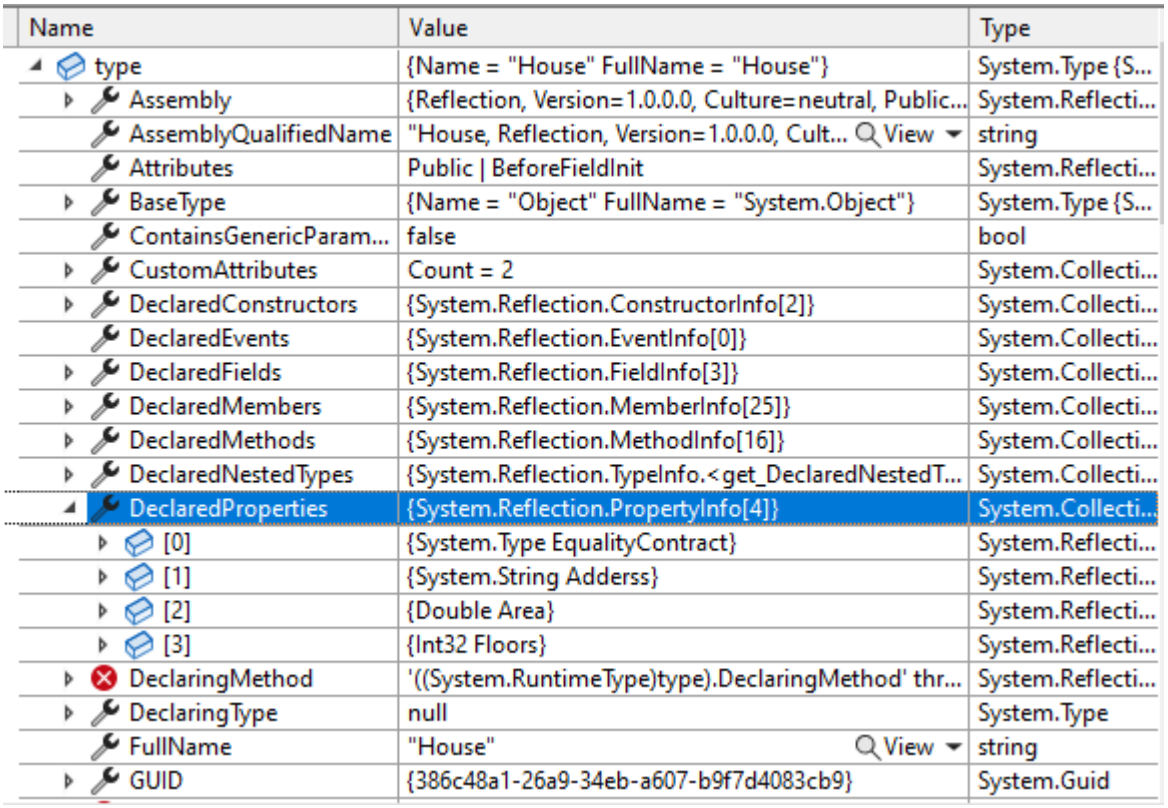
“Address is 123 Maple Road, Berrytown, Area is 170.6, Floors is 2”.

The problem is that in the Convert method, we have no idea what type we deal with. We can’t cast “obj” to anything concrete, as the types may vary. Also, to implement what we want we will not only need the values of the properties (which we could access if we only had more concrete type than System.Object) but we will also need the need their names, which is not available at runtime. In other words, when calling house.Floors we can get the number 2, but we can’t get the “Floors” string.

Well, actually, we can, but only if we use **reflection**. Reflection allows us to access information about some type at runtime. We can not only access the values of some fields but also their names. Moreover, we could access information about methods, constructors, access modifiers, and so on. Let’s see this in practice. First of all, we will use the GetType method on the obj object. It will return a Type object, which provides all information about a type:

```
0 references
public string Convert(object obj)
{
    Type type = obj.GetType();
}
```

Let’s see the **type** object in the debugger:



Name	Value	Type
type	{Name = "House" FullName = "House"}	System.Type {S...
Assembly	{Reflection, Version=1.0.0.0, Culture=neutral, Public...	System.Reflecti...
AssemblyQualifiedName	"House, Reflection, Version=1.0.0.0, Cult... View	string
Attributes	Public | BeforeFieldInit	System.Reflecti...
BaseType	{Name = "Object" FullName = "System.Object"}	System.Type {S...
ContainsGenericParam...	false	bool
CustomAttributes	Count = 2	System.Collecti...
DeclaredConstructors	{System.Reflection.ConstructorInfo[2]}	System.Collecti...
DeclaredEvents	{System.Reflection.EventInfo[0]}	System.Collecti...
DeclaredFields	{System.Reflection.FieldInfo[3]}	System.Collecti...
DeclaredMembers	{System.Reflection.MemberInfo[25]}	System.Collecti...
DeclaredMethods	{System.Reflection.MethodInfo[16]}	System.Collecti...
DeclaredNestedTypes	{System.Reflection.TypeInfo.<get_DeclaredNestedT...	System.Collecti...
DeclaredProperties	{System.Reflection.PropertyInfo[4]}	System.Collecti...
[0]	{System.Type EqualityContract}	System.Reflecti...
[1]	{System.String Adderss}	System.Reflecti...
[2]	{Double Area}	System.Reflecti...
[3]	{Int32 Floors}	System.Reflecti...
DeclaringMethod	'((System.RuntimeType)type).DeclaringMethod' thr...	System.Reflecti...
DeclaringType	null	System.Type
FullName	"House" View	string
GUID	{386c48a1-26a9-34eb-a607-b9f7d4083cb9}	System.Guid

As you can see there is quite a lot of data in here. We have some information about constructors, methods, base type, and also properties which I highlighted. We can see all properties we declared in the House type, and also an extra EqualityContract property which is is autogenerated for records. We will ignore it when converting the object to string.

All right. Let's use this data to achieve what we want. First, I want to read all properties from the given object, except the EqualityContract:

```
Type type = obj.GetType();

var properties = type
    .GetProperties()
    .Where(property => property.Name != "EqualityContract");
```

This gives me an IEnumerable<PropertyInfo>. Now I want to build a string for each PropertyInfos, accessing the property name as well as its value, and then join the strings together. I will use LINQ to do it:

```
return string.Join(
    ", ",
    properties
        .Select(property =>
        $"{property.Name} is {property.GetValue(obj)}"));
```

The Select method comes from LINQ, and it simply maps every property to a string.

All right. This is the final method:

```
public string Convert(object obj)
{
    Type type = obj.GetType();

    var properties = type
        .GetProperties()
        .Where(property => property.Name != "EqualityContract");

    return string.Join(
        ", ",
        properties
            .Select(property =>
            $"{property.Name} is {property.GetValue(obj)}"));
}
```

Let's make sure it works:

```
var converter = new ObjectToTextConverter();

Console.WriteLine(converter.Convert(
    new House("123 Maple Road, Berrytown", 170.6d, 2)));

Console.WriteLine(converter.Convert(
    new Pet("Taiga", PetType.Dog, 30)));
```

The result of this code is:

```
Adderss is 123 Maple Road, Berrytown, Area is 170.6, Floors is 2
Name is Taiga, PetType is Dog, Weight is 30
```

Great! Seems everything is working. We used reflection to access the information about some type at runtime and read the values and names of its properties.

Reflection gives us much more abilities. Here are some of them:

- loading dlls at runtime and using them
- instantiating a new instance of some object of a specific type at runtime. For example, we can create an object of a type defined in a dll we loaded reading private fields or properties, executing private methods (don't overuse it!)
- finding all classes derived from a specific base type or implementing a specific interface
- reading the attributes. This is for example what NUnit does when it runs the tests. It finds all methods with the [Test] attribute and executes them. We will learn more about attributes in the next lecture
- running a method by its name, for example, if the user of the application selected if from some dropdown
- debugging. For example, sometimes it is necessary to find out the list of currently loaded assemblies
- creating new types at runtime (System.Reflection.Emit namespace is used for that)
- and many more

As you can see reflection is a powerful tool, but as such should be used with caution. The code that heavily relies on reflection is usually hard to maintain and understand. It's also prone to errors. For example, when you call a method by its name, but someone changes the name without your knowledge, the code will crash the next time it's run because no method with the name exists anymore.

Also, Using reflection has a relatively big impact on **performance**. At one of the projects I worked on I was asked to improve the performance of some process. This application was using reflection a lot, mostly to load some types and attributes from dlls at runtime. It turned out that the results of those loads can be cached, and only this improvement made the process work twice as fast as before. We will learn more about this mechanism in the "What is caching?" lecture.

Use reflection with caution. If there is a convenient way of implementing the same logic without it, go for it. If not, reflection may be a lifesaver but keep an eye on the performance.

Let's summarize. Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example call a method with the name equal to a given string, or list all fields and their values belonging to a given object.

Bonus questions:

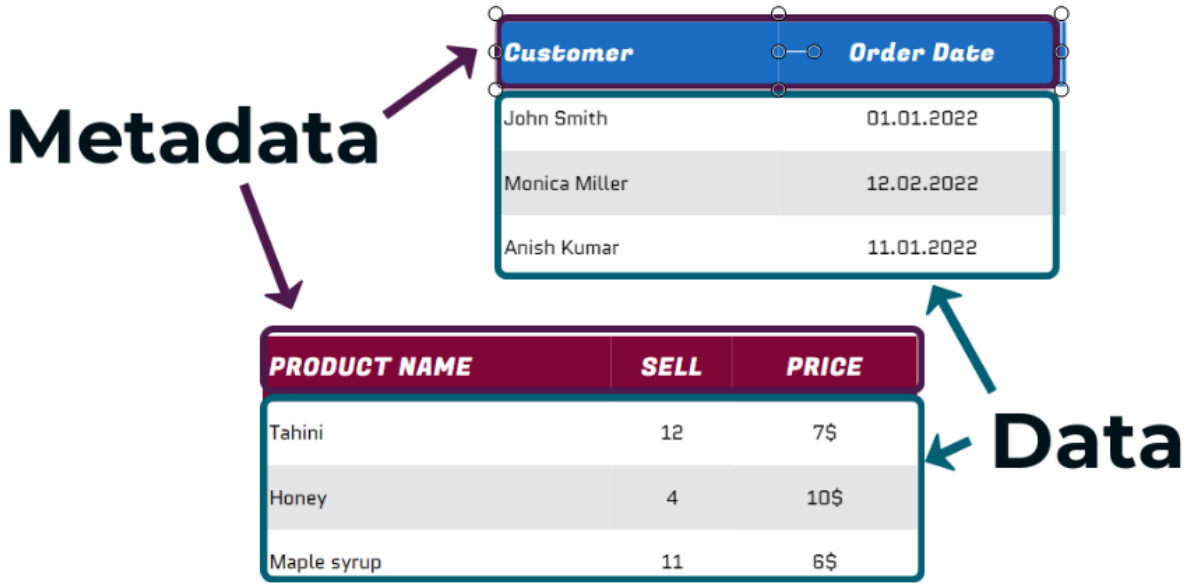
- "**What are the downsides of using reflection?**"
  Using reflection has a relatively big impact on performance. Also, it makes the code hard to understand and maintain. It may also tempt some programmers to "hack" some code, for example, to access private fields at runtime, which may lead to unexpected results and hard-to-understand bugs.

17. What are attributes?

Brief summary: Attributes add metadata to a type. In other words, they are a way to add information about a type or method to the metadata which describes that type or method.

Attributes add metadata to a type. In other words, they are a way to add information about a type or method to the existing metadata which describes that type or method, which we can read from the Type object.

First, let's understand what **metadata** is. Generally speaking, metadata is data providing information about other data. For example, when working with databases, the data stored inside the database is the actual data, while the structure of tables and relations between them is metadata.



In programming, metadata describes types used in an application.

First, let's consider this simple class:

```
public class Person
{
    2 references
    public string Name { get; }
    1 reference
    public int YearOfBirth { get; }

    0 references
    public Person(string name, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }

    0 references
    public Person(string name) => Name = name;
}
```

There is a lot of metadata describing this class. For example, the metadata contains the information that this class is named "Person", it is public, non-static, non-sealed, etc. It contains two get-only public properties called Name and YearOfBirth. It has one public constructor taking two string parameters, and one taking one parameter. The **actual data** stored in an instance of this class would be the string representing the name, and int representing the year of birth.

We can access all the class's metadata at runtime using **reflection**, which we learned about in the previous lecture.

```
var type = typeof(Person);
```

Sometimes we want to add extra metadata to a type or member, and this is what **attributes** are for.

Let's consider the following example. We want to have a common way of validating some data in the application. No matter the type, we want to be able to specify that its members of the string type must have a certain length. This is how it would look like:

```
1 reference
public class Dog
{
    1 reference
    public string Name { get; } //length must be between 2 and 10
    0 references
    public Dog(string name) => Name = name;
}
5 references
public class Person
{
    2 references
    public string Name { get; } //length must be between 2 and 25
    1 reference
    public int YearOfBirth { get; }
```

I want the Validator class to be able to take objects of **any class** and check if for any of their properties this validation is required. If so, it should check if the values of those properties are valid.

```
var validPerson = new Person("John", 1982);
var invalidDog = new Dog("R");
var validator = new Validator();

Console.WriteLine(validator.Validate(validPerson) ?
    "Person is valid" :
    "Person is not valid");
Console.WriteLine(validator.Validate(invalidDog) ?
    "Dog is valid" :
    "Dog is not valid");
```

All right. So what we want to do is to add some metadata to the Name properties in both Person and Dog types defining their minimal and maximal lengths. This is some “extra” metadata and to define it we must use a custom attribute. This is how it should look like:

```
2 references
public class Dog
{
    [StringLengthValidate(2, 10)]
    1 reference
    public string Name { get; }
    1 reference
    public Dog(string name) => Name = name;
}
4 references
public class Person
{
    [StringLengthValidate(2, 25)]
    2 references
    public string Name { get; }
```

As you can see, to add an attribute to a member or type, we simply must write its name in the brackets above the type or member we want to add it to. As you can see, the attribute we have requires two parameters - minimal and maximal length. Now, let's define the StringLengthValidateAttribute class.

```
class StringLengthValidateAttribute : Attribute
{
    3 references
    public int Min { get; }
    3 references
    public int Max { get; }

    2 references
    public StringLengthValidateAttribute(int min, int max)
    {
        Min = min;
        Max = max;
    }
}
```

All attributes must derive from the **Attribute** base class. Also, typically their names end with "Attribute". As you saw before, this postfix is omitted when we actually use the attribute:

```
[StringLengthValidate(2, 25)]
2 references
public string Name { get; }
```

One more thing. We can also define what the attribute can be applied to. In our case we want it to be applied to properties. To enforce that, we must actually use a built-in attribute called AttributeUsage:

```
[AttributeUsage(AttributeTargets.Property)]
7 references
class StringLengthValidateAttribute : Attribute
```

Great. Now all left to do is to define the Validator class.

```
class Validator
{
    2 references
    bool Validate(object obj)
```

This class will simply contain Validate method which can take any object. For this object, we will look for its properties with the StringLengthValidateAttribute defined.

```
bool Validate(object obj)
{
    var type = obj.GetType();
    var propertiesToValidate = type
        .GetProperties()
        .Where(property =>
            Attribute.IsDefined(
                property, typeof(StringLengthValidateAttribute)));
```

As you can see we selected the properties for which this attribute is defined using the LINQ's Where method along with Attribute.IsDefined method. Now, we can iterate those properties and check if their lengths are correct. But first, we must make sure that the property is a string. If not, we want to throw an exception, because it means that a developer added this attribute to a different type by mistake:

```
foreach (var property in propertiesToValidate)
{
    var attribute = (StringLengthValidateAttribute)
        property.GetCustomAttributes(
            typeof(StringLengthValidateAttribute), true).First();
    object? propertyValue = property.GetValue(obj);
    if(propertyValue is not string)
    {
        throw new InvalidOperationException(
            $"Attribute {nameof(StringLengthValidateAttribute)} " +
            $"can only be applied to strings.");
    }
```

Otherwise, we can validate the value:

```
if(propertyValue is not string)
{
    throw new InvalidOperationException(
        $"Attribute {nameof(StringLengthValidateAttribute)} " +
        $"can only be applied to strings.");
}
var value = (string)propertyValue;
if (value.Length < attribute.Min || value.Length > attribute.Max)
{
    Console.WriteLine($"Property {property.Name} is invalid. " +
        $"Value: {value}. The length must be between " +
        $"{attribute.Min} and {attribute.Max}");
    return false;
}
```

And this is the whole method:

```
bool Validate(object obj)
{
    var type = obj.GetType();
    var propertiesToValidate = type
        .GetProperties()
        .Where(property =>
            Attribute.IsDefined(
                property, typeof(StringLengthValidateAttribute)));

    foreach (var property in propertiesToValidate)
    {
        var attribute = (StringLengthValidateAttribute)
            property.GetCustomAttributes(
                typeof(StringLengthValidateAttribute), true).First();
        object? propertyValue = property.GetValue(obj);
        if(propertyValue is not string)
        {
            throw new InvalidOperationException(
                $"Attribute {nameof(StringLengthValidateAttribute)} " +
                $"can only be applied to strings.");
        }
        var value = (string)propertyValue;
        if (value.Length < attribute.Min || value.Length > attribute.Max)
        {
            Console.WriteLine($"Property {property.Name} is invalid. " +
                $"Value: {value}. The length must be between " +
                $"{attribute.Min} and {attribute.Max}");
            return false;
        }
    }
     return true;
}
```

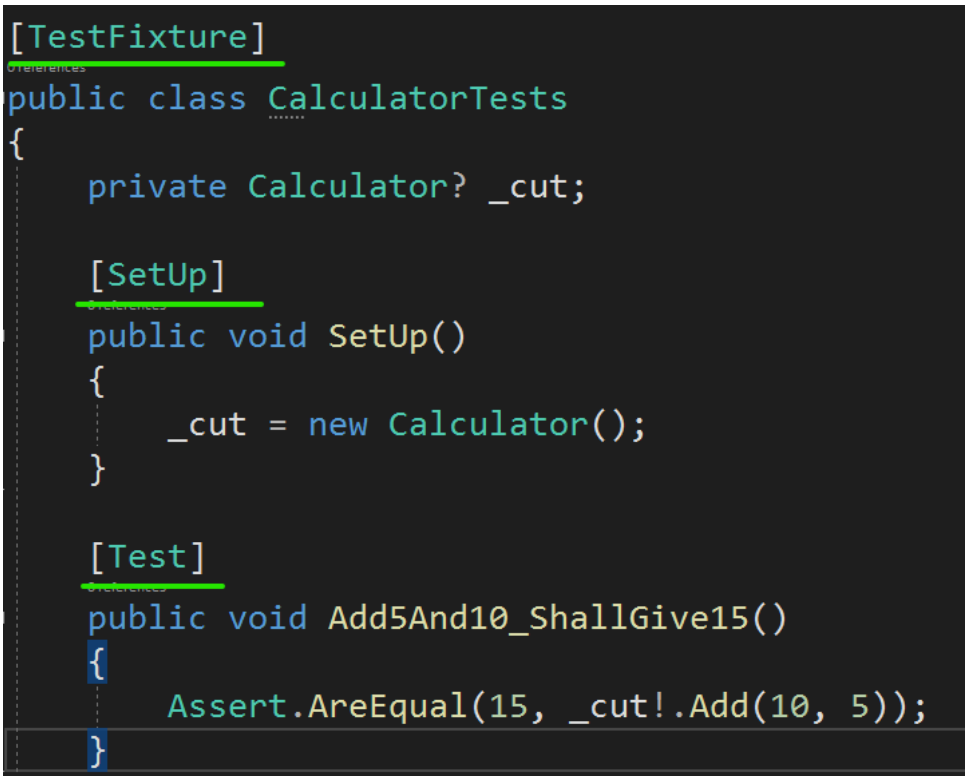
All right. Let's make sure it works:

```
Person is valid

Property Name is invalid. Value: R. The length must be
 between 2 and 10
Dog is not valid
```

Great! That's what we wanted.

As you can see attributes can be quite powerful. They are widely used in native .NET classes, as well as external libraries. For example, if you ever used NUnit, you must have used some of its attributes:



Let's summarize. Attributes add metadata to a type. In other words, are a way to add information about a type or method to the metadata which describes that type or method. To add an attribute to a member or type, we simply must write its name in the brackets above the type or member we want to add it to. There are plenty of built-in Attributes in C# standard library, but we can also create attributes of our own, simply by creating classes derived from the Attribute base class.

Bonus questions:

- "**What is metadata?**"
  Generally speaking, metadata is data providing information about other data. For example, when working with databases, the data stored inside the database is the actual data, while the structure of tables and relations between them is metadata. In programming, metadata describes types used in an application. We can access it in the runtime using reflection, to get the information about some type, for example, what methods or what constructors it contains.

- "**How to define a custom attribute?**"
  To define a custom attribute we must define a class that is derived from the Attribute base class.

18. What is serialization?

Brief summary: Serialization is the process of converting an object into a format that can be stored in memory or transmitted over a network. For example, the object can be converted into a text file containing JSON or XML, or a binary file.

Serialization is the process of converting an object into a format that can be stored in memory or transmitted over a network. For example, the object can be converted into a text file containing JSON or XML, or a binary file.

Deserialization is the opposite process - using the content of a file to recreate objects.

Let's write a program that reads personal data from the console, and then serializes it as an XML file. If the user restarts the program, this data can be reconstructed using the XML file stored in the computer's memory. Also, we could transfer this file to another computer, where it could also be used to recreate the object containing personal data. Let's see this in the code:

```
var name = Read("name");
var lastName = Read("last name");
var residence = Read("place of residence");
var hobby = Read("hobby");

var person = new Person(name, lastName, residence, hobby);

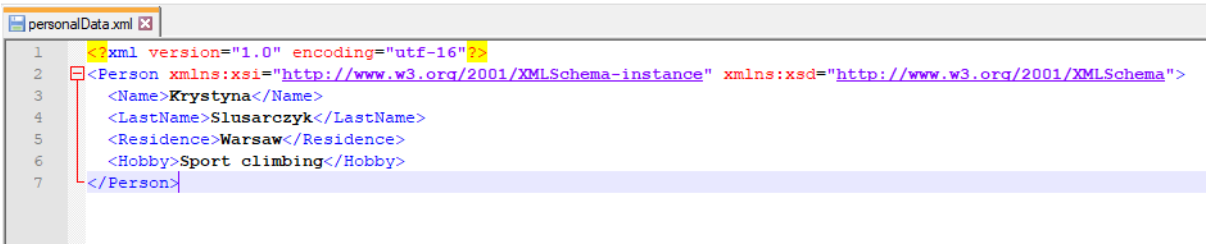
string xml = Serialize(person);
Console.WriteLine(
    $"The person object serialized to XML is:\n{xml}");

SaveXmlToFile("personalData.xml", xml);
```

```
static string Read(string what)
{
    Console.WriteLine($"Enter the {what}:");
    return Console.ReadLine();
}
```

For brevity, I skipped the code that does actual serialization and file writing. I used the built-in XmlSerializer class for serialization. Full code can be found in the solution published to Github.

I've run this code and entered some personal data. The file that was produced looks as follows:



As you can see, all the information that was stored in the Person object is saved to the XML file. We should now be able to read it in the program. Let's change the code so if the "personalData.xml" file exists, it reads its content instead of asking the user to enter the data.

```
const string filePath = "personalData.xml";
if (File.Exists(filePath))
{
    using Stream reader = new FileStream(filePath, FileMode.Open);
    var xmlSerializer = new XmlSerializer(typeof(Person));

    StreamReader stream = new StreamReader(reader, Encoding.UTF8);
    var person = (Person?)xmlSerializer.Deserialize(
        new XmlTextReader(stream));
    Console.WriteLine($"Personal data read from xml:\n{person}");
}
else
{
    var name = Read("name");
    var lastName = Read("last name");
```

The code checks if the file already exists - if so, it reads its content and deserializes it to recreate the Person object.

Not only XML can be used as the format to store objects. One of the most common formats is JSON. The name comes from JavaScript Object Notation because it derives from JavaScript objects format. JSON is typically used for communication over a network. For example, when you fill a form on a website, most likely the data from the form is wrapped in JSON format and sent to the server, which then reads it and (in the case of the C# backend) translates it to C# objects.

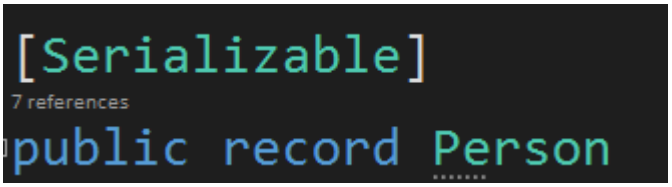
This is how the data showed previously as XML would look in JSON format:

```
{
  "Name": "Krystyna",
  "LastName": "Slusarczyk",
  "Residence": "Warsaw",
  "Hobby": "Sport climbing"
}
```

Probably the most popular library used for JSON serialization and deserialization is **JSON.Net** developed by **Newtonsoft**. It allows us to serialize and deserialize objects to/from JSON format very easily:

```
string json = JsonConvert.SerializeObject(person, Formatting.Indented);
Person recreatedPerson = JsonConvert.DeserializeObject<Person>(json);
```

One more thing that we should mention on this topic. The interviewer can ask you "What does the Serializable attribute do?".



This attribute indicates that instances of a class can be serialized with BinaryFormatter or SoapFormatter. It is not required for XML or JSON serialization. The BinaryFormatter serializes objects to a binary format (so, simply speaking, a chain of zeros and ones) and the SoapFormatter to the SOAP format, which is a little similar to XML. If you are curious, check out this article:
https://pl.wikipedia.org/wiki/SOAP

Let's summarize. Serialization is a process of translating objects and other data structures into a format that can be stored as a file or binary data, and potentially transmitted over a network. Serialized objects can later be reconstructed.

Bonus questions:

- "**What are the uses of serialization?**"
  It can be used to send objects over a network, or to store objects in a file for later reconstruction, or even to store them in a database - for example to save a "snapshot" of an object every time a user makes some changes to it, so we can log the history of the changes.

- "**What does the Serializable attribute do?**"
  This attribute indicates that instances of a class can be serialized with BinaryFormatter or SoapFormatter. It is not required for XML or JSON serialization.

- "**What is deserialization?**"
  Deserialization is the opposite of serialization: it's using the content of a file to recreate objects.

19. What is pattern matching?

Brief summary: Pattern matching is a technique where you test an expression to determine if it has certain characteristics.

Pattern matching is a technique where you test an expression to determine if it has certain characteristics.

The easiest way to understand pattern matching is with an example. Let's say I want to run some code if some value is null, and other if it isn't:

```
string NullCheck(object obj)
{
    if (obj == null)
    {
        return "Object is null!";
    }
    else
    {
        return "Object is null not null: " + obj.ToString();
    }
}
```

This code is pretty straightforward. There is only one problem - if I wouldn't know exactly what type the obj variable is, it might turn out that it has the == operator overloaded and that it actually does something else than simply checking if the value is null. To avoid this problem we can use **null check** pattern matching.

```
string NullCheck(object obj)
{
    if (obj is null)
    {
        return "Object is null!";
    }
```

All right. So far pattern matching seems very simple. It allowed us to check if an object **is null**. But it can give us many, many more abilities. Let's walk through some examples.

One of the most commonly used patterns is the **type test**. I want to run some code if a variable is of some type. Moreover, if it is, I want to cast it to this type. Without pattern matching, I would need to write something like this:

```
void TypeCheck(object obj)
{
    if(obj.GetType() == typeof(string))
    {
        var asString = obj as string;
        Console.WriteLine("String is: " + asString);
    }
    else
    {
        Console.WriteLine("Obj is not a string");
    }
}
```

With pattern matching, I can check if an object is a string and cast it in one line:

```
void TypeCheck(object obj)
{
    if(obj is string asString)
    {
        Console.WriteLine("String is: " + asString);
    }
    else
    {
        Console.WriteLine("Obj is not a string");
    }
}
```

This is quite convenient. Please note that the **asString** variable will be available only if the **obj** is a string, so I would not be able to use it anywhere else than inside the **if** statement.

We can also check some **particular properties** of the checked object:

```
string Properties(object obj)
{
    if (obj is Pet { Weight: > 10000, TypeOfPet: PetType.Fish })
    {
        return "It must be a whale shark!";
    }
    if(obj is Pet)
    {
        return "It's some kind of pet.";
    }
    return "It's definitely not a pet.";
}
```

Here we checked if an object is a Pet with Weight larger than 10000 and PetType equal to Fish.

All right. The next type of pattern matching is **comparing discrete values**. This is very similar to using a plain old switch statement. Let's say I have a method taking a string that should represent a number, and another string saying what type of number it is (int, decimal, or float). Depending on the second parameter I want to convert the first parameter to the given type:

```
object ComparingDiscreteValues(string number, string type)
{
    return type switch
    {
        "int" => int.Parse(number),
        "decimal" => decimal.Parse(number),
        "float" => float.Parse(number),
        _ => throw new ArgumentException($"{type} type is not supported"),
    };
}
```

Please notice the special "_" case. This is a **discard pattern** and it works similarly as **default** in the switch statement. It will be executed if the type parameter is not equal to any of the specified values.

Let's get to more complex types of pattern matching. The next one is a **relational pattern**. It allows us to check how a given value compares to constants:

```
string Relational(int age)
{
    return age switch
    {
        (> 20) and (< 60) => "middle-aged",
        < 20 => "teenager",
        > 60 => "senior",
    };
}
```

The cool thing about pattern matching is that it has very good IDE and compiler support, and we get an error when we do something silly. For example, let me add some more cases here:

```
string Relational(int age)
{
    return age switch
    {
        (> 20) and (< 60) => "middle-aged",
        < 20 => "teenager",
        > 60 => "senior",
        <11 => "child",
        18 => "just an andult, at least in some countries",
    };
}
```

This code doesn't compile, because the last two cases are unreachable. The cases are executed from top to bottom, so when the age parameter is 10, we will hit the "less than 20" case. We will never reach the "less than 11" case. Let's fix the order of the cases:

```
string Relational(int age)
{
    return age switch
    {
        18 => "just become adult, at least in some countries",
        (> 20) and (< 60) => "middle-aged",
        < 11 => "child",
        < 20 => "teenager",
        > 60 => "senior",
    };
}
```

Great. Now, this should work as expected. This code actually demonstrates one more pattern - a **logical pattern**. We used it when we checked if the age is less than 20 **and** more than 60.

We can also use pattern matching with deconstruction. Check out the “What is deconstruction?” lecture to learn more.

```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, TypeOfPet: PetType.Dog, Weight: 10) => "Small dog of any name",
        (Name: "Hannibal", TypeOfPet: PetType.Fish, _) => "Fish called Hannibal",
        _ => "Unknown!"
    };
}
```

We could omit the parameter names (but personally I would rather leave them for readability).

```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, PetType.Dog, 10) => "Small dog of any name",
        ("Hannibal", PetType.Fish, _) => "Fish called Hannibal",
        _ => "Unknown!"
    };
}
```

We can also mix deconstruction with checking particular properties:

```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, TypeOfPet: PetType.Dog, Weight: 10) => "Small dog of any name",
        (Name: "Hannibal", TypeOfPet: PetType.Fish, _) => "Fish called Hannibal",
        Pet { Weight: >100 } => "Heavy pet",
        _ => "Unknown!"
    };
}
```

All right. We learned some of the most basic usages of pattern matching. There is also a question when to use them, and when to use plain old if and switch statements. In my opinion, you should simply use those that you find more readable. You can mix both to get what's best in any of them.

To summarize: pattern matching is a technique where you test an expression to determine if it has certain characteristics.

Bonus questions:

- "**How can we check if an object is of a given type, and cast to it this type in the same statement?**"
  We can use pattern matching for that. For example, we could write "if obj is string text". This way, we will cast the object to the string variable called text, but only if this object is of type string.

20. How does the binary number system work?

Brief summary: The binary number system is used to represent numbers using only two digits - 0 and 1. For example, the number 13 (in the decimal number system) is 1101 in the binary number system. All data in a computer's memory is stored as sequences of bits, and so are all numbers.

The binary number system is used to represent numbers using only two digits - 0 and 1. For example, the number **13** (in the **decimal** number system) is **1101** in the **binary** number system.

As you probably know, every piece of data is stored in the computer's memory as a series of bits. Bit is the smallest unit of information and it can only have two values: 0 or 1. That means, every information we want to store in the computer's memory - a number, string, a complex object, or an entire program - is, in the end, stored as a series of zeros and ones. In this lecture, we will focus on numbers.

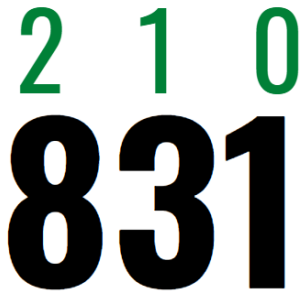
As we mentioned, the binary number system represents numbers as zeros and ones, so it fits perfectly how data is stored in a computer's memory. You may think that the binary number system is not something you need to understand, as it all happens under the hood. After all, there are programmers all around the world who have no idea how the binary number system works and they are doing fine.

But there are a lot of aspects of programming that are affected by how binary numbers work, and it's not possible to understand some of the programming caveats without knowing the binary system at all. If you are not convinced, let me give you a little spoiler from the next lecture: we will talk about how a banking application's client can lose all protection granted by daily transaction limits and in the end, have the account cleaned out by someone who accessed it illegally. It could be avoided if the programmer understood how operations on binary numbers work.

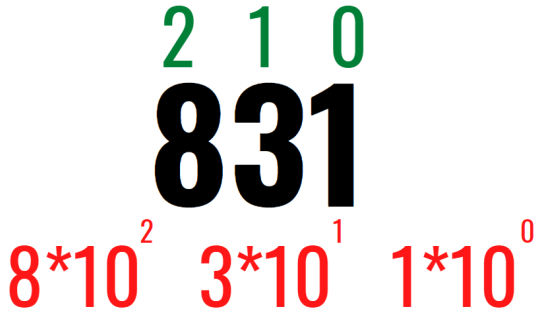
All right. Before we try to understand the binary number system, let's do so with the system we use on daily basis - the **decimal number system**. The base of this system is number 10. Actually, you can build a valid number system based on any number larger than 0, but 10 was probably most natural for the human race as we have 10 fingers, and we started our journey of understanding mathematics by counting them.

All right. Let's consider the following decimal number:

You probably don't need much explaining here - you simply know what this number is. You can imagine what it means to have 831 dollars (or any other currency you use), a folder with 831 pictures, or a book with 831 pages. We are so used to this system that we don't even think about the numbers - we simply see them and know by instinct what they mean. But let's break it down. Each digit has its place. The further to the left it is, the more significant it is - it means, it carries more "weight" of the number. 8 here means 800, 3 means 30 while 1 simply means 1. We could mark each digit with an index, counting from right to left:



Now, for each of the digits, we want to calculate 10 to the power of the index multiplied by the digit itself.



The sum of those numbers is the final number we want to represent. Let's make sure of that. 8*100 + 3*10 + 1*1 is 831. (Remember that any number to the power of zero is 1).

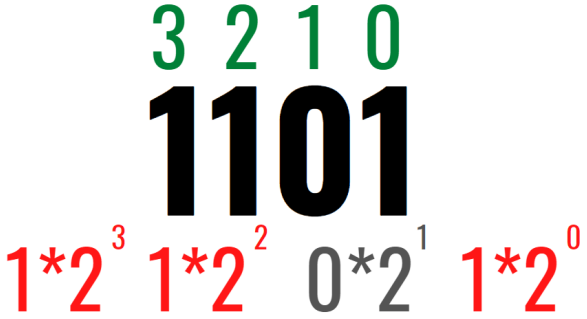
Now it is clear why numbers most to the left are most significant - because we will multiply them by 10 to the largest power.

Great. We now understand exactly how the decimal number system works. Let's move on to the **binary number system**. It actually works almost the same. The only difference is the base of the system. It will not be 10, but 2.

Let's consider this number:

This time you probably don't "feel" what the number means, but don't worry. We will figure it out in a second. Let's start the same as before - by marking each digit with its index, starting from the right.

In the decimal number system, we calculated the powers of 10 and then multiplied them by the digit itself. Here it's the same, but we calculate the powers of 2.



Let's calculate the sum. It's 8 + 4 + 0 + 1, which gives 13. That means, 1101 in the binary number system is 13 in the decimal number system.

Great. This gives us the basics that are needed to understand some operations related to programming.

The important thing to realize is that on a **limited number of bits we can store a limited number** (the same as in a decimal number system - for example the biggest number represented with 3 digits is 999). For example, with 4 bits the largest number that can be represented is 15 (because if each bit is set to one, then the number is 8 + 4 + 2 + 1 = 15). Each numeric type in C# occupies a certain number of bits in the memory. For example, an integer takes 32 bits. The largest number we can represent with int is 2147483648, which is a little over two billion.

And here is something interesting - this number is actually 2 to the power of 31, not 32! So what happened with one bit? Well, remember that with integers we can also represent negative numbers. This one bit is saved to store information whether the number is negative or not, which leaves us 31 bits for the actual number.

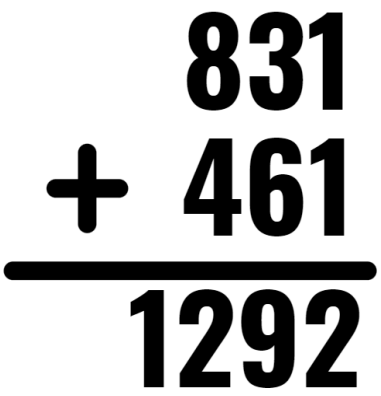
Here are sizes and ranges of the integral numeric types used in C#:

TYPE	BITS	MIN	MAX
sbyte (signed byte)	8	-128	127
byte	8	0	255
short	16	-32,768	32,767
ushort (unsigned short)	16	0	65,535
int	32	-2,147,483,648	2,147,483,647
uint (unsigned int)	32	0	4,294,967,295
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
ulong	64	0	18,446,744,073,709,551,615

All right. There is one more thing we must understand. Since each numeric type has its size limit and it simply can't represent a number that is larger, what happens when some arithmetic operation exceeds this limit?

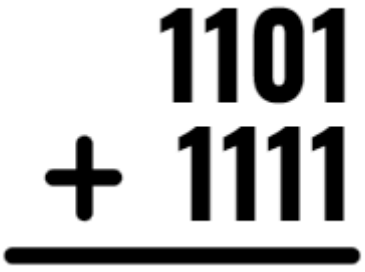
Well, in such situations, something quite interesting happens. For example, if I add 2 billion to two billion when operating on ints, I will get the result of -294967296. In this lecture, I will only explain to you how it works. In the next one, we will learn how to handle such situations when programming.

Before we can understand what happens, we must understand how adding binary numbers work. But as before, let's start with decimal numbers for simplicity:

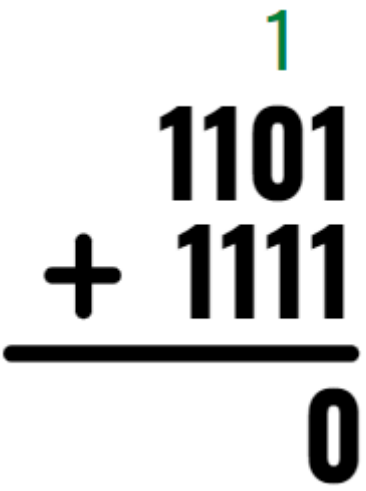


You probably know this technique of adding numbers. If not, please read this article first:
https://www.tutorialspoint.com/add_and_subtract_whole_numbers/addition_of_two_2digit_numbers_with_carry.htm

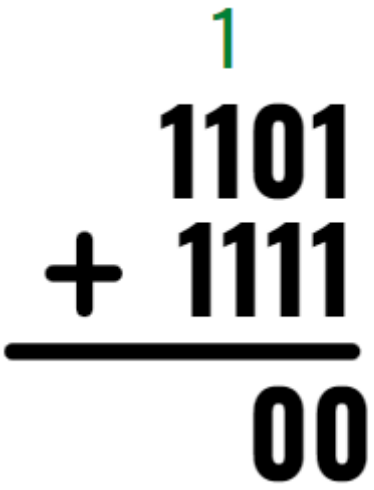
With binary numbers, it works the same. Let's add binary 13 to binary 15. Remember, 13 is 1101 and 15 is 1111:



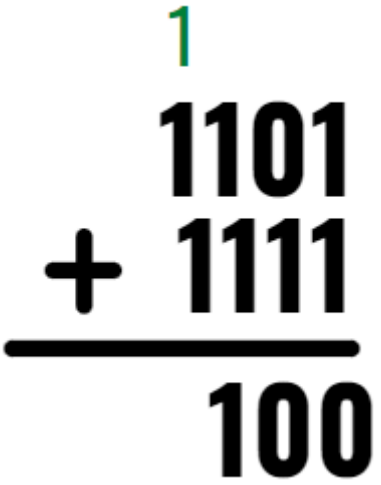
First, we add numbers from the first column from the right. 1+1 is 2, but we can't use 2 in the binary numbers system. That means, we need to carry it over to the next column. We will write 0 in the first column of the result because the modulo of the sum we calculated (2) and the base of the system (also 2) is 0.



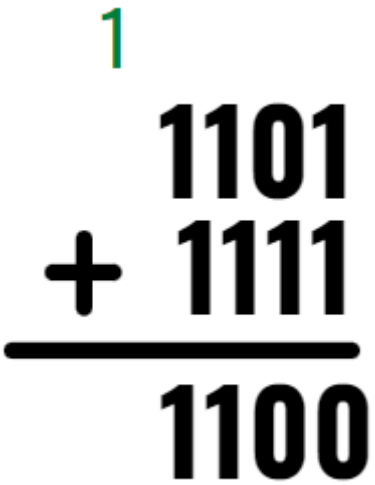
Now the second column. Again, the sum is 2, so we carry over to the next column again.



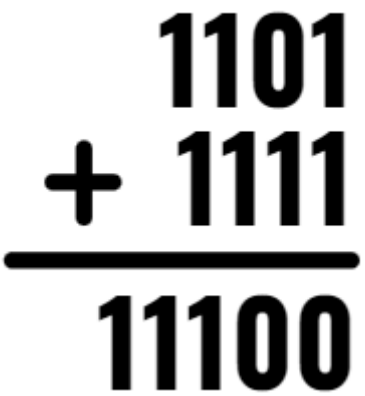
The third column. Now the sum is 3. We carry over 1 to the next column, and we leave 1 in the result. This is because the modulo of the sum we calculated (3) and the base of the system (2) is 1.



Finally, the fourth column. The sum is 3 again, so we carry over 1, and we leave 1 in the result:



It turned out that we actually need the fifth column to fit the 1 that we carried from the fourth column. This time it's simple. The sum is 1 and we add it to the result:



All right! We have our result. It's 16+8+4+0+0 = 28. This is correct because 13 + 15 is also 28.

But notice a very important thing - we needed to use one more digit to represent this number. Now, let's go back to thinking about computers. If we had a numeric type that only has 4 bits, it would simply not be able to hold the result we had. So what would happen? Well, the last, most significant bit would just be discarded. And the actual result the computer could see would not be 11100 which is 28, but 1100 which is 12 - something completely different and simply wrong from the arithmetics point of view.

Now you know why adding two billion to two billion gave some weird number before. If you are curious why it was negative, remember that the most significant bit represents a sign, so if it happens to be 1, then C# will interpret the whole number as negative (0 means positive number, 1 means negative).

All right. That lecture was touching very low-level topics, but now you understand the basics of the binary number system. In the next lecture, we will talk about how it affects our everyday programming.

Bonus questions:

- "**What is the decimal representation of number 101?**"
  It's 5 because it's 2 to the power of zero plus two to the power of 2, which gives 1 + 4 = 5.

- "**Why arithmetic operations in programming can give unexpected results, like for example adding two large integers can give a negative number?**"
  Because there is a limited number of bits reserved for each numeric type, for example for integer it's 32 bits. If the result of the arithmetic operation is so large that it doesn't fit on this amount of bits, some of the bits of the result will be trimmed, giving an unexpected result that is not valid.

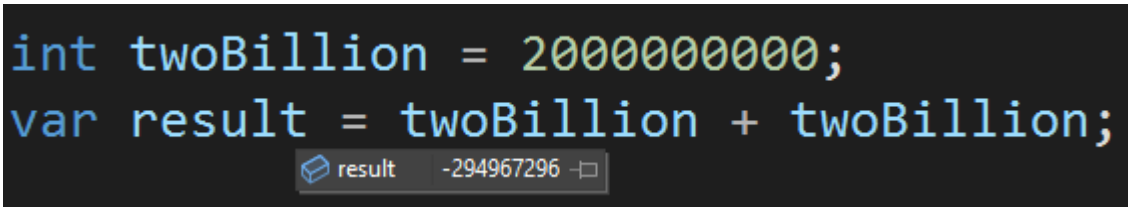
21. What is the purpose of the “checked” keyword?

Brief summary: The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow.

The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow.

To understand this slightly mysterious sentence we must first understand how arithmetic operations work in C# in general. In everyday programming, we don’t think too much about it, and perhaps we even assume the “programming arithmetics” is exactly the same as arithmetics we learned about in school. For example, we assume that the sum of two positive numbers must be a positive number. This is perfectly valid in real life, but not necessarily in programming.

For example, if I add two billion to two billion in C#, I will not get four billion. Instead, I will get this:



The result is -294967296. I added two positive numbers, and I got a negative number as a result.

To understand why this happened it is crucial to understand binary numbers, and how are they represented in the computer’s memory. Revisit the “How does the binary number system work?” lecture to find out.

I assume that by now you know that every number we use when programming is simply a sequence of bits. The important thing to realize is that on a limited number of bits we can store a limited number (the same as in a decimal number system - the biggest number represented with 3 digits is 999). For example, with 4 bits the largest number that can be represented is 15 (because if each bit is set to one, then the number is 1 + 2 + 4 + 8 = 15.

If we want to represent a bigger number, we simply need more bits of memory. Every basic numeric type in C# has a certain number of bits that it occupies in memory. For example, for integer, it's 32 bits (which FYI is 4 bytes - one byte is 8 bits). This means, the largest number an integer can be is 2147483647, which is a little more than two billion. So what happens when we add two billion to two billion? In "real" mathematics it would give 4 billion, but such a huge number is simply impossible to represent with integer type in C#. In this case, so-called "number overflow" happens, resulting in an unexpected result. This number is not random, and it's determined by how the addition of binary numbers works. You can read more about it here: https://www.sciencedirect.com/topics/computer-science/binary-addition#:~:text=Addition%20is%20said%20to%20overflow,in%20the%20remaining%20four%20bits.

It is crucial to understand that when number overflow happens **no exception is thrown** - the program continues to work normally. You may be a bit surprised by it - usually when we do something invalid, like accessing a nonexistent index in an array or dividing by zero - an exception is thrown, informing us what happened. Exceptions are a good thing, actually. It's better to be clearly informed that something went wrong.

The number overflow is a "**silent failure**" - the program doesn't work correctly, but it continues to work without exception. This can have disastrous effects. Invalid data may be stored in databases, overwriting old, valid data. Also, the program may continue and allow further invalid operations.

For example, imagine a banking system, which stores a sum of daily transactions and blocks any further payments if some limit has been exceeded. Let's imagine a very rich customer who is allowed to pay up to two billion of some currency daily. If the sum of daily payments exceeds two billion, the next payments will be blocked. Let's see a sample code:

```
public void MakePaymentNotChecked(int amount)
{
    var paymentsSumAfterPayment = _todaysPaymentsSum + amount;
    if (paymentsSumAfterPayment < MaxDailyPaymentsSum)
    {
        _todaysPaymentsSum = paymentsSumAfterPayment;
        Console.WriteLine($"[UNCHECKED] {amount} transferred! " +
            $"(Payments sum for today: {_todaysPaymentsSum})");
    }
    else
    {
        Console.WriteLine($"Transaction limit of " +
            $"{MaxDailyPaymentsSum} reached!");
    }
}
```

Now let's say the client makes a payment of 1 900 000 000 (almost two billion), and then tries to make the next one of 1 000 000 000 (one billion).

```
var account1 = new Account();
account1.MakePaymentNotChecked(1900000000);
account1.MakePaymentNotChecked(1000000000);
```

The second transaction should be blocked because the sum is over the limit of two billion, but actually, it will be allowed, because a daily sum becomes a negative number due to arithmetic overflow. And of course, any negative number is less than two billion.

```
[UNCHECKED] 1900000000 transferred! (Payments sum for today: 1900000000)
[UNCHECKED] 1000000000 transferred! (Payments sum for today: -1394967296)
```

We will allow the client to make more and more payments. And what if those payments are actually done by someone who hacked the client's account? Now the client may lose all the money instead of some limited sum.

I hope I convinced you that arithmetic overflows are dangerous. So how to deal with them? Well, this is where the "checked" keyword comes in handy. The "checked" keyword defines a scope in which arithmetic operations will be checked for overflow. If it happens, an exception will be thrown.

```
checked
{
    try
    {
        var paymentsSumAfterPayment = _todaysPaymentsSum + amount;
        if (paymentsSumAfterPayment < MaxDailyPaymentsSum)
        {
            _todaysPaymentsSum = paymentsSumAfterPayment;
            Console.WriteLine($"[CHECKED] {amount} transferred! " +
                $"(Payments sum for today: {_todaysPaymentsSum})");
        }
        else
        {
            Console.WriteLine($"Transaction limit of " +
                $"{MaxDailyPaymentsSum} reached!");
        }
    }
    catch (OverflowException)
    {
        Console.WriteLine($"Overflow exception happened!");
    }
}
```

In this scope, any overflow will throw an exception instead of failing silently.

```
[CHECKED] 1900000000 transferred! (Payments sum for today: 1900000000)
Overflow exception happened!
```

You may wonder “why isn’t this done by default?” Well, the reason is simple - **it’s performance**. Computers are very good at doing arithmetic operations and they do them extremely fast. On the other hand, checking for overflow is actually a relatively complex operation, and it takes some time. If we have a lot of arithmetic operations in the application, the performance impact may be noticeable.

I’ve created a quick little program that measures the performance difference for checked and unchecked operations. You can find it in the repository attached to the course. In short, this loop is executed and measured in both checked and unchecked context:

```
int a = 1;
int b = 2;

checked
{
    for (int i = 0; i < setSize; i++)
    {
        a = i + b + a;
        a = 1;
    }
}
```

On my computer, for **setSize** set to **one billion**, it takes on average 3257 milliseconds for **checked** context and 2431 for **unchecked**. That means the **checked operations took 33% more time**. As you can see the difference is not huge, but it is noticeable.

All right. We now have the basics of theory about the checked keyword. Let's think about how to apply it in practice. Here is a couple of tips:

- be aware of the limitations of the types you are using.
- choose proper numeric types for given usages. Do you need a counter of elements the user selected from the list that by design shows no more than 100 elements? Feel free to use **byte** - it's tiny, but the limitation to 255 is enough. On the other hand, what if you need a number representing the total number of financial transactions ever made in your banking application? **Int** sounds good, but what if your application becomes a roaring success and soon the number slightly over two billion is not enough? In this case, **long** may be a better choice - its max value is over 4 billion times larger than the max value of int.
- in case your number must be unlimited (for example you are an astronomer and you want to measure the galaxy size in millimeters) don't forget about **BigInteger** type. BigIntiger is only limited by the size of the memory of your computer, so you can represent gigantic numbers with it.
- remember that an overflow is not always a problem. For example, they are perfectly fine to happen when calculating a hash code of some object.

- if you have even the slightest concern that an undesired overflow may happen, you have two choices:
    - put this code in the checked context so an exception is thrown in case of an overflow
    - check for overflow before an actual operation, for example like this:

```
static int Multiply(int a, int b)
{
    if ((long)a * (long)b > int.MaxValue)
        throw new InvalidOperationException(
            "The multiplication will result in int overflow");
    return a * b;
}
```

    In my test application, it turned out that checking the overflow like this is actually better from the performance point of view than using the checked keyword (the test took 3257 milliseconds for checked scope and 3017 for manual checking for overflow). Please note that which one is performance-wise better depends a lot on a particular situation. If you are in doubt, it's best to run some benchmarks on your own.
- if you really need to, you can set the project setting to check arithmetic operations by default. In this case, if you want some code to be unchecked, you can use the "unchecked" keyword to define a scope in which the arithmetic operations are not checked.

Before we move on, there is an important caveat you must know about: the overflow check only applies to the immediate code block, not to any function calls inside the block. To understand this, let's consider the following code:

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

void SomeMethodWithCheckedScopeInside()
{
    checked
    {
        int i = Add(twoBillion, twoBillion);
    }
}
```

What do you think will happen? At the first glance, you might think that the OverflowException will be thrown. After all, we call the Add method in the checked scope, so adding two billion to two billion shall cause an overflow.
Well, actually it's not true. The "checked" keyword doesn't affect any methods that are called within it. If we want this code to actually be checked, we must add the "checked" keyword **inside** the Add method.

Let's summarize. The "checked" keyword is used to define a scope in which arithmetic operations will be checked for overflow. If this keyword will not be used (or overflow checking will not be enabled on the project level) the arithmetic overflow will not cause an exception, but will simply result in an invalid value.

Bonus questions:

- "**What is the purpose of the "unchecked" keyword?**"
  This keyword defines a scope in which check of arithmetic overflow is disabled. It makes sense to use it in projects in which the checking for overflow is enabled for an entire project (can be set on the project level settings).

- "**What is a silent failure?**"
  It's a kind of failure that happens without any notification to the users or developers - they are not informed that something went wrong, and the application moves on, possibly in an invalid state.

- "**What is the BigInteger type?**"
  It's a numeric type that can represent an integer of any size - it is limited only by the application's memory. It should be used to represent gigantic numbers (remember that max long is over 4 billion times larger than max int, which is a bit more than two billion, so BigInteger should be used instead of long only to represent unthinkably large numbers).

22. What is the difference between double and decimal?

Brief summary: The difference between double and decimal is that double is a floating-point **binary** number, while decimal is a floating-point **decimal** number. Double is optimized for performance, while decimal is optimized for precision. Doubles are much faster, they occupy less memory and they have a larger range, but they are less precise than decimals.

This is a question that you can hear quite often during interviews, especially in the financial sector. After this lecture, you will have a clear understanding of why.

The difference between double and decimal is that double is a *floating-point* ***binary*** *number*, while decimal is a *floating-point* ***decimal*** *number*. Double is optimized for performance, while decimal is optimized for precision.

First, let's understand the "floating-point" part. Floating-point numbers can not integers, but numbers like half, one-third, one-fourth, etc. We can actually represent such numbers *using* integers if we use the concept of **mantissa** and **exponent**. Mantissa gives some scaled representation of the number, while exponent says what the scale is - in other words, where the **decimal point** will be (that's why they are called "floating-point" numbers, as the point is moving).

This again is mysterious, so let's try to represent number 324.56 with mantissa and exponent.

All right. The pattern is simple - we multiply the mantissa by the base of the system raised to the power of the exponent. 10 to the power of -2 is 0.01, so the result is 324.56 as expected.

As you can see we managed to represent a floating-point number with integers only.

We said that double is a binary floating-point number, which means the base of the system it uses is two. For decimal, it is 10.

All right. Here comes the problem with floating-point numbers. If the mantissa has a limited precision (and it does in computers, or even if we try to write it down on a piece of paper) it means we can't represent some numbers in a perfectly precise way. For example, if you wanted to write ⅓ as a floating-point number on a piece of paper, you would fill it with 0.33333333333… and so on, but finally, you would run out of paper. Whatever number you would write, it would be some approximation of the number that is actually ⅓.

The same as with the paper, you can run out of bits when representing such a number in C#, which leads to representation that is not precise.

Double is a **binary** floating-point number occupying 64 bits. One bit is reserved for the sign (to know whether the number is positive or negative), 52 bits are reserved for mantissa and 11 bits are reserved for the exponent.

We know that numbers like ⅓ are impossible to be represented in the decimal system with a finite number of digits. In the binary system, the example of an unrepresentable number is 1/10. Let's see some code that will show how this lack of preciseness can be problematic:

```
0.3d == (0.1d + 0.2d)
```

From the point of view of real-world mathematics, this should evaluate to be true. Let's see what the computer's mathematics has to say about that:

```
0.3d == (0.1d + 0.2d): False
```

This shouldn't be that surprising, given that we already established that floating-point numbers are not represented precisely in the computer's memory.

And here is the important note - since doubles are approximations of numbers and are not very precise, we should avoid simply comparing them with the "=="

operator. Think of it like this: in real life, you are measuring boards to build something. You want the boards to be of the same length. You measure them carefully and you are happy to see that both have the length of exactly 273 centimeters and 0 millimeters you wanted. You can say to yourself "Yes, those boards are equal" and move on to constructing a shed for your gardening tools.

But… are they truly equal? I bet if you measured them with some advanced scientific machinery it would actually turn out that one of them is 273.01 while the other is 273.04 centimeters long. So they are not **exactly** equal, but they are equal enough for your needs.

It's the same thing with doubles. When checking for their equality, we should rather check if the difference between them is so small that we don't actually care about it. So the simplest implementation of a method checking doubles equality could be this:

```
bool AreEqual(double a, double b, double tolerance)
{
    return Math.Abs(a - b) < tolerance;
}
```

One more note about doubles. A variable of double type can have a specific value called **NaN** - **N**ot **a** **N**umber. It is reserved for representing undefined mathematical operations like for example dividing 0 by 0. Also, when checking if a number is NaN make sure to use double.IsNaN(value) method, because surprisingly, the equality operator for two NaNs gives false. I don't want to dwell into too much detail about it, as this lecture will be long enough already. Check out this article if you are curious:
https://docs.microsoft.com/en-us/dotnet/api/system.double.nan?view=net-6.0

All right. That closes the topic of doubles. Before we move on to decimals, let's mention floats. Well, float acts exactly the same as double The only difference is that float is stored on 32 bits while double is stored on 64.

All right. We learned that doubles are not very precise and we must be careful when making assumptions about them - for example, the equality of two double numbers may not be the same as it would be in real-life mathematics. For some scenarios we need precision and we can't make any compromises about that. For example, imagine a banking system that cross-checks some transactions. It must be sure that after the money transfer, the amount that was taken from your account is exactly the same as the amount that was added to the receiver's account. If we used double for representing money, this could lead to errors. Both amounts

would be represented as some approximation and they could not be exactly equal. That's why for representing money we should always use decimal.

Decimal is optimized for precision. It occupies more memory than a double, has a smaller range, and operations on it are slower, but it guarantees precision. Of course, like any C# numeric type, it has its size limitations and can't represent numbers smaller or larger than this, but in this range, the operations will give correct results, without any surprises.

First, let's see the code we had before, but this time let's use decimals:

```
Console.WriteLine(0.3d == 0.2d + 0.1d);
Console.WriteLine(0.3m == 0.2m + 0.1m);
```

At the top we have doubles and on the bottom decimals. And the result is:

```
False
True
```

We gained precision, but at what price? First, let's measure the performance. I created a simple program testing how long the same operations take when operating on doubles and decimals:

```
long DoubleTest(int iterations)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    double z = 0;
    for (int i = 0; i < iterations; i++)
    {
        double x = i;
        double y = x * i;
        z += y;
    }
    stopwatch.Stop();
    return stopwatch.ElapsedTicks;
}
```

There is also the second method, which does exactly the same but uses decimals. Let's see the test result:

```
Calculation of 30000000 elements for double took 3067260
ticks while for decimal it took 24023077
Decimal took 683.2096724764122% longer
```

Yikes. Calculating decimals took almost 7 times longer than doubles.

There is one more thing - doubles are not only faster, but they also have a much larger range (while taking less memory!). Let's see :

C# type/keyword	Approximate range	Precision	Size
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes

As you can see doubles have a ridiculously big range, which makes them perfect for some scientific uses where one often operates on very small or very large numbers.

In general, we should use **doubles** when representing some “natural” numbers, like physical measurements which by definition are not perfectly precise. They are great to represent things like length, speed, position on the map, and such. Due to their great performance, they are widely used in some industrial applications, games, etc. **Decimals** are much slower, they occupy more memory and have a smaller range, but they guarantee precision. We should use them when we can’t allow any approximations, so for example when representing money, points in games, and other human-made concepts that we need to represent precisely.

All right. Let’s summarize the topic of double vs decimal:

- doubles are optimized for performance, decimals are optimized for precision
- decimals have worse performance than doubles
- decimals have a smaller range than doubles - they can’t represent really tiny or really large numbers
- because of all that, decimals shall be used when we care about precision, for example, we want to compare two sums of money and tell if they are exactly equal or not. For the same usage, doubles are less precise but faster. They are perfect for representing numbers that are not human-made but rather come from nature or physics, like the speed of a car or the length of a wave. When checking two doubles for equality we should only check if they are close to each other within some tolerance.

Bonus questions:

- "**What is the difference between double and float?**"
  The only difference is that double occupies 64 bits of memory while float occupies 32, giving double a larger range. Except for that, they work exactly the same.

- "**What is the NaN?**"
  NaN is a special value that double and float can be. It means Not a Number, and it’s reserved for representing results of undefined mathematical operations, like dividing infinity by infinity.

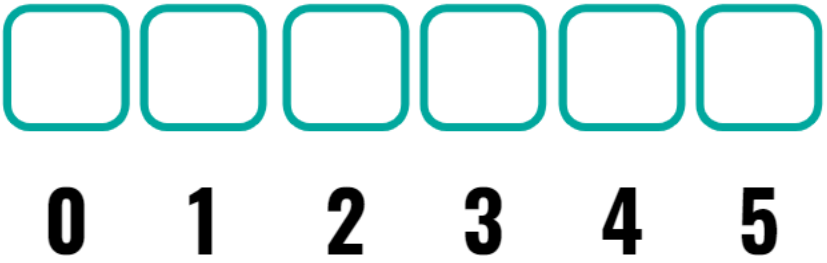
- "**What numeric type should we use to represent money?**"
  When representing money we should always use decimals.

23. What is an Array?

Brief summary: Array is the basic collection type in C#, storing elements in an indexed structure of fixed size. Arrays can be single-dimensional, multi-dimensional, or jagged.

With this lecture, we start a series about collection data structures in C#. We will begin with the simple but common question from the interviews: What is an **array**?

Array is the most basic collection in C#. You can think of an array as a collection of boxes, each one holding a single value. Each box has its index, **starting at zero** and ending at array length minus 1.



The important thing to understand is that **once an array has been created, its size cannot be changed**. Because of that arrays are not the best choice if the collection that we need is going to grow or shrink over time.

If we try to get or set the value at an index that's not in the array, we will get **IndexOutOfRangeException**. In the below code, we declare an array of size 5, so its last index is 4, but we try to access the element at index 10.

```
var array = new int[5];
array[10] = 7; //this will throw an IndexOutOfRangeException
```

When an array is created, it is filled with the default values for the given type. For example, an array of ints will be filled with zeros, and an array of strings will be filled with nulls.

We can set the values of the array right at the moment of initialization using the **collection initializer**. In this case, we don't need to specify the array's size, as it will be set to the count of provided elements:

```
var stringsArray = new [] {"a", "b", "c"};
```

The above code will naturally create an array of size 3.

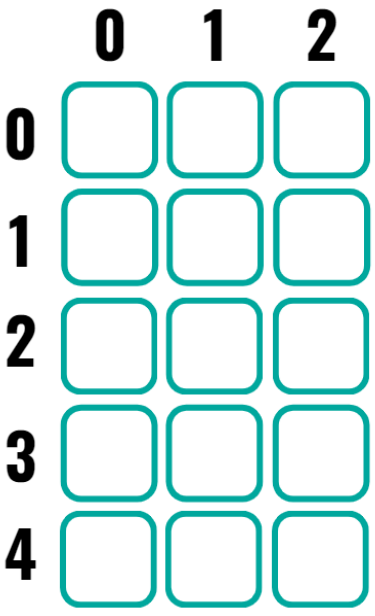
Arrays can store any objects of the same type. Arrays can be single-dimensional, multidimensional, or jagged. For example, this is a single-dimensional array that holds 5 ints.

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

We can also define multidimensional arrays that resemble matrices we know from mathematics. For example, this array can hold up to 15 elements:

```
var matrix = new int[3, 5];
```

The data held in this array can be visualized like this:



We can think of a multidimensional array as an array of arrays, which all have the same length. In this case, we have an array of size 3, and at each index, there is an array of size 5.

To get or set an element of a multidimensional array, we must simply use two indexes instead of one, separated by a comma:

```
matrix[1, 1] = 10;
```

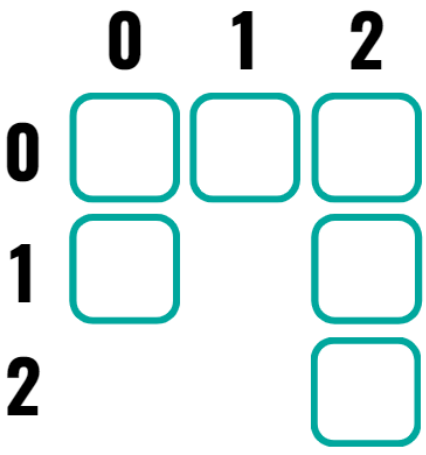
We can also define **jagged** arrays. A jagged array is an array of arrays, which don't need to be of the same length. Let's define a jagged array of integers.

```
var jagged = new int[3][];
```

Here we defined an array of size 3, for which each element will be an array of ints. Let's set those elements:

```
var jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[1];
jagged[2] = new int[3];
```

It means the structure of this jagged array looks like this:



At index 0 we have an array of length 2, at index 1 of length 1, and at index 2 of length 3.

Please notice the difference in accessing the elements of a jagged array, in opposite to a multidimensional array. We must use **two** sets of brackets:

```
jagged[2][1] = 7;
```

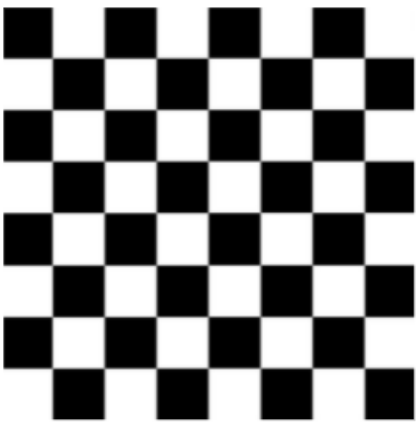
Before we continue, quick information for people with a background in languages like C or C++. Remember that in C# an **array is just an object as any others**. It's not an addressable region of memory like in those other languages. Arrays are **reference types** in C#.

All right. Let's see what arrays are best for, and when we should rather consider using other collection types.

First of all, arrays are **the most basic and native collection type** - they represent the data in a way that is very close to how the data is stored in the computer's memory. Many other collections, like for example Lists we will learn about in the next lecture, **use arrays as their underlying data structures**.

Arrays, as the most native collection type, have the advantage of being **fast**. We can get or set an element at the given index in a **constant time** (so "super-fast" in less technical terms). When we care about the **performance**, arrays are often the best choice.

Also, multidimensional arrays are commonly used in **mathematical operations**, as they represent matrices very well. They are also very useful wherever we need to represent any multidimensional structure - think of a 2D game with a tiled map, like chess or snake. The two-dimensional array of tiles would probably be the most natural and performance-efficient way to represent the game area:



The **disadvantage** of using arrays is that their **size is fixed**. In other words, it's not a dynamic collection - we can't really add or remove elements from it.

We could assume that some size of an array is large enough for our needs. Let's say we work on an e-commerce platform, and we want to store the items held in the shopping cart in an array. We can declare an array of size 100 and hope no one will need more space. But such assumptions are dangerous. What if someone is in a shopping rage and they really want to buy more things? Okay, so let's set it to 1000. But then, in 99% of the cases, we will allocate the memory for 1000 items, while actually only 2 or 3 will be added. This is a huge waste.

In short, arrays are great if we know the count of elements upfront, but not so much if we do not.

Bonus questions:

- "**What is a jagged array?**"
  A jagged array is an array of arrays, which can be all of the different lengths.

- "**What are the advantages of using arrays?**"
  They are fast when it comes to accessing an element at the given index. They are basic and easy to use and great for representing simple data of size that is known upfront.

- "**What are the disadvantages of using arrays?**"
  Arrays are of fixed size, which means once created, they can't be resized. It means that are not good for representing dynamic collections that grow or shrink over time. If we want to allocate the memory for all elements that may be stored, there is a chance we will allocate too much and waste it. We can also underestimate and not declare the array big enough for some edge cases.

- "**How to resize an array?**"
  It's not possible. An array is a collection of a fixed size and once created, it can't be resized.

24. What is a List?

Brief summary: List<T> is a strongly-typed, generic collection of objects. Lists are dynamic, which means we can add or remove the elements from them. It uses an array as the underlying collection type. As it grows, it may copy the existing array of elements to a new, larger array.

"What is a List?" may seem like a trivial question for you, and maybe you even rolled your eyes a little. You've probably used Lists thousands of times. Nevertheless, I think it's worth taking a while to understand how Lists work exactly, and what is going on under their hood.

List<T> is a strongly-typed, generic collection of objects. Lists are dynamic, which means we can add or remove the elements from them.

```
var list = new List<int> { 1,2,3 };
list.Add(4);
```

We can access List's element using an indexer, just like we do with an array:

```
list[0] = 5;
var last = list[list.Count - 1];
```

Lists provide a wide selection of built-in methods, which make them much more convenient than plain arrays. Let me show you some, but certainly not all of them:

```
list.RemoveAt(0);
list.Clear();
list.AddRange(new []{ 5, 6, 7 });
bool contains7 = list.Contains(7);
list.Prepend(10);
list.Sort();
```

Because of the dynamic nature of Lists and their convenience, they are probably the most often used collection type in C#.

All right. Let's take a look under the hood of the List. It turns out that the List is actually a fancy wrapper over an array, and all the List's data is held in a private array. Let me show you a short fragment of the List class source code:

```
[Serializable]
public class List<T> : IL
{
    private const int _de

    private T[] _items;
```

List exposes a property called Capacity. This property says what is the size of this private array held by the List. Let's see it in practice.

```
var newList = new List<int>();
Console.WriteLine(
    $"Newly created list capacity: {newList.Capacity}");

newList.Add(5);
Console.WriteLine(
    $"Capacity after adding 1 item: {newList.Capacity}");
```

Let's see what will be printed to the console:

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
```

Interesting. After a new list is created, its internal array's size is 0. Then, after the element is added, the size is "changed" to 4. I've put the "changed" into quotes because, as we learned in the previous lecture, the size of an array cannot change. I will explain what happens here in a minute, but first, let's add a couple of more elements to exceed the 4 elements limit:

```
newList.AddRange(new [] {6,7,8,9});
Console.WriteLine(
    $"Capacity when 5 items inside: {newList.Capacity}");
```

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
Capacity when 5 items inside: 8
```

It seems the size grew twice. Before explaining, let me just clear the list:

```
newList.Clear();
Console.WriteLine(
    $"Capacity after clearing: {newList.Capacity}");
```

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
Capacity when 5 items inside: 8
Capacity after clearing: 8
```

Even if the List has been emptied, its Capacity remained as it was.

All right. Let's see what is going on.

When we insert the first element to the List, it cautiously assumes that maybe the underlying array does not need to be very large. It creates an array of size 4 and assigns it to the private _items field, which we have seen in the snippet from the source code. This array is used as long as the count of elements in the List remains smaller or equal to 4.

But once this count is exceeded, the List can no longer fit elements in the underlying array - it's simply too small. So what it needs to do is this: first, it **creates a new array**, double the size of the old one. Then, it **copies** all elements from the old array to the new array. Then it can add the new element, that previously didn't fit into the smaller array.

So as you can see, it's not like the "size of the underlying array changes". It does not, as it's not possible to resize an array in C#. A brand-new array is created, and the old array is replaced by it.

On one hand, this is pretty clever, as it allows us to use a fixed-size underlying collection to actually represent dynamic data. On the other hand, it's not great from the performance point of view. Once the resizing is needed, the List must perform this quite complex operation of allocating a new array and copying the old one into it. That's why it's quite "generous" when allocating a new array, and it

makes it double the size of the old one. If the new array would be too small it could soon need to be resized again, and we want to avoid that.

On the other hand, it may of course happen that more memory than needed is allocated. For example, if we exceed the count of 1024 elements, the List will create a new array of size 2048. But it may be the case that in our business case 1025 is the absolute limitation above which the list will never grow.

In this case, we can "help" the list a little, add tell it what count of elements it should be ready for by using the constructor parameter:

```
var listOfCapacity1050 = new List<int>(1050);
Console.WriteLine(
    $"Capacity set to: {listOfCapacity1050.Capacity}");
```

```
Capacity set to: 1050
```

We should definitely consider doing that if we know upfront what is the expected size of the List. Remember that it's not set in stone, and once it's exceeded the List will resize as normal. But we will still avoid all the resizing operations that would normally happen between 0 and 1050 elements.

```
listOfCapacity1050.AddRange(Enumerable.Range(0, 2000));
Console.WriteLine(
    $"Capacity is now: {listOfCapacity1050.Capacity}");
```

```
Capacity set to: 1050
Capacity is now: 2100
```

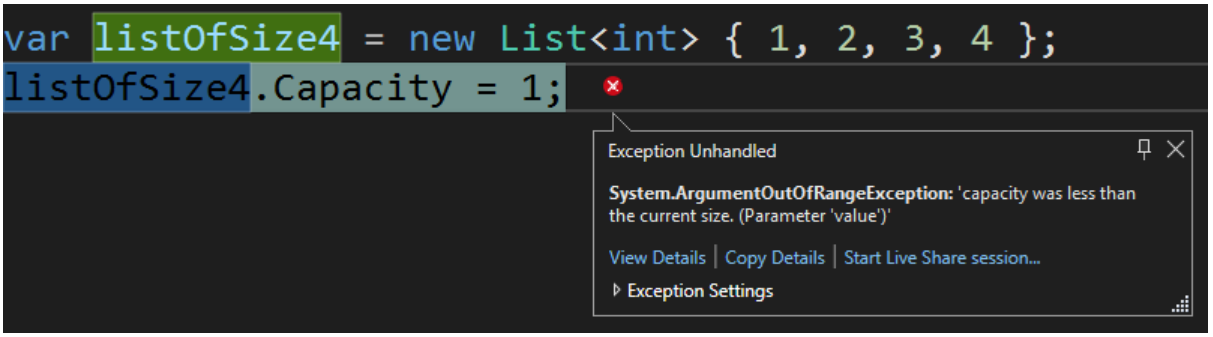
Because the operation of resizing is so heavy, the List doesn't reduce its size once elements are removed. It rather wants to assume that the allocated space will be needed sooner or later since it has already been needed once.

If we have good reasons to think that the larger space needed was a one-time thing, we can either set the Capacity to smaller manually or call the TrimExcess method, which will set the Capacity to the actual count of elements in the List.

```
var listOfCapacity1000= new List<int>(1000);
listOfCapacity1000.Add(5);
listOfCapacity1000.TrimExcess();
Console.WriteLine(
    $"Capacity after TrimExcess: {listOfCapacity1000.Capacity}");
```

```
Capacity after TrimExcess: 1
```

Remember that when setting the Capacity manually we can't make it smaller than the actual count of elements in the List. Otherwise, an exception will be thrown:



All right. We now know how things work under the hood of the List, and that in the end all data is held in an array. We must be aware that this has more implications than only resizing the array once its capacity is exceeded. Let's consider the following line:

```
var someList = new List<int> { 1, 2, 3, 5, 6 };
someList.Insert(3, 4);
```

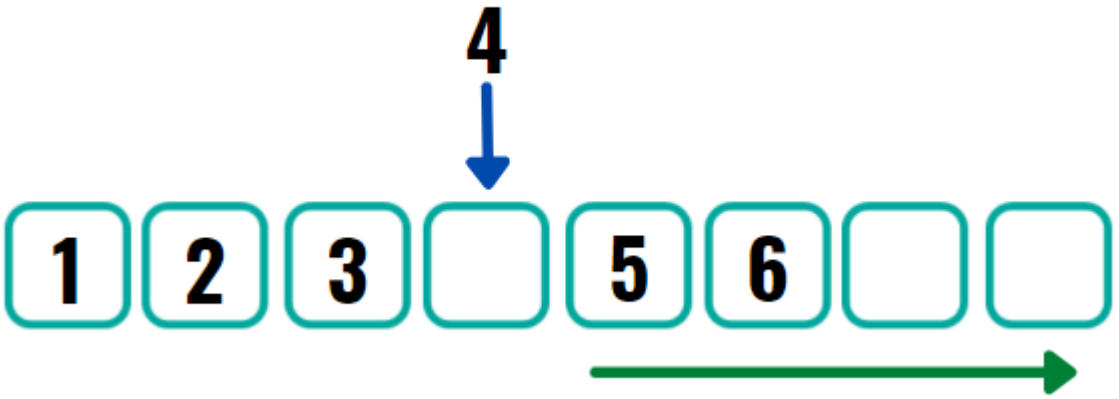
The **Insert** method takes two parameters - the index at which the new value will be placed in the List, and the value to be inserted. In this case, 4 will be inserted between 3 and 5.

Innocent as this operation seems, we must consider what happens with the array that is used underneath. Before the Insert operation, the array looks like this:

Please be aware that the last 3 elements are actually zeros (the default for int type that this Lists stores). The List knows they are not part of the represented data, because it remembers the count of elements it stores, and knows that anything

after the index Count-1 is not the actual data but the spare space that can be occupied by some new values later.

All right, back to the Insert method. We can't simply set the element at index 3 to value 4, because we would overwrite the 5. We must move all elements after the given index one index forward, to make room for the new value:



This is again impacting the performance. Worst case, we may need to move each element in the list (if we insert at index 0). This means the performance of the Insert operation is O(n), which means the count of operations will linearly grow with the collection's size.

The Insert method was just an example to show you that an operation that does something that the underlying array does not support - like in this case, inserting the element in the middle of the data - will always impact the performance, as the data in the array must be rearranged. This is something to be aware of when working with Lists, especially large ones, as the performance impact may be noticeable.

To summarize - Lists are great when it comes to representing collections that are dynamic. They give us a lot of useful methods, making working with them simple and efficient. But we must be aware that adding a feature of dynamic size to the fixed-sized collection like array comes with a cost, and this cost is performance.

Bonus questions:

- "**Why it is a good idea to set the Capacity of the List in the constructor if we know the expected count of elements upfront?**"
  Because this way we will avoid the performance-costly operation of copying the underlying array into a new, larger one, which happens when we exceed the count of 4, 8, 16… elements.

- "**What's the time complexity of the Insert method from the List class?**"
  The Insert method needs to move some of the elements of the underlying array forward, to make room for the new element. In the worst-case scenario, when we insert an element at the beginning of the List, we will need to move all existing elements. This means the complexity of this operation is O(N).

25. What is an ArrayList?

Brief summary: An ArrayList is a collection that can store elements of any type, as it considers them all instances of the System.Object. ArrayLists were widely used in older versions of C#, where the generics were not yet available. Nowadays they should not be used, as their performance is impacted by the fact that they need to box value types.

Let me take you on a journey back in time. A long, long time ago (before 2006) .NET was still at version 1. It was still a pretty new framework (its initial release was in 2002). The C# language itself did not look much as it does now.

At this version of .NET, there was no such thing as **generics**. If you wanted to have a collection of numbers and a collection of strings, arrays were your best choice. You couldn't count on things like the generic List<T> that can hold any type of items.

But as we learned in the lecture "What is an Array?", arrays can be pretty awkward. They have fixed sizes, they also don't provide any convenient methods like Add or Remove. In other words, with arrays only, creating a complete, efficient application that met some real business needs could have been a pain in the neck.

Luckily, there was another way than using plain arrays. The **ArrayList** type. An ArrayList is dynamic a collection, so a collection we can resize, that can hold any type of items. And just to be clear - at the same time. Single ArrayList can hold ints, strings, objects, DateTimes, and anything we want.

```
var arrayList = new ArrayList() {
    1, "hello", new DateTime(2022, 1, 1) };
```

In statically typed languages like C# this is at least weird. But how does it work? Well, the ArrayList simply treats everything it holds as instances of **System.Object** type. After all, everything in C# can be considered an Object, because every type is derived from the Object class. But there is a problem: if the item we want to store in ArrayList is of a **value type**, it must be **boxed** to be treated as Object, which is a reference type. Boxing is not a cheap operation - it requires moving the value from the stack to the heap and creating a reference for it. Also, at some point, we will need to unbox this item to access the underlying value.

We will talk more about the performance of the Array list later in this lecture.

Since ArrayList can hold any type of elements, we don't really know what they are and how can we use them. If I have a List<int>, I know I can, for example, calculate the sum of them. If I have a List<string> I know I can concatenate them. But what can I do with elements of an ArrayList?

The truth is, ArrayList was almost never used like this:

```
var arrayList = new ArrayList() {
    1, "hello", new DateTime(2022, 1, 1) };
```

In most practical cases, it was holding elements of the same type.

```
var numbers = new ArrayList() { 1, 2, 3};
var strings = new ArrayList() { "a", "b", "c"};
```

That looks "almost" like generic Lists, which again, were not present in .NET before version 2. But those variables are very problematic. Let's say I want to create a method that calculates the sum of elements in a collection of numbers:

```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach(var number in hopefullyNumbers)
    {
        result += number;
    }
    return result;
}
```

This doesn't compile. Each element of the ArrayList is an Object, so I can't simply add it to the result. I must first cast it and hope that it will succeed:

```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach(var number in hopefullyNumbers)
    {
        result += (int)number;
    }
    return result;
}
```

Just to be sure, let's handle the InvalidCastException in this method:

```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach (var number in hopefullyNumbers)
    {
        try
        {
            result += (int)number;
        }
        catch (InvalidCastException)
        {
            Console.WriteLine($"{number} is not really an int!");
            throw;
        }
    }
    return result;
}
```

As you can see, we were forced to create a lot of code that would not be needed if we knew what types exactly do we deal with. In other words, if we were given a List<int> instead of an ArrayList.

So we know the first big disadvantage of ArrayList - we don't know what is stored inside, so we must be ready for a lot of casting and error handling. The other disadvantage is the performance that I mentioned before - when storing value types in ArrayList, they must all be boxed which can impact the performance very much.

So in this case, the natural question is “When to use ArrayLists over Lists?”. Well, the answer is - never. Unless for some reason you must work in applications written in .NET 1, but I honestly hope that you don’t. Even if you do, consider upgrading the version of .NET rather than working in this ancient technology.

You may then ask, why do we learn about this, if it’s not a big deal since 2006. First of all, the questions about ArrayLists are quite liked by interviewers, as they can be a prelude to a discussion about static and dynamic typing, which we learned more about in the lecture about the “dynamic” keyword.

Secondly, as much as I hope you don’t need to work with ArrayLists, it may turn out that you’ll have to work with some legacy code that still uses them, and then it’s important that you know what you are dealing with. Also, only recently (in 2022) I’ve been working on a brand-new application where someone was using ArrayLists for reasons they couldn’t explain, and as it turned out, it was mostly storing value types. Changing them to Lists not only made the development process much easier, as the neverending casts and error handling could be omitted, but it also made the application over 25% faster.

There is one more case when using ArrayLists may seem tempting - when you *actually* need to store elements of different types in a single collection. But even then, it’s better to use a List<object> as it provides more functionality than ArrayList and will most likely be more consistent with the rest of the application.

Bonus questions:

- "**What is the difference between an array, a List, and an ArrayList?**"
  An array is a basic collection of fixed size that can store any declared type of elements. The List is a dynamic collection (it means, its size can change over time) that is generic, so it can also store any declared type of elements. An ArrayList is a dynamic collection that can store various types of elements at the same time, as it treats everything it stores as instances of the System.Object type.

- "**When to use ArrayList over a generic List<T>?**"
  Never, unless you work with a very old version of C#, which did not support generics. Even if you do, you should rather upgrade .NET to a higher version than work with ArrayLists.

26. What is the purpose of the GetHashCode method?

Brief summary: The GetHashCode method generates an integer for an object, based on this object's fields and properties. This integer, called hash, is most often used in hashed collections like HashSet or Dictionary.

We are not yet quite done with collections. In the next lecture, we will discuss Dictionaries. But to understand Dictionaries we must first understand the GetHashCode method, so let's do it in this lecture.

GetHashCode is one of the few methods that belong to the System.Object type. In other words, we can call it on any object in C#. Before we understand what it does, let's see it in action:

```
var anyObject1 = "123";
Console.WriteLine(
    $"'{anyObject1}' hashcode is {anyObject1.GetHashCode()}");

var anyObject2 = 123;
Console.WriteLine(
    $"{anyObject2} hashcode is {anyObject2.GetHashCode()}");
```

```
'123' hashcode is -2093679465
123 hashcode is 123
```

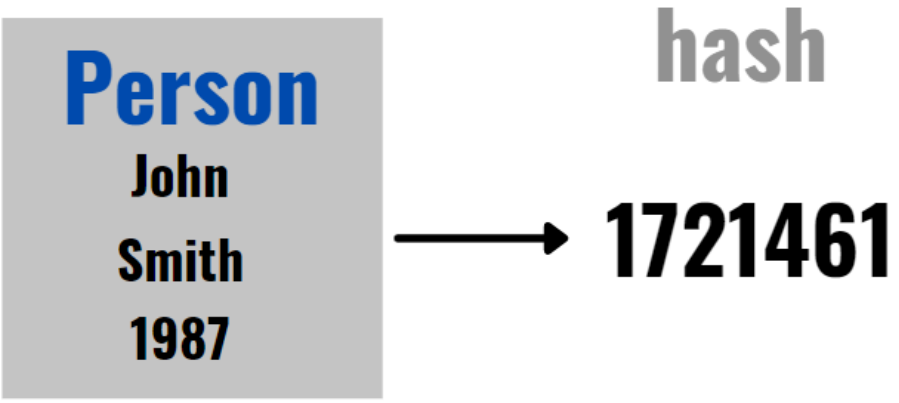
For now, those values look enigmatic, but hopefully, we will understand them better later in the lecture.

The GetHashCode method is a **hash function** implementation for an object. Let's see the definition of a hash function:

*"A **hash function** is a one-way cryptographic algorithm that maps an input of any size to a unique output of a fixed length of bits. "*

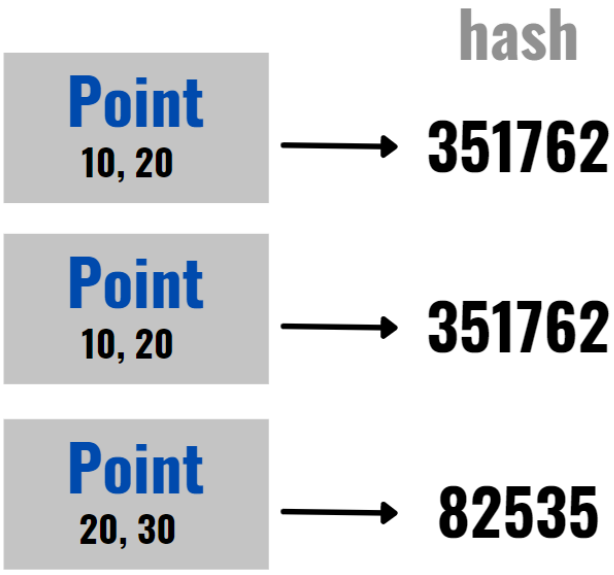
At least for me this sounds completely vague. First, let's understand what the result of this hash function is. In C# is an **integer**. In simple terms, hash is a number

calculated for some object from its components. Here is an object of Person class and some hash calculated for it.



We will take a closer look at how hash is actually calculated a bit later, but for now, let's just say that it's a function of the values of fields and properties belonging to the object. In this case, we could for example associate each letter with some number, which would allow us to translate words "John" and "Smith" to integers. Then, we would somehow combine those integers with the integer representing the year of birth, and as a result, we would have the hash code of the person. According to that, if we created another object of the Person class with name John, LastName Smith, and YearOfBirth 1987, the hash should be the same. On the other hand, if this other object had a different year of birth, its hash would be different.

Also, if we calculate the hash for the second time, the result should be the same as it was for the first time, assuming the object was not modified. Also, if we have two objects that are different instances, but we consider them equal (for example, two instances of the Point class, both having X=10 and Y=20) the hash code for both of them should be the same.



At this point you probably wonder “okay, but what is the use for hash codes?”. Well, their main use is that they work as keys in **hashed collections**. This may sound cryptic by now but don’t worry - we will soon learn about one of the most useful C#’s hashed collections - the Dictionary. If you used Dictionaries before you know that each value is stored under a key. The key can be any object, even a complex one, but the Dicionary needs to be able to translate it to an integer, and that’s exactly where the GetHashCode method comes in handy. We will learn more about it in the lecture about the Dictionaries.

Back to the hash functions, that “map” complex objects into integers. The very important trait of the hash function is that it should **uniformly distribute its values**. That means, if I call GetHashCode methods for 100000 **different** objects of the Point type, I should get very little or no duplicated hashcodes.

But **it is possible to have duplicated hash codes.** This situation is called “**hash code conflic**t” and it’s perfectly normal. Many people consider hash codes the be the “identifiers” of objects and think that two different objects of the same type can’t have the same hashcodes. But this is not true, and it cannot be. Let me prove it to you.

Consider a Point type. It contains two fields: X and Y. Both X and Y are ints, so each of them can have a value between int.MinValue to int.MaxValue, so in other words - the range of the integer. For simplicity, let’s say that the minimal value of the integer is -2 000 000 000 and maximal is 2 000 000 000. This means, we can have 4 billion different X coordinates and 4 billion Y coordinates, which in total gives **4 billion*4 billion** different Points, which is 16 quintillions! On the other hand, the **hash itself is an integer**, so we can only have **4 billion** different values, so much, much less than different Points. So when creating different Points, we will sooner or later simply run out of different hashes. It’s sometimes referred to as the “balls into bins problem”. If we have more balls than bins, and each ball is stored in some bin, it must mean that in some bins there is more than one ball.

Let’s summarize the hash function. If I have two different objects of some type, ideally their hash codes should be different. If I have plenty of different objects of the same type, there should be as few duplicated hash codes as possible. Finally, if I have two objects I consider equal, their hash code should be the same.

Let’s see some implementations of the GetHashCode method for some types. Here is the implementation for the int type:

```
// The absolute value of the int contained.
public override int GetHashCode() {
    return m_value;
}
```

For integers, the implementation is as simple as it can ever be. The integer value itself is a perfect hashcode. It will be the same for two equal integers, and it will be different for two different integers. There will be no duplicates at all because for each integer possible the hash code will be different.

Now, let's consider the Point type:

```
class Point
{
    1 reference
    public int X { get; }
    1 reference
    public int Y { get; }
    0 references
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

Before we think of our own implementation of the GetHashCode method, let's see what is the default. As we said, the GetHashCode method is defined in the System.Object class, so I can call it on any object even if I did not override it. Let's see some Points:

```
var point1 = new Point(10, 20);
var point2 = new Point(10, 20);
var point3 = new Point(20, 30);
```

As you can see **point1** and **point2** are the same, so I would like them to have the same hash code. **point3** is different, so it should have a different hash code. Let's see the result:

```
X=10, Y=20 hash code is 43942917
X=10, Y=20 hash code is 59941933
X=20, Y=30 hash code is 2606490
```

Well, that's not what we wanted. The two first hash codes should be the same. To understand why is that so, we must understand what's the default implementation of the GetHashCode method:

- for reference types, it bases on the reference itself, so the "address" of the object in memory
- for value types it is calculated based on the values stored in the object

That explains why two Points, even with the same X and Y, have different hash codes. We declared the Point as a class, so a reference type. point1 and point2 are two different objects with two different references. The hash code is built based on the reference, so it's different for both of them.

So if we want to have the same hashcodes for the Points with the same X and Y, we can simply change the Point class to a struct:

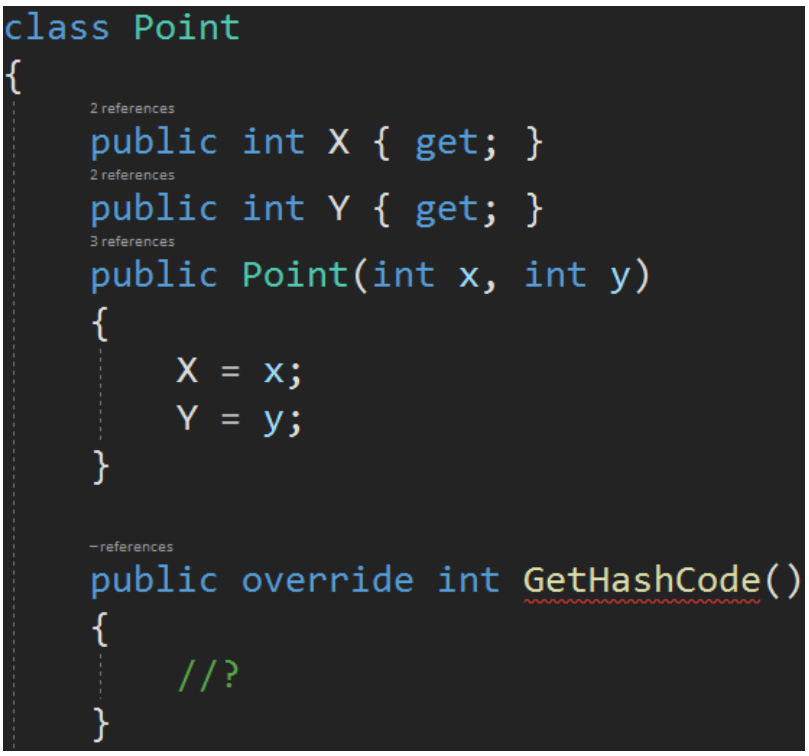
```
struct Point
```

And now, we can re-run the application:

```
X=10, Y=20 hash code is 1644919074
X=10, Y=20 hash code is 1644919074
X=20, Y=30 hash code is 1644919094
```

Now we have what we wanted - the first two Points have the same hash codes.

But let's change it back to a class, and let's try to implement the GetHashCode method ourselves:



Most of the base types in C# already provide a good implementation of the GetHashCode method. Those methods are usually strongly based on pure math and also pretty low-level, and because of that, I don't want to get into details on how they work. Just so you have an idea, here is a **fragment** of the GetHashCode implementation for string:

```
// 32 bit machines.
int* pint = (int *)src;
int len = this.Length;
while (len > 2)
{
    hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
    hash2 = ((hash2 << 5) + hash2 + (hash2 >> 27)) ^ pint[1];
    pint += 2;
    len  -= 4;
}

if (len > 0)
{
    hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
}

int     c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```

As you can see this is pretty low-level stuff. Luckily for us, the hard work has already been done by others. When defining custom types, we can simply combine the hashcodes of the values stored in the object into a single hashcode. For the Point class, it would look like this:

```
public override int GetHashCode()
{
    return HashCode.Combine(X, Y);
}
```

HashCode.Combine takes any objects as parameters, so for example for a person class we could easily use it like this:

```
class Person
{
    2 references
    public string FirstName { get; }
    2 references
    public string LastName { get; }
    2 references
    public int YearOfBirth { get; }

    0 references
    public override int GetHashCode()
    {
        return HashCode.Combine(FirstName, LastName, YearOfBirth);
    }
```

Also remember, that we do not always need to combine all properties and fields of a type to get a valid hash code. For example, if we had SocialSecurityNumber in the Person class, which by definition identifies a person, it would be perfectly fine to use it as the only component of the hash code. We always consider two Person objects equal if they have the same social security number, and we can ignore other fields (if they were different, it would most likely mean there is some error in data itself, as two different people should never have the same social security number).

We know how to implement the GetHashCode method now, but the question that we need to answer is this: when should do it?

The answer is simple - if the type is going to be used as a key of any hashed collection, like a Dictionary or the Hashtable, and the default implementation is not working for us.

For reference types, we usually don't want the default GetHashCode, as it compares objects by reference. As with the Point class - we had two Point objects with the same X and Y, yet their hash codes were different, so when used as keys in the Dictionary, they would be considered two different keys. In this case, we usually want to override the GetHashCode method and HashCode.Combine can be a great help (of course, in some situations hashes based on the reference itself are fine. It all depends on the context).

Later in the course, we will learn about records. **Records** are reference types that provide their own, value-based GetHashCode method.

For value types, it is a bit tricky. There is a default implementation that works fine and uses the values stored in the fields or properties of the type to calculate the hash code. The problem is that this default implementation uses **reflection**, and as we learned in the "What is reflection?" lecture, it's painfully slow. Because of that, it's a good idea to provide a custom implementation of the GetHashCode method

in value types we create, especially if they are going to be used as hashed collection keys a lot.

When overriding the GetHashCode method it is important to also override the Equals method. We will explain the reason for that in the lecture about the Dictionaries.

Let's summarize. The GetHashCode method generates an integer for an object, based on this object's fields and properties. This integer, called hash, is most often used in hashed collections like HashSet or Dictionary.

Bonus questions:

- "**Can two objects of the same type, different by value, have the same hash codes?**"
  Yes. Hash code duplications (or "hash code conflicts") can happen, simply because the count of distinct hash codes is equal to the range of the integer, and there are many types that can have much more distinct objects than this count.

- "**Why it may be a good idea to provide a custom implementation of the GetHashCode method for structs?**"
  Because the default implementation uses reflection, and because of that is slow. A custom implementation may be significantly faster, and if we use this struct as a key in hashed collections extensively, it may improve the performance very much.

27. What is a Dictionary?

Brief summary: A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique.

A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique.

Here is a Dictionary representing the mapping from the country name to its currency:

```
var currencies = new Dictionary<string, string>();
currencies["USA"] = "USD";
currencies["Japan"] = "JPY";
currencies["Brazil"] = "BRL";
```

The key and the value in a dictionary don't need to be of the same type. Below we have a Dictionary mapping from string to decimal. You can also see the Add method, which is an alternative for setting a value under each key with the indexer:

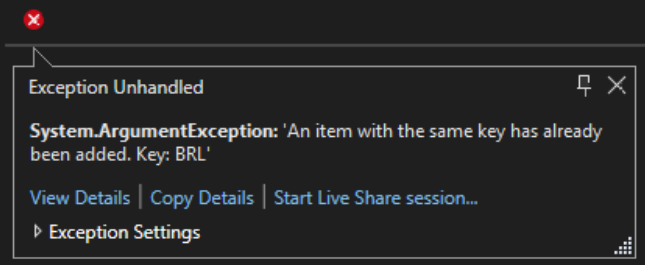
```
var currenciesValuesComparedToDollar = new Dictionary<string, decimal>();
currenciesValuesComparedToDollar.Add("USD", 1m);
currenciesValuesComparedToDollar.Add("JPY", 0.0086m);
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
```

We can use the collection initializer instead of adding the key-value pairs to the Dictionary one by one:

```
var savedGames = new Dictionary<string, string>
{
    ["save1"] = @"C:/saves/save1.dat",
    ["autosave"] = @"C:/saves/auto/save.dat",
    ["beforeBossFight"] = @"C:/saves/beforeBossFight.dat",
};
```

Remember that Dictionary's keys must be unique. That means, an attempt to add a new value under the same key will throw an exception:

```
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
```



When using an indexer, the old value under the given key will simply be replaced with the new:

```
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
currenciesValuesComparedToDollar["BRL"] = 0.19m;
```

The use cases for Dictionaries are endless. Whenever we need any kind of mapping, they are most likely the best choice. Let me give you a very simple example. We have a collection of Employees. Each Employee has a Department he or she works in and the Salary property. We want to create a method that calculates what is the average salary in each Department.

```
record Employee(Department Department, decimal Salary);
1 reference
enum Department { MissionControl, Xenobiology, PlanetTerraforming }
```

```
var employees = new List<Employee>
{
    new Employee(Department.Xenobiology, 15000),
    new Employee(Department.MissionControl, 10000),
    new Employee(Department.PlanetTerraforming, 9000),
    new Employee(Department.PlanetTerraforming, 8000),
    new Employee(Department.MissionControl, 11000),
    new Employee(Department.MissionControl, 12000),
};
```

The result of this method should be a mapping from Department to the average salary. A mapping is best represented with a Dictionary. I will use LINQ's GroupBy method to group the Employees by department, and then calculate the average salary for each group. I will transform the result into a Dictionary using the ToDictionary method.

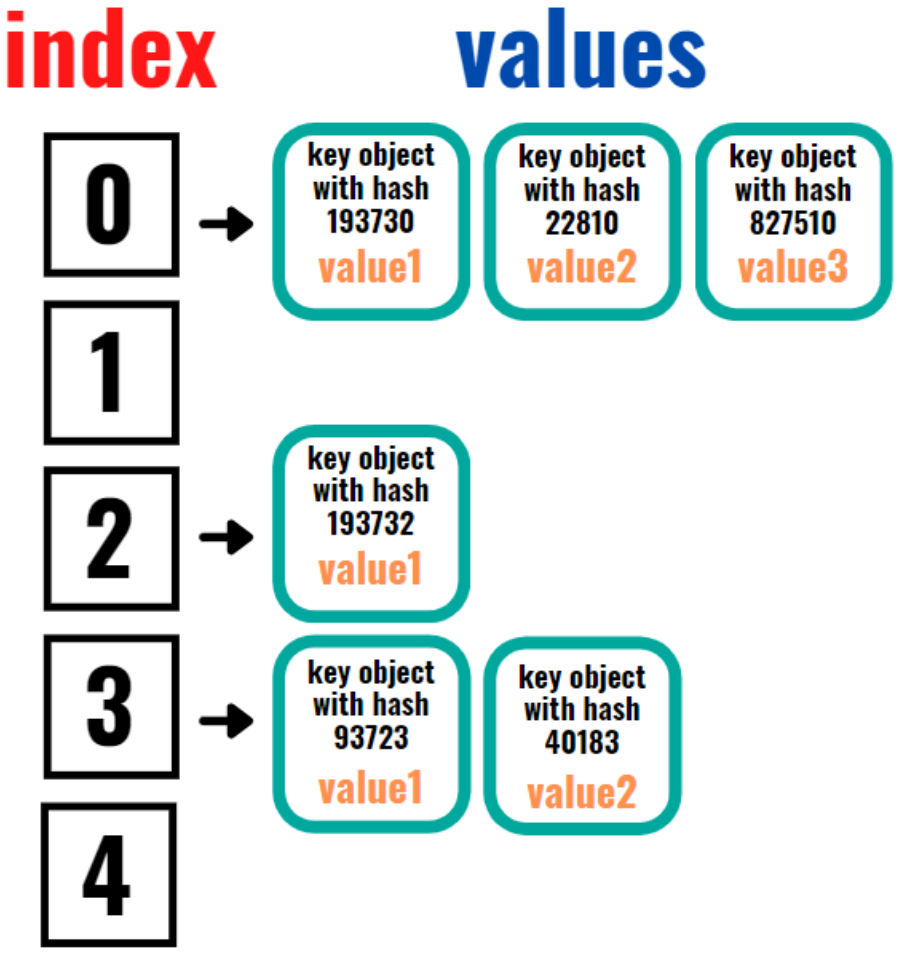
```
Dictionary<Department, decimal> departmentsAverageSalaries =
    employees
    .GroupBy(e => e.Department)
    .ToDictionary(
        grouping => grouping.Key,
        grouping => grouping.Average(g => g.Salary));
```

Let's see if the result is as expected:

```
Xenobiology: 15000
MissionControl: 11000
PlanetTerraforming: 8500
```

The result looks good, and Dictionary is a perfect data structure to represent it.

Now, let's take a look under the hood of Dictionaries. The underlying data structure of a Dictionary is a hash table. A hash table is basically an **array** of linked lists. We can imagine it like this:



Each element in the list has a **value** and the **key** object for which the hash code is calculated. The placing of the key-value pair is not random in the array. The index is calculated like this:

hash code % array size

This, naturally, gives a number from 0 to array size minus one, which is a valid index.

When an item is inserted into the hash table, its hash code is calculated (we learned about hash codes in the "What is the purpose of the GetHashCode method?"). Then, the index in the array is calculated using the above formula. Finally, the key-value pair is added to the list stored under the given index. This means, under a certain index objects with different hash codes can be stored.

But what happens if we add a new key-value pair to the Dictionary, and the key has the same hash code as some other key that is already stored in it? Well, the Dictionary must ask this: is this actually the same key, or is it a different key that accidentally has the same hash code? There is one method that can answer this question: the **Equals method**.

If two keys have the same hash codes, and the **Equals** method returns **true** for them, it means it's actually the same key. Then the dictionary will either throw an exception (if the key-value pair was added with the Add method, which expects that this key is not yet present in the Dictionary) or, if it was set with the indexer, it will simply update the value under the key.

But if the new and the old key have the same hash codes, but the **Equals** method returns **false** for them, it means there are actually two different keys, that have the same hashcode by accident. In this case, the Dictionary stores the new key-value pair under the same index, and it adds it at the end of the linked list that is stored under this index. When we try to retrieve the value under this key again, the Dictionary will quickly calculate the hash code, and based on this hash, the index in the array of linked lists. Then, it will iterate the list, looking for an object with the same key - so the key for which the Equals method returns true if compared with the key for which we try to retrieve the value.

Because the **Equals** method is needed in case of hashcode conflicts to properly identify the key, we should **always override it when overriding the GetHashCode** method, so their implementations are consistent. For example, if GetHashCode returns the social security number for a Person object, it means we consider this

number the Person's identifier. The Equals method should also only compare the social security numbers.

Calculating the hashcode is (or at least should be) very fast. Accessing the array element at a given index is extremely fast. This means, as long as the list under each index is small, accessing the Dictionary value under a given key should be super fast, and this is the main power of Dictionaries.

So, how big are the lists stored under each index? Well, it depends on two factors:

- what is the size of the array that represents the hash table (this is something we don't control, the Dictionary itself adjusts the size similarly as List did)
- how often the hash codes for different objects are the same. If, for example, we implemented the GetHashCode method for some type as "return 1", it would mean that all hashcodes will be duplicated. There will be only one element in the array representing the hash table, and this element will be a very long list of key-value pairs. In other words, in this particular case, the performance of the Dictionary will be similar to the performance of a List. This is the reason for which the hash functions should be uniformly distributed. The fewer hash code conflicts, the better the Dictionary's performance.

Let's summarize. A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique. When a key is added to the Dictionary, it calculates its hash code using the GetHashCode method. It uses this hashcode to properly place the value for the given key in the hash table that is the underlying data structure of a Dictionary.

Bonus questions:

- "**What is a hash table?**"
  A hash table is a data structure that stores values in an array of collections. The index in the array is calculated using the hash code. It allows quick retrieval of objects with given hashcode. A hash table is the underlying data structure of Dictionary.

- "**Will the Dictionary work correctly if we have hash code conflict for two of its keys?**"
  Yes. The Dictionary still can tell which key is which using the Equals method, so it will not mistake them only because they have the same hash codes.

- "**Why should we override the Equals method when we override the GetHashCode method?**"
  Because the Equals method is needed for the Dictionary to distinguish two keys in case of the hash code conflict, and because of that its implementation

should be in line with the implementation of the GetHashCode method. For example, if GetHashCode returns the social security number for a Person object, it means we consider this number the Person's identifier. The Equals method should also only compare the social security numbers.

28. What are indexers?

Brief summary: Indexers allow instances of a type to be indexed just like arrays. In this way, they resemble properties except that they take parameters. For example, a Dictionary<string, int> has an indexer that allows calling "dictionaryVariable["some key"]" to access the value under some key.

Indexers are something we use all the time, usually without giving it much thought. Whenever accessing an element under a specific index of a list, we are actually calling the List's indexer:

```
var list = new List<decimal> { 5.5m, 3m, 0m };
var thirdElement = list[2];
```

This is exactly the same as accessing the third element of an array. This code may seem simple, but there is a lot going on in the List class whenever we use its indexer. Let's take a look into List's source code:

```
// Sets or Gets the element at the given index.
//
public T this[int index] {
    get {
        // Following trick can reduce the range check by one
        if ((uint) index >= (uint)_size) {
            ThrowHelper.ThrowArgumentOutOfRangeException();
        }
        Contract.EndContractBlock();
        return _items[index];
    }

    set {
        if ((uint) index >= (uint)_size) {
            ThrowHelper.ThrowArgumentOutOfRangeException();
        }
        Contract.EndContractBlock();
        _items[index] = value;
        _version++;
    }
}
```

We can recognize the definition of the indexer by the "this[someType paramName]" code. For the case of a List, the indexer is quite complex, but in the end, its main job is to get or set the value of an internal array called "_items". In the lecture "What is a List?" we learned that they use an array as the underlying collection, and this is exactly what we see here.

Indexers don't necessarily use integers as parameters. For example, when using Dictionaries we use indexers to access an element under a given key - and a key of a Dictionary can be whatever we want. For example, if the key of the Dictionary is a string, then the parameter of the indexer will also be a string:

```
var currencies = new Dictionary<string, string>
{
    ["USA"] = "USD",
    ["Great Britain"] = "GBP",
    ["Poland"] = "PLN"
};
var currencyInGreatBritain = currencies["Great Britain"];
```

We can define our own indexers in the types we created. Let's define a simple class that works as a wrapper for an array:

```
2 references
class MyList<T>
{
    private T[] _numbers;

    1 reference
    public MyList(T[] numbers)
    {
        _numbers = numbers;
    }
}
```

For now, when trying to use an indexer on an object of this class, we will get a compilation error, because this class does not support it:

```
var myList = new MyList<int>(numbers);
var secondValue = myList[1];
```

To make it work, we must define an indexer accepting an int in the MyList class:

```
public T this[int index]
{
    get => _numbers[index];
    set => _numbers[index] = value;
}
```

On **get**, this indexer simply retrieves the value from the array, and on **set** it overwrites it with the provided value.

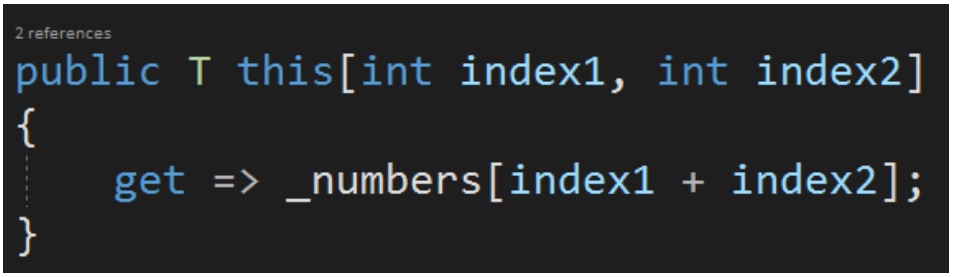
We can define as many indexers as we want if they only differ by types or count of parameters. For example, I can add an indexer accepting a string to this class:

```
public T this[string textIndex]
{
    get => _numbers[int.Parse(textIndex)];
    set => _numbers[int.Parse(textIndex)] = value;
}
```

Indexers with multiple parameters are also allowed:

```
1 reference
public T this[int index1, int index2]
{
    get => _numbers[index1 + index2];
    set => _numbers[index1 + index2] = value;
}
```

Please note that it is possible to have an indexer with getter only (or with setter only, but this is more unusual).



In this case, we will be able to access an element at a given index, but we won't be able to overwrite it.

Let's sum up. Indexers allow instances of a type to be indexed just like arrays. In this way, they resemble properties except that they take parameters. For example, a Dictionary<string, int> has an indexer that allows calling "dictionaryVariable["some key"]" to access the value under some key.

Indexers are most often used with types representing collections, but we can add them to any type. Indexers are simple and natural to use for developers, and we should consider adding them to our types, especially if we already have some methods like "GetValueAtIndex". In this case, we should definitely consider refactoring and introducing an indexer.

Bonus questions:

- "**Is it possible to have a class with an indexer accepting a string as a parameter?**"
  Yes. We can define indexers with any parameters. An example of such a class can be a Dictionary<string, int> as we access its elements like "dict["abc"]".

- "**Can we have more than one indexer defined in a class?**"
  Yes. Just like with method overloading, we can have as many indexers as we want, as long as they differ by the type, count, or order of parameters.

29. What is caching?

Brief summary: Caching is a mechanism that allows storing some data in memory, so next time it is needed, it can be served faster.

Caching is a mechanism that allows storing some data in memory, so next time it is needed, it can be served faster. To understand it better, let's consider this class:

```
class PeopleController
{
    private readonly IRepository<Person> _peopleRepository;

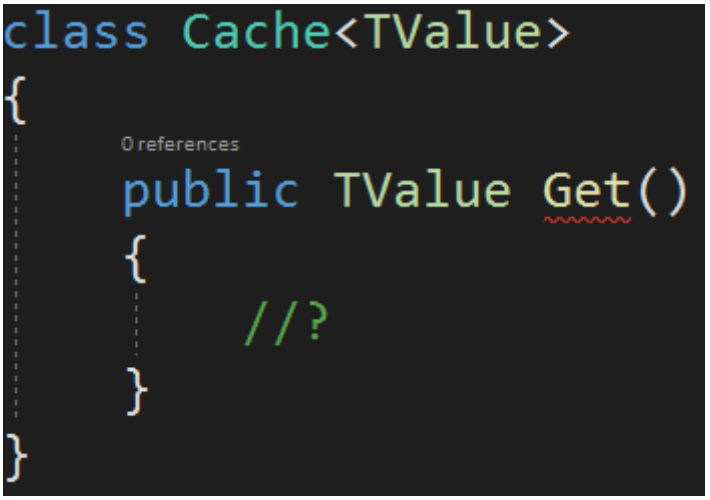
    1 reference
    public PeopleController(IRepository<Person> peopleRepository)
    {
        _peopleRepository = peopleRepository;
    }

    3 references
    public Person? GetByName(string firstName, string lastName)
    {
        return _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault();
    }
}
```

This class is responsible for retrieving a Person object from some repository using a person's first and last name. This code is very simple, but we must be aware of one thing - in a real-life project, accessing data from an external source may be **slow**. Maybe the class implementing the IRepository interface connects to some bulky database, or retrieves data from some API? It may be the case that every call to this external data source takes some considerable time, and if many calls are executed, the application may start to work slowly. We don't want that.

How can we make it better? Well, one solution could be this: if we already accessed the data for some particular first and last name - for example, John Smith - we could store it in the application memory and next time when we want to find John Smith, we will access his data immediately, instead of asking the external system to provide it again. This mechanism is called **caching**. We store some data in a cache, so a piece of memory of the application, making it available immediately. Let's

implement a very simple caching mechanism. We will start with defining a cache class.



I made this class generic, so it can store any type. In our case, it will store objects of type Person.

Now we need some kind of container to store the cached Person objects. What data structure would be the best for us? We will want to retrieve them by providing first and last names. So perhaps the best choice would be a Dictionary in which a tuple of two strings (for both names) would be the key, and the Person object would be the value.

```
class Cache<TValue>
{
    private readonly Dictionary<(string, string), TValue> _cachedData = new();

    0 references
    public TValue Get()
    {
        //?
    }
}
```

As you can see I used a ValueTuple for the key because for Dictionary keys we want to have a value-based equality comparison (If I used regular tuple, which is a reference type, two tuples holding the same first and last name would not be considered the same key by the Dictionary).

We want to keep the Cache generic, so the Dictionary key type should also be parameterized:

```
class Cache<TKey, TValue> where TKey: notnull
{
    private readonly Dictionary<TKey, TValue> _cachedData = new();

    0 references
    public TValue Get()
    {
        //?
    }
}
```

As you can see I also added the **notnull** constraint. The Dictionary keys should never be null.

Now about the Get method. First of all, it should take a key as a parameter:

If the Dictionary already stores the value with a given key, it should simply be returned:

```
public TValue Get(TKey key)
{
    if(_cachedData.ContainsKey(key))
    {
        return _cachedData[key];
    }
}
```

But if not, it means the value is not yet cached and we need to access it somehow. In the case of the PeopleController, it would be read from the repository. But we can't do it like this:

```
if(_cachedData.ContainsKey(key))
{
    return _cachedData[key];
}
else
{
    _cachedData[key] = _peopleRepository
        .GetByName(firstName, lastName)
        .FirstOrDefault();
}
```

The Cache class is generic and it must work not only for People read from some specific repository but for anything else too. In other words, we must also provide some generic mechanism for reading the values that will be then stored in the cache. The simplest solution is to pass a Func to the Get method. This Func will return an object of the TValue type. It will be up to the caller of the Get method to provide a specific method of retrieval. In our case, the PeopleController will be using the cache, and it will pass a function reading the Person object from the database to the Get method.

```
0 references
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if(_cachedData.ContainsKey(key))
    {
        return _cachedData[key];
    }
    else
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    return _cachedData[key];
}
```

We can now simplify this code:

```
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if (!_cachedData.ContainsKey(key))
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    return _cachedData[key];
}
```

Great. This is exactly what we wanted. The value is being read only once. It is stored in the Dictionary, and next time it will be needed it will be retrieved from it. Let's now use the Cache class in the PeopleController. First of all, I will declare and initialize the private Cache field in this class:

```
class PeopleController
{
    private readonly Cache<(string, string), Person?> _cache = new();
```

Now I can use it in the GetByName method. As a reminder: this is how this method looks without using caching:

```
public Person? GetByName(string firstName, string lastName)
{
    return _peopleRepository
        .GetByName(firstName, lastName)
        .FirstOrDefault();
}
```

And this is how it changes when the Cache is used:

```
public Person? GetByName(string firstName, string lastName)
{
    return _cache.Get(
        (firstName, lastName),
        () => _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault());
}
```

The first parameter of the Get method is the key, in our case a ValueTuple holding first and last name. The second parameter is a function that retrieves the Person with those names from the repository.

Let's test if it works as expected. If it does, the GetByName method from the repository should be called only once when I try to access the John Smith object two times.

```
var john = peopleController.GetByName("John", "Smith");
john = peopleController.GetByName("John", "Smith");
```

I've set a breakpoint in the GetByName method:

```
class PeopleRepositoryMock : IRepository<Person>
{
    2 references
    public IEnumerable<Person> GetByName(string firstName, string lastName)
    {
        if(firstName == "John" && lastName == "Smith")
        {
            return new[] { new Person("John", "Smith") };
        }
        throw new NotImplementedException();
    }
}
```

Now, when I run this program, I will see that we only hit this breakpoint once. In a real-world application, we could ask for John Smith hundreds of times and instead of asking the database or some API for his data this many times, we will only ask once.

All right. We implemented a super simple cache ourselves. Of course, we could make this cache much more complex. For example, we don't actually remove anything from the cache now, and it may happen that during the execution of the program it will grow extremely big and will finally drain our application of free memory. Usually, some kind of cleanup mechanism is added, that removes the data that is older than some specified time. Also, this cache is not thread-safe, so it shouldn't be used in multithreaded applications.

There are of course some existing libraries that provide the caching mechanisms. For example, we can use NuGet to install **Microsoft.Extensions.Caching.Memory**. It works very similarly to the cache we implemented. This is how the PeopleController would look like if we used Microsoft's implementation of the cache:

```
private readonly MemoryCache _memoryCache =
    new MemoryCache(new MemoryCacheOptions());

//alternatively we can use the Microsoft's MemotyCache
0 references
public Person? GetByNameMemoryCache(string firstName, string lastName)
{
    return _memoryCache.GetOrCreate(
        (firstName, lastName),
        cacheEntry => _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault());
}
```

As you can see it's almost exactly the same as our code, and I highly recommend using this package. Nevertheless, I wanted to show you how to implement a cache on our own, so you understand how it works under the hood.

From other caching tools you should be aware of, one of the most commonly used third-party tools is called **Redis**. It provides more functionality than a regular cache and it's known for its excellent performance.

All right. Before we wrap up a word of caution. Caching is great, but for some scenarios only. If we don't retrieve the data identified by the same key repeatedly, but we keep using different keys all the time, it doesn't really give us anything. You can always check the cache success rate by adding a simple field to your cache:

```
private int _cacheSuccessRate;

1 reference
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if (!_cachedData.ContainsKey(key))
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    else
    {
        _cacheSuccessRate++;
    }
    return _cachedData[key];
}
```

This counter will be incremented each time we use the cached data instead of retrieving it from the data source. You can check its value during debugging and

see if your cache is successful. The higher the counter goes, the better your cache is performing in your application.

So, use caching when it really makes sense.

Caching is most often used to retrieve data from some external sources, but remember that even the data calculated locally can be cached. For example, if your program does some performance-costly mathematical operations that take a lot of time, you could consider caching the results too.

Also, remember that the underlying data can change after it has been first retrieved by the cache. It means the cache will be providing us with **stale** data. That's why caching is best when the underlying data doesn't change often. In this case, it's enough to have some mechanism that removes the piece of data from the cache after some specific time has passed.

Bonus questions:

- "**What are the benefits of using caching?**"
  Caching can give us a performance boost if we repeatedly retrieve data identified by the same key. It can help not only with data retrieved from an external data source but even calculated locally if the calculation itself is heavy (for example some complex mathematical operations).

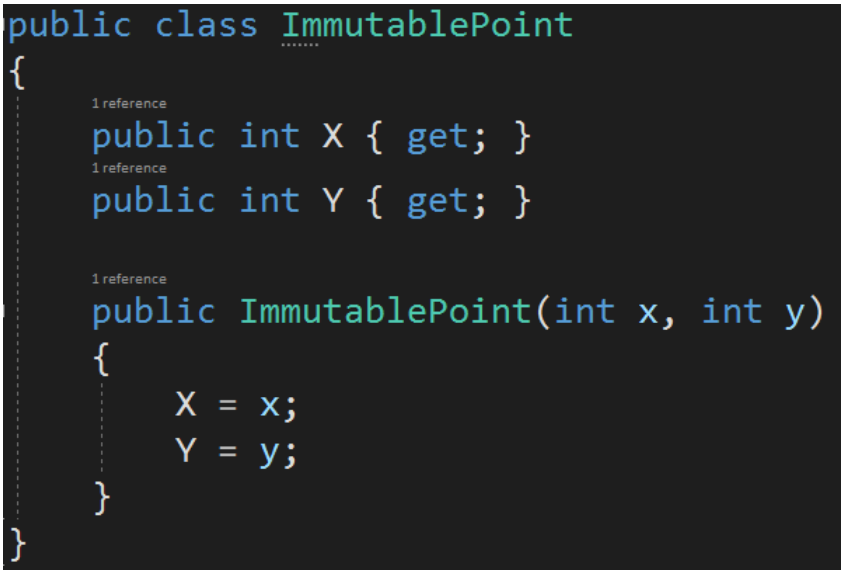
- "**What are the downsides of using caching?**"
  Cache occupies the application's memory. It may grow over time, and some kind of cleanup mechanism should be introduced to avoid OutOfMemoryExceptions. Such mechanisms are usually based on the expiration time of the data. Also, the data in the cache may become stale, which means it changed at the source but the old version is cached and used in the application. Because of that, caching is most useful when retrieving data that doesn't change often.

30. What are immutable types and what's their purpose?

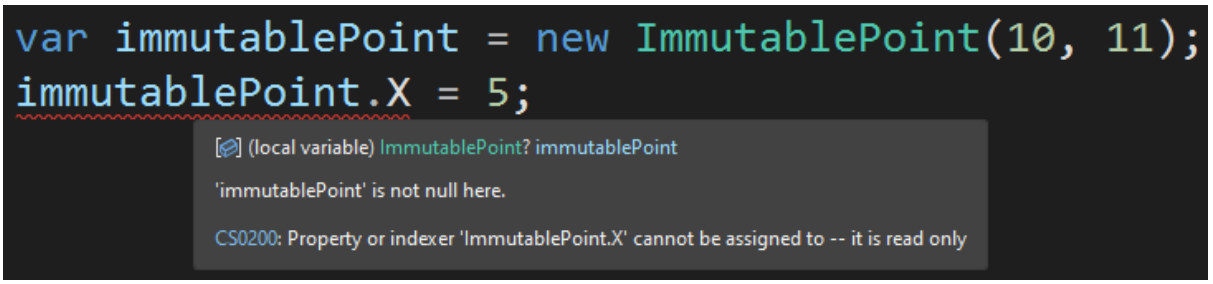
Brief summary: Immutability of a type means that once an object of this type is created none of its fields of properties can be updated. Using immutable types over mutable ones gives a lot of benefits, like making the code simpler to understand, maintain and test, as well as making it thread-safe.

Immutability of a type means that once an object of this type is created none of its fields of properties can be updated.

Let's see a simple immutable type:



The objects of this class are immutable because the X and Y properties do not have setters. Once we create an object using the constructor (the only place where we can assign values to X and Y) it will not be possible to modify it:



As you can see the concept of immutability is very simple. It can extend to more complex types, for example, collections - once we create an immutable collection, it can't be changed, so no element can be added, removed, or altered.

The question is - why should we bother in creating immutable types?

Let's see a couple of the most important benefits of having immutable types:

1) **Clarity and simplicity of the code**
   First, let's see a piece of code that seems simple:

```
var point = new Point(5, 10);
SomeMethod(point);
SomeOtherMethod(point);
Console.WriteLine($"X:{point.X}, Y:{point.Y}");
```

   What do you think will be printed from the fourth line? Well, it's impossible to say, because we don't know what happens in SomeMethod and SomeOtherMethod. Maybe they simply read the values of the point, but maybe they alter it?

```
void SomeMethod(Point point)
{
    point.X = 500;
}
```

   We won't be sure what the code does and how it behaves until we follow the flow of the code very carefully, checking what exactly every method does with the Point object.
   If the Point was immutable, we wouldn't need to worry - we would be sure that once created, its value remains the same.
2) **Pure functions**
   Pure functions are functions whose results only depend on the input parameters, and they do not have any side effects - they don't alter any state of the application, they don't modify the input parameters. We can call a pure function any time we want with the same set of parameters, in any order, and it will always yield the same result. Because of that, we can **cache** the result, making the parameters the key of the cache. Pure functions are **simple to understand**, as we don't need to be aware of the context in which they are called. **Testing** them is extremely simple, as we only check if their result is as expected. For testing purposes, we don't need to set up any

context in which those functions work, as they only depend on the input parameters, and not, for example, the state of the class they live in. Using immutable types and creating pure functions work very well together, and actually, they are two tenets of **functional programming** - a coding paradigm that grows more and more popular for its clarity as well as working great in multithreaded applications. This leads us to the next point:

3) **Thread safety**

When working with multithreaded applications we must always be very cautious when it comes to making any assumptions about the state of an object - because it can always be the case that another thread altered this state without our knowledge. Using immutable objects wipes this problem out. If an object can't be altered, there is no need to worry that some other thread altered it, right? This makes the creation of multithreaded applications much simpler and less error-prone, and you must know that finding bugs in multithreaded applications can be extremely hard, as they often happen in a non-deterministic manner that can be extremely hard to reproduce.

4) **No invalid objects**

Let's consider the Person class and its constructor:

```
public Person(string id, string name, int yearOfBirth)
{
    if (string.IsNullOrEmpty(id) ||
        id.Length != 9 ||
        !id.All(character => char.IsDigit(character)))
    {
        throw new ArgumentException("Id is invalid");
    }
    if (string.IsNullOrEmpty(name) || name.Length < 2 || name.Length > 25)
    {
        throw new ArgumentException("Name is invalid");
    }
    if (yearOfBirth < 1900 | yearOfBirth > DateTime.Now.Year)
    {
        throw new ArgumentException("YearOfBirth is invalid");
    }
    Id = id;
    Name = name;
    YearOfBirth = yearOfBirth;
}
```

Someone clearly put up a lot of work to make sure that when an object of the Person class is created, it is valid - its Id is correct, the name is not empty, the year of birth is reasonable. If objects of the Person class could be mutated, it would mean that at any time of the application execution they can be rendered invalid:

```
var person = new Person("123456789", "John", 1987);
person.Id = null;
```

If Person's class objects can become invalid at any moment, that would mean that our code would quickly fill up with lines like that:

```
if(string.IsNullOrEmpty(person.Name) ||
    person.Name.Length < 2 ||
    person.Name.Length > 25)
{
    //do something
}
else
{
    //handle error case
}
```

This not only is a code duplication (because we already defined this checks in the Person's class constructor) but it also creates noise in the code, making it harder to understand, maintain and test - because in each of those places we must create tests that will handle both valid and invalid person objects. The easier testing is another benefit of immutable objects, but before we move on to this point, let's consider another aspect of making objects invalid:

5) **Prevention of identity mutation.**
Imagine we want to use the Person class object as a key in the dictionary. We want to use the Person's **Id** as the hash code that the dictionary will use.

```
public override int GetHashCode()
{
    return Id.GetHashCode();
}
```

If the Id is mutable, we will lose the object in the Dictionary:

```
var dict = new Dictionary<Person, string>();
dict[person] = "aaa";
person.Id = "new id";
Console.WriteLine(dict[person]);
```

First, we've used the person object as the key in the Dictionary, using its Id's hash code. Then the id changed. Because of that, the fourth line will throw an exception, because there is no key with the hashcode built by the "new id" string in the Dictionary - the only key there is the one built with the old id. As a rule of thumb, if an object is meant to be a key in the dictionary, it should be immutable.

6) **Easier testing**
   Immutable objects make code easier to understand, and they also give us a guarantee that once a valid object had been created, it will remain valid forever. This makes testing much simpler because we have fewer paths of code to test, as handling of invalid objects is simply not needed. Also, we don't need to test if a state of an object had been changed, which is sometimes tricky especially if it's the private state that changes. As mentioned before, using immutable objects makes it easier to create pure functions, and they are extremely simple to test.

All right. Seems like immutable objects can be really beneficial. But following this "nothing ever changes" rule can be demanding. After all, we sometimes need to change *something*. Let's consider the DateTime type, which is immutable in C#. It provides a method called AddDays:

```
var januaryThe1st = new DateTime(2022, 1, 1);
var januaryThe8th = januaryThe1st.AddDays(7);
```

Adding 7 days to January the 1st won't make it a different date. It will produce another date. It makes perfect sense - after all, a date never changes, and January the 1st 2022 will always be January the 1st 2022.

Such "apparent modification" of immutable objects is called **a non-destructive mutation**. It is an operation of creating a new object based on another immutable object. The immutable object won't be modified, but the result of "modification" will become a new object. We will learn more about it in the next lecture "**What are records and record structs?**"

All right. We learned what immutable types are and what are the most important benefits of using them. But we must also be aware of the important **disadvantage** they have: with the non-destructive mutation, each update of an object actually creates a new object, allocating new memory. The old object must be cleaned up by the Garbage Collector. It's usually not an issue with small types, but remember, even collections can be immutable. Imagine having a list of million elements, and that adding a new item to it means actually building a whole new collection of size million and one.

It may sound scary, but don't be discouraged to use immutable types. First of all - there are implementations of collections that actually make this quite efficient. Second - not all applications suffer from performance loss when using immutable types, and the benefits are often bigger than the costs. Garbage Collector is a smart tool, and most often you won't even notice the performance impact of introducing immutable types. Nevertheless, there are cases when performance is critical. For example, I wouldn't recommend making every type immutable when developing video games, as it would make the Garbage Collector kick off more often, and remember that when Garbage Collector works, all other threads are frozen until it finishes. In the case of video games, it could lead to a performance decrease that would be noticed by the players, and we definitely don't want that to happen.

Bonus questions:

- "**What are pure functions?**"
  Pure functions are functions whose results only depend on the input parameters, and they do not have any side effects like changing the state of the class they belong to or modifying the objects passed as an input.

- "**What are the benefits of using immutable types?**"
  The code using immutable types is simple to understand. Immutable types make it easy to create pure functions. Using immutable types makes it easier to work with multithreaded applications, as there is no risk that one thread will modify a value that the other thread is using. Immutable objects retain their identity and validity. Mutable objects make testing problematic. Testing code using immutable types is simpler.

- "**What is the non-destructive mutation?**"
  The non-destructive mutation is an operation of creating a new object based on another immutable object. The immutable object won't be modified, but the result of "modification" will become a new object. The real-life analogy could be adding 7 days to a date of January the 1st. It will not change the date of January the 1st, but it will produce a new date of January the 8th.

31. What are records and record structs?

Brief summary: Records and record structs are new types introduced in C# 9 and 10. They are mostly used to define simple types representing data. They support value-based equality. They make it easy to create immutable types.

Important: records are available since C# 9. Record structs are available since c# 10.

When programming, we often need to define simple data structures, that don't really hold any business logic - they simply store data. Let's define a Point class.

```
public class PointClass
{
    1 reference
    public int X { get; }
    1 reference
    public int Y { get; }

    0 references
    public PointClass(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

This class only holds two integers X and Y. I want objects of this class to be **immutable**, so once set, they will not be updated. That's why I only added getters to the X and Y properties. Setters are not available.

I would like the objects of this class to be **nicely printed.** Now, if I call Console.WriteLine(somePoint) I will get the full type name printed to the console, so "Namespace.PointClass". This is not very convenient. Let's override the ToString method:

```
public override string ToString()
{
    return $"X:{X}, Y:{Y}";
}
```

Now, something like “X:10, Y:5” will be printed.

Next, I would like to use objects of the Point class to as **keys in the Dictionary**. Currently, since Point is a class, its objects are compared by reference. That means, even if I have two points equal by value, they will be considered two different keys in a Dictionary:

```
var somePoint = new PointClass(10, 5);
var otherPointEqualByValue = new PointClass(10, 5);
var dict = new Dictionary<PointClass, string>();
dict[somePoint] = "aaa";
dict[otherPointEqualByValue] = "bbb";
```

After this code is executed, the Dictionary will have a size of **two**, because each point is considered a different key, as they differ by reference. I would like other behavior - if two points are equal by value, they are considered the same key by the Dictionary. To achieve this, I must overwrite the GetHashCode and Equals methods:

```
0 references
public override int GetHashCode()
{
    return HashCode.Combine(X, Y);
}

2 references
public override bool Equals(object? obj)
{
    return obj is PointClass && Equals((PointClass)obj);
}
```

Now I also must provide the Equals method that accepts a **PointClass** object, not an **object**. That means, my PointClass shall implement the IEquatable<PointClass> interface:

```
public bool Equals(PointClass? other)
{
    return other is not null && other.X == X && other.Y == Y;
}
```

Let's see the whole class:

```
public class PointClass : IEquatable<PointClass>
{
    3 references
    public int X { get; }
    3 references
    public int Y { get; }

    2 references
    public PointClass(int x, int y)
    {
        X = x;
        Y = y;
    }

    0 references
    public override string ToString()
    {
        return $"X:{X}, Y:{Y}";
    }

    0 references
    public override int GetHashCode()
    {
        return HashCode.Combine(X, Y);
    }

    0 references
    public override bool Equals(object? obj)
    {
        return obj is PointClass && Equals((PointClass)obj);
    }

    2 references
    public bool Equals(PointClass? other)
    {
        return other is not null && other.X == X && other.Y == Y;
    }
}
```

Well… it works, but it's a lot of code, and all of it only to implement a simple data structure that:

- prints itself nicely

- is compared by value
- provides custom GetHashCode and Equals methods implementations so it can safely be used in hashed collections

At some point, the creators of C# realized that this is a common issue. As a solution, they introduced **records**. Before I explain exactly what records are, let me show you how exactly the same behavior as we defined in the type above can be achieved with records:

```
public record PointRecord(int X, int Y);
```

That's it. Only one line of code, and it does the same thing as 31 lines of code we needed to define the PointClass.

Records are new types, joining classes and structs. They are available starting with C# 9. Let's list the most important information about records:

- records are reference types
- …but they base on value-type equality, which means, two records with identical values of properties will be considered equal even if they differ by reference
- like classes, they support inheritance
- the compiler generates the following methods for records:
    - an override of Equals(object?) method
    - a virtual Equals(ThisRecord?) method (this method comes from the IEquatable<ThisRecord> interface which records implement)
    - and override for the GetHashCode method
    - overloads of == and != operators
    - an override of the ToString method, which prints the names of the properties with their values

The record we defined above is even more special: it's a so-called **positional record**, so a record that doesn't even have a body. Later in the article, we will learn how to define non-positional records. For now let's just note that for positional records, the compiler also generates:

- a primary constructor whose parameters match the positional parameters on the record declaration
- public properties for each parameter of a primary constructor. Those properties are read-only (but they are **not** for **record structs**, which we will learn about a bit later)
- a Deconstruct method to extract properties from the record

All right. Let's see a regular, non-positional record now:

```
public record PointNonPositionalRecord
{
    2 references
    public int X { get; set; } //non-positional records can be read-write
    3 references
    public int Y { get; set; }

    1 reference
    public PointNonPositionalRecord(int x, int y)
    {
        X = x;
        Y = y;
    }

    0 references
    public int Sum() //we can add methods to records
    {
        return X + Y;
    }
}
```

As you can see we need to write a little more code (like explicitly defining the properties and the constructor) but as a reward, we can add methods or make the properties writable. Remember that methods like GetHashCode, Equals, or ToString are still generated by the compiler and we don't need to worry about them.

As we learned in the last lecture, the **immutability** of types is a desired trait. Records are perfect for representing immutable types. To make things even easier, they provide non-destructive mutation implemented with the **with** keyword. This may sound cryptic, so let's see an example. Let's say I have some point, and I want to update its Y property:

```
var somePointRecord = new PointRecord(2, 3);
var somePointBythNewY = somePointRecord with { Y = 6 };
```

With the **with** keyword, I created a **new** Point equal to the old one, but with Y set to the new value of 6. The old point is immutable, so it cannot be changed. I can change as many properties as I want with the "with" keyword.

Starting with C# 10, **record structs** were introduced. They are similar to records, with some differences:

- they are value types
- positional record structs are read-write by default, which means their properties are mutable
- record structs can be declared as readonly, making them immutable

```
public readonly record struct PointReadonlyRecordStruct(int X, int Y);
```

- For record structs, the compiler also generates a parameterless constructor which sets all its properties to the default values

When deciding whether to use records or record structs, you should take the same things into consideration as when deciding whether to use classes or structs. In general - if the type is simple and you want value-type behavior like passing parameters by value, you should go for the record structs.

Also, records and record structs support deconstruction. We learned more about it in the "What is deconstruction?" lecture.

```
var somePointRecord = new PointRecord(2, 3);
var somePointBythNewY = somePointRecord with { Y = 6 };
var (localX, localY) = somePointBythNewY;
```

Let's summarize. Records and record structs are new types introduced in C# 9 and 10. They are mostly used to define simple types representing data. They support value-based equality. They make it easy to create immutable types.

Bonus questions:

- "**What is the purpose of the "with" keyword?**"
  The "with" keyword is used to create a copy of a record object with some properties set to new values. In other words, it's used to perform a non-destructive mutation of records.

- "**What are positional records?**"
  Positional records are records with no bodies. The compiler generates properties, constructor, and the Deconstruct method for them. They are a shorter way of defining records, but we can't add custom methods or writable properties to a positional record.

32. Why does string behave like a value type even though it is a reference type?

Brief summary: String is a reference type with the value type semantics. All strings are immutable, which means when they seem to be modified, actually, a new, altered string is created. String has value-type semantics as this is more convenient for developers, but it can't be a value type because string objects can be large, and value types are stored on the stack which has a limited size.

In C#, there is a big difference in how value types and reference types behave. Let's see this difference in code:

```
var valueType = 5;
Console.WriteLine($"Value type is {valueType}");
Console.WriteLine("Executing AddOneToValueType method");
AddOneToValueType(valueType);
Console.WriteLine($"Now value type is {valueType}");
```

```
void AddOneToValueType(int valueType)
{
    ++valueType;
}
```

In this example, we have a variable of type int, which is a **value type**. It is passed to a method that increments it. Because integers are value types, when they are passed as parameters to methods, a copy of the value is created. That's why after this method is executed, the value of the original variable will not be changed.

```
Value type is 5
Executing AddOneToValueType method
Now value type is 5
```

Let's consider a similar example now, but with **reference types**:

```
var referenceType = new List<int> { 5 };
Console.WriteLine($"Reference type has {referenceType.Count} elements");
Console.WriteLine("Executing AddOneToReferenceType method");
AddOneToReferenceType(referenceType);
Console.WriteLine($"Now reference type has {referenceType.Count} elements");
```

```
void AddOneToReferenceType(List<int> referenceType)
{
    referenceType.Add(6);
}
```

A List<int> is a reference type. It is passed to a method by reference, which means inside the method we add an element to the original List<int> object. That's why after the method is executed, the count of elements in the list is 2:

```
Reference type has 1 elements
Executing AddOneToReferenceType method
Now reference type has 2 elements
```

Now, let's see the last example. This time we will deal with **strings**:

```
var text = "Hello!";
Console.WriteLine($"text is {text}");
Console.WriteLine("Executing AddOneToString method");
AddOneToString(text);
Console.WriteLine($"Now text is {text}");
```

```
void AddOneToString(string text)
{
    text += "1";
}
```

And now, let's see the result:

```
text is Hello!
Executing AddOneToString method
Now text is Hello!
```

The string has **not** been modified. It's the same behavior we've seen for value types. But here is the plot twist: **string is a reference type** in C#! So what is going on?

To understand it, we must first realize, that under the hood **string is an array of chars**. As we learned in the "What is an Array?" lecture, arrays are collections of fixed size. Once an array is created, its size never changes. If we want to add an element to an array, we must declare a new, bigger array, copy the old array to it, and set the value under the last index to the new element. And that's exactly what happens when we modify a string. It's not really changing the original string. It is creating a new string which then gets assigned to the variable:

```
var text = "abc";
text += "1";
```

In this case, a new string gets created, containing the original string with "1" added to the end.

This means strings in C# are **immutable**. A string object is never modified. Even if it seems like it, a new object is actually created under the hood.

As we already know, when passed as a parameter to a method, the string behaves like a value type. But this is not the end of similarities of string to value types. Also, its == operator is overloaded, so it compares strings by value, not by reference. For reference types, equality is compared by reference, so those two Lists will not be equal, because they are two different objects pointed to by two different references:

```
List<int> list1 = new List<int> { 1, 2, 3 };
List<int> list2 = new List<int> { 1, 2, 3 };
```

On the other hand, this equality comparison will return true, because strings are compared by value, even though they are reference types:

```
var string1 = "abc";
var string2 = "abc";
```

Technical reasons aside, it is actually desired for strings to behave more like value types. I think most programmers would be surprised if for the above strings the equality check with the == operator would return false as it should for regular reference types. Also, it would be pretty challenging if the modification of a string in a method to which it was passed as a parameter would affect the original string object. In general, people tend to think of strings in a similar way as they think of value types, and learning to use them as other reference types would most likely make C# quite disliked in the programming community.

You may wonder: since string behaves like a value type, why isn't it one? Maybe another design than using the array of characters would be possible, and string could be a value type like numbers or DateTime?

Well, the answer is (as so often) related to performance. Value types are stored on the stack, which has a limited size (1 MB for 32-bit processes and 4MB for 64-bit processes). Strings can be quite huge, and they could simply not fit on the stack. They are stored on the heap instead, along with other reference types.

One more thing before we wrap up. Because strings are immutable, if we have multiple strings of the same value, we can use the optimization called **Interning**. **Interning** means that if multiple string variables hold strings that are known to be equal, the runtime actually points their references to a single string object, thereby saving memory. This optimization wouldn't work if strings were mutable, because then changing one string variable would have unpredictable results on other string variables.

Let's summarize. String is a reference type with the value type semantics. All strings are immutable, which means when they seem to be modified, actually, a new, altered string is created. String has value-type semantics as this is more convenient for developers, but it can't be a value type because string objects can be large, and value types are stored on the stack which has a limited size.

Because this copy-and-alter way of modifying strings can be performance-costly, it is recommended to use the StringBuilder class when building strings incrementally. We will learn more about it in the next lecture.

Bonus questions:

- "**What is interning of strings?**"
  Interning means that if multiple strings are known to be equal, the runtime can just use a single string, thereby saving memory. This optimization wouldn't work if strings were mutable, because then changing one string would have unpredictable results on other strings.

- "**What is the size of the stack in megabytes?**"
  It's 1 MB for 32-bit processes and 4 MB for 64-bit processes.

- "**What is the underlying data structure for strings?**"
  It's an array of chars. Arrays by definition have fixed size, which is a reason why strings are immutable - we couldn't modify a string by adding new characters to it, because they wouldn't fit in the underlying array.

33. What is the difference between string and StringBuilder?

Brief summary: String is a type used for representing textual data. StringBuilder is a utility class created for optimal concatenation of strings.

String is a type used for representing textual data. StringBuilder is a utility class created for optimal concatenation of strings.

We all know strings. We use them all the time to represent some text. What some people don't know is that all strings are immutable, which means once created they can't be modified.

You may now be surprised. You probably mutated strings plenty of times by now. For example, this code is perfectly valid:

```
var someString = "abc";
someString = "def";
someString += "g";
Console.WriteLine("someString is " + someString);
```

Ta-dah. We modified a string. First, we changed the value from "abc" to "def" and then we added "g" to it, which resulted in the final value of "defg". So what's the fuss about the immutability?

Well, we actually didn't modify the "abc" string. We created a brand new "def" string and we simply pointed the reference stored in **someString** to this new string. A similar thing happened in the next line. We created a new string by concatenating "def" with "g" and then pointed the reference to this new string. At some point in time, the Garbage Collector will see those old strings as objects to whom no reference points, and will remove them from memory.

All right. So we now know that when we "modify" a value of string the following things need to happen:

- a new object needs to be created, which involves allocating memory for it
- the variable that was pointing to the original string must be pointed to the new string

- at some point, the Garbage Collector must clean out the old string

That's relatively a lot of work. In many cases it's ok and we don't notice any performance impact when we add "Mr. " to the "John Smith" string. But a need to build strings gradually from parts is pretty common. For example, imagine your application is downloading some data from the web, and it does it in chunks, as the data is pretty large. You need to create some kind of extract from this data.

```
string finalResult = string.Empty;
using var webConnection = new WebConnection(
    "weatherDatabaseUrl");
while (webConnection.CanRead("weatherData"))
{
    string weatherData = webConnection.Read("weatherData");
    string weatherForDay = weatherData.Substring(0, 100);
    foreach(var weatherForHour in weatherForDay.Split(";"))
    {
        finalResult += weatherForHour;
    }
}
```

For each chunk read (and it can be thousands of them) you need to build some pretty complex string, and then append this string to the string representing the final result. It can involve millions of concatenation operations. As the process continues the final result is growing. At some point, it can be a huge string, and still, every concatenation keeps copying it to a new (huge!) part of memory, adding some tiny part, and then replacing the reference stored in the original variable. Not to mention that behind the scenes the Garbage Collector is struggling to clean up all those large, but unused strings from memory.

And for such use cases, the StringBuilder class has been created.

```
StringBuilder stringBuilder = new StringBuilder();
using var webConnection = new WebConnection(
    "weatherDatabaseUrl");
while (webConnection.CanRead("weatherData"))
{
    string weatherData = webConnection.Read("weatherData");
    string weatherForDay = weatherData.Substring(0, 100);
    foreach(var weatherForHour in weatherForDay.Split(";"))
    {
        stringBuilder.Append(weatherForHour);
    }
}
var finalResult = stringBuilder.ToString();
```

The StringBuilder can add or remove pieces from the final result, but without this laborious copying of the old string, adding a part, and removing the old string. StringBuilder object maintains a buffer to accommodate expansions to the string. New data is appended to the buffer if there is any space in it left. Otherwise, a new, larger buffer is allocated, data from the original buffer is copied to the new buffer, and the new data is then appended to the new buffer. As you can see the only scenario when the entire result is being copied is when the buffer needs to be enlarged.

All right. Let's see a simple program that will measure the performance of string and StringBuilder:

```
(long, string) StringTest(int iterations)
{
    Stopwatch stopWatch = Stopwatch.StartNew();
    string a = "";
    for (int i = 0; i < iterations; i++)
    {
        a += "a";
    }
    stopWatch.Stop();
    return (stopWatch.ElapsedTicks, a);
}

(long, string) StringBuilderTest(int iterations)
{
    Stopwatch stopWatch = Stopwatch.StartNew();
    StringBuilder a = new StringBuilder();
    for (int i = 0; i < iterations; i++)
    {
        a.Append("a");
    }
    stopWatch.Stop();
    return (stopWatch.ElapsedTicks, a.ToString());
}
```

As you can see both methods simply build a string of letters “a” of the length given in the parameter. Let’s see how they will handle 100000 iterations:

```
Concatenation of 100000 strings for string took 43509688
ticks while for StringBuilder it took 5702
string took 762960.1192564012% longer
Are results equal? True
```

Wow. StringBuilder built the result string over 7000 times faster than string.

I hope I convinced you that when you implement some process of incremental building of strings, the StringBuilder should be your choice.

Of course, for simple uses like concatenating a couple of strings, using the StringBuilder is an overkill and it only complicates the code. And if you wanted to have at least a tiny performance boost, I must disappoint you. If the string is not

built incrementally but is composed in a single instruction, plain old string addition actually works faster:

```
var firstName = "John";
var lastName = "Smith";
string name1 = firstName + " " + lastName;
```

The above is faster than the below, not to mention how much simpler it looks:

```
StringBuilder builder = new StringBuilder();
builder.Append(firstName);
builder.Append(" ");
builder.Append(lastName);
string name2 = builder.ToString();
```

The performance of the first code is better because behind the scenes this code is translated into:

```
string name = String.Concat(firstName, " ", lastName);
```

Which is actually quite efficient. Remember, this can only be done if concatenation happens in a single instruction, so it would not work if, for example, we used a loop to build a string from pieces.

All right. The most important thing you need to remember from this lecture is to use StringBuilder when incrementally building large strings, as it gives much better performance than using a simple string.

Bonus questions:

- "**What does it mean that strings are immutable?**"
  It means once a string is created, it can't be modified. When we modify a string, actually a brand-new string is created and the variable that stored it simply has a new reference to this new object.

34. What is operator overloading?

Brief summary: Operator overloading is a mechanism that allows us to provide custom behavior when objects of the type we defined are used as operands for some operators. For example, we can define what will “obj1+obj2” do.

C# provides many operators, for example +, -, ++, ?: etc. The important thing to understand about operators is that their behavior differs depending on what types they are used with. For example, adding two numbers with the + operator will simply calculate the sum of numbers, while adding two strings with the same operator will concatenate those two strings.

When defining our own types, we would often like to provide a custom implementation for some of the operators. Let’s consider a simple Point type:

```
record struct Point
{
    4 references
    public float X { get; }
    4 references
    public float Y { get; }

    6 references
    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}
```

We would like to define the operation of adding two points - it should work by adding their X and Y coordinates, for example adding (10,5) point to (3, -2) shall give a new Point with coordinates (13, 3). We can achieve it by defining the Add method in the Point record struct:

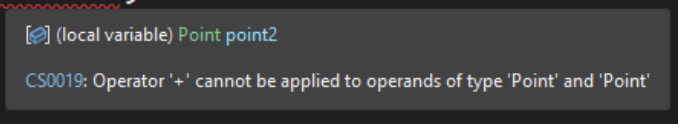
```
public Point Add(Point other)
{
    return new Point(X + other.X, Y + other.Y);
}
```

This is correct, but it's a bit awkward to use. To add two Points we will need to write something like this:

```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1.Add(point2);
```

It would be more natural to perform the addition with the + operator: it is, after all, the addition operator. Unfortunately, this doesn't work:

```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1 + point2;
```



The compiler doesn't know how to add two points yet. To enable the addition of two objects of this type we must overload the addition operator:

```
public static Point operator +(Point point1, Point point2) =>
    new Point(point1.X + point1.X, point1.Y + point2.Y);
```

As you can see to overload the operator we must define a **static** method using the "**operator**" keyword. We must define the parameters and the return type just like in regular methods. In the case of an addition, there are two operands, so we have two parameters. Remember - the operand is the thing to which the operator is applied, for example when adding 3+5 we have two operands: 3 and 5.

Please note that they are operators taking less or more operands. For example, the ++ operator that increases the number by one only takes one operand.

```
int a = 5;
++a;
```

On the other hand, the ternary conditional operator takes three operands: the condition, value if true, and value if false:

```
int a = 5;
var text = a > 1000 ? "big number" : "small number";
```

All right. We overloaded the addition operator for Point type, and now we can safely write this:

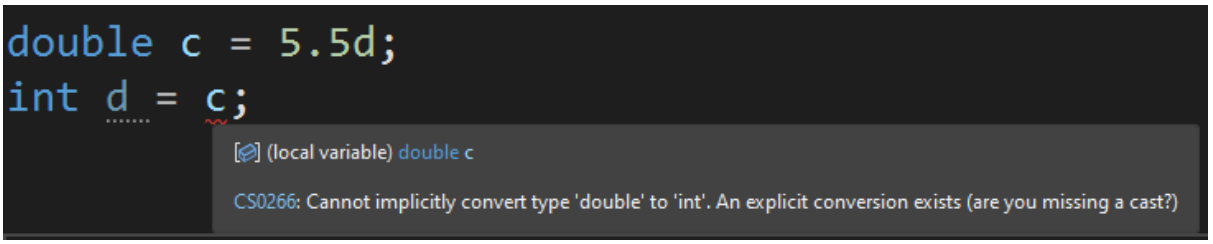
```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1 + point2;
```

We can overload most of the C# operators, but not all of them. For example, we can't overload lambda operator =>, member access operator (a dot, like in obj.Property), or "new" operator. You can find the full list of overloadable operators here, and at the bottom of the table all non-overloadable operators are listed:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading

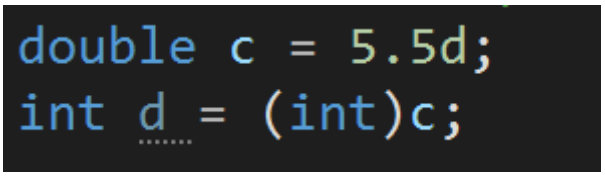
I don't want to show you the overloads for all operators because the code would mostly be the same. But let me show you two interesting and commonly used operators: the **explicit and implicit conversion operators**. First, let me show you some examples of their usage for built-in types.

```
int a = 5;
double b = a;
```

This code looks innocent, but there is more going on here than it seems. After all, we assign an integer to a double. They are two different types, so how does it work? Well, it works because implicit conversion happens. The "a" integer is implicitly converted to a double. Now, let's see the opposite assignment:

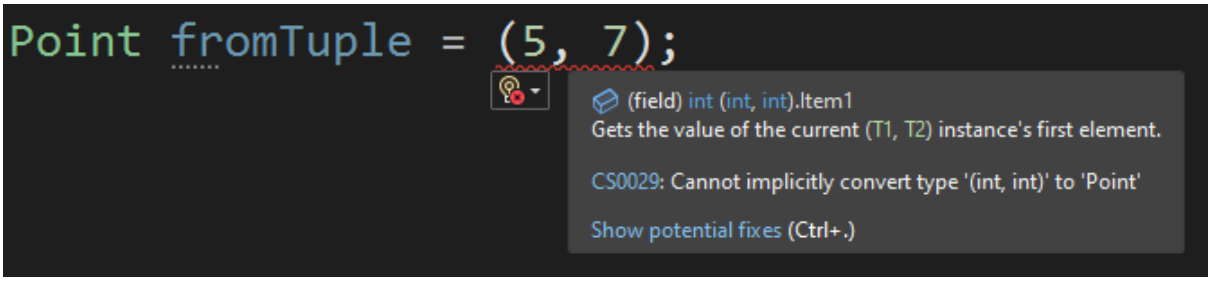


As you can see, this doesn't work. You might be asking, why did it work when assigning an int to a double, but it doesn't work for the opposite operation? The reason for that is simple: the conversion of an int to a double is lossless. The double type can represent the value of 5 that was stored in an int variable. On the other hand, the integer can't represent the number 5.5. When converting 5.5 to int, we will lose some accuracy of the data. The result will be trimmed to a full 5. That's why we must perform such conversion explicitly, so there is no chance we will do it by accident. When using explicit cast we say "I know what I'm doing and I'm aware that the value might actually change during the conversion - I'm ready to take this risk and handle it".



By adding "(int)" I performed the explicit conversion from 5.5 double to int. The result will be 5.

It is quite a common use case that we want to overload the conversion operators. Let's go back to the Point type example. Let's say that our application is getting the points data from some external source and that the points are delivered to us as tuples. We would like to be able to simply assign a tuple of two floats to a variable of Point type, thus performing implicit conversion:



Well, it doesn't work. The compiler doesn't know how to cast a tuple of two numbers to the Point type. We must implement our own implicit conversion operator:

```
public static implicit operator Point((float, float) externalPoint) =>
    new Point(externalPoint.Item1, externalPoint.Item2);
```

We can also overload the explicit conversion operator. If only the explicit conversion operator is implemented, we will have to cast the tuple to Point explicitly:

```
Point fromTuple = (Point)(5, 7);
```

To overload the explicit cast operator we must write this:

```
public static explicit operator Point((float, float) externalPoint) =>
    new Point(externalPoint.Item1, externalPoint.Item2);
```

As you can see, for conversion operators overloading the “explicit” or “implicit” keyword is needed.

Before we wrap up, let’s think about when we should use the implicit, and when the explicit casting operator overloading. We can safely use implicit casting when the cast is lossless, so it won’t change the underlying data - for example, it won’t change its precision. In all other cases, we should use explicit casting, so no data-losing operations are executed behind the scenes, without the programmer’s intention.

Let’s summarize. We can provide custom behavior for operators use in our own types by using operators overloading. We can overload most, but not all C# operators. We can also overload the implicit and explicit conversion operators.

Bonus questions:

- "**What is the purpose of the "operator" keyword?**"
  It is used when overloading an operator for a type.

- "**What is the difference between explicit and implicit conversion?**"
  Implicit conversion happens when we assign a value of one type to a variable of another type, without specifying the target type in the parenthesis. For example, it happens when assigning an int to a double. Explicit conversion requires specifying the type in parenthesis, for example when assigning a double to an int.

35. What are anonymous types?

Brief summary: Anonymous types are types without names. They provide a convenient way of encapsulating a set of read-only properties into a single object without having to explicitly define a type first.

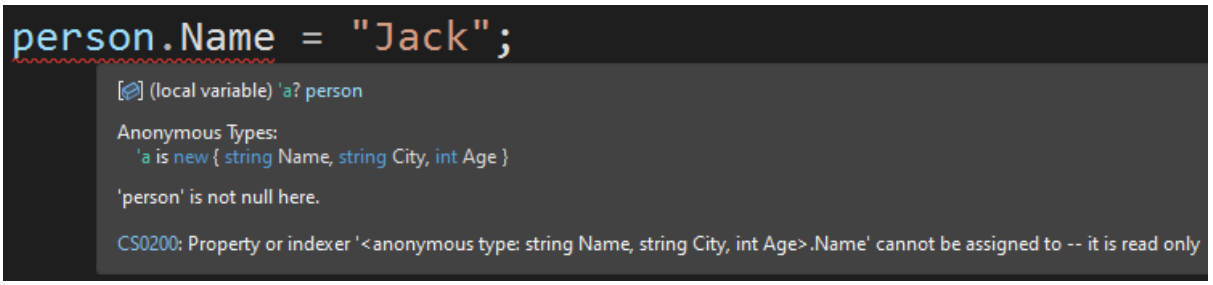
Anonymous types are types without names. Anonymous types provide a convenient way of encapsulating a set of read-only properties into a single object without having to explicitly define a type first.

```
var person = new { Name = "Martin", City = "Savannah", Age = 45 };

Console.WriteLine($"This person's name is {person.Name}," +
    $"he lives in {person.City} and is {person.Age} years old");
```

As you can see, to create an object of an anonymous type we simply use the "new" keyword and then put any properties we want in the curly braces. Here we created an anonymous type with three properties - Name of type string, City of type string, and Age of type int.

The properties of anonymous types are read-only, so code modifying them will not compile:



To understand better what may be the use case for anonymous types, let's consider a simple coding challenge. First, let's define a collection of Pets:

```
var pets = new[]
    {
        new Pet("Hannibal", PetType.Fish, 1.1f),
        new Pet("Anthony", PetType.Cat, 2f),
        new Pet("Ed", PetType.Cat, 0.7f),
        new Pet("Taiga", PetType.Dog, 35f),
        new Pet("Rex", PetType.Dog, 40f),
        new Pet("Lucky", PetType.Dog, 5f),
        new Pet("Storm", PetType.Cat, 0.9f),
        new Pet("Nyan", PetType.Cat, 2.2f)
    };
```

Each Pet has a name, type, and weight. What we want to do is to build a collection of strings that will contain data about each pet type and average weight for pets of this type. The result should be sorted by weight ascending. In other words, it should look like this:

```
Average weight for type Fish is 1.1
Average weight for type Cat is 1.45
Average weight for type Dog is 26.666666
```

We will use LINQ to do it. If you don't know LINQ, check out my other course "LINQ tutorial: Master the Key C# Library". In the last lecture of this course, you can find a discount coupon.

All right. We need to group those pets by type:

```
var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)
```

For each of the groups, I want to calculate the average weight:

```
var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)
    .Select(grouping => grouping.Average(pet => pet.Weight))
```

This is what I want, but there is one problem. I only selected the **average weights** of each group now, but I lost the information about the name of each of those

groups. I must change this code to not select floats (as the average weight is a float) but pairs of PetTypes-floats.

But how should I represent those pairs? I could define a class, struct, or a record for it:

```
record PetTypeAverageWeightPair(
    PetType PetType, float AverageWeight);
```

…but this seems like a relatively big effort. I created a whole separate type for this very specific piece of data. I will probably never use it in a different context. Not to mention that its name is a bit awkward, but how else should we call it? There is really no good name for this very specific set of data.

The solution is to use an anonymous type. An anonymous type is a type defined right where it's needed, without even giving it a name. It's perfect for use cases like ours - where the type is **small and temporary**, **and we don't intend to use it anywhere else:**

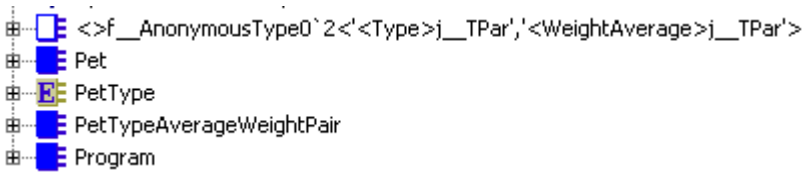
```
var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)
    .Select(grouping => new
    {
        Type = grouping.Key,
        WeightAverage = grouping.Average(pet => pet.Weight)
    })
```

The final code would look like this:

```
var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)
    .Select(grouping => new
    {
        Type = grouping.Key,
        WeightAverage = grouping.Average(pet => pet.Weight)
    })
    .OrderBy(data => data.WeightAverage)
    .Select(data => $"Average weight for type " +
    $"{data.Type} is {data.WeightAverage}");
```

Because the anonymous type we declared doesn't even have a name, we will not be able to use it anywhere else - because how could we refer to it if we don't know its name?

Actually, the compiler gives it a name that can be seen in the Common Intermediate Language, but even if we use the decompiler to find it, it won't be possible to use it. Just to satisfy your curiosity, I checked how the compiler named this particular type:



The name of the anonymous type is at the top. As you can see it's not very readable. Please note that from the perspective of Common Language Runtime anonymous types are no different than any other types.

Let's list the **most important information** about anonymous types:

- they contain only read-only properties
- no other kinds of class members, such as methods or events, are valid
- if no names are given to the properties of the anonymous type, the compiler will use the name of the property that was used to set the value of the anonymous type's property. For example, if instead of this:

```
.Select(grouping => new
{
    Type = grouping.Key,
    WeightAverage = grouping.Average(pet => pet.Weight)
})
```

…we would have this:

```
.Select(grouping => new
{
    grouping.Key,
    WeightAverage = grouping.Average(pet => pet.Weight)
})
```

…the name of the first property would be "Key", the same as the name of the property we assigned to it. When the value is not a property or a field, it

must be given a name. So in the case of WeightAverage, whose value is calculated, we must give it a name - otherwise, it will not compile, which we can see here:

```
.Select(grouping => new
{
    grouping.Key,
    grouping.Average(pet => pet.Weight)
})
```

- Anonymous types are class objects, derived directly from System.Object. They can't be cast to any other type.
- They override the Equals and GetHashCode methods to support value-based equality. Two anonymous objects with the same values will have the same hashcodes, and the Equals method will return true for them. Please note that the == operator is not overloaded, so it will return false (because they differ by reference).
- They support non-destructive mutation with the "with" keyword. Remember: non-destructive mutation is not changing the original object, but rather creating a new one with changed values.

```
var someData = new { number = 5, text = "hello!" };
var changedData = someData with { number = 10 };
```

Bonus questions:

- "**Can we modify the value of an anonymous type property?**"
  No. All properties of anonymous types are read-only.

- "**When should we, and when should we not use anonymous types?**"
  The best use case for anonymous types is when the type we want to use is simple and local to some specific context and it will not be used anywhere else. It's very often used as a temporary object in complex LINQ queries. If the type is complex or we want to reuse it, it should not be anonymous. Also, anonymous types can only provide read-only properties; they can't have methods, fields, events, etc, so if we need any of those features the anonymous types will not work for us.

- "**Are anonymous types value or reference types?**"
  They are reference types since they are classes, but they support value-based Equality with the Equals method. In other words, two anonymous objects with the same values of properties will be considered equal by the Equals method even if their references are different.

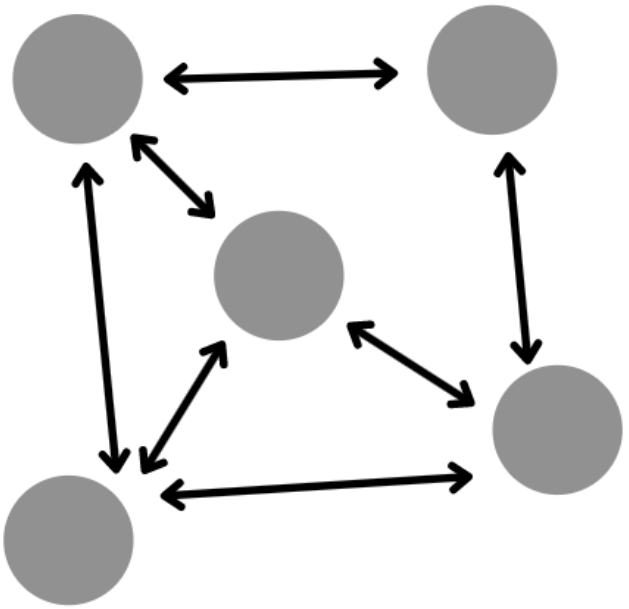
36. What is cohesion?

Brief summary: Cohesion is the degree to which elements of a module belong together. In simpler words, it measures how strong the relationship is between members of a class. High cohesion is a desirable trait of the classes and modules.

Cohesion is the degree to which elements of a module belong together. In simpler words, it measures how strong the relationship is between members of this class or module. The closer related the members of a class are, the better.

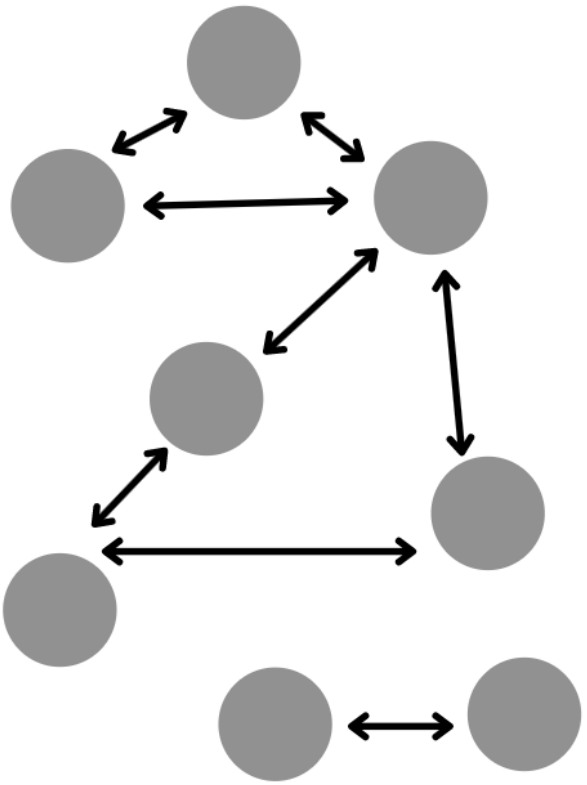
High cohesion is a desirable trait of classes and modules.

This illustrates a highly cohesive class or module:

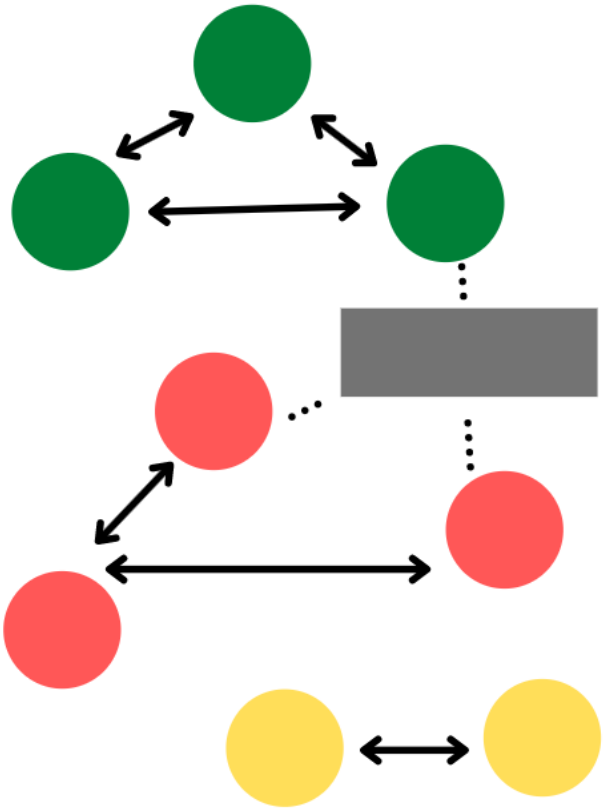


As you can see no piece seems to be “lonely”. There is a lot of connections between them, and it would be hard to draw any line in which this module could be split.

Now, let’s see a module that’s not cohesive:



In this case, there are some pieces that seem to have very little or nothing to do with others. We can easily see how this module could be divided into highly-cohesive modules:



As you can see I've kept the connection between the green and red parts, but I made it go through some abstraction - in C# this would most likely be an interface.

Let's see some code now:

```
class PetsCollection
{
    private List<Pet> _pets = new ();

    0 references
    public void Add(Pet pet) => _pets.Add(pet);
    0 references
    public int Count => _pets.Count;
    1 reference
    public IEnumerable<PetType> GetCurrentlyStoredTypes() =>
        _pets.Select(pet => pet.PetType).Distinct();
    0 references
    public bool Contains(PetType petType) =>
        GetCurrentlyStoredTypes().Any(type => type == petType);
}
```

This class is characterized by **high cohesion**. All those methods use the underlying collection called _pets. The Contains method uses the GetCurrentlyStoredTypes method. None of those methods could be easily moved away from this class. There is no easy way to split this class into separate classes, nor it would make much sense, as those methods naturally belong together.

Now, let's consider a different class:

```
public class HousePricer
{
    private IOwnersDatabase _ownersDatabase;
    private decimal _dollarsPerSquareMeter;
    0 references
    public HousePricer(
        decimal dollarsPerSquareMeter,
        IOwnersDatabase ownersDatabase)
    {
        _dollarsPerSquareMeter = dollarsPerSquareMeter;
        _ownersDatabase = ownersDatabase;
    }

    1 reference
    public decimal GetPrice(House house) =>
        _dollarsPerSquareMeter * (decimal)house.Area *
        GetPriceMultiplierBasedOnFloors(house.Floors);

    1 reference
    private decimal GetPriceMultiplierBasedOnFloors(int floors) =>
        floors switch { 1 => 1m, 2 => 1.5m, _ => 1.6m };

    0 references
    public void SendPriceToOwner(House house) =>
        Console.WriteLine($"Sending price {GetPrice(house)}" +
            $" to {FindOwnerEmail(house.Address)}");

    1 reference
    private string FindOwnerEmail(string address) =>
        _ownersDatabase.GetEmailByAddress(address);
}
```

This class is **not** cohesive. It has two quite separate responsibilities. First, it evaluates a price of a house, and second, it notifies the owner about the calculated price. The only point where those two responsibilities meet is that the SendPriceToOwner method needs the information about the price, but this is something that can be easily refactored.

Let’s create two highly-cohesive classes:

```
public class HousePricer
{
    private decimal _dollarsPerSquareMeter;
    0 references
    public HousePricer(decimal dollarsPerSquareMeter)
    {
        _dollarsPerSquareMeter = dollarsPerSquareMeter;
    }

    0 references
    public decimal GetPrice(House house) =>
        _dollarsPerSquareMeter * (decimal)(house.Area) *
        GetPriceMultiplierBasedOnFloors(house.Floors);

    1 reference
    private decimal GetPriceMultiplierBasedOnFloors(int floors) =>
        floors switch { 1 => 1m, 2 => 1.5m, _ => 1.6m };
}
```

```
public class OwnerNotifier
{
    private IOwnersDatabase _ownersDatabase;
    1 reference
    public OwnerNotifier(IOwnersDatabase ownersDatabase)
    {
        _ownersDatabase = ownersDatabase;
    }

    0 references
    public void SendToOwner(string information, string address) =>
        Console.WriteLine($"Sending {information}" +
            $" to {FindOwnerEmail(address)}");

    1 reference
    private string FindOwnerEmail(string address) =>
        _ownersDatabase.GetEmailByAddress(address);
}
```

Now we can simply use them one after another:

```
var housePricer = new HousePricer(2000);
var price = housePricer.GetPrice(house);
var ownerNotifier = new OwnerNotifier(ownersDatabase);
ownerNotifier.SendToOwner(price.ToString(), house.Address);
```

By now you may probably be thinking "Oh, so high cohesion and Single Responsibility Principle are the same things?". Well, no, but it's common that a highly cohesive class meets the SRP and vice versa.

High cohesion means that the data and methods that belong together, are kept together. If following only the SRP, we **could** (but it doesn't mean we should!) keep splitting classes into smaller pieces until every class would have only one public method. Each of those tiny classes would definitely meet the SRP, as they would only have a single responsibility and single reason to change. But they wouldn't be cohesive, as they should belong together.

But, does it mean we should do it?

Well, no! Imagine what would happen if the List class was split into tiny classes, like ListAdder, ListRemover, ListClearer, ListCountGetter, etc. That would be unmaintainable and hard to understand. Now all those methods - Add, Remove, Clear and the Count property belong to a highly-cohesive List class. This class is focused on providing a generic, dynamic collection, and this is its responsibility. It still meets the SRP, because it has one reason to change - it will change if the idea of how such collection structure should be represented in C# changes.

If you want to read more about the relation between the SRP and high cohesion, I recommend this thread on Stack Overflow:
https://stackoverflow.com/questions/11215141/is-high-cohesion-a-synonym-for-the-single-responsibility-principle

High cohesion is not something we should create. It's something we observe and our job is not to break it. So how to recognize high cohesion?

High cohesion	Low cohesion
most or all members use the same private data and they reuse member methods	some private members are used by a group of members only; other members are used by a different group
the functionalities of a class have much in common	the functionalities of a class are unrelated
the class would be hard to split - if we did it, a lot of private data would need to be passed from one part to another	the class is easy to split and the line of splitting is natural and obvious
class is easy to name and its name is accurate	class is hard to name precisely or its name lies about what it does

If you see high cohesion - don't break it. High cohesion gives us a lot of benefits:

- Highly cohesive classes are easier to understand and use. They provide a highly-focused set of operations instead of more functionality than we need. Think of our OwnerNotifier class - it could easily be reused to send some other information to the person living at some address.
- When a change is needed, it's easier to introduce, as it affects fewer modules.
- Cohesive classes are easy to test.
- They are reusable.

On the other hand, when you see a class that is not highly cohesive, consider refactoring it and splitting it into highly-cohesive pieces.

Let's summarize. Cohesion is the degree to which elements of a module belong together. In simpler words, it measures how strong the relationship is between members of a class. High cohesion is a desirable trait of the classes and modules.

Bonus questions:

- "**Is following the Single Responsibility Principle and keeping high cohesion the same thing?**"
  No, but it's common that a highly cohesive class meets the SRP and vice versa. High cohesion means that the data and methods that belong together, are kept together. If following only the SRP, we ***could*** *(but it doesn't mean we should!) keep splitting classes into smaller pieces until every class would have only one public method. Each of those tiny classes would definitely meet the SRP, as they would only have a single responsibility and single reason to change. But they wouldn't be cohesive, as they should belong together.*

37. What is coupling?

Brief summary: Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

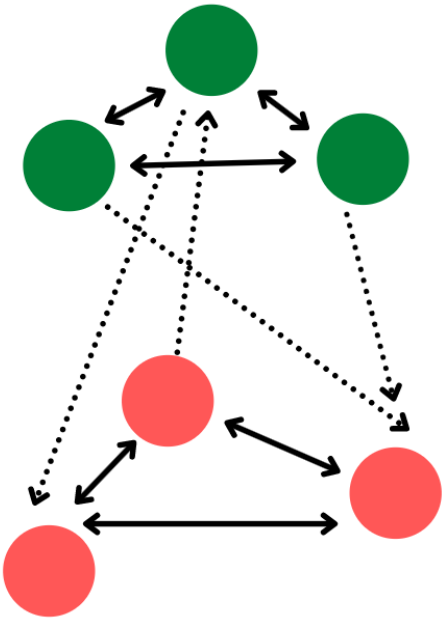
Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

Have you ever needed to introduce a small change in a class, but it actually forced you to also introduce changes in many other classes? Well, it seems like those classes were highly coupled with each other. It made them brittle - they got broken and needed to be fixed when a change was introduced somewhere else.

The high (or "strong") coupling means that one class knows too much about what is going on under the hood of another class.

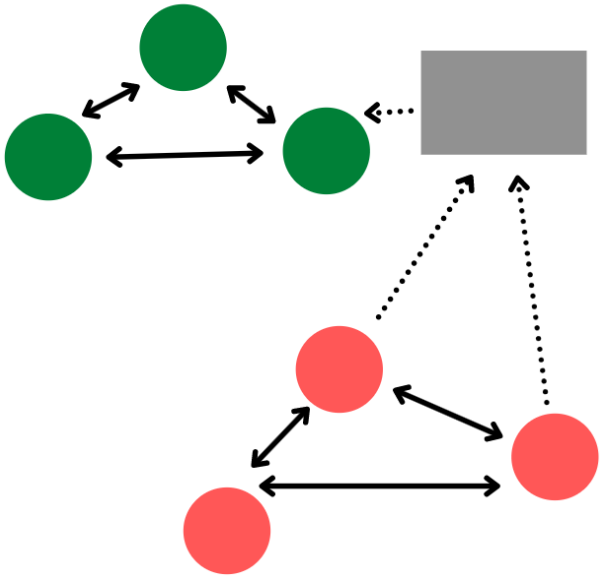
Low (or "loose") coupling is a desirable trait of classes.

This illustration shows the strong coupling between two classes:



Those classes, although separate, know way too much about each other, and they communicate directly between themselves. To reduce coupling, we should

introduce a simple, well-defined, and abstract interface, that will be the channel through which they communicate.



This way, if something changes in one of the classes, the other will not be affected, as long as the interface doesn't change. And remember, the implementation details change much more frequently than interfaces.

Let’s see some strongly-coupled classes.

```
public record Subscriber(string Email, bool IsPremium);

2 references
public class Subscribers
{
    public Subscriber[] Items;
}
1 reference
public class NewsletterSender
{
    private Subscribers _subscribers;

    0 references
    public NewsletterSender(Subscribers subscribers)
    {
        _subscribers = subscribers;
    }

    0 references
    public void SendTo(bool premiumSubscribersOnly)
    {
        for (int i = 0; i < _subscribers.Items.Length; i++)
        {
            if(!premiumSubscribersOnly ||
                _subscribers.Items[i].IsPremium)
            {
                Console.WriteLine(
                    $"Newsletter sent to " +
                    $"{_subscribers.Items[i].Email}");
            }
        }
    }
}
```

At first glance, it may look all right. But notice how the SendTo method (and thus the whole NewsletterSender class) depends on implementation details of the Subscribers class. It is not only aware that it holds a very concrete type of collection (an array) but it could even modify its elements. Let’s see what would happen if I wanted to change the collection that the Subscribers class use from an array to HashSet:

```
public class Subscribers
{
    4 references
    public HashSet<Subscriber> Items { get; }
    0 references
    public Subscribers(HashSet<Subscriber> items) => Items = items;
}
```

The SendTo method breaks:

```
public void SendTo(bool premiumSubscribersOnly)
{
    for (int i = 0; i < _subscribers.Items.Length; i++)
    {
        if(!premiumSubscribersOnly ||
            _subscribers.Items[i].IsPremium)
        {
            Console.WriteLine(
                $"Newsletter sent to " +
                $"{_subscribers.Items[i].Email}");
        }
    }
}
```

I changed an **implementation detail** in the Subscribers class and it **shouldn't affect** any other classes. It did, which proves that our code is brittle.

Let's fix it. The Subscribers class should only expose an abstract collection of items - let's make it IEnumerable. The consumers of this class don't need to know whether is an array, a HashSet, or anything else:

```
public class Subscribers
{
    3 references
    public IEnumerable<Subscriber> Items => _items;
    2 references
    private HashSet<Subscriber> _items { get; }
    0 references
    public Subscribers(HashSet<Subscriber> items) => _items = items;
}
```

Now, let's adjust the code in the NewsletterSender class:

```
public void SendTo(bool premiumSubscribersOnly)
{
    foreach(var subscriber in _subscribers.Items.Where(s =>
        !premiumSubscribersOnly || s.IsPremium))
    {
        Console.WriteLine(
                $"Newsletter sent to " +
                $"{subscriber.Email}");
    }
}
```

Great. Now the NewsletterSender class is not aware of any implementation details of other classes. As far as it's concerned, the Subscribers class only provides a collection that can be enumerated. Whether it's an array, a List, or anything else is irrelevant, and can change without the NewsletterSender class even knowing.

You can recognize high coupling by observing the following:

- One type uses another type directly, without having any abstraction in between.
- Even a small change in a class leads to a cascade of changes all around the project.
- Classes are not independent. To make some object work, we need to set up some state in other objects. This is particularly visible in testing - when setting up a test, you must do a lot of work on other objects than the one that you actually want to test.

The question is, what can we do when we observe that our code is tightly coupled? The best solution is to simply reduce the direct connections between concrete types.

Let me illustrate it like this: let's say I want to go for a trip by the sea. If I am tightly coupled with the Car class and I only accept it as the mean of transportation. It may mean that my weekend will be ruined if my car breaks down or, for example, my driving license expires. On the other hand, if I would only depend on some IMeanOfTransport service, it would mean that I am not coupled with any concrete type implementing it, and I could easily switch whatever I use to a plane or a train. And my weekend would be saved. I wouldn't depend on the technical details of the mean of transport. I would only need to be provided with something I can use to travel, and what it is or how it works under the hood, I don't really care as long as it takes me to the beach.

As you can see, to reduce coupling we should have different types communicate over interfaces, not directly. If you know the Dependency Inversion Principle from the SOLID principles, you can see that its main purpose is reducing coupling: according to this principle, types should not depend on concrete implementations, but rather on abstractions. By following this principle, we remove the direct way of communication between classes, making them more independent from each other.

The perfect classes and modules should be highly cohesive and loosely coupled.

Let's summarize. Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

Bonus questions:

- "**How to recognize strongly couples types?**"
  One type uses another type directly, without having any abstraction in between. We often recognize strong coupling the hard way: when we see that even a small change in a class leads to a cascade of changes all around the project. It proves that the types are not independent.

- "**Which of the SOLID principles allow us to reduce coupling?**"
  The Dependency Inversion Principle, which says that classes shouldn't depend on concrete implementations, but rather on abstractions. When following this principle we remove the direct way of communication between classes, making them more independent from each other.

38. What is the Strategy design pattern?

Brief summary: The Strategy Design pattern is a pattern that allows us to define a family of algorithms to perform some tasks. The concrete strategy can be chosen at runtime.

The Strategy design pattern is a pattern that allows us to define a family of algorithms that perform some tasks. The concrete strategy can be chosen at runtime.

Let's imagine we implement a platform selling video games. Here is the Game type and some games we currently have in our database:

```
record Game(
    string Title,
    decimal Price,
    decimal Rating,
    DateTime ReleaseDate,
    bool IsAvailable);
```

```
var games = new List<Game>
{
    new Game(
        "Stardew Valley", 19.99m, 98,
        new DateTime(2016, 2, 26), true),
    new Game(
        "Red Dead Redemption II", 60m, 92,
        new DateTime(2018, 10, 26), true),
    new Game(
        "Spiritfarer", 25m, 95,
        new DateTime(2020, 8, 18), false),
    new Game(
        "Heroes III", 10m, 82,
        new DateTime(1999, 3, 3), false),
    new Game(
        "God of War", 60m, 97,
        new DateTime(2018, 4, 20), true),

};
```

At the first version of the platform, the user can only search for games by their title. By definition, we don't want to show games that are not available. The code to implement this behavior could look like this:

```
IEnumerable<Game> FindByTitle(
    IEnumerable<Game> games,
    string searchWord)
{
    return games.Where(g => g.isAvailable &&
    g.Title.Contains(searchWord));
}
```

Great. After some time our platform evolves, and we are asked to add some pre-defined filters to the search options. The first one is "Best games" which returns games with a rating of 95 or more:

```
IEnumerable<Game> FindBestGames(
    IEnumerable<Game> games)
{
    return games.Where(g => g.isAvailable &&
    g.Rating > 95);
}
```

All right. This method is quite similar to the one that we had before, but let's not jump to refactoring yet - we perhaps have better things to do. But soon after, we are asked to add other predefined filters: "Games of this year" showing games released in the current year, and "Best deals" finding games with prices below 25$.

```
IEnumerable<Game> FindGamesOfThisYear(
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable &&
    g.ReleaseDate.Year == DateTime.Now.Year);
}

IEnumerable<Game> FindBestDeals(
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable &&
    g.Price < 25);
}
```

Well… this starts to look unmanageable. All those methods are almost identical, and the code is duplicated. Before we start refactoring, let's see how this code could be used:

```
var selectedOption = FilteringType.BestGames;
var searchWord = "Red";
IEnumerable<Game> filteredGames = null;
switch (selectedOption)
{
    case FilteringType.ByTitle:
        filteredGames = FindByTitle(games, searchWord);
        break;
    case FilteringType.BestGames:
        filteredGames = FindBestGames(games);
        break;
    case FilteringType.GamesOfThisYear:
        filteredGames = FindGamesOfThisYear(games);
        break;
    case FilteringType.BestDeals:
        filteredGames = FindBestDeals(games);
        break;
}
```

All right. This doesn't look good. It's high time to introduce the Strategy design pattern. According to this pattern, we should be able to define a family of algorithms that can be injected into some other code at runtime. In our case, the

family of algorithms will contain all predicate methods, that decide whether a game should be included in filtered results or not:

```
Func<Game,bool> SelectStrategy(
    FilteringType selectedOption, string searchWord)
{
    switch (selectedOption)
    {
        case FilteringType.ByTitle:
            return game => game.Title.Contains(searchWord);
        case FilteringType.BestGames:
            return game => game.Rating > 95;
        case FilteringType.GamesOfThisYear:
            return game => game.ReleaseDate.Year ==
                DateTime.Now.Year;
        case FilteringType.BestDeals:
            return game => game.Price < 25;
        default:
            throw new ArgumentException("Invalid option");
    }
}
```

Each strategy is an algorithm enclosed in executable code. In this case, I return it as a Func, but returning it as an object implementing an interface would also be valid and in line with this design pattern. Remember, after all, a Func is like an interface with a single method.

We can now plug this strategy into code doing actual filtering:

```
IEnumerable<Game> FindBy(
    Func<Game, bool> strategy,
    IEnumerable<Game> games)
{
    return games.Where(g => g.IsAvailable && strategy(g));
}
```

And this is how it all can be used together:

```
var strategy = SelectStrategy(selectedOption, searchWord);

var filteredGames = FindBy(strategy, games);
foreach (var game in filteredGames)
{
    Console.WriteLine(game);
}
```

As you can see, this is quite simple. You probably used this pattern before, even if you did not know it. Everywhere where you pass some interchangeable code as a parameter - especially a Func or various objects representing a single interface - you were using the Strategy design pattern.

Using this pattern allowed us to remove code duplications. We have now one clear place where the algorithms are defined. Adding a new way of filtering would now only mean that we must add another **case** to the **switch** in which we define methods of filtering. The generic filtering algorithm (defined in the FindBy method) and specific subfilters are now separated. This makes the code simpler and more easily testable.

So, to summarize: the Strategy design pattern is a pattern that allows us to define a family of algorithms that perform some tasks. The concrete strategy can be chosen at runtime.

Bonus questions:

- "**What are the benefits of using the Strategy design pattern?**"
  It helps to reduce code duplications, makes the code cleaner and more easily testable. It separates the code that needs to be changed often (the particular strategy) from the code that doesn't change that much (the code using the strategy).

39. What is the Dependency Injection design pattern?

Brief summary: Dependency Injection is providing the objects some class needs (its dependencies) from the outside, instead of having it construct them itself.

Dependency Injection means providing the objects that some class needs (its dependencies) from the outside, instead of having it construct them itself.

Let's see this in practice. First, the code that does **not** use the Dependency Injection:

```
class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var peopleDataReader = new PeopleDataReader();

        var people = peopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

The PersonalDataFormatter needs to use the PepleDataReader - it means, the PeopleDataReader is its **dependency**. In this code, the PersonalDataFormatter creates the PepleDataReader object itself using the **new** operator.

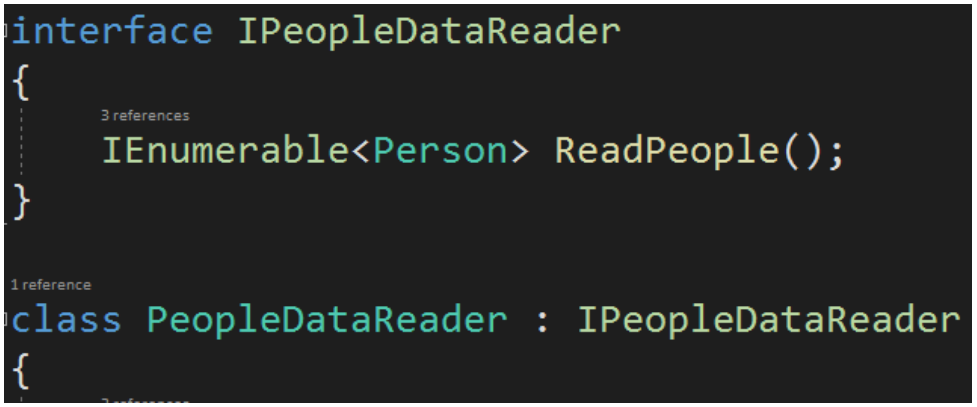
There are a couple of issues with this design:

- PersonalDataFormatter depends on a very particular implementation of people's data reading logic. What if we wanted to use a different data source? We would not have any way of doing this, as this class commits to using the specific PeopleDataReader object by creating it with the **new** operator. Now those two classes are **tightly coupled**.
- This is particularly problematic when we want to unit test this code. Let's assume the PeopleDataReader connects to a real database and sources

people's information from there. If we created the PersonalDataFormatter in tests, it would instantiate the PeopleDataReader, which would try to access the database. This is not acceptable in unit tests. We must have a way of providing a mock implementation instead. With the current design, it's not possible. We will learn more about mocks in the "**What are mocks?**" lecture.

- We are breaking the Single Responsibility Principle here. The PersonalDataFormatter should only be responsible for formatting personal data, but now it is also responsible for creating a PeopleDataReader object. In this simple code this may not seem like an issue, but keep in mind that in real-life applications it's often much more complicated to create an object, as it may have many dependencies of its own.

All right. Let's refactor this code to use Dependency Injection. First of all, let's make the PeopleDataReader implement an interface:



And now, let's inject this dependency to the PersonalDataFormatter, instead of creating it right in it:

```
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

    0 references
    public PersonalDataFormatter(
        IPeopleDataReader peopleDataReader)
    {
        _peopleDataReader = peopleDataReader;
    }

    0 references
    public string Format()
    {
        var people = _peopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

This **solves** all problems mentioned before:

- The classes are now **loosely coupled**. We can easily switch the object we pass to the PersonalDataFormatter's constructor to any other object implementing IPeopleDataReader interface. We can also do it at runtime. PersonalDataFormatter doesn't know anything about the concrete PeopleDataReader class. All it cares about is that it's being provided a dependency that can retrieve people's data - it doesn't care how it is done exactly.
- Because of that, we can easily provide a **mock** of the IPeopleDataReader in tests, to avoid connecting to a real database.
- The PersonalDataFormatter is no longer responsible for creating PeopleDataReader object. The creation of this object and using it are separated. The Single Responsiblilty Principle is not broken and we maintain the separation of concerns.

As you can see, the Dependency Injection is a straightforward pattern, yet it solves a lot of problems.

In C#, we most typically use the constructor injection - so the dependency is injected to a class via its constructor. It is also possible to inject dependency via a setter, but this is much less popular (as, in general, having a public setter is a risky business):

```
class PersonalDataFormatter
{
    1 reference
    public IPeopleDataReader PeopleDataReader { get; set; }

    0 references
    public string Format()
    {
        var people = PeopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

Before we mentioned that classes should not be responsible for creating their dependencies. Well, who should be responsible for it, then? Most typically we have two places where we construct objects, depending on our needs:

- if we need objects that can be constructed right at the program start (for example a logger that will be reused throughout the application) we can create them at the **entry point** of the application, like the Main method, and then pass them down to whatever class that need them:

```
public static void Main(string[] args)
{
    //creation of objects
    var logger = new Logger();
    var peopleDataReader = new PeopleDataReader(logger);
    var personalDataFormatter = new PersonalDataFormatter(
        peopleDataReader, logger);

    //actual run of the application
    Console.WriteLine(personalDataFormatter.Format());

    Console.ReadKey();
}
```

Please notice that in many real-life projects the creation of objects is not done manually, but with **Dependency Injection frameworks**. They are mechanisms that automatically create dependencies and inject them into objects that need them. Dependency Injection frameworks are configurable, so we can decide what concrete types will be injected into objects. They can also be configured to reuse one instance of some type or to create separate

instances for each object that needs them. Some of the popular Dependency Injection frameworks in C# are Autofac or Ninject.

- If we are not sure what objects exactly we need (a concrete type may depend on some parameter or configuration provided at runtime), or whether we will need them at all, we can use a **factory**. Let's say that in PersonalDataFormatter we can either use the default formatting or formatting provided from the outside. The decision which one will be used is done at runtime, and it depends on the value of a parameter of the Format method:

```
public string Format(bool isDefaultFormatting)
{
    _logger.Log("Formatter running...");
    var people = _peopleDataReader.ReadPeople();

    if (isDefaultFormatting)
    {
        return string.Join("\n",
          people.Select(p => $"{p.Name} born in" +
          $" {p.Country} on {p.YearOfBirth}"));
    }
    var formatter = _formatterFactory.Create();
    return formatter.Format(people);
}
```

If the isDefaultFormatting parameter is set to true, we don't need to create a Formatter object at all. In other words, the action of creating an object must happen right in this class, so this object can't be injected. But we don't want to lose the benefits of dependency injection.

A factory allows us to achieve this. Please note that the factory returns an interface, so for testing purposes, we can provide a mock of the factory, that will create a mock of an actual Formatter.

```
interface IFormatterFactory
{
    2 references
    IFormatter Create();
}

1 reference
class FormatterFactory : IFormatterFactory
{
    2 references
    public IFormatter Create()
    {
        return new NameOnlyFormatter();
    }
}
```

```
interface IFormatter
{
    2 references
    string Format(IEnumerable<Person> people);
}

1 reference
class NameOnlyFormatter : IFormatter
{
    2 references
    public string Format(IEnumerable<Person> people)
    {
        return string.Join("\n",
            people.Select(p => p.Name));
    }
}
```

This way we don't need to create the Formatter object upfront. Maybe it will not be created at all if the Format method is never called with the isDefaultFormatting parameter set to false. Of course, in this sample code it wouldn't matter that much, but again: in a real-life application the creation of an object might be more complicated and performance-costly.

All right. We learned that Dependency Injection is a design pattern, according to which we should provide the dependencies that an object needs instead of having it construct them itself.

Dependency Injection is a specific kind of Inversion of Control, which we will learn about in the next lecture.

Bonus questions:

- "**What are Dependency Injection frameworks?**"
  *Dependency Injection frameworks are mechanisms that automatically create dependencies and inject them into objects that need them. They are configurable, so we can decide what concrete types will be injected into objects depending on some abstractions. They can also be configured to reuse one instance of some type or to create separate instances for each object that needs them. Some of the popular Dependency Injection frameworks in C# are Autofac or Ninjec*t.

- "**What are the benefits of using Dependency Injection?**"
  Dependency Injection decouples a class from its dependencies. The class doesn't make the decision of what concrete type it will use, it only declares in the constructor what interfaces it will need. Thanks to that, we can easily switch the dependencies according to our needs, which is particularly useful when injecting mock implementations for testing purposes.

40. What is the Template Method design pattern?

Brief summary: Template Method is a design pattern that defines the skeleton of an algorithm in the base class. Specific steps of this algorithm are implemented in derived classes.

Template Method is a design pattern that defines the skeleton of an algorithm in the base class. Specific steps of this algorithm are implemented in derived classes.

Let's consider the following example: we are developing a platform that allows users to play board games online. The first board game we deliver is Settlers of Catan. Here is the (slightly simplified) implementation:

```
class SettlersOfCatan
{
    private Random _random = new Random();

    1 reference
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    private void SetupBoard()
    {
        Console.WriteLine("Randomly placing hexagonal tiles.");
    }

    1 reference
    private bool PlayTurn()
    {
        Console.WriteLine("Building, trading, etc.");
        return _random.Next(5) >= 4;
    }

    1 reference
    private void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one who first got 12 points");
    }
}
```

All right. Soon after we are asked to implement another game - this time it's Terraforming Mars:

```
class TerraformingMars
{
    private Random _random = new Random();

    0 references
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    private void SetupBoard()
    {
        Console.WriteLine("Choosing from two available maps.");
    }

    1 reference
    private bool PlayTurn()
    {
        Console.WriteLine(
            "Raising oxygen level, placing oceans, etc.");
        return _random.Next(5) >= 4;
    }

    1 reference
    private void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one with most points at game's end.");
    }
}
```

Huh. This is quite similar to the code we had before. After implementing couple more board games, we come to a revelation: all bords games follow a similar **template**! We first set up the board, then we play turns until the game is finished, and finally, we select the winner.

Instead of repeating this logic in each class, we could define it once in the base class, and ask the subclasses to only provide the details of the implementation of each step. This way, if the template changes for some reason, we will only have one place to fix.

Let's use the Template Method design pattern in this code. First, let's define the template itself. It will be done by using an abstract class:

```
abstract class BoardGame
{
    protected Random Random = new Random();

    0 references
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    protected abstract void SetupBoard();
    1 reference
    protected abstract bool PlayTurn();
    1 reference
    protected abstract void SelectWinner();
}
```

Now we can implement the concrete games:

```
class SettlersOfCatan : BoardGame
{
    2 references
    protected override void SetupBoard()
    {
        Console.WriteLine("Randomly placing hexagonal tiles.");
    }

    2 references
    protected override bool PlayTurn()
    {
        Console.WriteLine("Building, trading, etc.");
        return Random.Next(5) >= 4;
    }

    2 references
    protected override void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one who first got 12 points");
    }
}
```

```
class TerraformingMars : BoardGame
{
    private Random _random = new Random();

    2 references
    protected override void SetupBoard()
    {
        Console.WriteLine("Choosing from two available maps.");
    }

    2 references
    protected override bool PlayTurn()
    {
        Console.WriteLine(
            "Raising oxygen level, placing oceans, etc.");
        return _random.Next(5) >= 4;
    }

    2 references
    protected override void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one with most points at game's end.");
    }
}
```

Great. Now the thing that those classes had in common - so the general template of each game - is enclosed in the base type. If this template changes, we will only need to adjust the base class. The derived classes only define what makes each board game special, and they don't replicate what they have in common.

The Template Method design pattern is useful everywhere where some base algorithm is needed, but the specific parts of it vary. A practical example could be the execution flow of tests in the unit tests framework. Typically such execution looks like this:

Foreach test:

1) **Run the SetUpMethod**
2) **Execute test**
3) **Run the TearDown method**

This could easily be achieved with the Template Method design pattern. First, let's define the base class for all test fixtures:

```
abstract class TestFixture
{
    1 reference
    public bool Run()
    {
        int failedTestsCount = 0;
        foreach (var test in GetTests())
        {
            SetUp();
            if (!test())
            {
                failedTestsCount++;
            }
            TearDown();
        }
        return failedTestsCount == 0;
    }

    2 references
    protected abstract IEnumerable<Func<bool>> GetTests();
    2 references
    protected abstract void SetUp();
    2 references
    protected abstract void TearDown();
}
```

And now, let's define some actual tests. We will be testing this super-complicated class:

```
class Calculator
{
    0 references
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

```
class CalculatorTests : TestFixture
{
    private Calculator _cut;

    2 references
    protected override void SetUp()
    {
        _cut = new Calculator();
        Console.WriteLine("SetUp of MyTests class");
    }

    2 references
    protected override IEnumerable<Func<bool>> GetTests()
    {
        return new List<Func<bool>>()
        {
            () =>
            {
                Console.WriteLine("3 + 2 shall be 5");
                return _cut.Add(3,2) == 5;
            },
            () =>
            {
                Console.WriteLine("10 + (-10) shall be 0");
                return _cut.Add(10, -10) == 0;
            }
        };
    }

    2 references
    protected override void TearDown()
    {
        Console.WriteLine("TearDown of MyTests class");
    }
}
```

When we run those tests, we will see that the SetUp and TearDown methods are executed before and after each test, as expected:

```
var calculatorTests = new CalculatorTests();
Console.WriteLine(calculatorTests.Run() ? "Success!" : "Failure");
```

```
SetUp of MyTests class
3 + 2 shall be 5
TearDown of MyTests class
SetUp of MyTests class
10 + (-10) shall be 0
TearDown of MyTests class
Success!
```

All right! As you can see, the Template Method design pattern can be quite handy everywhere where a generic algorithm shall be defined once, but the implementations of the particular steps of this algorithm may vary.

Bonus questions:

- "**What is the difference between the Template Method design pattern and the Strategy design pattern?**"
  Both patterns allow specifying what concrete algorithm or a piece of the algorithm will be used. The main difference is that with the Template Method, it is selected at compile-time, as this pattern uses the inheritance. With the Strategy pattern, the decision is made at runtime, as this pattern uses composition.

41. What is the Decorator design pattern?

Brief summary: Decorator is a design pattern that dynamically adds extra functionality to an existing object, without affecting the behavior of other objects from the same class.

Decorator is a design pattern that dynamically adds extra functionality to an existing object, without affecting the behavior of other objects from the same class.

Let's start with something simple. We have a class that reads information about people from some data source:

```
class PeopleDataReader : IPeopleDataReader
{
    4 references
    public IEnumerable<Person> Read()
    {
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
    }
}
```

This class is nice, simple, and focused. *For now.*

At some point, we are asked to add an optional feature of logging how many elements have been read. Let's add this feature to the class:

```
class PeopleDataReader: IPeopleDataReader
{
    bool _shallLog;
    private readonly ILogger _log;

    3 references
    public PeopleDataReader(bool shallLog, ILogger log)
    {
        _shallLog = shallLog;
        _log = log;
    }

    6 references
    public IEnumerable<Person> Read()
    {

        var data = new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
        if (_shallLog)
        {
            _log.Log($"Read {data.Count()} elements");
        }
        return data;
    }
}
```

Ouch. It was such a pretty class, and now it grew large and ugly. Well, never mind… at least it does what it's supposed to.

Soon after that change, we are asked to add one more optional feature: to be able to limit data to some given count of People. Let's try to add this:

```
class PeopleDataReader: IPeopleDataReader
{
    bool _shallLog;
    bool _shallLimitCount;
    int _countLimit;
    private readonly ILogger _log;

    3 references
    public PeopleDataReader(
        bool shallLog,
        bool shallLimitCount,
        ILogger log,
        int countLimit = 0)
    {
        _shallLog = shallLog;
        _log = log;
        _shallLimitCount = shallLimitCount;
        _countLimit = countLimit;
    }

    6 references
    public IEnumerable<Person> Read()
    {
        var data = new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
        if (_shallLog)
        {
            _log.Log($"Read {data.Count()} elements");
        }

        return _shallLimitCount ?
            data.Take(_countLimit) :
            data;
    }
}
```

This code is terrible. This class has big chunks of logic which will or will not be executed depending on the flags. Its logic, so simple before, is now messy and complex. It takes more parameters than it may need (we don't need a logger if logging is not enabled, and we don't need **countLimit** if limiting is not enabled). It will be a nightmare to test it.

As more and more extra features are required to be added to this class, it will keep growing, becoming an unmanageable mess that no one wants to work with.

It's time to introduce the **Decorator** design pattern. This pattern allows adding some behavior to an object dynamically, without touching its code. If you know the **Open-Closed Principle** from SOLID, you know this is a good thing. It also allows us to keep the **Single Responsibility Principle** happy.

First, let's revert this class to how it was before changes:

```
class PeopleDataReader : IPeopleDataReader
{
    6 references
    public IEnumerable<Person> Read()
    {
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
            new Person("Michael", 1980, "Canada"),
            new Person("Anne", 1974, "New Zealand"),
        };
    }
}
```

Beautiful in its simplicity. Now, let's add LoggingDecorator class. It will "decorate" the PeopleDataReader with the ability of logging.

Implementing the Decorator design pattern boils down to two steps:

- making the Decorator **implement the same interface as the decorated object**
- making the Decorator **own an object implementing this interface**. It will be the decorated class itself or another Decorator, which allows us to compose many Decorators together

Let’s see how it looks in code:

```
class LoggingDecorator : IPeopleDataReader
{
    private readonly IPeopleDataReader _decoratedReader;
    private readonly ILogger _log;

    2 references
    public LoggingDecorator(
        ILogger log,
        IPeopleDataReader decoratedReader)
    {
        _decoratedReader = decoratedReader;
        _log = log;
    }

    6 references
    public IEnumerable<Person> Read()
    {
        var data = _decoratedReader.Read();
        _log.Log($"Read {data.Count()} elements");
        return data;
    }
}
```

As you can see, the Decorator owns an object that it wants to decorate. It implements the same interface. In the Read method that comes from the interface, it calls whatever implementation is provided, but it adds a little something from itself - in this case, it writes to a log.

Remember that the **_decoratedReader** doesn’t need to be the plain PeopleDataReader object - it **can be anything implementing the IPeopleDataReader interface, including another Decorator**. Of course, at some point one of the Decorators in this structure must own the basic decorated object of PeopleDataReader type.

Let’s now add the Decorator that will be limiting the count of returned Person objects:

```
class CountLimitingDecorator : IPeopleDataReader
{
    private readonly IPeopleDataReader _decoratedReader;
    private readonly int _countLimit;

    2 references
    public CountLimitingDecorator(
        int countLimit,
        IPeopleDataReader decoratedReader)
    {
        _decoratedReader = decoratedReader;
        _countLimit = countLimit;
    }

    6 references
    public IEnumerable<Person> Read()
    {
        Console.WriteLine(
            $"LIMITING the result to {_countLimit} elements");
        return _decoratedReader.Read().Take(_countLimit);
    }
}
```

Great. We can now compose those Decorators to our liking. Let's create an object that reads people data, logs the original count, and then limits it:

```
IPeopleDataReader loggingCountLimitingPeopleDataReader =
    new CountLimitingDecorator(3,
        new LoggingDecorator(new Logger(),
            new PeopleDataReader()));

var people1 = loggingCountLimitingPeopleDataReader.Read();
foreach(var person in people1)
{
    Console.WriteLine(person);
}
```

Here the real magic happens. Each Decorator takes any object implementing the IPeopleDataReader interface as a parameter but also implements this interface itself. It means, we can pass a Decorator as a parameter to other Decorator, stacking their functionalities. That's why the final object will be able to both log and limit the count of elements:

```
Both logging and count limiting
LIMITING the result to 3 elements
[LOG] Read 5 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
```

Please be aware that the order of the Decorators creation matters. If we change this code to this…

```
IPeopleDataReader loggingCountLimitingPeopleDataReader =
    new LoggingDecorator(new Logger(),
        new CountLimitingDecorator(3,
            new PeopleDataReader()));
```

…the result will be different because the limiting Decorator's Read method will be executed before the logging Decorator's Read method. From the point of view of the LoggingDecorator the count of data will be 3, not 5.

```
Both logging and count limiting
LIMITING the result to 3 elements
[LOG] Read 3 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
```

The features of logging and limiting are optional, but with the Decorator pattern, it's easy to choose what we need. Let's create a PeopleDataReader that only logs some information, but does not limit the count:

```
IPeopleDataReader loggingPeopleDataReader =
    new LoggingDecorator(new Logger(),
        new PeopleDataReader());
```

In the result we will see all 5 elements:

```
Only logging
[LOG] Read 5 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country = F
rance }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Jap
an }
Person { FirstName = Selene, YearOfBirth = 1944, Country = G
reat Britain }
Person { FirstName = Michael, YearOfBirth = 1980, Country =
Canada }
Person { FirstName = Anne, YearOfBirth = 1974, Country = New
 Zealand }
```

And now, let's create an object that does not log, but it does limit the data:

```
IPeopleDataReader countLimitingPeopleDataReader =
    new CountLimitingDecorator(3,
        new PeopleDataReader());
```

```
Only count limiting
LIMITING the result to 3 elements
Person { FirstName = Martin, YearOfBirth = 1972, Country =
France }
Person { FirstName = Aiko, YearOfBirth = 1995, Country = Ja
pan }
Person { FirstName = Selene, YearOfBirth = 1944, Country =
Great Britain }
```

As you can see, there is no "[LOG]" string in this result.

All right. As you can see the Decorator pattern allows us to easily add functionality to objects, without touching the original classes, so it's very much in line with the Open-Closed Principle. It allows us to keep classes simple. It also helps us to be in line with the Single Responsibility Principle, as each class now has a very focused responsibility. They would be easy to test, maintain, and generally pleasant to works with.

Bonus questions:

- "**What are the benefits of using the Decorator design pattern?**"
  The Decorator pattern allows us to easily add functionality to objects, without touching the original classes, so it's very much in line with the Open-Closed Principle. *It allows us to keep classes simple. It makes it easy to stack functionalities together, building complex objects from simple classes. It also helps us to be in line with the Single Responsibility Principle, as each class now has a very focused responsibility. They would be easy to test, maintain, and generally pleasant to works with.*

42. What is the Observer design pattern?

Brief summary: The Observer design pattern allows objects to notify other objects about changes in their state.

Observer design pattern allows objects to notify other objects about changes in their state.

Let's consider the following example. We have some class that is able to read the current Bitcoin price. In a real-life application it would read it from some public API, but for the example's sake let's make it return a random number from 0 to 50000 (looking at cryptocurrencies prices fluctuations, I would say it's not that far away from the truth).

```
public class BitcoinPriceReader
{
    private decimal _currentBitcoinPrice;

    3 references
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
    }
}
```

Now, let's say we want to create a couple of mechanisms that will notify the application's users if the price has grown over a certain threshold. Let's say we want to be able to send users emails and/or push notifications.

```
class EmailPriceChangeNotifier
{
    private readonly decimal _notificationThreshold;

    1 reference
    public EmailPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    0 references
    public void Update(decimal currentBitcoinPrice)
    {
        if (currentBitcoinPrice > _notificationThreshold)
        {
            Console.WriteLine($"Sending an email saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {currentBitcoinPrice}");
        }
    }
}
```

The class for sending push notifications would be almost the same, except that the message would be different. Please notice that this is a simplification, and in a real project, those classes would actually send emails or push notifications. We could also implement more classes to perform other types of notifications.

All right, so here is the big picture: we have the **BitcoinPriceReader** that reads the price, and two classes that wait to be notified about the price change - **EmailPriceChangeNotifier** and **PushPriceChangeNotifier**. When the price is read from the BitcoinPriceReader, we want it to execute the **Update** method from both the classes that wait for the information about the new price:

```
public class BitcoinPriceReader
{
    private decimal _currentBitcoinPrice;

    private readonly EmailPriceChangeNotifier _emailPriceChangeNotifier;
    private readonly PushPriceChangeNotifier _pushPriceChangeNotifier;

    1 reference
    public BitcoinPriceReader(
        EmailPriceChangeNotifier emailPriceChangeNotifier,
        PushPriceChangeNotifier pushPriceChangeNotifier)
    {
        _emailPriceChangeNotifier = emailPriceChangeNotifier;
        _pushPriceChangeNotifier = pushPriceChangeNotifier;
    }

    3 references
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
        _emailPriceChangeNotifier.Update(_currentBitcoinPrice);
        _pushPriceChangeNotifier.Update(_currentBitcoinPrice);
    }
}
```

Well.. this is awkward, at least. First of all, it **tightly couples** the BitcoinPriceReader with the other two classes. Secondly, this way we will only notify a single EmailPriceChangeNotifier object and a single PushPriceChangeNotifier object. What if we wanted to notify a whole group of them? Lastly, what if some of those objects will no longer be interested in listening about the price changes (for example the user of the application decides to sell all his or her crypto and move to live in the Bahamas?). We won't have any control over what objects we notify.

It's time to introduce the **Observer design pattern**. Let's do it step by step.

First of all, we want to decouple the BitcoinPriceReader (the **Observable**) from the EmailPriceChangeNotifier and PushPriceChangeNotifier (the **Observers**). We will need to define interfaces over which they can communicate. The first question we need to ask is "what **data** will be sent from the Observable to the Observers?". In our case, it will be the current Bitcoin price, so a decimal, but let's make the interfaces generic, so they can work with any payload. First, let's define the IObserver interface, which will be implemented by EmailPriceChangeNotifier and PushPriceChangeNotifier. This interface will contain a single Update method, which will be called by the Observable to send the data to the Observers:

```
3 references
public interface IObserver<TData>
{
    2 references
    void Update(TData data);
}
```

Let's use this interface before we move on to IObservable:

```
public class EmailPriceChangeNotifier : IObserver<decimal>
{
    private readonly decimal _notificationThreshold;

    1 reference
    public EmailPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    2 references
    public void Update(decimal currentBitcoinPrice)
    {
        if (currentBitcoinPrice > _notificationThreshold)
        {
            Console.WriteLine($"Sending an email saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {currentBitcoinPrice}");
        }
    }
}
```

In this case, the method was already implemented, so not much to do here. In general, the Update method is the one that receives the notification from the Observable and decides what to do about it. I also added the interface implementation to the PushPriceChangeNotifier.

Let's now define the IObservable interface.

```
public interface IObservable<TData>
{
    0 references
    void AttachObserver(IObserver<TData> observer);
    0 references
    void DetachObserver(IObserver<TData> observer);
    0 references
    void NotifyObservers();
}
```

The first two methods are used to attach (or “subscribe”) the observer to the observable. This way we will have control over who is notified. We can detach (or “unsubscribe”) the observers at any time if they are no longer interested in receiving the notifications from the Observable.

The last method will be executed to send the notification to all subscribed observers.

Let’s implement this interface in the BitcoinPriceReader class. First, we need to define a collection of Observers:

```
public class BitcoinPriceReader : IObservable<decimal>
{
    private decimal _currentBitcoinPrice;
    private List<IObserver<decimal>> _observers =
        new List<IObserver<decimal>>();

    3 references
    public void AttachObserver(IObserver<decimal> observer)
    {
        _observers.Add(observer);
    }

    2 references
    public void DetachObserver(IObserver<decimal> observer)
    {
        _observers.Remove(observer);
    }
```

The NotifyObservers method will simply iterate the List of Observers and execute the Update method on them with the _currentBitcoinPrice:

```
public void NotifyObservers()
{
    foreach (var observer in _observers)
    {
        observer.Update(_currentBitcoinPrice);
    }
}
```

The only thing left to do is to call the NotifyObservers method after the latest Bitcoin price has been read:

```
2 references
public void ReadCurrentPrice()
{
    _currentBitcoinPrice = new Random().Next(0, 50000);

    NotifyObservers();
}
```

As you can see the NotifyObservers method could be private, but I'll leave it public as this is the most typical implementation of the Observer design pattern.

All right, let's put it all together. First, let's create the Observers and attach them to the Observable. Let's say the email should be sent if the price exceeds 25000, and push notification - when it exceeds 40000.

```
var bitcoinPriceReader = new BitcoinPriceReader();

var emailPriceChangeNotifier = new EmailPriceChangeNotifier(25000);
bitcoinPriceReader.AttachObserver(emailPriceChangeNotifier);

var pushPriceChangeNotifier = new PushPriceChangeNotifier(40000);
bitcoinPriceReader.AttachObserver(pushPriceChangeNotifier);
```

Now, let's execute the ReadCurrentPrice method couple of times:

```
bitcoinPriceReader.ReadCurrentPrice();
bitcoinPriceReader.ReadCurrentPrice();
```

And here is the result:

```
Sending an email saying that the Bitcoin price exceeded 25000 and
is now 44326

Sending a push notification saying that the Bitcoin price exceeded
 40000 and is now 44326
```

It seems like one of the calls triggered both email and push notifications, and the other did not trigger any of them (so the price must have been below 25000).

Now, let's detach the PushPriceChangeNotifier:

```
Console.WriteLine("Push notifications OFF");
bitcoinPriceReader.DetachObserver(pushPriceChangeNotifier);
```

And call the ReadCurrentPrice method again:

```
Sending an email saying that the Bitcoin price exceeded 25000
 and is now 35705

Push notifications OFF
Sending an email saying that the Bitcoin price exceeded 25000
 and is now 47023
```

As you can see, after the push notifications have been unsubscribed, they are not sent even if the price exceeded 40000.

Remember that this code bases on random numbers, so when you execute it, you will have different results. Run it a couple of times and see what happens!

All right. We implemented the basic Observer design patterns.

Please note that there is an existing Microsoft's implementation of this pattern, but it's a bit more complex. I wanted to show you custom implementation so you

see exactly what is going on. If you are curious about Microsoft's implementation, make sure to read this article:
https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern

We will revisit the topic of the Observer design pattern in the next lecture, where we will talk about events, as they have very much in common.

Bonus questions:

- "**In the Observer design pattern, what is the Observable and what is the Observer?**"
  The Observable is the object that's being observed by Observers. The Observable notifies the Observers about the change in its state.

43. What are events?

Brief summary: Events are the .NET way of implementing the Observer design pattern. They are used to send a notification from an object to all objects subscribed.

An event is a message sent by an object to signal the occurrence of an action. Events are the .NET implementation of the Observer design pattern.

Let's implement the same logic we had in the lecture about the Observer design pattern. We want the BitcoinPriceReader object to notify other objects about the reading of the Bitcoin price.

This time, we will do so using events. We will start by defining a delegate that will represent the function or functions that will be executed once the price reading event has been raised:

```
public delegate void PriceRead(decimal price);
```

The BitcoinPriceReader will own an **event** of this delegate's type:

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;
    public event PriceRead? PriceRead;
```

Let's take a moment to stop and think about what happened here. We declared an **event belonging** to the BitcoinPriceReader class. An event is always of a delegate type. Remember, a delegate is a type whose instances hold a reference to a method with a particular parameter list and return type.

So what is the difference between an event, and a regular field of delegate type? In other words, what will be the difference between those two?

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;

    public event PriceRead? PriceRead;
    public PriceRead? PriceReadNotAnEvent;
```

I will try to attach some method to both of them.

```
void SomeMethod(decimal price)
{
    Console.WriteLine($"Price is {price}");
}
```

```
var bitcoinPriceReader = new BitcoinPriceReader();
bitcoinPriceReader.PriceRead += SomeMethod;
bitcoinPriceReader.PriceReadNotAnEvent += SomeMethod;
```

So far, an event and a field of delegate type act exactly the same. But here is the difference:

```
bitcoinPriceReader.PriceRead(100);
bitcoinPriceReader.PriceReadNotAnEvent(100);
```

As you can see, I can't invoke the event from outside the class it belongs to. I can invoke the non-event delegate without a problem.

This is a **critical difference**. Only the class that owns an event can raise it. Events are used to send notifications about some action, so imagine what would happen if any class could raise them: any code could raise the PriceRead event with any price they want, triggering invalid notifications all around the system. The event delegate **must be public** so the subscribers can subscribe to it, but it must only be invokable from within the class that owns it. And this is what the "event" keyword enforces.

All right. Let's move on with the implementation. We defined an event in the BitcoinPriceReader class.

```
public event PriceRead? PriceRead;
```

As you can see I declared it as nullable because before any subscriber subscribes to it, it will be null.

All right. When the Bitcoin price is read, we want to "raise" the event, so simply invoke all methods stored in the event delegate. That means, all subscribers will be notified that the event occurred:

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;

    public event PriceRead? PriceRead;
    public PriceRead? PriceReadNonEvent;

    1 reference
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
        OnPriceRead(_currentBitcoinPrice);
    }

    1 reference
    protected virtual void OnPriceRead(decimal price)
    {
        PriceRead?.Invoke(price);
    }
}
```

As you can see I execute the event delegate by the **Invoke** method. This is because the "normal" execution does not allow using the null-conditional operator, and as we said, the PriceRead event might be null:

```
1 reference
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?(price);
}
```

As you can see, the above code doesn't compile. I don't want to simply call "PriceRead(price)" because I would be at risk of causing the NullReferenceException.

All right. Let's summarize what happened in this class. We declared an event, which is of a delegate type. When the Bitcoin price is read, we want to raise the event so all subscribers are notified.

Let's move on to the subscribers. Each subscriber must contain a method that is compatible with the event delegate (in our case, a void method accepting a decimal).

```
public delegate void PriceRead(decimal price);
```

Let's see such a method in the PushPriceChangeNotifier class:

```
public class PushPriceChangeNotifier
{
    private readonly decimal _notificationThreshold;

    1 reference
    public PushPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    0 references
    public void Update(decimal price)
    {
        if (price > _notificationThreshold)
        {
            Console.WriteLine($"Sending a push notification saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {price}\n");
        }
    }
}
```

A very similar method exists in EmailPriceChangeNotifier class. We can now subscribe those two classes to be notified when the event is raised:

```
var emailPriceChangeNotifier = new EmailPriceChangeNotifier(25000);
var pushPriceChangeNotifier = new PushPriceChangeNotifier(40000);

bitcoinPriceReader.PriceRead += emailPriceChangeNotifier.Update;
bitcoinPriceReader.PriceRead += pushPriceChangeNotifier.Update;
```

And that's it! All that's left is to call the ReadCurrentPrice method:

```
bitcoinPriceReader.ReadCurrentPrice();
```

And the result is:

```
Sending an email saying that the Bitcoin price exceeded 25000
and is now 49741

Sending a push notification saying that the Bitcoin price exce
eded 40000 and is now 49741
```

It seems like everything is working. Let's see again what happened. When the ReadCurrentPrice method is executed, it raises the PriceRead event.

```
public void ReadCurrentPrice()
{
    _currentBitcoinPrice = new Random().Next(0, 50000);
    OnPriceRead(_currentBitcoinPrice);
}

1 reference
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?.Invoke(price);
}
```

The PriceRead event is invoked, and since the Update methods from EmailPriceChangeNotifier and PushPriceChangeNotifier are attached to the event delegate, they get executed.

All right. This works as expected. There is one improvement we can make, though. Instead of using our own PriceRead delegate for the event, we can use the EventHandler delegate that is predefined in C#:

```
public event EventHandler PriceRead;
```

This is the signature of this delegate:

```
public delegate void EventHandler(object? sender, EventArgs e);
```

As you can see it carries the information about the object that raised the event (sender) and event arguments. EventArgs is a base class for any event arguments we want. In our case, the argument of an event is the Bitcoin price that has been read. Let's create our own type derived from EventArgs:

```
public class PriceReadEventArgs : EventArgs
{
    1 reference
    public decimal Price { get; }

    0 references
    public PriceReadEventArgs(decimal price)
    {
        Price = price;
    }
}
```

To make sure this type of event argument will be used, we must use the generic EventHandler:

```
public event EventHandler<PriceReadEventArgs> PriceRead;
```

We must now change the code that raises the event, to match the EventHandler<PriceReadEventArgs> delegate type:

```
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?.Invoke(this, new PriceReadEventArgs(price));
}
```

As you can see, as the first argument we pass “this” so the sender of the event. The second argument is the PriceReadEventArgs object holding the price.

The last thing left to do is to change the Update method in EmailPriceChangeNotifier and PushPriceChangeNotifier classes:

```
public void Update(object sender, PriceReadEventArgs eventArgs)
{
    if (eventArgs.Price > _notificationThreshold)
    {
        Console.WriteLine($"Sending an email saying that " +
            $"the Bitcoin price exceeded {_notificationThreshold} " +
            $"and is now {eventArgs.Price}\n");
    }
}
```

All right. Now everything works as expected.

You may think that this is more complicated than the code that we had before, and this is true. Nevertheless, I wanted to show you this EventHandler type, as it is used in many Microsoft’s frameworks based on events. For example, in Windows Forms or Windows Presentation Foundation desktop applications. All user actions like clicking a button or closing a window trigger events and those events are defined with the EventHandler delegate. In those use cases, the sender argument is used by the subscribers, which we did not need to do in our code.

One last thing before we wrap up. This is generally a good practice to unsubscribe from the event handler before the object getting notified about the event is discarded:

```
bitcoinPriceReader.PriceRead -= emailPriceChangeNotifier.Update;
bitcoinPriceReader.PriceRead -= pushPriceChangeNotifier.Update;
```

I don’t want to get into details about why is it important (as this lecture is quite long already), so let me just give you a very quick overview. When a subscriber subscribes to be notified about a state change in some object, a **hidden reference** is created between them. It is needed because the EventHandler holds a reference to a method stored in the object that will be notified about the event, so a reference to this object is necessary. It may happen that we think an object is no longer in use, while the Garbage Collector still sees a reference to it, and will not remove it from memory. For example, consider a desktop application with some MainWindow, and a ChildWindow that opens when we click some button. The ChildWindow is subscribed to an event of the MainWindow, so the reference from the MainWindow to ChildWindow exists. When we close the ChildWindow it should be removed from memory, but the Garbage Collector will see this reference and will decide this object should not be removed. As the application’s user keeps opening and closing ChildWindows, more and more memory is being used, but none of it is being freed. This situation is called a “memory leak”. Over time, it may slow the application down, or even cause it to crash due to OutOfMemoryException. Unsubscribing from events when it’s possible (for example, when the ChildWindow is being closed) is a way of preventing that.

Let’s summarize. Events are .NET way of implementing the Observer design pattern. They are used to send a notification from an object to all objects subscribed. The pattern of using events is as follows:

- the class that will be sending notifications owns an event, which is a delegate
- objects that want to be notified about an event can attach their own methods to this delegate
- when the observable class raises the event, it does so by invoking the methods stored in the delegate. This way, all methods from the observers will be executed

Bonus questions:

- "**What is the difference between an event and a field of the delegate type?**"
  A public field of a delegate type can be invoked from anywhere in the code. Events can only be invoked from the class they belong to.

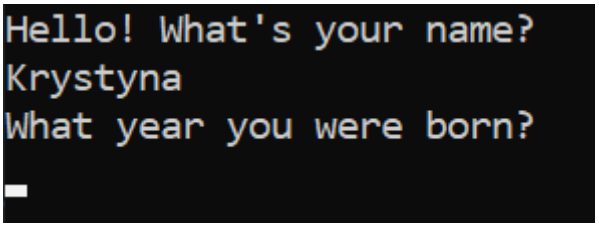
- "**Why is it a good practice to unsubscribe from events when a subscribed object is no longer needed?**"
  Because as long as it is subscribed, a hidden reference between the observable and the observer exists, and it will prevent the Garbage Collector from removing the observer object from memory.

44. What is Inversion of Control?

Brief summary: Inversion of Control is the design approach according to which the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it.

Inversion of Control is the design approach according to which the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it.

Let's consider two simple examples. First is a simple console application interacting with the user.



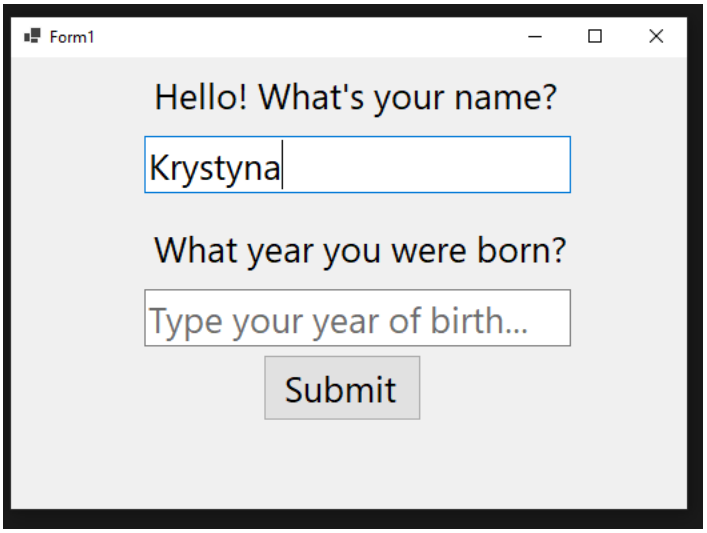
In this program, the code is in control - it decides when the user answers the questions shown in the console. Here is the implementation of this program:

```
Console.WriteLine("Hello! What's your name?");
var name = Console.ReadLine();
int yearOfBirth;
do
{
    Console.WriteLine("What year you were born?");
}
while (!int.TryParse(Console.ReadLine(), out yearOfBirth));

Console.WriteLine(
    $"Hello {name}, you are " +
    $"{DateTime.Now.Year - yearOfBirth} years old!");

Console.ReadKey();
```

Now, let's see a different approach:



In this application, the code doesn't control when exactly the user will fill in the form, and when the final message will be printed. The action of the user (clicking on the Submit button) will trigger an event that will handle printing the output:

```
private void SubmitButton_Click(object sender, EventArgs e)
{
    SetBackColor(NameTextBox);
    SetBackColor(YearOfBirthTextBox);

    if(!string.IsNullOrEmpty(NameTextBox.Text) &&
       !string.IsNullOrEmpty(YearOfBirthTextBox.Text))
    {
        int age = DateTime.Now.Year - int.Parse(
            YearOfBirthTextBox.Text);
        ResultTextBox.Text = $"Hello {NameTextBox.Text}! You are
            $"{age} years old!";
    }
}

2 references
private void SetBackColor(TextBox textBox)
{
    if (string.IsNullOrEmpty(textBox.Text))
    {
        textBox.BackColor = Color.IndianRed;
    }
```

The control flow of the program is **inverted** compared to the "traditional" flow, where the code decides when exactly some action happens. Here, the framework (in this case, Windows Forms) is in charge, and it executes particular pieces of code based on the user actions that trigger events.

Inversion of Control is sometimes referred to as "the Hollywood Principle" which says "don't call us, we will call you". In this case, we don't call a method. The framework calls us, letting us know via an event that some code needs to be executed.

According to Martin Fowler (author of the great book "Refactoring" and in general authority in topics of software development, design, etc.) the Inversion of Control is what makes the **difference between a framework and library**:

"*A **library** is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.*

*A **framework** embodies some abstract design, with more behavior built in. In order to use it, you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.*"

There are many ways in which the control can be inverted. In the example we've seen, this was implemented by using events. Events were triggered by the user's actions on the GUI, thus executing some particular methods in code.

Speaking more generally, the Inversion of Control happens whenever some kind of a callback is defined. A callback is an executable code (a method in C#) that gets passed as an argument to some other code. Let's consider this simple example:

```
ReadLineByLine(input, () => Console.WriteLine("Finished!"));
Console.ReadKey();

void ReadLineByLine(string input, Action onFinished)
{
    string[] lines = input.Split(Environment.NewLine);
    for (int i = 0; i < lines.Length; i++)
    {
        string line = lines[i];
        Console.WriteLine($"[Line number {i}] {line}");
    }
    onFinished();
}
```

The ReadLineByLine method uses a callback - an Action passed as a parameter. Once the entire input has been read, the callback will be executed. In real-life projects it often happens that after some data is read (from a database, API, or anything else that takes time to execute) a callback is invoked, informing some other piece of the code that it can start its work, as the data it requires is ready to be used.

Another example of Inversion of Control could be the **Template Method**. In the lecture about it, we mentioned the example of SetUp and TearDown methods from NUnit framework. It's another case when the framework calls the methods we defined. The template is defined in NUnit itself, where it is decided that first the SetUp must be called, then the actual test, and then the TearDown. But the actual implementation of those steps is defined by the programmer.

Dependency Injection is another example of Inversion of Control. The code that some class needs to execute is injected from the outside. We don't have control over what method exactly will be called. This decision is made for us by someone who provides the concrete type as the constructor parameter. We only declare that we need some dependency.

Using an interface is similar to having a callback. After all, an interface is like a bundle of methods. It would actually be possible to have Dependency Injection without interfaces, but by simply providing a class with Funcs that will be executed, similarly as the Action that we saw in an example above.

```
class PersonalDataFormatterWithFunc
{
    private readonly Func<IEnumerable<Person>> _readPeople;

    0 references
    public PersonalDataFormatterWithFunc(
        Func<IEnumerable<Person>> readPeople)
    {
        _readPeople = readPeople;
    }

    0 references
    public string Format()
    {
        var people = _readPeople();

        return string.Join("\n",
          people.Select(p => $"{p.Name} born in" +
          $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

Let's summarize. Inversion of Control is the design approach according to which the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it.

Bonus questions:

- "**What is a callback?**"
  A callback is an executable code (a method in C#) that gets passed as an argument to some other code.

- "**What is the difference between a framework and a library?**"
  *According to Martin Fowler: "A **library** is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client. A **framework** embodies some abstract design, with more behavior built in. In order to use it, you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points." So in short, the framework relies on Inversion of Control, but the library does not.*

45. What is the “composition over inheritance” principle?

Brief summary: “Composition over inheritance” is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within other objects, than inheriting one from another.

“Composition over inheritance” is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within other objects, than inheriting one from another.
Let’s see a practical example.

```
class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    1 reference
    private IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

This class reads people's data from a database and formats it as a single string. It's obviously breaking the Single Responsibility Principle, but let's by now not focus on that.

One day the business requirements change, and we are told that sometimes the people's information will be read from the database, but sometimes from an Excel file. We want to be able to make this decision at runtime.

We can solve it in two ways - by either using composition, and injecting an object implementing some IPeopleDataReader interface with this class's constructor, or we can use inheritance.

First, let's solve this with inheritance. I will make the PersonalDataFormatter class abstract:

```
abstract class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    3 references
    protected abstract IEnumerable<Person> ReadPeople();
}
```

The ReadPeople method is also abstract, so it will have to be overridden in inheritors:

```
class DatabaseSourcedPersonalDataFormatter : PersonalDataFormatter
{
    2 references
    protected override IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

```
class ExcelSourcedPersonalDataFormatter : PersonalDataFormatter
{
    2 references
    protected override IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from an Excel file");
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
        };
    }
}
```

Great. We achieved what we wanted - we can now format the personal data sourced from both databases and Excel files. We don't have any code duplications, and we can decide the type at runtime, using some Factory.

```
var factory = new PersonalDataFormatterFactory();

var fromExcel = factory.Create(Source.Excel);
Console.WriteLine(fromExcel.Format());

Console.WriteLine();

var fromDatabase = factory.Create(Source.Database);
Console.WriteLine(fromDatabase.Format());
```

```
class PersonalDataFormatterFactory
{
    2 references
    public PersonalDataFormatter Create(Source source)
    {
        switch (source)
        {
            case Source.Database:
                return new DatabaseSourcedPersonalDataFormatter();
            case Source.Excel:
                return new ExcelSourcedPersonalDataFormatter();
            default:
                throw new ArgumentException("Invalid source");
        }
    }
}
```

Everything looks good, doesn't it?

Well… not so fast. Let me tell you why using inheritance, in this case, wasn't our brightest idea.

- The PersonalDataFormatter class is tightly coupled with its inheritors now. Any change in the base class will affect the child classes. We can't really use any of those types without the others provided, so if I wanted to use ReadPeople method anywhere else, I would not be able without engaging this entire hierarchy of classes. You can learn more about coupling in the "What is coupling?" lecture.
- The relation between those particular classes is rigid - it is defined at compile time. If I had some other mechanism that can read people's information from some source, I wouldn't be able to use it here without creating another derived type.
- Also, we have all the limitations of inheritance here, especially the fact that we can only inherit from a single base class.
- This example is simple, but if we needed some other changes that would make those classes different, we would have the inheritance hierarchy growing really fast. For example, if the way of formatting the final string would also need to be configurable, we would need to create even more classes, like:
  DatabaseSourcedPersonalData**Short**Formatter,
  DatabaseSourcedPersonalData**Full**Formatter,
  ExcelSourcedPersonalData**Short**Formatter,
  ExcelSourcedPersonalData**Full**Formatter
  Such hierarchy would soon become unmanageable.

- If we wanted to create unit tests for those classes, it would be tricky, and there are actually two approaches for testing abstract classes and their inheritors, both with their own disadvantages:
    - We can test both inheritors, but in both of them, we will also test the common part belonging to the base class. Our tests will be partially duplicated.
    - We can test inheritors ignoring the logic belonging to the base type as much as possible. Then, we can test the base abstract class logic by creating for the testing purposes a special, dedicated concrete type derived from it. In the tests of this class, we would focus on testing the base class logic. This is even worse than the first point - if you need to create special inheritors classes for testing purposes only, it means you messed your design up badly.
- It is often the case that we inherit more than we would actually want. The base class is exposing the implementation details to inheritors.
- There is one more reason for avoiding inheritance that is not really related to this example, but I want to mention it anyway: if we use inheritance in a hierarchy of objects that we intend to store in a database using some Object-Relational Mapping tools like Entity Framework, it may be a challenge to store those objects properly. Databases don't easily "understand" inheritance, so mapping the C#'s hierarchy of inheritance into a flat structure of tables is tricky, and often leads to overcomplicating the model in the database.

So how to solve all of it?

Well, in this case, we should definitely apply the "composition over inheritance" principle. Let's refactor this code:

First of all, I will introduce an interface:

```
interface IPeopleDataReader
{
    3 references
    IEnumerable<Person> ReadPeople();
}
```

I will have two classes implementing it:

```
class DatabasePeopleDataReader : IPeopleDataReader
{
    2 references
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

```
class ExcelPeopleDataReader : IPeopleDataReader
{
    2 references
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from an Excel file");
        return new List<Person>
        {
            new Person("Martin", 1972, "France"),
            new Person("Aiko", 1995, "Japan"),
            new Person("Selene", 1944, "Great Britain"),
        };
    }
}
```

Instead of using inheritance, I will compose The PersonalDataFormatter with a type implementing the IPeopleDataReader interface:

```
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

    2 references
    public PersonalDataFormatter(IPeopleDataReader peopleDataReader)
    {
        _peopleDataReader = peopleDataReader;
    }

    2 references
    public string Format()
    {
        var people = _peopleDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

Finally, I will adjust the Factory:

```
class PersonalDataFormatterFactory
{
    2 references
    public PersonalDataFormatter Create(Source source)
    {
        switch (source)
        {
            case Source.Database:
                return new PersonalDataFormatter(
                    new DatabasePeopleDataReader());
            case Source.Excel:
                return new PersonalDataFormatter(
                    new ExcelPeopleDataReader());
            default:
                throw new ArgumentException("Invalid source");
        }
    }
}
```

Great. Everything works as before, but no problems mentioned above occur now:

- The classes are **loosely coupled**. They live in complete separation, and they only communicate by an interface
- The relationship between classes is **not rigid** anymore. It is defined at runtime when we actually inject a concrete PeopleDataReader to

PersonalDataFormatter object with the constructor. Before, the relation was defined at compile time.

- If we needed to add more changes, the inheritance hierarchy wouldn't grow. We would only add a new interface and classes implementing it, for example, an IPersonFormatter implemented by PersonShortFormatter and PersonFullFormatter.
- Testing would be simple. We would test each class in separation, and no tests would be duplicated.
- No class exposes any implementation details to another class.

All right. I hope you see now that in this case "composition over inheritance" was a rule worth following. I would say it is in the majority of cases, and when in doubt, you should follow the composition design rather than inheritance. To be honest, at my everyday work I use inheritance extremely rarely.

One more thing before we move on. If you know the Bridge design pattern, this all may sound very familiar to you. This is because the Bridge pattern is simply a way of implementing the composition over inheritance principle. You can read more about the **Junior e-book**.

All right. We said that having composition instead of inheritance has a lot of benefits. But it doesn't mean that inheritance should be avoided at any cost. Let's take a look at the Person type:

```
public class Person
{
    2 references
    public string FirstName { get; }
    2 references
    public string LastName { get; }
    2 references
    public int YearOfBirth { get; }

    0 references
    public Person(string firstName, string lastName, int yearOfBirth)
    {
        FirstName = firstName;
        LastName = lastName;
        YearOfBirth = yearOfBirth;
    }
}
```

Now, let's say we want to introduce an Employee type to the project. An Employee is still a Person, and it should have FirstName, LastName, and YearOfBirth properties. Besides that, this type should have a "Position" property.

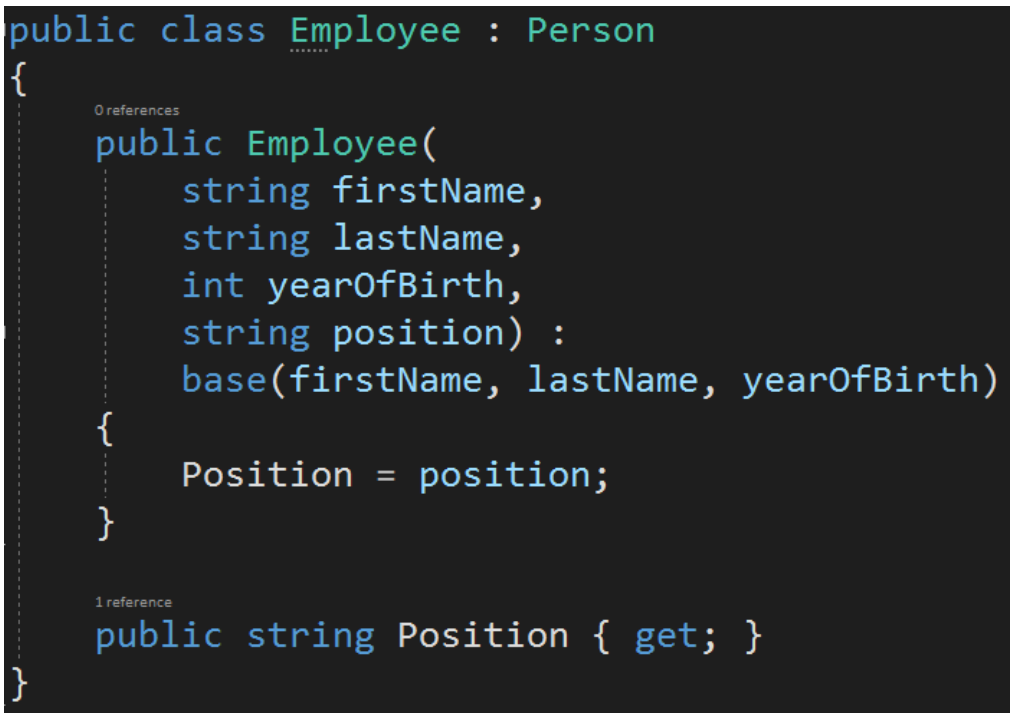
Let’s say we are so excited about using the “composition over inheritance” that we decide not to use inheritance ever again. And this is the code we create:

```
public class Employee
{
    private Person _person;
    0 references
    public string FirstName => _person.FirstName;
    0 references
    public string LastName => _person.LastName;
    0 references
    public int YearOfBirth => _person.YearOfBirth;
    1 reference
    public string Position { get; }

    0 references
    public Employee(Person person, string position)
    {
        _person = person;
        Position = position;
    }
}
```

Is this design good? Well, I wouldn’t say so. What looks a bit fishy are **the forwarding methods** - so the methods that only exist to call methods from some inner object. In our case, those methods are the FirstName, LastName, and YearOfBirth properties (remember that properties are like special kinds of methods).

Let’s see what this code would look like if we used inheritance:



Well, I think it looks much simpler. The forwarding methods are not there, as the properties we want to have in the Employee class are simply inherited from the Person class. All we need to define is the new Position property that actually makes the Employee different from a Person.

How to decide whether to use composition over inheritance? Well, first of all, you need to answer this question: when thinking about your types, can you say that one of them IS the other one? Do they have the same structure and similar functionality, with only some extended behavior in the derived type? If so, inheritance can be the right choice. Otherwise, you should rather opt for composition. When in doubt, go for composition and in the worst case, you will adjust your design if it turns out it's not working out.

You can read more about the details of making the "composition or inheritance" decision in this article:
https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Let's summarize. "Composition over inheritance" is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within different objects, than inheriting one from another.

Bonus questions:

- "**What is the problem with using composition only?**"
  If we decide not to use inheritance at all, we make it harder for ourselves to define types that are indeed in an "IS-A" relation - so when one type IS the other one. For example, a Dog IS an Animal, or an Employee IS a person. When implementing such hierarchy with the composition we create very similar types that wrap other types only adding a bit of new functionality, and they mostly contain forwarding methods.

- "**What are forwarding methods?**"
  They are methods that don't do anything else than calling almost identical methods from some other type. Forwarding methods indicate a very close relationship between types, which may mean that one type should be inherited from another.

46. What are mocks?

Brief summary: Mocks are objects that can be used to substitute real dependencies for testing purposes. For example, we don't want to use a real database connection in unit tests. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data. We can set up what will be the results of the methods called on mocks, as well as verify if a particular method has been called. Mocks are an essential part of unit testing, and it's nearly impossible to test a real-life application without them.

Mocks are objects that "pretend" to be other objects and are used mostly for testing purposes. For example, we don't want to use a real database connection in unit tests, and we will explain why in a minute. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data.

Let's say we want to unit test this class:

```
class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    1 reference
    private IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from real database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

Imagine the ReadPeople method connects to a real database, performing all necessary steps like opening the database connection, executing some SQL queries, etc.

The tests of the PersonalDataFormatter class could look like this:

```
    [TestFixture]
    0 references
    public class PersonalDataFormatterTests
    {
        private PersonalDataFormatter _cut = new PersonalDataFormatter();

        [Test]
        | 0 references
        public void ShallFormatPersonalDataCorrectly()
        {
            var result = _cut.Format();
            var expectedResult = @"John born in USA on 1982
Aja born in India on 1992
Tom born in Australia on 1954";
            Assert.AreEqual(expectedResult, result);
        }
    }
```

This may even work under some circumstances, but there are numerous **problems** with this approach:

- A test that connects to a database is **not a unit test**. A unit test should test only one piece of functionality. Here we test the class, the database connection, and the database itself.
- Also, unit tests should be fast, and connecting to a database **takes time**.
- This test only reads from the database, but what if other tests would also write to it? If some other test would add a new person to the database, this test would start to fail, as the result would contain one more line. As tests would run, the database **state would change** constantly, affecting the results. Because of that, we would be forced to reset the database to some desired state before each test, which would again take significant time.
- What if the database is **not set up** on the computer of another developer? This test may work for us, but it may not work for others.
- What if the database contains **millions of entries**? Then the expected value in this test would be an enormous string, which would obviously be problematic, especially if the test failed and in this huge string we would try to find the exact part that doesn't match the expected result.

To solve all those issues, we need a mechanism that will allow us to **mock** the database connection. Instead of using an object connecting to a real database, we will use a **fake** one, that will return a predefined set of data used for testing purposes only.

But first, we must refactor this code to use **Dependency Injection**, so we are not tightly coupled with the implementation that connects to a real database:

```
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

    1 reference
    public PersonalDataFormatter(
        IPeopleDataReader peopleDataReader)
    {
        _peopleDataReader = peopleDataReader;
    }

    1 reference | 1/1 passing
    public string Format()
    {
        var people = _peopleDataReader.ReadPeople();
        return string.Join(Environment.NewLine,
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

```
public interface IPeopleDataReader
{
    3 references | 1/1 passing
    IEnumerable<Person> ReadPeople();
}

0 references
public class DatabasePeopleDataReader : IPeopleDataReader
{
    3 references | 1/1 passing
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from real database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

Great. Now, in the production code, we can inject the implementation that connects to a real database:

```
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        var personalDataFormatter = new PersonalDataFormatter(
            new DatabasePeopleDataReader());

        Console.WriteLine(personalDataFormatter.Format());
    }
}
```

But for unit tests, we will use a mock. I will be using the **Moq** library for that, which is one of the most popular mocking libraries for C#. To create a mock of some interface, we can simply use the Mock<T> class:

```
[TestFixture]
0 references
public class PersonalDataFormatterTests
{
    private Mock<IPeopleDataReader> _peopleDataReaderMock;
    private PersonalDataFormatter _cut;

    [SetUp]
    0 references
    public void SetUp()
    {
        _peopleDataReaderMock = new Mock<IPeopleDataReader>();
        _cut = new PersonalDataFormatter(
            _peopleDataReaderMock.Object);
    }
```

As you can see I moved the creation of the _cut object to the SetUp method. This is because I want a brand-new mock for each test, which is a good practice since the mocks have their own state (they can track what methods had been called upon them, which is used for validating mock behavior. We will talk more about it later in the lecture).

Let's now use the mock in the test. I will set it up to return some predefined People objects when the ReadPeople method is called:

```
        [Test]
        0 references
        public void ShallFormatPersonalDataCorrectly()
        {
            _peopleDataReaderMock.Setup(mock => mock.ReadPeople())
                .Returns(new List<Person>
                    {
                        new Person("Person1", 1982, "Country1"),
                        new Person("Person2", 1992, "Country2"),
                        new Person("Person3", 1954, "Country3"),
                    });

            var result = _cut.Format();
            var expectedResult = @"Person1 born in Country1 on 1982
Person2 born in Country2 on 1992
Person3 born in Country3 on 1954";
            Assert.AreEqual(expectedResult, result);
        }
    }
```

Great. Now when the _cut object uses the ReadPeople method from the IPeopleDataReader interface that is its dependency, the mock will be used. It will return the predefined collection of people.

This **solves** all problems mentioned before:

- This test is now a **real unit test**. It tests the PersonalDataFormatter class in isolation.
- It is **fast** because it doesn't connect to a database.
- It has no way of **affecting other tests**, as it doesn't modify any shared state (with the test not using mocks, if the test would write to a database, it would modify its content for all other tests).
- The test will work on any machine, no matter if some database is present on it or not.
- We have full control over the data. We can define a small set of people that is enough for testing the PersonalDataFormatter. We won't be affected by the fact that there are millions of people in the database.

All right. Please notice that mocks have one more powerful ability - we can **verify** if some methods have been called upon them as part of the test verification. Let's consider this class.

```
public class EnthusiasticGreeter
{
    0 references
    public void PrintHelloNTimes(int n)
    {
        for(int i = 0; i < n; i++)
        {
            Console.WriteLine("Hello!");
        }
    }
}
```

This class is quite simple, but unfortunately, it is not easy to test. The PrintHelloNTimes method is void, so there is no result to be compared with the expected result.

The test that validates this class should basically have a way of checking if the "Hello!" was printed to the console given count of times. It could possibly be done by actually running the program (which would make this test non-unit) and

somehow intercepting the output printed to the console. But this would be complex, tricky, and non-unitary. After all, we would be testing the Console class as much as the EnthusiasticGreeter class.

The solution is again, to use mock. But what to mock here, exactly? Well, ideally it would be to mock the Console class, but this is impossible since it's static. In most frameworks, including Moq, the mocking mechanism is based on inheritance or interface implementations, so a mock object is basically a derived type from the type we want to mock or it implements the mocked interface. We can't have classes derived from static classes. Again, we will need to use Dependency Injection:

```
public class EnthusiasticGreeter
{
    readonly Action<string> _printToConsole;

    0 references
    public EnthusiasticGreeter(Action<string> printToConsole)
    {
        _printToConsole = printToConsole;
    }

    0 references
    public void PrintHelloNTimes(int n)
    {
        for(int i = 0; i < n; i++)
        {
            _printToConsole("Hello!");
        }
    }
}
```

In the production code, we will simply inject an action that uses Console.Writeline:

```
var enthusiasticGreeter = new EnthusiasticGreeter(
    message => Console.WriteLine(message));

enthusiasticGreeter.PrintHelloNTimes(5);
```

But for testing purposes, we will use a mock of the Action object:

```
[TestFixture]
0 references
public class EnthusiasticGreeterTests
{
    private Mock<Action<string>> _printToConsoleMock;
    private EnthusiasticGreeter _cut;

    [SetUp]
    0 references
    public void SetUp()
    {
        _printToConsoleMock = new Mock<Action<string>>();
        _cut = new EnthusiasticGreeter(
            _printToConsoleMock.Object);
    }
```

We can now write a test that checks that “Hello!” has been printed as many times as the number provided with the parameter:

```
[Test]
| 0 references
public void ShallPrintHello5Times_WhenCalledPrintHello5Times()
{
    _cut.PrintHelloNTimes(5);
    _printToConsoleMock.Verify(
        mock => mock("Hello!"), Times.Exactly(5));
}
```

As you can see, using mocks allowed us to test code that doesn’t return a value. Instead, we tested that a specific method was called with a given parameter and a given number of times.

Let’s summarize. Mocks are objects that can be used to substitute real dependencies for testing purposes. For example, we don’t want to use a real database connection in unit tests. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data. We can set up what will be the results of the methods called on mocks, as well as verify if a particular method has been called. Mocks are an essential part of unit testing, and it’s nearly impossible to test a real-life application without them.

Bonus questions:

- "**What is Moq?**"
  Moq is a popular mocking library for C#. It allows us to easily create mocks of interfaces, classes, Funcs, or Actions. It gives us the ability to decide what result

will be returned from the mocked functions, as well as validate if some function has been called, how many times, and with what parameters.

- "**What is the relation between mocking and Dependency Injection?**"
  Mocking is hard to implement without the Dependency Injection. Dependency Injection allows us to inject some dependencies to a class, so we can choose whether we inject real implementations or mocks. If the dependency of the class would not be injected but rather created right in the class, we could not switch it to a mock implementation for testing purposes.

47. What are NuGet packages?

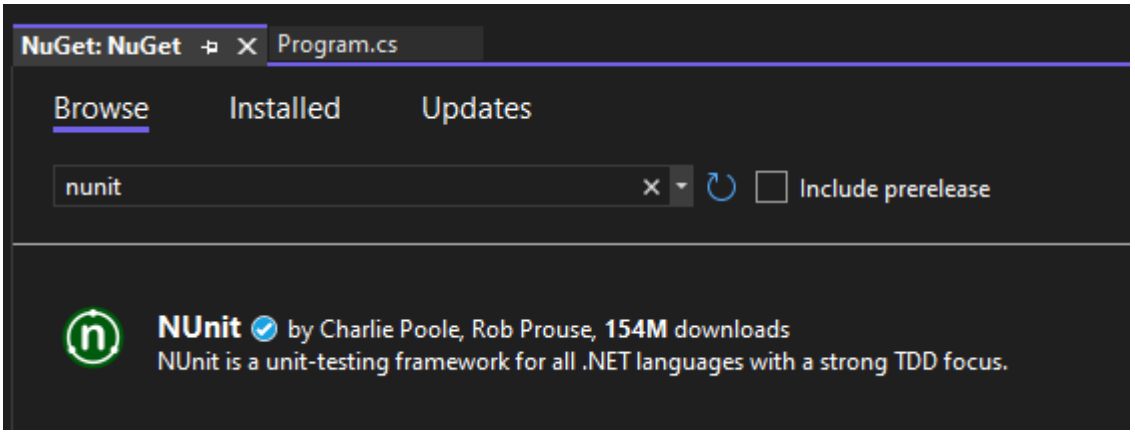
Brief summary: NuGet packages contain compiled code that someone else created, that we can reuse in our projects. The tool used to install and manage them is called NuGet Package Manager.

NuGet is a Microsoft-supported package manager, so a tool through which developers can create, share, and consume useful code.

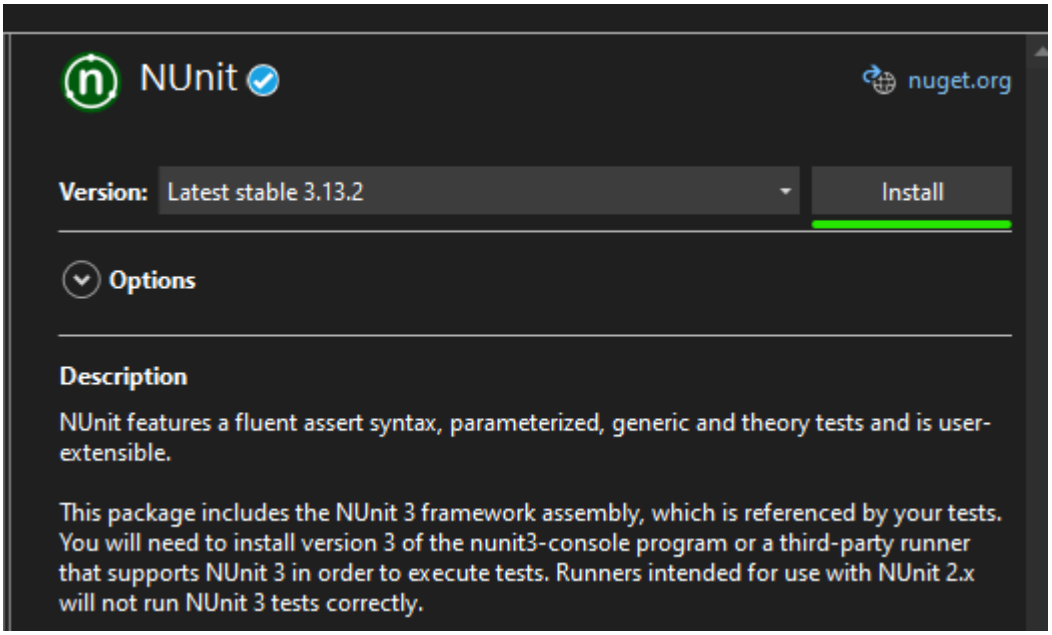
There are tons of libraries that other developers created, which we can use in our own projects. NuGet Package Manager is the tool that allows us to access them. Each package contains the dlls built from the code that someone else developed.

For instance, let's add NUnit and Moq to a project. NUnit is one of the most popular unit testing frameworks for C#, and Moq is a mocking library. Together they are two essential tools that we can use to create unit tests for our code.

The easiest way to install a NuGet package is by right-clicking on the project and selecting "Manage NuGet Packages". On the screen that opens we can search for the package that we want to install:



After selecting it, we can choose the version we want to install. Let's select the latest one.



After installing the package, we can start using it:

```
using NUnit.Framework;

[TestFixture]
0 references
public class Tests
{

}
```

There is another, sometimes even more convenient way of installing NuGet packages. We can simply start using the types from the package, and once Visual Studio complains that it doesn't know them, we can choose to install the package from the context menu:

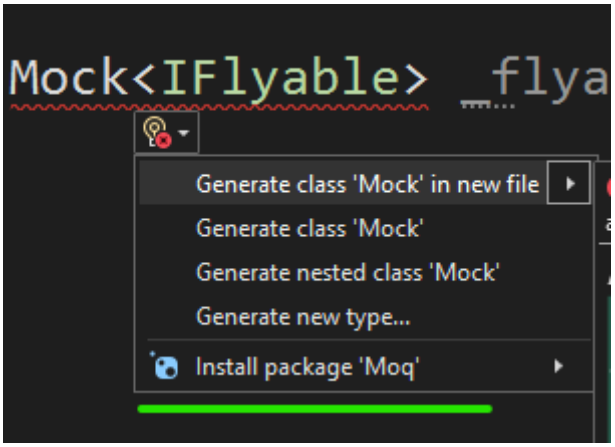
```
public interface IFlyable
{
    0 references
    public void Fly();
}

[TestFixture]
0 references
public class Tests
{
    private Mock<IFlyable> _flyableMock;
}
```

CS0246: The type or namespace name 'Mock<>' could not be found (are you missing a using directive or an assembly reference?)

Show potential fixes (Ctrl+.)

In this case, I'm trying to use Mock type from the Moq framework, which is not currently installed. I can click on the suggestion button to see that Visual Studio kindly offers to install this NuGet package for me:



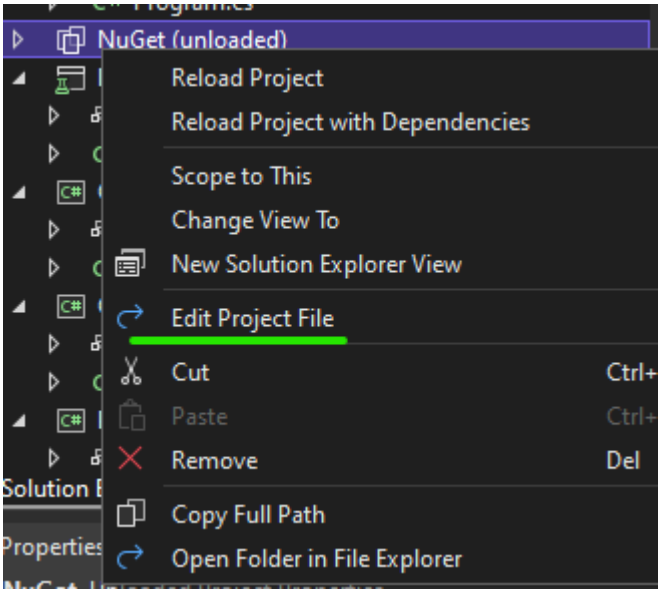
After doing so, the code compiles correctly:

```
using Moq;
using NUnit.Framework;

1 reference
public interface IFlyable
{
    0 references
    public void Fly();
}

[TestFixture]
0 references
public class Tests
{
    private Mock<IFlyable> _flyableMock;
}
```

Let's take a look at how the *.csproj file changed after installing those two packages. To see the *.csproj file of the project we must first unload it. Right-click on it and select "unload". After, you can right-click on the unloaded project again and select "Edit Project File".



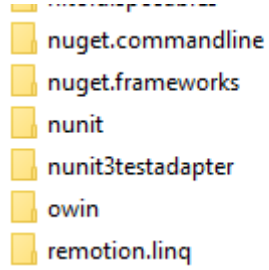
In the *.csproj file that will open we will see the entries that have been added by NuGet:

```
<ItemGroup>
  <PackageReference Include="Moq" Version="4.16.1" />
  <PackageReference Include="NUnit" Version="3.13.2" />
</ItemGroup>
```

The question is: where exactly did the packages get installed? Well, this evolved with the versions of .NET, but in .NET 6 which we use in this course, it by default gets installed in your Windows's user folder, for example in a path like this:

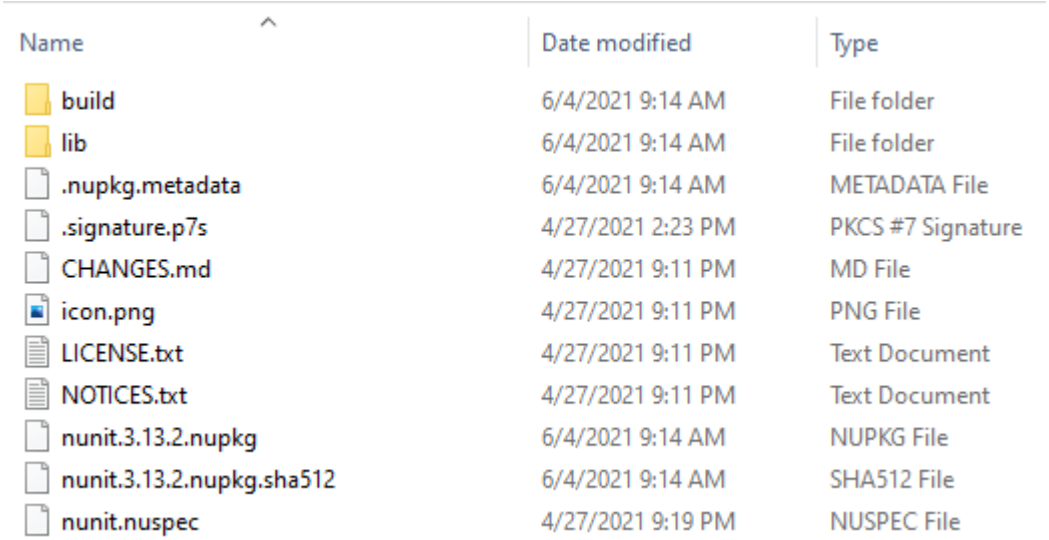
C:\Users\Krystyna\.nuget\packages

And here we can see the nunit folder:



Don't worry if for you it looks different. I have dozens of different coding projects on my computer and overall 344 NuGet packages installed.

Thanks to the fact that the package gets installed in the user folder, it can be reused between different projects. I have multiple projects using NUnit, but it only exists in a single copy on my machine. Let's take a look at what's inside such a NuGet package:



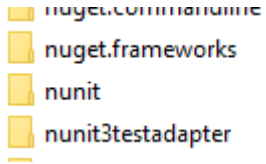
In the **lib** folder, we can find the actual dlls that get referenced from our project.

Now, I'll do something mean. I will remove the entire nunit folder from the .nuget\packages directory.

Let's see if the project will build correctly. After all, the dlls it needs have been deleted.

```
1>Done building project "NuGet.csproj".
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

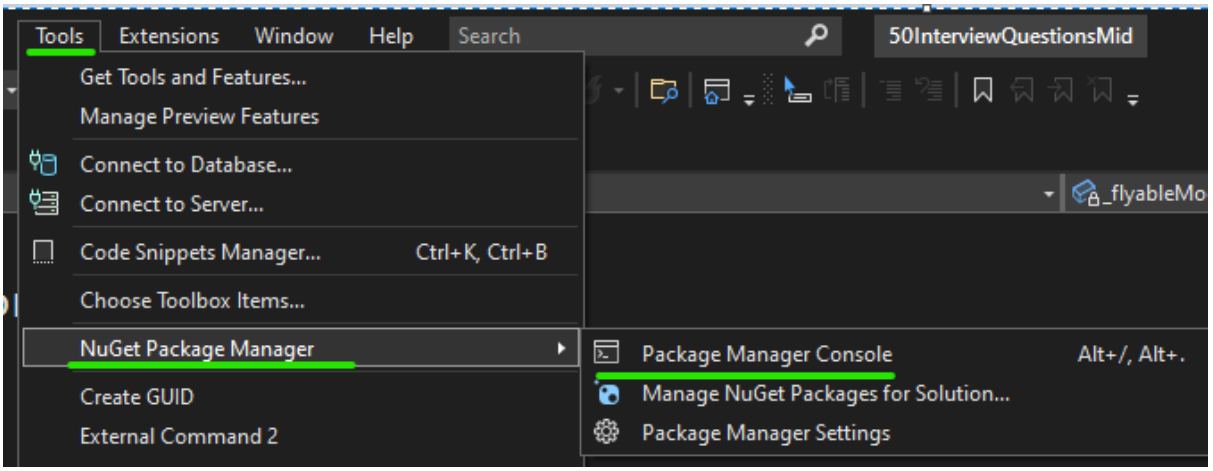
That's a bit surprising. The build was successful. Let's take a look into .nuget\packages directory again.



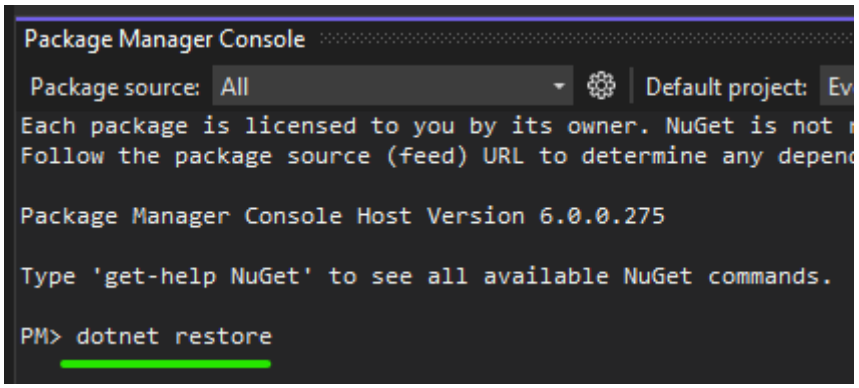
It seems like the nunit folder has "magically" reappeared.

Actually, it's no magic. The *.csproj file now clearly declares what packages it needs. Visual Studio knows that if the package is missing from the packages folder, it must simply reinstall it. This is quite convenient, especially if we share the code via some kind of repository. We only commit the code and package references to the repository, not the packages themselves. Once another programmer downloads the code and builds it, the packages get installed on his or her machine automatically.

One more thing. Sometimes Visual Studio messes something up and is not able to restore the packages. If this happens, you can always run Tools-> NuGet Package Manager -> Package Manager Console…



…and from this console, run "**dotnet restore**" command, which will restore all packages referenced in the solution.



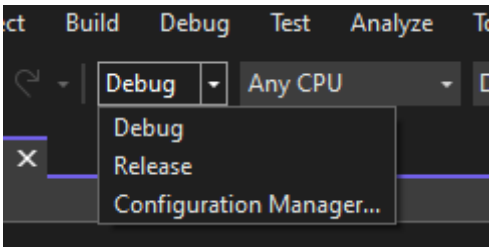
All right. We now know how to use the packages that someone else created. I highly recommend you use this beautiful concept, and not reinvent the wheel each time you need something done. In general, when you think of something not specific to your project, but rather something that could likely be used by other developers, there is a 99% chance there is a NuGet package that already does that.

If you have some nice ideas of your own, you can always create and publish your own NuGet packages. Here is a series of articles about it:
https://docs.microsoft.com/en-us/nuget/create-packages/overview-and-workflow

48. What is the difference between Debug and Release builds?

Brief summary: During the Release build, the compiler applies optimizations it finds appropriate. Because of that, the result of the build is often smaller and it works faster. On the other hand, it's harder to debug because the compiled result doesn't match the source code exactly.

When using Visual Studio, it's hard not to notice this little option in the top menu:



When developing the code, we most likely use the Debug mode and don't really think much about the other option - the Release mode. As their names suggest, the Debug mode is most appropriate when debugging the application, so mostly during the development, and the Release mode should be used when we intend to release the application so it can be used by users or other programs.

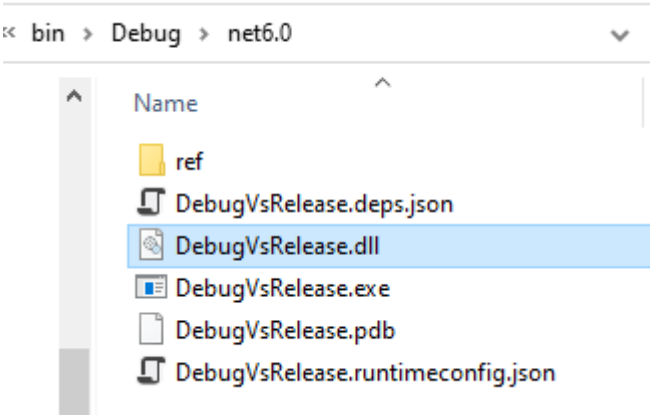
But what is the difference between them, exactly?

The main difference is that in the Release build, the compiler introduces some optimizations it finds appropriate. They can be things like removing unused variables or simplifying method calls. It helps to make the final CIL code smaller, simpler, and often faster. Remember, CIL stands for the Common Intermediate Language, which is the language to which the compiler compiles C# code. Those optimizations may make debugging harder because the compiled result doesn't match exactly the source code.

Let's see the simple compiler optimizations in practice. First, let me show you some code that is not optimal:

```
public static void Main(string[] args)
{
    int someUnusedVariable = 5;
    const string Hello = "Hello!";
    Console.WriteLine(Hello + Hello);
    Console.ReadKey();
}
```

As you can see, we have an unused variable here. Let me build this project in **Debug** mode. The build output will end up in the Debug folder:

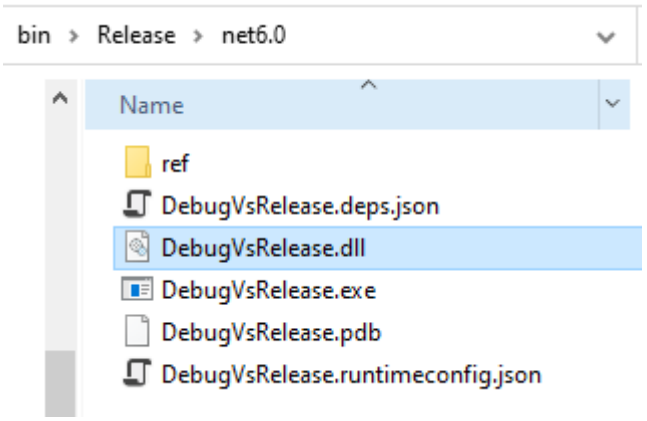


Here we can see the dll being the result of the program compilation. What I'm going to do now is to first see the CIL code using **ildasm**, and then I will use SharpLab.io to translate it back to C#, to see how the compilation affected the structure of the code:

```
public class Program
{
    [System.Runtime.CompilerServices.NullableContext(1)]
    public static void Main(string[] args)
    {
        int num = 5;
        Console.WriteLine("Hello!Hello!");
        Console.ReadKey();
    }
}
```

Well, it's not very exciting. We can see that the unused variable has been renamed to something shorter and that the const strings have been inlined. Now, let's do

the same thing but in the **Release** build. First of all, we will see it ends up in a different folder:



As you can see, the Release folder is used now. I will again take the CIL code that has been created by the compiler and will paste it to SharpLab to see what C# it translates to:

```
public class Program
{
    [System.Runtime.CompilerServices.NullableContext(1)]
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello!Hello!");
        Console.ReadKey();
    }
}
```

This time, the unused variable has been removed completely.

This was just an extremely simple example of what the compiler can optimize in the Release mode, but be aware that those optimizations can be much more serious. The compiler can simplify or inline method calls, remove whole pieces of code that it doesn't find useful, and so on. The algorithm behind that is quite complex and it depends on the compiler version, as well as such low-level details like how big the methods are. We can configure some of the optimizations settings with attributes like MethodImplOptions.NoInlining. You can read more about it here:
https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.methodimploptions?view=net-6.0

The code optimization done by the compiler is the main difference between the Debug and Release modes. Please note that if we want some code to be executed

only in one of those modes, we can put it in **#if DEBUG** or **#if RELEASE** conditional preprocessor directives:

```
#if DEBUG
Console.WriteLine("We are in Debug mode!");
#endif

#if RELEASE
Console.WriteLine("We are in Release mode!");
#endif
```

Let's summarize. During the Release build, the compiler applies optimizations it finds appropriate. Because of that, the result of the build is often smaller and it works faster. On the other hand, it's harder to debug because the compiled result doesn't match the source code exactly.

Bonus questions:

- "**How can we execute some piece of code only in the Debug, or only in the Release mode?**"
  *By placing it inside a **#if DEBUG** or **#if RELEASE** conditional preprocessor directives.*

49. What are preprocessor directives?

Brief summary: Preprocessor directives help us control the compilation process from the level of the code itself. We can choose if some part of the code will be compiled or not, we can disable or enable some compilation warnings, or we can even check for the .NET version and execute different code depending on it.

Preprocessor directives help us control the compilation process from the level of the code itself. We can choose if some part of the code will be compiled or not, we can disable or enable some compilation warnings, or we can even check for the .NET version and execute different code depending on it.

A **preprocessor** (also known as the "precompiler") is a program that runs before the actual compiler, that can apply some operations on code before it's compiled. Although the C# compiler doesn't have a separate preprocessor, the directives described in this lecture are processed as if there were one.

We can recognize preprocessor directives by the fact that they start with the # symbol. Preprocessor directive must be the only code in a line.

Let's see some of the most useful preprocessor directives in C#.

In the last lecture, we mentioned the **#if DEBUG** and **#if RELEASE** directives, which allow us to include some code into compilation only if we are in Debug or Release mode:

```
#if DEBUG
        Console.WriteLine("We are in Debug mode!");
#endif

#if RELEASE
Console.WriteLine("We are in Release mode!");
#endif
```

We can use #if, #elif, and #else directives to control what we want to compile. Besides checking for Debug or Release mode, we can also check things like the version of .NET or the app target, like iOS or Android. Let's see this in code:

```
#if (DEBUG && NET5_0_OR_GREATER)
    Console.WriteLine("We are in Debug mode in .NET 5 or greater.");
#elif (DEBUG && !NET5_0_OR_GREATER)
    Console.WriteLine("We are in Debug mode in .NET older than 5.");
#elif (RELEASE && NET6_0_IOS)
    Console.WriteLine("We are in Release mode and we target iOS.");
#endif
```

Those directives are very useful when we build an application that targets more than one .NET version or is meant to work on different platforms.

Another commonly used preprocessor directive is **#region**. #region allows us to define a region of code that can be collapsed in Visual Studio, so it doesn't affect the actual compilation process:

```
#region someCollapsibleCode
Console.WriteLine("Not much going on here");
Console.WriteLine("...");
Console.WriteLine("...");
#endregion
```

The above region can be collapsed into this:

Regions are often used with autogenerated code, that we don't really want to read that often. Some developers define regions in large files, so some parts of them can be collapsed, making the file seem smaller and easier to read. I wouldn't recommend that, as this is simply sweeping the problem under the rug. Refactoring is definitely a better approach.

We can also use **#error** and **#warning** preprocessor directives to explicitly create a compiler's warning or error:

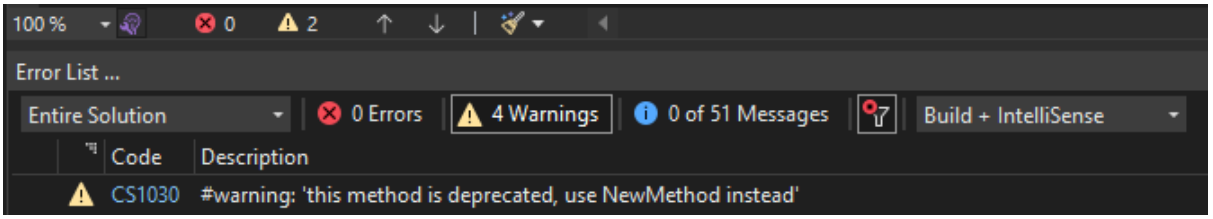
```
#warning this method is deprecated, use NewMethod instead
OldMethod();

Console.ReadKey();

void OldMethod()
{
    Console.WriteLine(
        "This method shouldn't be used, use NewMethod instead");
}

void NewMethod()
{
    Console.WriteLine("Bran new method");
}
```

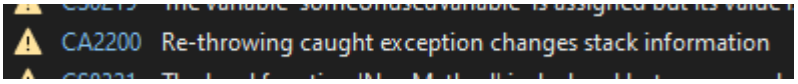
After adding the #warning, we will see it in the build output:



Speaking of **warnings** - we can **disable** some of them in a file, using another preprocessor directive: **#pragma warning disable**. Let's consider this simple code:

```
try
{
    //some code that can throw
}
catch (Exception ex)
{
    throw ex;
}
```

As we learned in the “What is the difference between throw and throw ex?” lecture, we should rather use “throw” instead of “throw ex”. The compiler actually warns us if we do this mistake:



If we are 100% sure what we are doing, we can disable this warning in the code using #pragma warning disable:

```
try
{
    //some code that can throw
}
catch (Exception ex)
{
#pragma warning disable CA2200
    throw ex;
#pragma warning restore CA2200
}
```

As you can see, we must specify the type of warning by its code. In this case, it is CA2200, which we saw in the compilation output next to the yellow triangle.

#pragma warning disable disables the warning till the end of the file, so it’s a good practice to restore it after the last line it should affect.

In the lecture “What are nullable reference types?” we will learn that enabling nullability checks for reference types may cause an enormous count of warnings in the project. We can disable those warnings for some files or code blocks using the **#nullable disable directive**:

```
#nullable disable
string nullable = null;
#nullable enable
```

This can be pretty handy if we want to fix the nullability warnings file by file. We can then disable them for the entire project, and enable them gradually in the files we fix one by one.

All right. We learned about some (but not all!) preprocessor directives in C#. As you can see they can be pretty useful when it comes to controlling what code is compiled under given circumstances, or what warnings are shown. If you want to read about all available preprocessor directives, make sure to check out this article from .NET documentation:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives

Bonus questions:

- "**What is the preprocessor?**"
  The preprocessor (also known as the "precompiler") is a program that runs before the actual compiler, that can apply some operations on code before it's compiled.

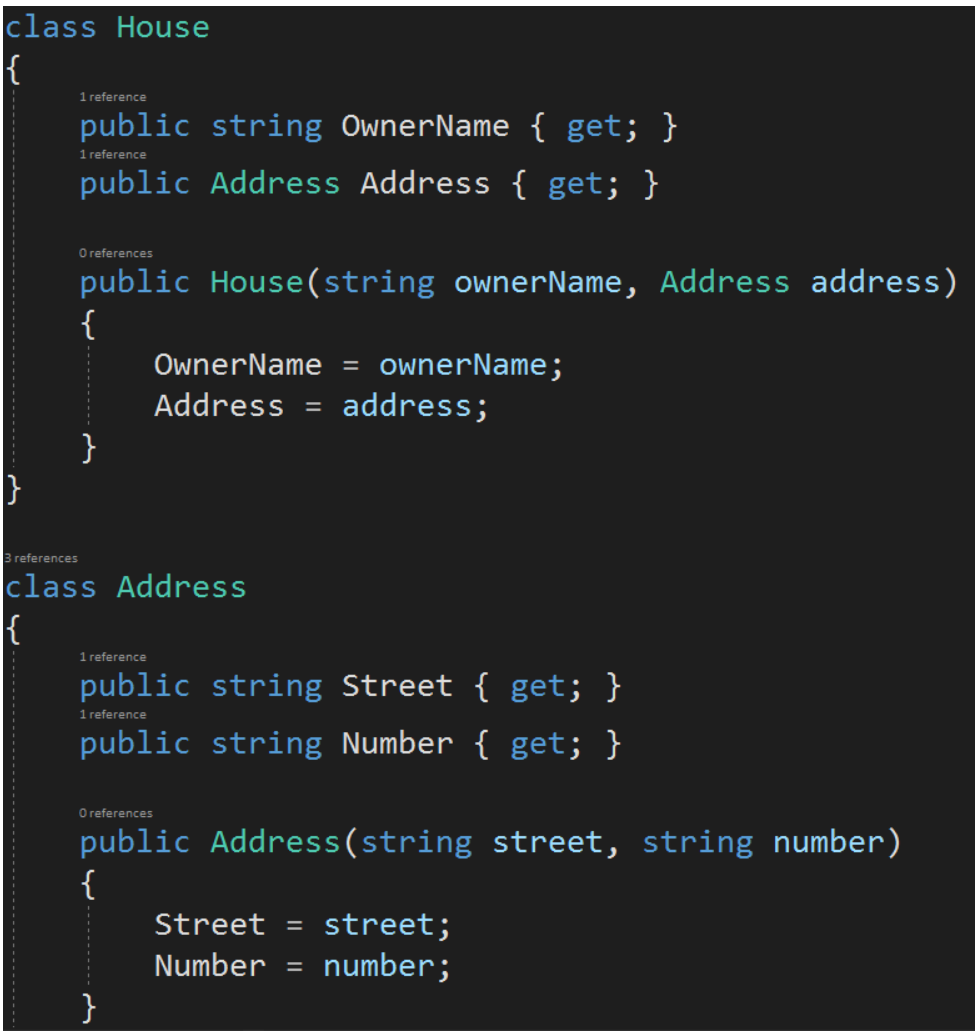
- "**How to disable selected warning in a file?**"
  By using the #pragma warning disable preprocessor directive. It takes the warning code as the parameter, so for example to disable the "Don't use throw ex" warning we can do "#pragma warning disable CA2200".

50. What are nullable reference types?

Brief summary: Nullable reference types is a feature introduced with C# 8, that enables explicit declaration of a reference type as nullable or not. The compiler will issue a warning when it recognizes the code in which a non-nullable object has a chance of being null, or when we use nullable reference types without null check, risking the NullReferenceException. This feature doesn't change the actual way of executing C# code; it only changes the generated warnings.

"*What are nullable reference types?*" At first glance, you may think this question is silly. After all, **all reference types are nullable** in C#, right? Well, yes, this is true and did not change with introducing the feature called "nullable reference types" with C# 8.

First, let me show you this code:



Let's say we work on an application that manages houses data. We use those two types - House and Address - all around it. Let's see a tiny fragment of this application, but keep in mind that the entire application can be huge.

```
string FormatHousesData(IEnumerable<House> houses)
{
    return string.Join("\n",
        houses.Select(house =>
        $"Owner is {house.OwnerName}, " +
        $"address is {house.Address.Number} " +
        $"{house.Address.Street}"));
}
```

All right. We submit this code for code review and wait for the feedback. Soon after we see a new comment:

"*Looking good, but if house.Address is null, NullReferenceException will be thrown. You can see Visual Studio warning you about that with the green underline*".

It is true. After a short discussion with the reviewer, we make a decision - the owner of the house should never be a null string. Also, the Address can't be null, and both Street and Number can't be null either.

See what happened here: we made a **decision** that in this **particular** case, the **nullable** type that is string and Address, should actually **not be nullable**.

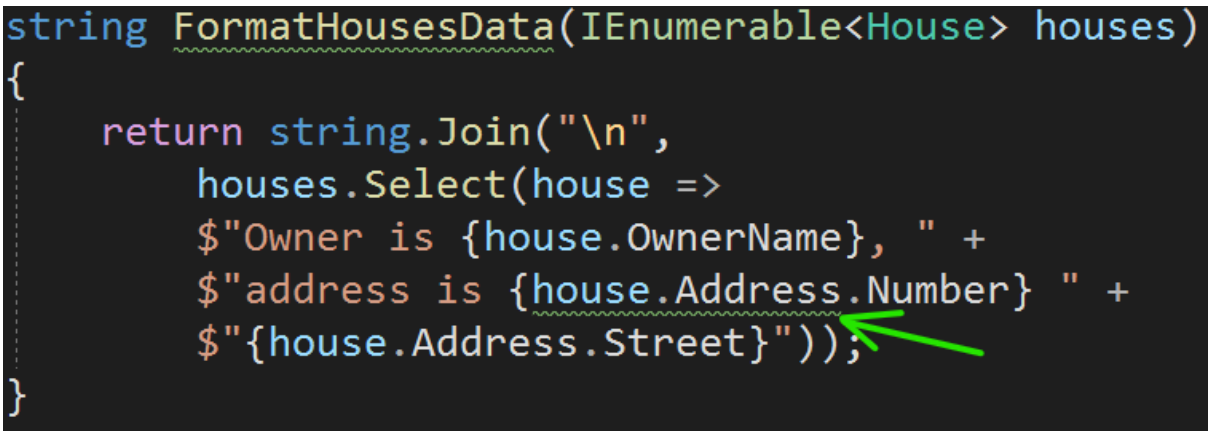
All right. We have work to do. If those properties should not be nullable, we must add some logic to the constructors:

```
public House(string ownerName, Address address)
{
    if(ownerName == null)
    {
        throw new ArgumentNullException(nameof(ownerName));
    }
    if (address == null)
    {
        throw new ArgumentNullException(nameof(address));
    }
    OwnerName = ownerName;
    Address = address;
}
```

```
public Address(string street, string number)
{
    if (street == null)
    {
        throw new ArgumentNullException(nameof(street));
    }
    if (number == null)
    {
        throw new ArgumentNullException(nameof(number));
    }
    Street = street;
    Number = number;
}
```

Great. Now it will simply be impossible to create a House in which the OwnerName is null, nor an Address with null Street or Number. We made them **practically not nullable,** even if as reference types **technically they are nullable**.

The problem is, Visual Studio (before it was updated to support C# 8) was not smart enough to know that we actually ensure that the Address is not null, and it would keep giving us the warning about possible NullReferenceException.



We could ignore those warnings, but it's not really a solution. Other developers, who are not aware that the constructor is enforcing values not to be null, may still be suspicious and will feel more comfortable with filling their code with endless checks for null values.

I guess you probably have seen such code quite often:

```
bool ValidateAddress(House house)
{
    if(house == null)
    {
        Console.WriteLine("house is null");
        return false;
    }
    if(house.Address == null)
    {
        Console.WriteLine("address is null");
        return false;
    }
    if(house.Address.Number == null)
    {
        Console.WriteLine("address/number is null");
        return false;
    }
    if(house.Address.Street == null)
    {
        Console.WriteLine("address/street is null");
        return false;
    }
    if(house.Address.Street == null)
    {
        Console.WriteLine("address/street is null");
        return false;
    }
    if(house.Address.Number.Length == 0)
    {
        Console.WriteLine("address/number is empty");
        return false;
    }
    if (house.Address.Street.Length == 0)
    {
        Console.WriteLine("address/street is empty");
        return false;
    }
    return true;
}
```

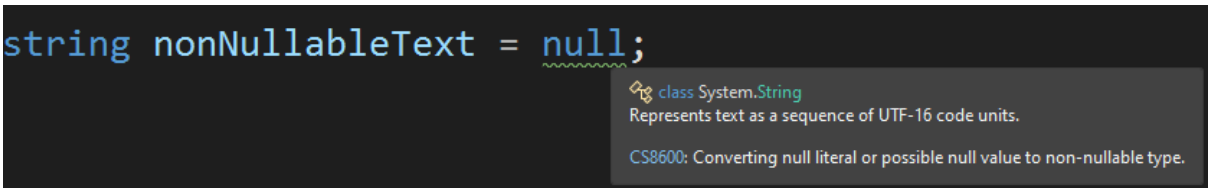
Such code sometimes takes a significant part of the entire codebase, making it bulky, hard to read, and unmaintainable.

Wouldn't it be just simpler if we could **agree** that the Address and its components can't be null and that we promise to enforce it at the constructor level?

Well, the need for such an agreement was exactly the reason for introducing **nullable reference types**.

This feature gives us the ability to declare a reference type as nullable or not nullable. If a type is declared as **not nullable**, the compiler will give us warnings in any context in which there is a risk that the value may actually be null. If a type **is nullable**, the compiler will give us a warning where a NullReferenceException could happen.

With C# 8 and newer, the "old" way of declaring reference types will make them not nullable:



As you can see, the compiler gives us a warning. We declared the variable as a non-nullable string, but we assigned null to it. This doesn't make much sense, hence the compiler warning. We can fix it by declaring the variable as nullable, the same way as we would declare nullable value types - by adding a question mark:

```
string? nullableText = null;
```

Now there is no warning. This variable is a nullable string, so assigning null to it is fine.

A very important note: **nullable reference types feature does not change anything in how the program is executed**. Even non-nullable values will still throw NullReferenceExceptions when null. **This feature only changes how compiler warnings are issued.**

All right. Now that we know the essence of nullable reference types, let's take another look at this type:

```
class Address
{
    4 references
    public string Street { get; }
    4 references
    public string Number { get; }

    0 references
    public Address(string street, string number)
    {
        Street = street;
        Number = number;
    }
}
```

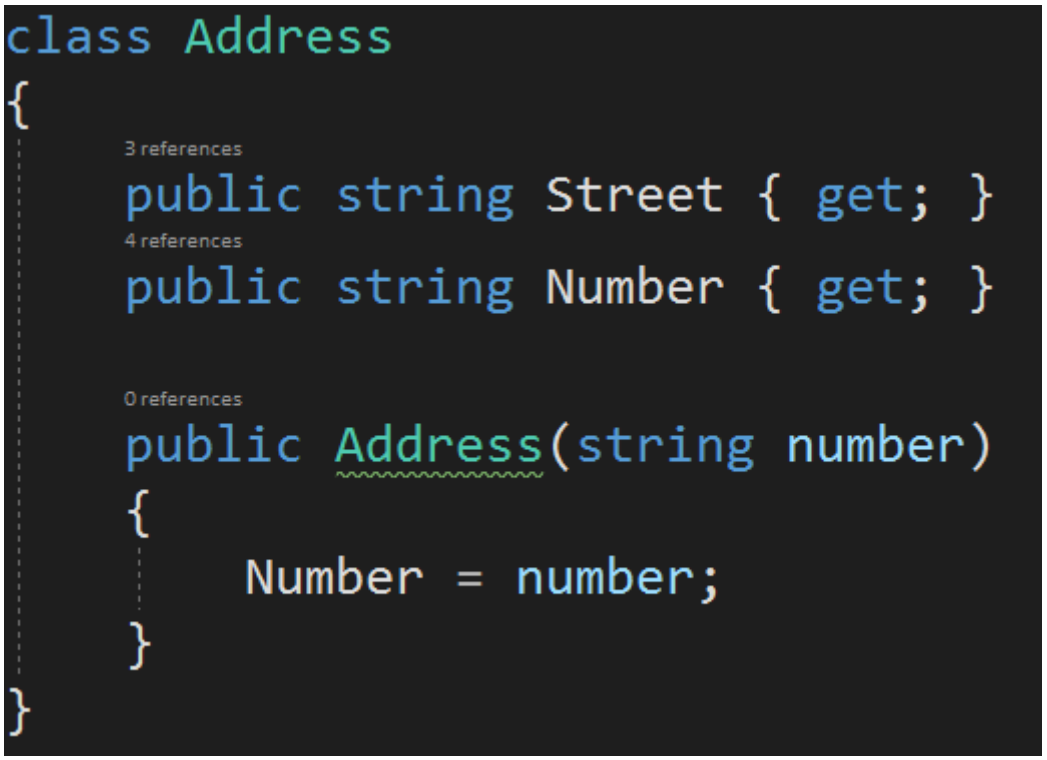
There are no compiler warnings here because the constructor parameters are non-nullable strings. But let me change something:

```
class Address
{
    4 references
    public string Street { get; }
    4 references
    public string Number { get; }

    0 references
    public Address(string? street, string number)
    {
        Street = street;
        Number = number;
    }
}
```

I made the street parameter nullable, and now we see warnings. They are here because we assign a nullable parameter to a non-nullable property, which obviously doesn't make much sense, and it may make this seemingly non-nullable property null.

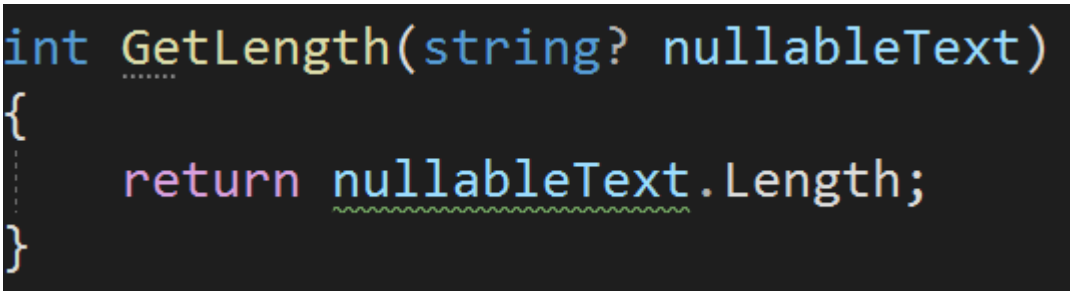
Let me show you one more case:



Now I removed the assignment to the Street property completely. The warning at the constructor makes sense - the Street property should not be null, but it will be because it's not assigned anything.

And this actually answers quite a tricky question - what is the default value for non-nullable reference types? Well, ironically, it's null (because what else could it be?)

The great thing about the nullable reference types feature is that it has good support from Visual Studio and other modern IDEs. Let me show you some code:



Here the warning is expected, because **nullableText** may be null. But let me add a null check:

```
int GetLength(string? nullableText)
{
    if(nullableText == null)
    {
        return 0;
    }
    return nullableText.Length;
}
```

The warning is gone because Visual Studio knows that in the last line of this method we can be sure that the parameter is not null, as this has already been checked.

Please notice that this IDE support is not infallible, and sometimes it can be tricked:

```
var array = new string[10];
Console.WriteLine(array[0].Length);
```

As you can see here I do something silly - I declare an array of non-nullable strings, yet by default, it is filled with nulls. No compiler warning appears, though, even if the "array[0].Length" will throw NullReferenceException.

It doesn't mean this feature is useless. Let me quote Jon Skeet on that: "*Being able to know when things might or might not be null, even when it's only to 90% confidence, is a lot better than 0% confidence.*"

If you are curious what else Jon Skeet has to say about nullable reference types, check out his lecture about this feature from GOTO Copenhagen 2019 Conference: https://www.youtube.com/watch?v=1tpyAQZFlZY

All right. There are use cases when we actually know better than the compiler if something is or is not null. One of the outstanding examples is when some field is set with the SetUp method in unit tests:

```
class Calculator
{
    1 reference
    public int Add(int a, int b) => a + b;
}

[TestFixture]
0 references
public class CalculatorTests
{
    private Calculator _cut;

    [SetUp]
    0 references
    public void SetUp()
    {
        _cut = new Calculator();
    }

    [Test]
    0 references
    public void Add5And10_ShallGive15()
    {
        Assert.AreEqual(15, _cut.Add(10, 5));
    }
}
```

If you don't know NUnit, let me give you a very quick introduction. The method with the **SetUp** attribute will be executed before each test, so it gives us the same guarantee as the constructor, that the **_cut** (Class Under Test) field will not be null. Yet, the compiler warns us that the **_cut** field may be null before exiting the constructor. C# compiler doesn't know how NUnit works, so it's not aware this field will never be null when the test is executed.

To get rid of the compiler warning, let's declare this field as nullable:

```
private Calculator? _cut;

[SetUp]
0 references
public void SetUp()
{
    _cut = new Calculator();
}

[Test]
0 references
public void Add5And10_ShallGive15()
{
    Assert.AreEqual(15, _cut.Add(10, 5));
}
```

Ugh… one warning disappeared, but another showed up. Now the compiler warns us that the _cut may be null when calling the Add method on it. Making the code compliant with the compiler's requirements about the nullable reference types is a bit like playing Whack-A-Mole. One warning disappears, but another pops out.

But in this case, **I know better**. I know the SetUp method will be executed first. I want to say "Quiet, compiler! I know it's not null!". And exactly for such situations, the **null-forgiving operator** was introduced:

```
_cut!.Add(10, 5));
```

As you can see, I can simply put "!" after a nullable reference type object which I know is not null to suppress the warning.

It can actually add it even if I know it's null, which is sometimes needed in unit tests. Let me show you an example.

First, let's take another look at the House class. Even if the ownerName and address parameters are non-nullable, I want to perform a null check here. This is a good practice - after all, one still can pass nulls there, because as we said, this

feature doesn't change how the code works - it only issues new warnings. Of course, the person that calls this constructor will see a warning when passing null as a parameter, but what if he or she will ignore it? After the House object is created, we want to be sure that we can really trust what was declared - that the OwnerName and Address properties are not null. That's why it's best to enforce it once and for all in the constructor.

```
class House
{
    2 references
    public string OwnerName { get; }
    8 references
    public Address Address { get; }

    1 reference
    public House(string ownerName, Address address)
    {
        if(ownerName == null)
        {
            throw new ArgumentNullException(nameof(ownerName));
        }
        if (address == null)
        {
            throw new ArgumentNullException(nameof(address));
        }
        OwnerName = ownerName;
        Address = address;
    }
}
```

This looks good. After this validation, we can be sure the OwnerName and Address will not be null anywhere in our code. We can forget about the neverending null-checks. The only thing left to do is to add unit tests for this constructor:

```
[TestFixture]
0 references
public class HouseTests
{
    [Test]
    0 references
    public void NullOwnerName_ShallThrowException()
    {
        Assert.Throws<ArgumentNullException>(() =>
            new House(null, new Address("Maple Street", "12B")));
    }
}
```

This test checks if an ArgumentNullException will be thrown if OwnerName is null. But even in this test scenario, the compiler gives me a warning. But I know what I'm doing. I want this null here, so I kindly ask the compiler to give me a break:

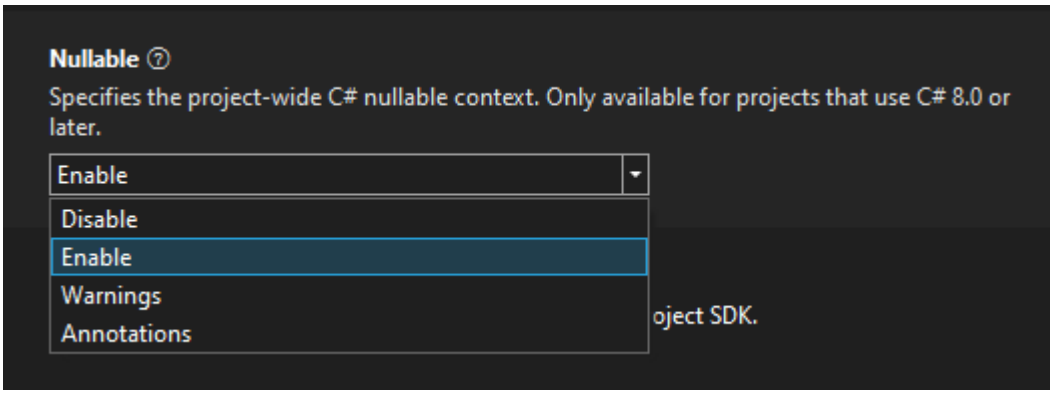
```
Assert.Throws<ArgumentNullException>(() =>
    new House(null!, new Address("Maple Street", "12B")));
```

Again, the null-forgiving operator proved to be useful.

Now we understand how the nullable reference types work. The question that remains is **when should we use them**.

Well, my advice is this: think about the types in your code, their fields, and properties, as well as local variables and parameters. Can they ever be null, or do you always ensure that they are not? If they can be null, make them nullable explicitly. This will clearly show what your intention was, and anyone working with your code will know that this thing may be null and needs to be checked for it.

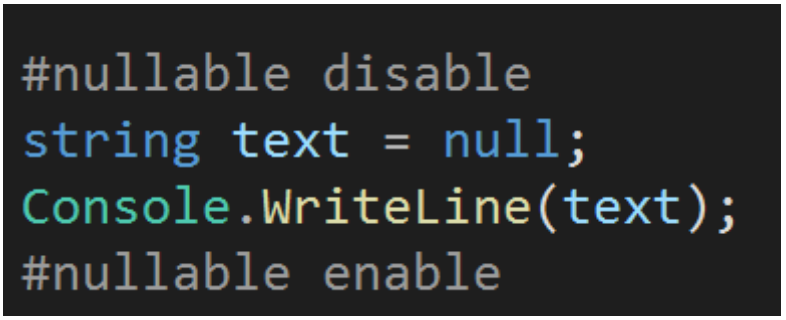
Also, a word of caution about migration. Since the "old" type declarations are made non-nullable starting with C# 8, it may mean that after updating your .NET and C# version you will suddenly get an overwhelming number of warnings. Don't worry - they are a good thing and will help you migrate to this new feature. But if you really don't want to see them, you can disable this feature in project properties:



The migration process itself can be a bit tiring (remember the Whack-A-Mole metaphor?) but having nullable reference types can really save you a lot of pain, errors and null checks, making the code cleaner, more expressive, and easier to maintain.

Even if you don't decide to introduce this feature in an existing project due to complex migration, I highly recommend using it in new code. You can disable or

enable this feature per file or even a code fragment. This way, you can improve your code step by step, and not drown in an ocean of warnings.



As you can see there is no warning here, even if we assign null to a non-nullable string.

Let's summarize. Nullable reference types is a feature introduced with C# 8, that enables explicit declaration of a reference type as nullable or not. The compiler will issue a warning when it recognizes the code in which a non-nullable object has a chance of being null, or when we use nullable reference types without null check, risking the NullReferenceException. This feature doesn't change the actual way of executing C# code; it only changes the generated warnings.

Bonus questions:

- "**What is the default value of non-nullable reference types?**"
  It is null.

- "**What is the purpose of the null-forgiving operator?**"
  It allows us to suppress a compiler warning related to nullability.

- "**Is it possible to enable or disable compiler warnings related to nullable reference types on the file level? If so, how to do it?**"
  *It is possible. We can do it by using **#nullable enable** and **#nullable disable** preprocessor directives.*

FINAL WORD

Thanks for reading this ebook! I hope it will help you during your next interview.

Check out my Udemy courses:

C#/.NET - 50 Essential Interview Questions (Junior Level)

Link: https://bit.ly/3hSRpOq

C#/.NET - 50 Essential Interview Questions (Mid Level)

Link: https://bit.ly/3sC7FsW

LINQ Tutorial: Master the Key C# Library

Link: https://bit.ly/3HqGR33