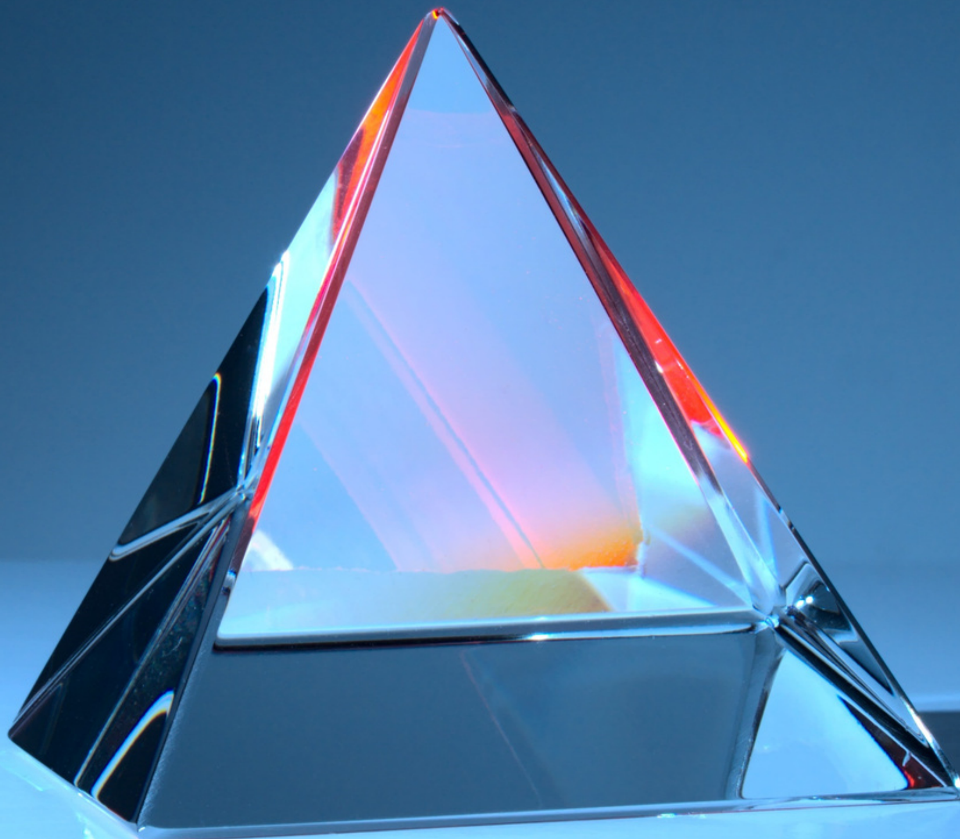
Krystyna Ślusarczyk

C#/.NET

15 ESSENTIAL INTERVIEW QUESTIONS

Junior Level



HELLO!

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

https://bit.ly/3sC7FsW

It contains 15 chapters, all with in-depth explanations of C#-related topics that you may need to understand some of the mid-level lectures.

All those 15 topics are part of my 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

1. What is the Common Intermediate Language (CIL)?
2. What is the Common Language Runtime?
3. What is the difference between value types and reference types?
4. What is boxing and unboxing?
5. What is the difference between a class and a struct?
6. What is LINQ?
7. What are extension methods?
8. What is IEnumerable?
9. What is the Garbage Collector?
10. What are nullable types?
11. What are generics?
12. What is the difference between an interface and an abstract class?
13. What is the Bridge design pattern?
14. What is the Single Responsibility Principle?
15. What is the Open-Closed Principle?

1. What is the Common Intermediate Language?

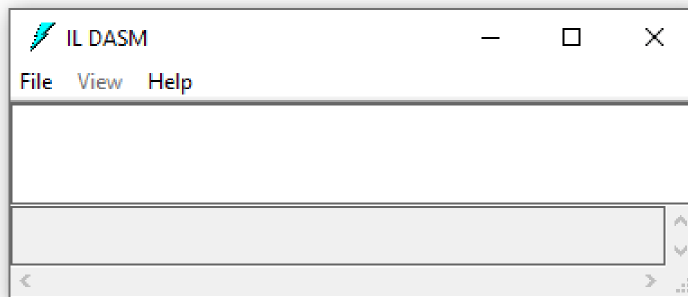
Brief summary: The Common Intermediate Language is a programming language that all .NET-compatible languages like C#, Visual Basic, or F# get compiled to.

The **Common Intermediate Language** (sometimes, for short, referred to as the Intermediate Language) is a programming language. When the source code written in .NET-compatible languages like C#, Visual Basic, or F# gets compiled, it is transformed into code written in Common Intermediate Language. When the application is started, the Common Language Runtime's **JIT Compiler** translates the CIL code to binary code (JIT stands for "Just-In-Time", which means that a particular piece of code will be translated from CIL to binary code just before it is executed for the first time).

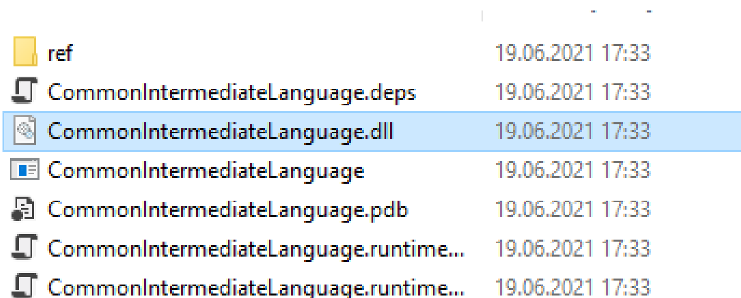
Let's write a simple C# code and see what it looks like after being translated to CIL:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello! What's your name?");
        var name = Console.ReadLine();

        Console.WriteLine($"Nice to meet you, {name}. How old are you?");

        var ageAsText = Console.ReadLine();

        if(int.TryParse(ageAsText, out int age))
        {
            Console.WriteLine($"That young, only {age}?");
        }
        else
        {
            Console.WriteLine("Sorry, I didn't get that.");
        }
        Console.WriteLine("Well, it was nice to meet you! Bye, bye!");
        Console.ReadKey();
    }
}
```

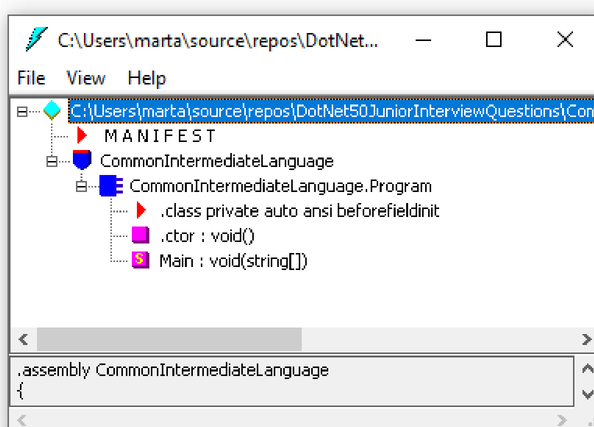
To see the CIL code, we must first make sure to build the solution in the Visual Studio. Then we are going to use **Ildasm** to view the CIL code. Ildasm is the Intermediate Language Disassembler, and it gets installed when you install .NET on your machine. On my machine it got installed in C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\ildasm.exe. Let's run this tool:



Now, we must find the *.dll for which we want to see the CIL code. The simplest solution is to right-click the project in the Visual Studio and then select "Open Folder in File Explorer". Then, we must go to the output folder. In my case it is /bin/Debug/net5.0. There I can find the *.dll that was built by the Visual Studio:



We can simply drag and drop it to Ildasm:



All right, let's see how the Main method looks in CIL code. Let's double-click on the Main method in Ildasm. This is what we will see:

CommonIntermediateLanguage.Program::Main : void(string[])

Find Find Next

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       115 (0x73)
  .maxstack  3
  .locals init (string V_0,
           string V_1,
           int32 V_2,
           bool V_3)
  IL_0000:  nop
  IL_0001:  ldstr      "Hello! What's your name\?"
  IL_0006:  call       void [System.Console]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  call       string [System.Console]System.Console::ReadLine()
  IL_0011:  stloc.0
  IL_0012:  ldstr      "Nice to meet you, "
  IL_0017:  ldloc.0
  IL_0018:  ldstr      ". How old are you\?"
  IL_001d:  call       string [System.Runtime]System.String::Concat(string,
                                                                    string,
                                                                    string)
  IL_0022:  call       void [System.Console]System.Console::WriteLine(string)
  IL_0027:  nop
  IL_0028:  call       string [System.Console]System.Console::ReadLine()
  IL_002d:  stloc.1
  IL_002e:  ldloc.1
  IL_002f:  ldloca.s   V_2
  IL_0031:  call       bool [System.Runtime]System.Int32::TryParse(string,
                                                                   int32&)
  IL_0036:  stloc.3
  IL_0037:  ldloc.3
  IL_0038:  brfalse.s  IL_0054
  IL_003a:  nop
  IL_003b:  ldstr      "That young, only {0}\?"
  IL_0040:  ldloc.2
  IL_0041:  box        [System.Runtime]System.Int32
  IL_0046:  call       string [System.Runtime]System.String::Format(string,
                                                                    object)
  IL_004b:  call       void [System.Console]System.Console::WriteLine(string)
  IL_0050:  nop
  IL_0051:  nop
  IL_0052:  br.s       IL_0061
  IL_0054:  nop
  IL_0055:  ldstr      "Sorry, I didn't get that."
  IL_005a:  call       void [System.Console]System.Console::WriteLine(string)
  IL_005f:  nop
  IL_0060:  nop
  IL_0061:  ldstr      "It was nice to meet you!"
  IL_0066:  call       void [System.Console]System.Console::WriteLine(string)
  IL_006b:  nop
  IL_006c:  call       valuetype [System.Console]System.ConsoleKeyInfo [System.Console]System.Console::ReadKey()
  IL_0071:  pop
  IL_0072:  ret
} // end of method Program::Main
```

This might not be the most beautiful programming language you've ever seen, but on the other hand, it is not completely unreadable, as one could expect.

Remember - all .NET compatible languages, not only C#, get compiled to the CIL. That enables communication between, for example, a C# and an F# libraries. For example, we can have a C# class derived from an F# class exactly because they both get compiled to the same programming language - the CIL.

Tip: other interview questions on this topic:

- "**How is it possible that a C# class can derive from, for example, an F# class?**"
  It is possible because both those languages are .NET compatible and they get compiled to the Common Intermediate Language.

- "**Does C# compiler compile C# source code directly to binary code?**"
  No, it compiles it to the Intermediate Language, which is compiled to binary code by the Just-In-Time compiler in runtime.

- "**How can you see the CIL code a project got compiled to?**"
  *Some tools can decompile a *.dll file and read the CIL code. One of those tools is Ildasm.*

- *"***What is the Just-In-Time compiler?** *"*
  Just-In-Time compiler is a feature of the Common Language Runtime (CLR), which translates the Common Intermediate Language (CIL) code to binary code during the program execution.

2. What is the Common Language Runtime (CLR)?

Brief summary: The Common Language Runtime is a runtime environment that manages the execution of the .NET applications.

The **Common Language Runtime** is a runtime environment that manages the execution of .NET applications. The CLR works as a special "operating system" for .NET applications, that manages all operations (like memory management) that otherwise must have been dealt with by a programmer. The CLR stands between the actual operating system (for example Windows) and the application.

Before we move on, I would like to say one thing: you may think that CLR is not a junior-level topic, and you might be right - this is a pretty low-level feature of .NET and you certainly can start programming .NET applications without even knowing that the CLR exists. Nevertheless, I wanted to introduce this topic as throughout this course I mention the CLR very often, as it affects many aspects of .NET programming. I want to make sure every subject in this course is explained in detail, even if it sometimes exceeds the junior level. **Some of the topics of this course simply can't be understood thoroughly without a basic understanding of the role of the CLR.**

All right, let's move on then. The important thing to understand is that the CLR is the .NET component that is not exclusive for C# applications. All programs written for the .NET are executed by the CLR. All code executed under the CLR is called the **managed code**. Thanks to the CLR, cross-language integration is supported in .NET. For example, you can have a C#'s class derived from a class defined in F#, because the CLR can understand both languages (because both are compiled to the Intermediate Language).

The CLR is responsible for many operations essential for any .NET application to work. Some of them are:

- **JIT (Just-in-time) compilation** - the compilation of the Common Intermediate Language to the binary code. Thanks to that the .NET applications can be used cross-platform because the code is compiled to platform-specific binary code only right before execution. See the "What is the Common Intermediate Language (CIL)?" lecture for more information on that.

- **Memory management** - CLR allocates the memory needed for every object created within the application. CLR also includes the Garbage Collector, which is responsible for releasing and defragmenting the memory. See the "What is the Garbage Collector?" lecture for more information.
- **Exception handling** - when the exception is thrown, the CLR makes sure the code execution is redirected to the proper catch clause. See the "How are exceptions handled in C#?" lecture for more information.
- **Thread management** - threads are beyond junior level, so let's just shortly say that the CLR manages the execution of the multi-threaded applications, making sure all threads work together well
- **Type safety** - part of the CLR is the **CTS - Common Type System**. CTS defines the standard for all .NET-compatible languages. Thanks to that, the CLR can understand types defined in C#, F#, Visual Basic, and so on, enabling cross-language integration.
- And many more

The CLR is the implementation of the CLI - **Common Language Infrastructure**. CLI was originally created by Microsoft and is standardized by ISO and ECMA. Sounds confusing? Let's explain it in this way - CLI is like a design of a house - it describes where walls, windows, piping, and electricity goes. It was originally created by Microsoft. ISO and ECMA are like state authorities who approve the design and make sure it is safe and reasonable, and that houses built based on this design will all function properly. Using this design, Microsoft builds a house. In this metaphor, the design is the CLI and the house built by Microsoft is the CLR. The thing about designs is that we can build many things based on the same design. That means another company could implement its own version of the project in accordance with the CLI. Does it ever happen? Actually, it does! For example, there is Mono Runtime, a counterpart of the CLR developed by a company called Ximian.

By now you should have a general idea of what the CLR does, but before we wrap up let's go **step-by-step** through a (simplified) process of creating and running the application to see when and how the CLR plays its role.

1. The programmer writes the program, which at first is just a bunch of text files.
2. The compiler compiles the text file to Common Intermediate Language, which is platform-independent. The compiler also prepares the metadata that describes all the types along with the methods they include.
3. Now the CLR comes into play. It starts the program under the specific operating system.
4. The CLR's Just-In-Time compiler compiles the Intermediate Language to binary code that can be interpreted by the machine's operating system. It uses the metadata prepared by the compiler.

5. As the application runs, the CLR manages all its low-level aspects - memory management, threads, exceptions handling, and so on.
6. When the application stops, the CLR's job is done.

As you can see, the **CLR is the critical component of .NET**. It manages basically everything that happens under the hood of the running application, allowing us, the programmers, to focus on business aspects of the development. Before tools like the CLR were introduced, programmers must have dealt with all things like memory management, which is the case in languages like C (can you imagine allocating memory for the objects by hand?).

Tip: other interview questions on this topic:

- "**What is the difference between CLR, CLI, and CIL?**"
  CLR (Common Language Runtime) is an implementation of the CLI (Common Language Infrastructure). CIL is Common Intermediate Language, to which all .NET-compatible languages get compiled.

- "**What is the CTS?**"
  CTS is the Common Type System, which is a standardized type system for all .NET-compatible languages, which makes them interoperable - for example, we can have a C# class derived from an F# class.

- "**Is the CLR the only implementation of the CIL**?"
  No. Anyone can create their implementation of the CIL. One of the examples is Mono Runtime.

3. What is the difference between value types and reference types?

Brief summary: The differences between value types and reference types are:

1. Value types inherit from System.ValueType while reference types inherit from System.Object.
2. Value types are passed by copy, reference types are passed by reference.
3. On assignment, a variable of a value type is copied. For reference types, only a reference is copied.
4. All value types are sealed (which means, they cannot be inherited)
5. Value types are stored on the stack, reference types are stored on the heap (because of that, the Garbage Collector only cleans up reference types)

In C# we distinguish **two** types of variables:

- First are **value types**. Simple built-in types like int, double, DateTime, bool are value types. Also, all structs are value types.
- Second are **reference types**. Object, String, StringBuilder, List, Array, HttpClient, XmlSerializer, and all user-defined classes are reference types.

The fundamental difference between them is that a reference type variable only holds a reference (you can think of it as a link or an address) to the actual data, while the value type variable holds the actual data.

To better understand this let's consider two real-life scenarios:

- I went for lunch with some interesting person - let's call him Bob - and after the meeting, I've noted Bob's phone number on the piece of paper. Then I've met with you and told you how nice it was to meet Bob. You asked me for his phone number. You noted it down on your own piece of paper. Bob's phone number is a **value type**. You created a **copy** of the actual piece of information. Now, if I **change** what I wrote down on my piece of paper, it **will not affect** what you have on yours. The variable - in this case, the piece of paper - is a piece of information itself.
- I went to the library and asked where I can find Jon Skeet's "C# in depth". The librarian gave me a piece of paper with the number of the row and the bookshelf. I went over there and enjoyed reading. Then I met with you and told you how great this book is. You've noted down the "**address**" of the book. In this metaphor, the book is a reference type. I can only give you the

address - called "**reference**" in C#. If you go to the library, and you draw some doodles on the pages (please don't!) it will also affect the book I would be reading - because this is the same book! There is only one copy in the memory, but it can be pointed to by multiple references.

Now, let's look closer and more technically at the differences between value and reference types.

The first basic difference is that all reference types inherit from System.Object whereas all value types inherit from System.ValueType.

Another difference is that **value types are passed by copy whereas reference types are passed by reference**. Let's see some code to understand what it means:

```
private static void AddOne(int number)
{
    ++number;
}

private static void AddOneToList(List<int> list)
{
    list.Add(1);
}
```

As you can see we defined two methods. Both of them alter the parameter that was passed to them. The fundamental difference is that the AddOne method takes a value type, while the AddOneToList takes a reference type. What do you think this code will print?

```
int a = 5;
Console.WriteLine($"Number is {a}");
AddOne(a);
Console.WriteLine($"Now number is {a}");
```

Well, it will print "5" twice! This is because the a variable has been passed to the AddMethod by a copy. The AddMethod incremented the copy, so the original a variable has not been affected.

```
Number is 5
Now number is 5
```

Let's see a similar situation, but for the reference types:

```
var list = new List<int>();
Console.WriteLine($"List contains {list.Count} elements");
AddOneToList(list);
Console.WriteLine($"Now list contains {list.Count} elements\n");
```

What do you think will be printed? Well, since the List is a reference type, the AddOneToList does not operate on its copy, but on the same object that is referenced by the list variable. Because of that, the number of elements in the list variable has been incremented by one.

```
List contains 0 elements
Now list contains 1 elements
```

Another difference between value and reference types is that when assigning the value of the value type to the new variable, **a copy is created**. The change in the original object will not affect the new object. For reference types, when assigning the value to the new variable, **only the reference is copied**, while the original object still exists in one copy only. That means that the change in the original object will also affect what is stored in the new variable.

Let's see this in code for value types:

```
int b = 10;
int c = b;
++c;
Console.WriteLine($"Number 'b' is {b}");
Console.WriteLine($"Number 'c' is {c}\n");
```

This is the result printed to the console:

```
Number 'b' is 10
Number 'c' is 11
```

Incrementing the c variable did not affect the b variable, since a copy was created at the assignment.

Now, let's take a look at the reference types:

```
List<int> listB = new List<int> { 1, 2, 3 };
List<int> listC = listB;
listC.Add(4);
Console.WriteLine($"listB contains {listB.Count} elements");
Console.WriteLine($"listC contains {listC.Count} elements\n");
```

For this code the result will be:

```
listB contains 4 elements
listC contains 4 elements
```

At assignment, no copy of the listB was created. Only the reference was copied, and it still points to the same object. Because of that adding an element to the listC variable also affected listB - because listC and listB point to the same object.

There are also some more technical differences between value types and reference types. Firstly, all value types are sealed, which means other types can't inherit from them. Because of that, value types can't have virtual or abstract members. See the "What is the purpose of the "sealed" modifier?" lecture for more information.

Secondly, value types are stored on the **stack** whereas reference types are stored on the **heap** (only the reference itself is stored on the stack). The value of the value type variable is cleaned out from the stack when the code execution leaves the scope this variable lived in. For reference types it is not the case - the object addressed by the reference will be cleaned up by the Garbage Collector and the exact time of that is unknown.

Let's summarize the differences between value types and reference types:

- All reference types inherit from System.Object whereas all value types inherit from System.ValueType.
- Value types are passed by copy whereas reference types are passed by reference.
- When assigning the value of the value type to the new variable, a copy is created. The change in the original object will not affect the new object. For reference types, when assigning the value to the new variable, only the reference is copied, while the original object still exists in one copy only. That means that the change in the original object will also affect what is stored in the new variable.

- All value types are sealed, which means other types can't inherit from them. Because of that, value types can't have virtual or abstract members.
- Value types are stored on the stack whereas reference types are stored on the heap (only the reference itself is stored on the stack).
- The value of the value type variable is cleaned out from the stack when the code execution leaves the scope this variable lived in. For reference types it is not the case - the object addressed by the reference will be cleaned up by the Garbage Collector and the exact time of that is unknown.

Tip: other interview questions on this topic.

- "**What will happen if you pass an integer to a method and you increase it by one in the method's body? Will the variable you passed to the method be incremented?**"
  The number will be increased in the scope of the method's body, but the variable outside this method will stay unmodified because a copy was passed to the method.

- "**Assuming you want the modification to the integer parameter to affect the variable that was passed to a method, how would you achieve that?**"
  By using ref parameter. See the question "What is the difference between the "ref" and the "out" keywords?".

4. What is boxing and unboxing?

Brief summary: Boxing is the process of wrapping a value type into an instance of a type System.Object. Unboxing is the opposite - the process of converting the boxed value back to a value type.

To understand what boxing and unboxing are, it is essential to understand what value and reference types are. You can learn about them from the "What is the difference between value types and reference types?" lecture.

Boxing is the process of converting a **value type to the System.Object type**. **Unboxing** is the **opposite** - the process of converting the boxed value back to value type. When the value is boxed, it is wrapped inside an instance of the System.Object class and stored on the heap.

As we know, value types are stored on the **stack** while reference types are stored on the **heap**. Only the reference itself (so an "address" or "pointer" to the object stored on the heap) is stored on the stack. Let's see a short piece of code:

```
int number = 5;
string word = "abc";
```

In this situation the following data is stored in the memory:

- On the stack
  - The value of number variable (5), as an integer is a **value** type
  - The reference to the word variable stored on the heap, as string is a **reference** type
- On the heap
  - The value of the word variable ("abc")

Let's box the value of the number variable:

```
int number = 5;
object boxedNumber = number;
```

A new variable of type object is created. Object is a reference type, so its value is stored on the heap. Let's see the state of the stack and the heap now:

- On the stack
  - The value of number variable (5), as an integer is a value type
  - The reference to the word variable stored on the heap
  - The reference to the boxedNumber variable stored on the heap
- On the heap
  - The value of the word variable ("abc")
  - The value of the boxedNumber variable (5)

As you can see, boxing is done implicitly. On the other hand, the unboxing must be done explicitly by using a cast:

```
int number = 5;
object boxedNumber = number;
int unboxedNumber = (int)boxedNumber;
```

Unboxing unwraps the original value from the object and assigns it to a value type variable.

The unboxing requires the exact type match. For example, this would throw an exception, because integer is not the same as short:

```
//this will throw
//int unboxedShortNumber = (int)boxedShortNumber;

short otherShortNumber = 3;

//this will work fine - no boxing or unboxing here
int otherShortNumberCastToInt = (int)otherShortNumber;
```

Without boxing and unboxing, casting a short to an integer works fine.

Please be aware that boxing and unboxing come with a **performance penalty**. Unlike regular variables assignment, boxing requires the creation of a new object and allocating memory on the heap for it. The unboxing requires a cast, which is also computationally expensive.

Now we know how boxing and unboxing are done and what exactly they do. **But what's their use?** Well, boxing and unboxing are necessary for providing a **unified type system** - that is, that we can treat any variable in C# as an **object**. Without boxing and unboxing, we couldn't have the ultimately generic code that accepts

any type of variable - we would have to distinguish value and reference types and possibly provide separate implementations for both of them. That was particularly useful before the generic types were introduced, and classes like ArrayList (used for storing any type of data) were commonly used. Even nowadays it is still used, for example in ADO.NET which is used to store objects in databases - at some point, this framework treats every piece of data as an object. Without boxing, it wouldn't be able to handle value types.

Tip: other interview questions on this topic:

- "**What is the penalty for using boxing and unboxing?**"
  The main penalty is performance - when boxing, a new object must be created, which involves allocating memory for it. The unboxing requires a cast which is also expensive from the performance point of view.

- "**Is assigning a string to a variable of type object boxing?**"
  No, because string is not a value type. The point of boxing is to wrap a value type into an object (which is a reference type).

5. What is the difference between a class and a struct?

Brief summary:

1. Structs are value types and classes are reference types.
2. Structs can only have a constructor with parameters, and all the struct's fields must be assigned in this constructor.
3. Structs can't have explicit parameterless constructors.
4. Structs can't have destructors.

There are several differences between structs and classes:

- **Structs are value types and classes are reference types.** Please refer to the "What is the difference between value types and reference types?" lecture for more information. Structs, like all value types:
    - inherit from System.ValueType
    - are passed by copy
    - are sealed (which means, they cannot be inherited)
    - are stored on the stack

- Structs can only have a **constructor with parameters**, and all the struct's fields must be assigned in this constructor. Structs can't have explicit parameterless constructors. (The reason for that is quite low-level and technical, and probably beyond Junior's level of knowledge. If you are curious, I recommend this thread, where Jon Skeet explains this topic thoroughly: https://stackoverflow.com/questions/333829/why-cant-i-define-a-default-constructor-for-a-struct-in-net)

```
struct Point
{
    public int x;
    public int y;

    //does not compile - struct can't have
    //explicit parameterless constuctor
    public Point()
    {

    }

    //does not compile - all fields must be
    //assigned in the constructor
    public Point(int x)
    {

    }
}
```

- **Structs can't have destructors** (finalizers). A destructor is a special method that is executed when the object is being removed from the memory. To understand why structs can't have destructors, you must remember that they are value types, and whenever we assign one struct variable to another or pass a struct as a parameter, a copy of this struct is created. Now, imagine you would have a struct that manages a database connection, and in the Finalize method, you would like to close this connection. Then, let's say you pass your struct as a parameter to a method. A copy of the struct is created, and at the end of the method's scope, the struct's Finalize would be called, closing the database connection. But what about the struct you actually passed to a method? The one that a copy was created from? It would suddenly stop working, as the connection would be closed. This is why a struct can't have a destructor defined.

```
//does not compile - structs can't have destructors
public ~Point()
{

}
```

When should we use structs?

- The type is logically small (for example it represents a single value, like int, double, or bool)
- It is small from the memory point of view (under 16 bytes)
- It is immutable (once an instance is created, it is not modified)
- It is commonly short-lived
- It is commonly embedded in other objects
- You want value type semantics (creating a copy on assignment, passing a parameter by copy)
- It will not be boxed frequently

If some of those criteria are not met, you should rather use a class.

Tip: other interview questions on this topic:

- "**What is the base type for structs?**"
  System.ValueType.

- "**Is it possible to inherit from a struct?**"
  No, all structs are sealed.

- "**How would you represent a point in the cartesian coordinate system?**"
  I would create a struct that has two float readonly properties - X and Y.

6. What is LINQ?

Brief summary: LINQ is a set of technologies that allow simple and efficient querying over different kinds of data.

Data can be stored in various types of containers - C# data structures like Lists or arrays, databases, XML documents, and many more. LINQ allows us to query such data in a uniform way, allowing the programmer to focus on what the program is supposed to do with the data, not on the technical details of accessing different data containers. We can use different LINQ providers, like **LINQ to SQL** that allows querying over SQL databases, or **LINQ to XML** that allows querying over XML documents. Each LINQ provider must implement IQueryProvider and IQueryable interfaces. We can create our own LINQ providers if we need to support querying over a new type of data container.

Let's try to use LINQ in practice. We will work on the following data:

```
var pirates = new List<Person>
{
    new Person("Anne", "Bonny", 1698),
    new Person("Charles", "Vane", 1680),
    new Person("Mary", "Read", 1690),
    new Person("Bartolomew", "Roberts", 1682),
};
```

LINQ allows us to use two alternative ways of querying over data:

- **Query syntax**:

```
var bornAfter1685QuerySyntax = from pirate in pirates
                               where pirate.YearOfBirth > 1685
                               select pirate;
```

- **Method syntax**:

```
var bornAfter1685MethodSyntax = pirates.Where(pirate => pirate.YearOfBirth > 1685);
```

Both expressions do the same thing - they filter out those pirates who were born after 1685. There isn't any distinct advantage of one over the other. Any query syntax can be transformed into method syntax.

From my personal experience, most developers prefer method syntax, as it is just pure C#, over query syntax which is kind of a new language. But it is up to you (or your team) which one you should use.

Let's see some of the most useful methods from the System.LINQ namespace. From now on I'm going to stick to method syntax.

```
IEnumerable<Person> orderedByLastName = pirates.OrderBy(pirate => pirate.LastName);

IEnumerable<int> onlyYearsOfBirth = pirates.Select(pirate => pirate.YearOfBirth);

double averageYearOfBirth = pirates.Average(pirate => pirate.YearOfBirth);

bool isAnyPirateBornBefore1650 = pirates.Any(pirate => pirate.YearOfBirth < 1650);

bool areAllPiratesBornAfter1650 = pirates.All(pirate => pirate.YearOfBirth > 1650);

IEnumerable<Person> piratesWithLastNameStartingWithR = pirates.Where(
    pirate => pirate.LastName.StartsWith("R"));

Person firstPirateByAlphabet = pirates.OrderBy(pirate => pirate.LastName).First();

IEnumerable<Person> piratesFromYoungestToOldest = pirates.OrderBy(
    pirate => pirate.YearOfBirth).Reverse();
```

As you can see, those expressions are really easy to read and understand. LINQ is widely considered an amazing library with intuitive syntax that was well designed by its developers. If you are not familiar with LINQ, I highly recommend you to start learning it, as it may be one of the most powerful tools .NET developers use.

Tip: other interview questions on this topic:

- "**What are the benefits of using LINQ?**"
  It offers a common syntax for querying any type of data source. It provides a simple yet powerful way of manipulating data. LINQ methods are chainable, which means you can have a single expression with multiple LINQ methods. It allows writing cohesive, readable, and flexible code.

- "**What is a LINQ provider?**"
  L*INQ provider is any class that implements* *IQueryProvider* *and* *IQueryable* *interfaces. LINQ providers are used for querying over a particular source of data. Examples of LINQ providers may be LINQ to SQL or LINQ to XML.*

7. What are extension methods?

Brief summary: An extension method is a method defined outside a class, that can be called upon this class's objects as a regular member method. Extension methods allow you to add new functionality to a class without modifying it.

Extension methods allow us to add new functionality to a class without actually modifying it, or without adding a new class inheriting from it. We define them in a special static class, and because of that, you don't need to recompile the original class. Even if they are defined in a separate class, you can still call them as regular member methods of the extended class.

Let's see this in practice. Imagine your code operates on long, multiline strings. You need a method that, given a string, counts a number of lines. Unfortunately, there is no such method in the built-in String class. We must create our own method.

```
var multilineString = @"Said the Duck to the Kangaroo,
                        Good gracious! how you hop
                        Over the fields, and the water too,
                        As if you never would stop!
                        My life is a bore in this nasty pond;
                        And I long to go out in the world beyond:
                        I wish I could hop like you,
                        Said the Duck to the Kangaroo.";

var numberOfLines = GetNumberOfLines(multilineString);
```

Something like that. But where to define the new GetNumberOfLines method? Ideally, it would belong to some static class aggregating all string operations. Let's do it:

```
public static class StringOperations
{
    public static int GetNumberOfLines(string multilineString)
    {
        return multilineString.Split("\n").Length;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var multilineString = @"Said the Duck to the Kangaroo,
                                Good gracious! how you hop
                                Over the fields, and the water too,
                                As if you never would stop!
                                My life is a bore in this nasty pond;
                                And I long to go out in the world beyond:
                                I wish I could hop like you,
                                Said the Duck to the Kangaroo.";

        var numberOfLines = StringOperations.GetNumberOfLines(multilineString);
```

All right - this code works, but it seems a bit clumsy. Over time, more and more methods could join the StringOperations class, and each of them would have to be called with StringOperations.MethodName syntax. Wouldn't it be better if we could simply call "multilineString.NumberOfLines()"? Unfortunately, we can't modify the String class. But we can use extension methods, and that's exactly their purpose. Let's do it:

```
public static class StringExtensions
{
    public static int NumberOfLines(this string input)
    {
        return input.Split("\n").Length;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var multilineString = @"Said the Duck to the Kangaroo,
                                Good gracious! how you hop
                                Over the fields, and the water too,
                                As if you never would stop!
                                My life is a bore in this nasty pond;
                                And I long to go out in the world beyond:
                                I wish I could hop like you,
                                Said the Duck to the Kangaroo.";

        var numberOfLines = multilineString.NumberOfLines();
```

Now the NumberOfLines method is an extension method for the String class. Please note "this" before the parameter type. We need to use the "this" keyword in order to create an extension method. It also must be static and must belong to a static, non-generic class. Notice how we call this method now - exactly like it belonged to the String class.

I also renamed StringOperations to StringExtensions as this is the more conventional way of naming the classes containing extension methods.

As you can see, the extension method is an easy way to add functionality to a class without modifying it. It's most commonly used with external classes (not defined in our project, so built-in classes or classes from external libraries) but we can also use it with our own classes:

```
public class Duck
{
}

public static class DuckExtensions
{
    public static string Quack(this Duck duck)
    {
        return "Quack, quack, I'm a duck";
    }
}
```

If the class already contains a method with the same signature as the extension method, the member method will be called and the extension method will be ignored:

```
public class Duck
{
    public string Quack()
    {
        return "(Not an extension method) Quack, quack, I'm a duck";
    }
}

public static class DuckExtensions
{
    public static string Quack(this Duck duck)
    {
        return "(This is an extension method) Quack, quack, I'm a duck";
    }
}
```

Please note that both of those methods would be called upon a duck object with "duck.Quack()". In this case, calling the Quack method on a duck object will result in:

```
(Not an extension method) Quack, quack, I'm a duck
```

As you can see the non-extension method has been called, as it has a priority over extension methods.

To summarize: extension methods allow us to add functionality to a class without modifying it, which is especially useful when we use an external class and we don't have access to its source code. Extension methods are used very often and you might even have used them without knowing it - for example, whenever calling LINQ's methods OrderBy, GroupBy, or Where, you are actually calling extension methods for IEnumerable. Here is a snippet from LINQ source code to prove it:

```
public static partial class Enumerable
    {
        public static IEnumerable<TSource> Where<TSource>(
            this IEnumerable<TSource> source, Func<TSource, bool> predicate)
            {
```

Requirements:

- The class in which the extension methods are defined must be static and non-generic
- The extension method must take the object of the extended class as a first parameter, with "this" modifier proceeding this parameter

Tip: other interview questions on this topic:

- "**How would you add new functionality to an existing class, without modifying this class?**"
  By using extension methods.

- "**What will happen if you call a member method that has the same signature as the existing extension method?**"
  The member method has priority and it will be the one to be called. The extension method will not be called.

8. What is IEnumerable?

Brief summary: IEnumerable is an interface that enables iterating over a collection with a foreach loop.

IEnumerable is an interface that enables iterating over a collection with a **foreach** loop:

```
var words = new[] { "a", "little", "duck" };

foreach(var word in words)
{
    Console.WriteLine(word);
}
```

The above code works because the array we used implements IEnumerable interface. If it hadn't, the code would not compile:

```
foreach(var word in customCollection)
{
    Console.WriteLine(w
}
```

(local variable) CustomCollection customCollection

CS1579: foreach statement cannot operate on variables of type 'CustomCollection' because 'CustomCollection' does not contain a public instance or extension definition for 'GetEnumerator'

There is also a generic counterpart of IEnumerable interface - IEnumerable<T> that can hold any type of item. The important thing to note here is that LINQ provides a lot of extension methods for operating on IEnumerable<T> objects. Those methods can be used for ordering, filtering, or aggregating the data, and many more. For simplicity, we will focus on non-generic IEnumerable.

Another important feature of IEnumerable interface is that it provides **read-only** access to a collection - it doesn't expose any methods that allow modification of the collection. When creating a method returning a collection it is recommended to return it as a read-only collection unless we really want to let the user of this method modify the collection.

All right. Let's take a closer look at this interface. It contains a single method:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

As you can see, GetEnumerator simply returns IEnumerator:

```
public interface IEnumerator
{
    bool MoveNext();
    Object Current { get; }
    void Reset();
}
```

An **enumerator** is a mechanism that allows iterating over collection elements. You can imagine it as a kind of a pointer that points to a "current" element in the collection (that element can be accessed with the Current property). To iterate forward, the MoveNext method is executed. It moves the pointer one element up, assuming we are not already at the end of the collection. If we are, MoveNext will return false. There is also the Reset method, which moves the pointer back to the beginning of the collection.

We will implement our own type supporting enumeration in a second, but first, let's see how foreach loop is interpreted by .NET. Let's see the code with a foreach loop again:

```
var words = new[] { "a", "little", "duck" };

foreach (var word in words)
{
    Console.WriteLine(word);
}
```

.NET actually translates this code to something like this:

```
IEnumerator wordsEnumerator = words.GetEnumerator();
string word;
while (wordsEnumerator.MoveNext())
{
    word = (string)wordsEnumerator.Current;
    Console.WriteLine(word);
}
```

Now it is obvious why we need to implement IEnumerable to support usage of foreach loops - if we didn't, the above code would simply not compile.

All right, let's create our own collection that will hold a group of strings, and that will support being iterated over by foreach loop. To do so, we need to create a class that implements IEnumerable interface:

```
class WordsCollection : System.Collections.IEnumerable
{
    private string[] _words;

    public WordsCollection(string[] words)
    {
        _words = words;
    }

    public IEnumerator GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Now, we will have to create the WordsEnumerator class that will implement IEnumerator interface:

```
class WordsCollection : System.Collections.IEnumerable
{
    private string[] _words;

    public WordsCollection(string[] words)
    {
        _words = words;
    }

    public IEnumerator GetEnumerator()
    {
        return new WordsIterator(_words);
    }
}

class WordsIterator : IEnumerator
{
    private string[] _words;

    public WordsIterator(string[] words)
    {
        _words = words;
    }

    public object Current => throw new NotImplementedException();

    public bool MoveNext()
    {
        throw new NotImplementedException();
    }

    public void Reset()
    {
        throw new NotImplementedException();
    }
}
```

As you remember, we said that an enumerator is kind of a pointer to a current element. We must store the element this pointer points to as a private field. We will use an integer named _position that will refer to the index in the _words array:

```
class WordsIterator : IEnumerator
{
    private string[] _words;
    private int _position = -1;
```

You may be surprised why we used -1, not 0. To understand it, let's see again how .NET interprets the foreach loop:

```
IEnumerator wordsEnumerator = words.GetEnumerator();
string word;
while (wordsEnumerator.MoveNext())
{
    word = (string)wordsEnumerator.Current;
    Console.WriteLine(word);
}
```

As you can see, the MoveNext method is called **first**, and **then** we access the Current element. The MoveNext method will be incrementing the value of _position field by one. If we initialized the _position field with 0, it would have a value of 1 after the MoveNext method was called. And because of that, the "first" element returned by the Current property would actually be the **second** in the collection. This is why we set the position to -1, so after the first use of MoveNext it is 0.

All right, now it is pretty obvious what the Reset method will do:

```
public void Reset()
{
    _position = -1;
}
```

MoveNext will simply increment the _position by one. Please note that MoveNext returns a bool. This bool should be true if we successfully advanced to the next element, and false if we passed the end of the collection. Let's implement that:

```
public bool MoveNext()
{
    _position++;
    return _position < _words.Length;
}
```

Finally, let's implement the Current property:

```
public object Current
{
    get
    {
        try
        {
            return _words[_position];
        }
        catch (IndexOutOfRangeException)
        {
            throw new InvalidOperationException("Collection end reached");
        }
    }
}
```

That's it! Now it works correctly with the foreach loop:

```
var wordsCollection = new WordsCollection(words);
foreach (var word in wordsCollection)
{
    Console.WriteLine(word);
}
```

All right, let's summarize what we learned about IEnumerable:

- It allows looping over collections with a foreach loop
- It works with LINQ query expressions
- It allows read-only access to a collection
- To implement it, we must create an enumerator class, that will provide MoveNext and Reset methods, as well as Current property

Tip: other interview questions on this topic:

- "**What is an enumerator?**"
  An enumerator is a mechanism that allows iterating over collection elements. It's a kind of a pointer that points to a "current" element in the collection.

- "**Assuming a method returns a collection of some kind, how to best express your intent if you don't want the user to modify this collection?**"
  By returning it as IEnumerable or another readonly collection type.

9. What is the Garbage Collector?

Brief summary: The Garbage Collector is a mechanism that manages the memory used by the application. If an object is no longer used, the Garbage Collector will free the memory it occupies. The Garbage Collector is also responsible for defragmenting the application's memory.

The **Garbage Collector** is the CLR's (Common Language Runtime) mechanism that manages the memory used by the application.

In some languages (like, for example, C) it was a developer's responsibility to allocate and deallocate the memory needed by objects. It created a lot of noise in the code and was a common source of errors.

In many modern languages, including C#, memory management is built-in and we as developers do not need to worry about it (most of the time).

As soon as the application starts it begins to create new objects, and those objects need to be stored in memory. **C#'s objects live in two areas of memory: the stack and the heap.**

The stack holds value types, so types like ints, bools, floats, and structs. Memory allocated for those objects is automatically freed when the control reaches the end of the scope those objects live in:

```
if(args != null)
{
    int a = 5;

    //end of the scope for the variable "a" - it will be cleaned up right away
}
```

When variable a was created, it was put on the stack. When the control reaches the end of the scope this variable lives in - so the "if" clause - this variable is removed from the stack and its memory is freed. This is NOT done by the Garbage Collector, but by the CLR itself.

Garbage Collector manages objects that live on **the heap**, so the reference types (strings, Lists, arrays, and all other objects that are defined as classes). Garbage

Collector determines if there are any existing references to the object - if not, it decides this object is no longer used and frees the memory used by this object.

```
if(args != null)
{
    string text = " abc";

    //end of the scope for the variable "text" - Garbage Collector will clean it up, but not immediately
}
```

The important thing here is that Garbage Collector **will not clean the memory immediately** - it is important to know that we can't deterministically say when exactly will that happen (so if you **must** clean up some resources at once, don't leave that to Garbage Collector - implement IDisposable interface and use the Dispose method). There is a way to force collection of the memory by the Garbage Collector by using **GC.Collect** method, but it shouldn't be the default solution if you want to have the memory immediately freed.

So when does the Garbage Collector decide to clean up memory?

- When the system has low physical memory (the operating system notifies about that)
- When the memory that's used by allocated objects on the heap surpasses a given threshold. This threshold is continuously adjusted as the process runs
- When the GC.Collect method is called

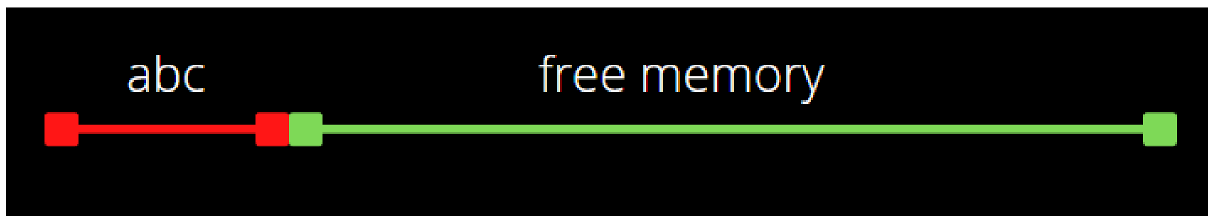
So basically, the Garbage Collector does not start to work unless there is a need for it.

The important thing to understand is that **Garbage Collector runs on its own, separate thread**, and as this happens all other threads are being **stopped** until Garbage Collector finishes its work. This might obviously cause performance issues. For example, consider a video game created in C# (in case you don't know, one of the most popular video games development environments is Unity, where you can create games in C#). If plenty of short-lived objects would be created every second, Garbage Collector would have a lot of work to do, and it would often "freeze" the game for a fraction of a second to do its work. The experience for the player would not be perfect. In such cases, it is recommended to avoid frequent triggering of the Garbage Collector work (there are some techniques to do it, for example by using a pool of objects that are being reused, instead of frequent creation and destruction of short-lived objects). The important thing to remember is that Garbage Collector freezes all other threads for the time of the collection.

After Garbage Collector finishes freeing the memory, it also executes **memory defragmentation**.

To understand what it is, imagine a computer's memory as a long array of bits (that's actually pretty close to the real thing). Let's say we create a variable of type string, and value of "abc". It consists of 3 characters, and each char in C# is 2 bytes. One byte is 8 bits. So we need 3 * 2 * 8 bits of memory, so 48 bits.

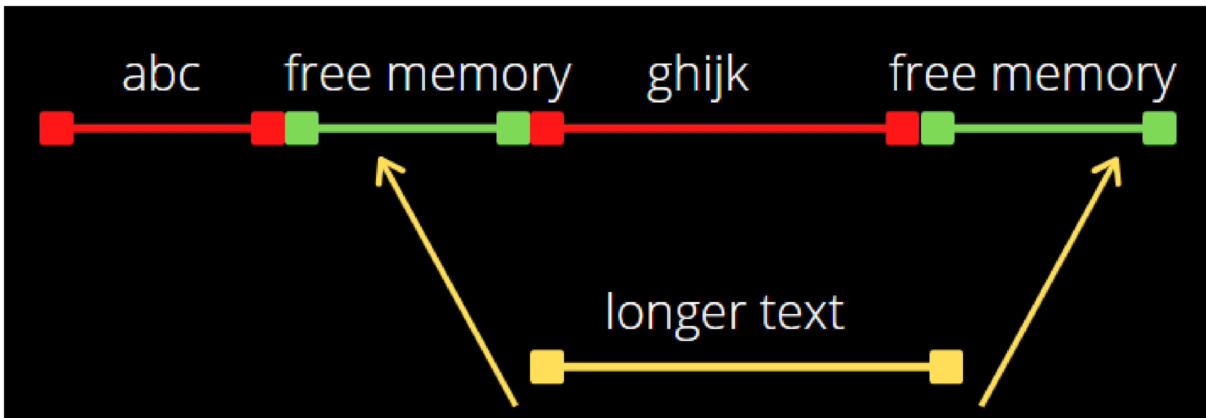
The Garbage Collector finds a 48-bits long part of empty memory in this long array of memory assigned to the process in which our application runs, and reserves it for the "abc" string. Now the memory looks like this:



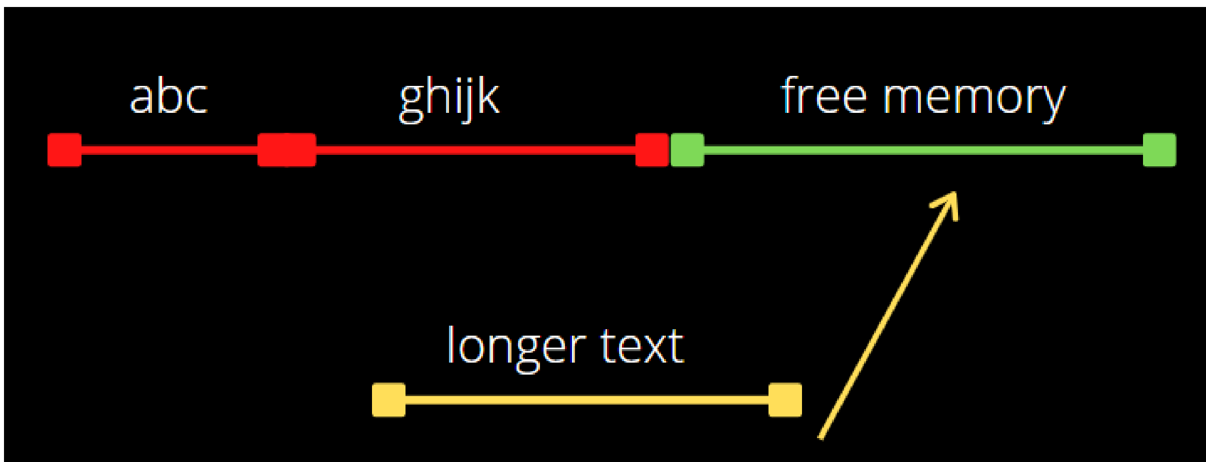
Let's say some other strings had been created:

Now, let's assume the "def" string is no longer used and its memory is freed by the Garbage Collector.

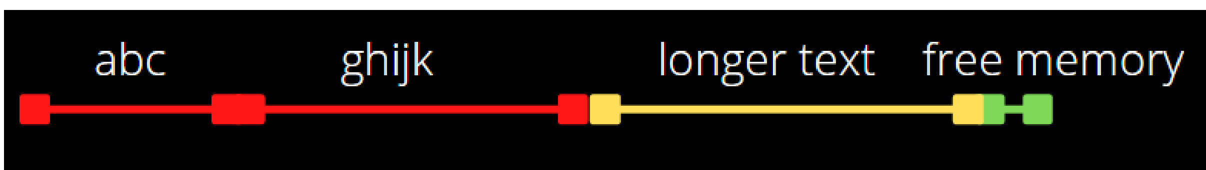
Now, if we want to add some new, long string to the memory, we might actually not find a place for it to fit. We do have two blocks of free memory, but none of them is long enough.



What the Garbage Collector needs to do, is to move some pieces of memory and make them contiguous, thus creating a bigger free block of memory:



Now the new, long string can easily fit in:



This process is called **memory defragmentation**.

So as we can see, the Garbage Collector is a pretty smart tool. It makes our work much easier, but it doesn't mean we can forget completely about memory issues. Let's consider **memory leaks**. A memory leak is a situation when some piece of memory is not being cleaned up, even if the object using it is no longer in use. It is important to understand that **Garbage Collector does not give us 100% protection from memory leaks**.

One of the most common sources of memory leaks in .NET is related to event handlers. Events are topics beyond Junior level, but let's for now just assume they are used to handle some specific situations (like clicking on a button) by "attaching" a method that will be called to the event "handler".

Imagine you have a window-based application. There is the main window, and when a button in this window is clicked, a child window opens. Here is a very simplified code for that:

```
public class MainWindow
{
    public event EventHandler<EventArgs> mainWindowEventHandler;

    private void buttonClick_OpenChildWindow(object sender, EventArgs e)
    {
        var childWindow = new ChildWindow(this);
        childWindow.Show();
    }
}
```

```
public class ChildWindow
{
    private MainWindow _mainApplicationWindow;

    public ChildWindow(MainWindow mainApplicationWindow)
    {
        _mainApplicationWindow = mainApplicationWindow;
        _mainApplicationWindow.mainWindowEventHandler += HandleEventFromMainWindowInChildWindow;
    }

    private void HandleEventFromMainWindowInChildWindow(object sender, EventArgs e)
    {
    }

    internal void Show()
    {
        //opening a window
    }
}
```

Note what happens: when MainWindow's button is clicked, a ChildWindow is being created. We want something to happen in the ChildWindow when an event happens in the MainWindow. In its constructor, the ChildWindow subscribes to the MainWindow's event handler. This is a very important thing: by subscribing to the event handler, a reference from MainWindow to ChildWindow is created. It is necessary because if the event will be triggered in the MainWindow, the ChildWindow must be notified about that.
Now, let's say the user closes the ChildWindow. It would seem like the object of ChildWindow would be cleaned up by the Garbage Collector. But it will not be!

Because there is still a "hidden" reference from the MainWindow to the ChildWindow. Garbage Collector sees this reference and decides that the ChildWindow object is still in use. Now, imagine what will happen if the user keeps opening and closing the ChildWindow. New objects will be created, but they will never be freed. Finally, we will run out of memory.

In this case, the solution is to unsubscribe from the event handler - for example on form closing:

```
void OnWindowClosing()
{
    _mainApplicationWindow.mainWindowEventHandler -= HandleEventFromMainWindowInChildWindow;
}
```

This was just an example of how we can encounter memory leaks that GarbageCollector is unable to protect us against. You should always be considerate of the memory and remember that Garbage Collector is just a tool, even if pretty clever.

There are many low-level details on how exactly Garbage Collector works, but they are beyond Junior level. The most important things you need to remember about Garbage Collector are:

- It manages the memory of the application
- It frees up memory allocated for objects that are no longer referenced
- It's hard to say when exactly it will collect the memory - it has its' own mechanism that depends on how much memory is being used
- It defragments the memory
- It does not guarantee protection from memory leaks

Tip: other interview questions on this topic:

- "**Would the Garbage Collector free up the memory occupied by an integer?**"
  No, since an integer is a value type and Garbage Collector only handles memory occupied by reference types. Value types are stored on the stack which has its own mechanism for freeing the memory.

- "**How to manually trigger the Garbage Collector memory collection?**"
  By calling GC.Collect method from System namespace.

- "**What is memory fragmentation and defragmentation?**"
  When pieces of memory are allocated and freed, the memory becomes defragmented, which means free memory is in small pieces rather than in one long piece. This is called memory fragmentation. If there is a need to

put a big object into memory, it might be impossible to find a block of free memory that is long enough. The process of moving the pieces of allocated memory so they stick together, to create a big chunk of free memory is called defragmentation.

- "**What are memory leaks? Does the Garbage Collector guarantee protection from them?**"
  Memory leaks happen when memory is not freed even if an object is no longer used. No, GC does not guarantee protection from them.

10. What are nullable types?

Brief summary: Nullable type is any type that can be assigned a value of null. Nullable<T> struct is a wrapper for a value type allowing assigning null to the variable of this type. For example, we can't assign null to an integer, but we can to a variable of type Nullable<int>.

Nullable type is any type that can be assigned a value of **null**. In C#, all reference types are nullable by default, and value types are non-nullable by default. That means, we can't assign null to an integer variable, but it is fine to assign null to, for example, a List variable, since List is a reference type and int is a value type.

Null represents "lack of value", or "missing value". There are many business cases when such special value is needed, even for value types. For example, imagine you have a collection of people, and each Person has a Height property. If we decide that Height is an integer (so a non-nullable value type) we might encounter a problem - what if we don't know some person's height? We could make a risky assumption that in this case, we would use some special value, like 0 or -1. But then, what if we want to calculate the average height of all people in this collection? Having people of height 160, -1, 185, -1, 170, we would end up with an average value of 102,6, which obviously doesn't make much sense. It's better to represent height as a nullable integer type, and only include non-null values in the average height calculation.

In C#, we can declare value type as nullable with the "?" operator:

```
public int? Height;
```

Using "?" operator is just a short syntax for:

```
public Nullable<int> Height;
```

We can assign null to a nullable:

```
int? nullableNumber = null;
```

Please note that Nullable<T> is a **struct**, which means Nullable is a value type (as all structs are). As we said, we can't assign null to a value type - so how is it possible that we assign a null to a Nullable<T>? Actually, it's a trick of C#'s compiler. Behind the scenes, it interprets the above code as:

```
Nullable<int> nullableNumber = new Nullable<int>();
```

So as you can see, it's not *really* assigning a null.

Nullable exposes useful properties like HasValue, which indicates whether the value is null or not, or Value, which unpacks the internal value (so for the nullable type int? Value property will return an int).

Tip: other interview questions on this topic:

- "**Can a value type be assigned null?**" *No, it can't. It must be wrapped in the Nullable<T> struct first if we want to make it nullable.*

- "**Is it possible to have a variable of type Nullable<T> where T is a reference type? For example, Nullable<string>?**"
No, the Nullable struct has a type constraint on T that requires the T to be a value type. Providing a reference type as T will result in a compilation error.

11. What are generics?

Brief summary: Generic classes or methods are parametrized by type - like, for example, a List<T> that can store any type of elements.

Generics allow us to create classes or methods that are **parametrized by type**.

Let's consider a simple example of a List. Without generics, we would need to have a separate class for all data types we want to store in a list:

```
public class IntegersList
{
    public void Add(int item)
    {
        //...
    }
}

public class StringsList
{
    public void Add(string item)
    {
        //...
    }
}

public class DoublesList
{
    public void Add(double item)
    {
        //...
    }
}
```

That would be absolutely awful. Not only would we need to copy this code each time we want a new type to be stored in a list, but also if we decided a change must be made in the List classes (for example, if we found a bug) we would need to modify each and every one of them.

This is where generics come in handy - we can create a single class, that will be parameterized by a type:

```
public class List<T>
{
    public void Add(T item)
    {
        //...
    }
}
```

Now, we can store **any** type in the List.

Of course, C# already provides a very good implementation of a List<T> - it can be found in System.Collections.Generic namespace.

We can parametrize a class or a method with more than one type parameter. An example of such a class is a Dictionary. Dictionary is a data structure that holds pairs of keys and values. We can use any types as keys and values:

```
var currencies = new Dictionary<string, string>
{
    ["USA"] = "USD",
    ["Great Britain"] = "GBP"
};

var yearsOfBirth = new Dictionary<string, int>
{
    ["John Smith"] = 1980,
    ["Monica Smith"] = 1983
};
```

In such a case, when defining a generic type, we simply provide two type parameters. Let's see how the C#'s dictionary is defined:

```
public class Dictionary<TKey,TValue>: IDictionary<TKey,TValue>,
```

We sometimes need to put some kind of constraint on a generic type. For example, the Nullable<T> allows only value types to be used as T parameter because non-value types are nullable by definition. To put a constraint on a type we use a "**where**" keyword. Let's see some of the basic type constraints in C#:

Class - the type must be a reference type:

```
public class OnlyReferenceTypes<T> where T : class
{

}
```

Struct - the type must be a value type:

```
public class OnlyValueTypes<T> where T : struct
{

}
```

New() - the type must provide a public parameterless constructor:

```
public class OnlyTypesWithParameterlessConstructor<T> where T : new()
{

}
```

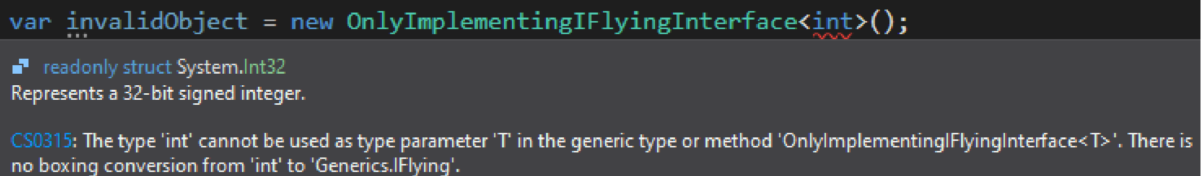
The type must be derived from a base class:

```
public class BaseClass
{

}

public class OnlyDerivedFromBaseClass<T> where T : BaseClass
{

}
```

The type must implement an interface:

```
public interface IFlying
{

}

public class OnlyImplementingIFlyingInterface<T> where T : IFlying
{

}
```

If the type does not meet the criteria defined in the constraint, a compilation error appears:



Tip: other interview questions on this topic:

- "**What are type constraints?**" Type constraints allow limiting the usage of a generic type only to the types that meet specific criteria. For example, we may require the type to be a value type, or require that this type provides a public parameterless constructor.

- "**What is the "where" keyword used for?**" It's used to define type constraints. Also, it is used for filtering when using LINQ.

- "**What are the benefits of using generics?**" They allow us to reduce code duplication by creating a single class that can work with any type. Reducing code duplication makes the code easier to maintain and less error-prone.

12. What is the difference between an interface and an abstract class?

Brief summary: An interface defines what set of operations will be provided by any class implementing it - it does not provide any implementation on its own. An abstract class is like a general blueprint for derived classes. It may provide implementations of methods, contain fields, etc.

Before we delve into more technical details, let's try to understand the difference between interfaces and abstract classes at the conceptual level:

- An interface is an abstraction over **behavior**. It defines what an object can **do**. When you have a group of objects and they share similar behavior, they might have a common **interface**. For example, a bird, a kite, and a plane fly - so it makes sense for all of them to implement the IFlyable interface. When you are given an object implementing IFlyable interface, you might not be sure what that **is** - but you'll know it is **able** to fly. When you try to find what some objects have in common and a **verb** comes to mind, it means you probably want to use an interface.
- An abstract class is an abstraction over alikeness. It defines what an object **is**. When you have a group of objects, and they all belong to some general category of things, they might inherit from the same abstract class. For example, a bird, a snake and a dog all are animals - so it makes sense for them to inherit from abstract class Animal. When you are given an object that inherits from the Animal abstract class, you might not be sure how it **behaves** - but you know that it **is** some kind of animal. When you try to find what some objects have in common and a **noun** comes to mind, it means you probably want to use an abstract class.

Let's look more technically on what an interface and an abstract class are:

- An interface is a set of **definitions** of methods - it **does not** provide any implementation (at least in 99% of the cases - see the note about interfaces change in C# 8.0 at the end of this lecture). It specifies a **contract** that an implementing method will have to fulfill. When you implement an interface in your class, it means you declare that this class will provide all the methods from this interface. For example, one of the most commonly used interfaces in C# is ICollection - an interface that defines a set of methods related to working on collections - methods like Add, Contains, Remove, Clear, etc. ICollection only defines what methods a collection must provide, but they are not implemented in the interface itself. The concrete classes

implementing this interface - for example List - provide the implementations.

```
interface IFlyable
{
    void Fly(); //no "public", no method body
}

class Bird : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using fuel of grain and worms.");
    }
}

class Drone : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using energy stored in battery.");
    }
}
```

- An abstract class is a type that is too - *well* - abstract for the actual instances of it to exist. It represents some general category of things. It can have method implementations, but it can also contain abstract methods - methods with no bodies, that will have to be implemented in the inheriting classes. As in the example before, you can imagine an Animal abstract class. You can't have an object of type Animal - it's always some specific kind of animal, like a dog or a horse. Animal is just an abstraction over a whole category of creatures a bit similar to each other (and not similar at all to plants or fungi).

```
abstract class Animal
{
    public abstract void Move();
}

abstract class Mammal : Animal //abstract class inheriting from abstract class
{
    public void ProduceMilk()
    {
        Console.WriteLine("Producing milk to feed its young");
    }
}

class Snake : Animal
{
    public override void Move()
    {
        Console.WriteLine("Slithering on belly");
    }
}

class Dog : Mammal
{
    public override void Move()
    {
        Console.WriteLine("Running using four legs");
    }
}

class Cat : Mammal //does not compile - MUST provide implementation of the Move method
{
}
```

As you can see you can't create instances of an abstract class:

```
var animal = new Animal(); //can't create an instance of an abstract class
var mammal = new Mammal(); //can't create an instance of an abstract class

var dog = new Dog();
var snake = new Snake();
```

To summarize - the differences between an interface and abstract class are:

- Interface can't provide any implementation of the methods, an abstract class can (but doesn't have to if the method is abstract)
- All interface methods are by default public and they can't have any other access modifier specified. They can not be sealed or static (that wouldn't make sense because sealed or static methods can't be overridden). They also can't declare methods as abstract or virtual (that would be redundant because as methods with no bodies, meant to be implemented in classes implementing the interface, they are already kind of abstract and virtual).

- An interface can only contain methods or properties definitions - it can not have fields or constructors, while an abstract class can
- A class can implement multiple interfaces, but it can only inherit from one abstract class

	Interface	Abstract class
Abstraction over	behavior	alikeness
Defines what an object ...	...can do	...is
Group of objects share...	...behavior	...general category of things
Example	bird, kite and plane **can** fly	bird, snake and dog **are** animals
Not sure what it...	...is	...is able to do
Sure what it...	...is able to do	...is
Part of speech	verb	noun

Before we end this lecture I would like to mention one thing - **starting with C# 8.0 interfaces can have methods with bodies**. I decided to skip it in this course, as it exceeds junior level. If you are interested, please see https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/default-interface-methods-versions

Tip: other interview questions on this topic:

- "**Why can't you specify the accessibility modifier for a method defined in the interface?**"
  The point of creating an interface is to specify the public contract that a class implementing it will expose. Modifiers other than "public" would not make sense. Because of that, the "public" modifier is the default and we don't need to specify it explicitly.

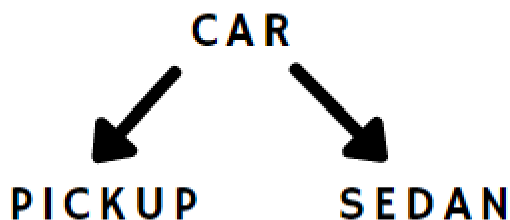
- "**Why can't we create instances of abstract classes?**"
  Because an abstract class may have abstract methods which do not contain a body. What would happen if such a method was called? It doesn't have a body so the behavior would be undefined.

13. What is the Bridge design pattern?

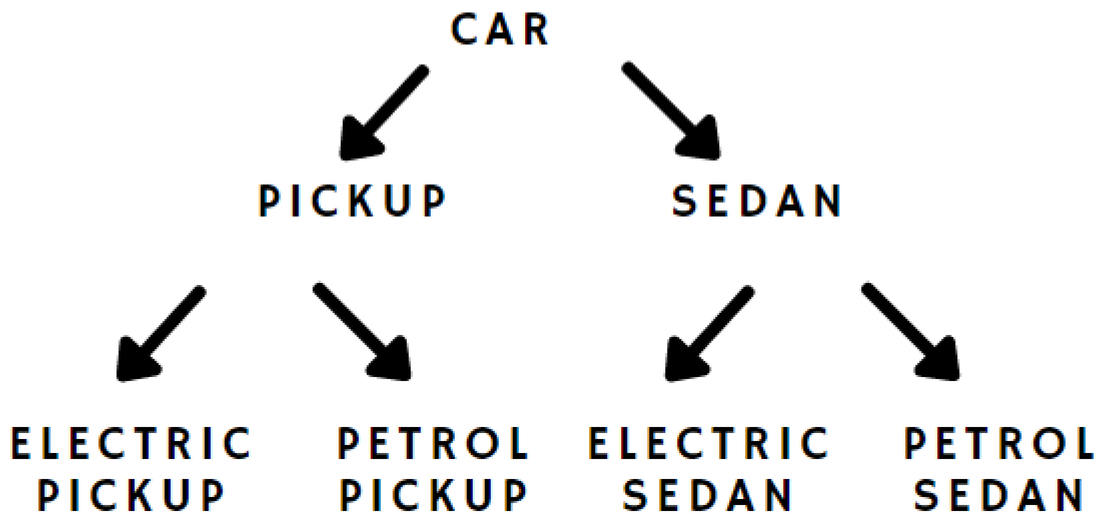
Brief summary: The Bridge design pattern allows us to split an inheritance hierarchy into a set of hierarchies. It is the implementation of the "composition over inheritance" principle.

The **Bridge** design pattern allows us to split an inheritance hierarchy into a set of hierarchies, which can then be developed separately from each other. It is the implementation of the "composition over inheritance" principle, which states that it is better to introduce new features to a class by extending what this class **contains**, instead of extending the **inheritance hierarchy**.

This all sounds a bit complicated, but let's consider a really simple example: we have a base class Car with two inheritors: Pickup and Sedan.

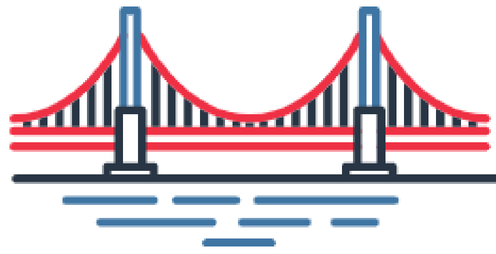


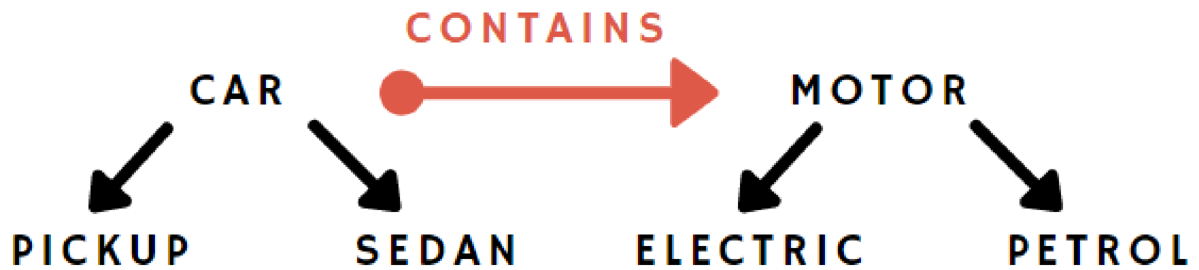
That looks very simple for now. But then the application you develop grows, and there is a need to distinguish electric cars from petrol cars. Let's see how this could affect the inheritance hierarchy:



The hierarchy grew a lot. We suddenly have **seven** classes instead of the **three** we had before. What if we are asked to add another trait that characterizes a car, like manual and automatic gear? We would have to create classes like ManualElectricPickup and AutomaticElectricPickup, and then ManualPetrolPickup and AutomaticPetrolPickup, and so on, and so forth. That would be an explosion of classes, completely unmanageable.

What is the alternative? Well, we can use the **Bridge design pattern**. Instead of expressing a trait of a Car as another layer in the inheritance hierarchy, we **split** the inheritance hierarchy in two. We simply add a new class - like Motor - and then add two inheritors - ElectricMotor and PetrolMotor. Then, the Car class will **contain** a Motor object within. Let's see it in a diagram:





Now, the Car and the Motor are separate entities. We can extend them to our needs, without affecting one another. Also, adding another entity to the picture (like ManualGear or AutomaticGear) is not a problem at all - we will simply add another class hierarchy that represents them.

Let's see this in the C# code. First, the classes representing cars:

```
class Car
{
    public Motor Motor { get; }
    public Gear Gear { get; }

    public Car(Motor motor, Gear gear)
    {
        Motor = motor;
        Gear = gear;
    }
}

class Pickup : Car
{
    public Pickup(Motor motor, Gear gear) : base(motor, gear)
    {
    }
}

class Sedan : Car
{
    public Sedan(Motor motor, Gear gear) : base(motor, gear)
    {
    }
}
```

Now, Gear and Motor:

```
class Motor { }
class ElectricMotor : Motor { }
class PetrolMotor : Motor { }

class Gear { }
class ManualGear : Gear { }
class AutomaticGear : Gear { }
```

Now it should be simple to create any car we want - for example, an electric pickup with manual gear or a petrol sedan with automatic gear:

```
var electricPickupWithManualGear = new Pickup(new ElectricMotor(), new ManualGear());
var petrolSedanWithAutomaticGear = new Sedan(new PetrolMotor(), new AutomaticGear());
```

Thanks to the Bridge pattern, our inheritance hierarchy is kept simple and clean. We won't have any problem with adding new characteristics to the Car class. We can work on each of the families of classes without affecting the other.

Tip: other interview questions on this topic:

- "**What is "composition over inheritance"?**" *It is a principle that states that it is better to design polymorphic and reusable code by using composition rather than inheritance.*

`

14. What is the Single Responsibility Principle?

Brief summary: "S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should be responsible for only one thing. Sometimes the alternative definition is used: that a class should have no more than one reason to change.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should **be responsible for only one thing**. Sometimes the alternative definition is used: that a class **should have no more than one reason to change**.

`

Let's consider the following class:

```
public class PeopleInformationPrinter
{
    private readonly string _connectionString;
    private readonly string _resultFilePath;

    public PeopleInformationPrinter(string connectionString, string resultFilePath)
    {
        _connectionString = connectionString;
        _resultFilePath = resultFilePath;
    }

    public void Print()
    {
        var people = ReadFromDatabase();
        var text = BuildText(people);
        File.WriteAllText(_resultFilePath, text);
    }
    private IEnumerable<Person> ReadFromDatabase()
    {
        var people = new List<Person>();
        using (var connection = new SqlConnection(_connectionString))
        {
            using (var command = new SqlCommand("select * from People", connection))
            {
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        people.Add(
                            new Person(
                                reader["Name"] as string,
                                reader["LastName"] as string,
                                (int)reader["Username"]));
                    }
                }
            }
        }
        return people;
    }

    private string BuildText(IEnumerable<Person> people)
    {
        return string.Join("\n", people.Select(person => person.ToString()));
    }
}
```

Is this class responsible for only one thing? Of course not!

- It is responsible for connecting to the database
- It is responsible for transforming a list of people into text
- It is responsible for writing this text to a text file

`

What reasons to change it may have?

- It may need to change if the data source will change - for example, if we decide to read the information about people from the Excel file rather than a database, or if we switch to another database engine
- It may need to change if the formatting of the text will change
- It may need to change if the way of writing the data will change - for example, we decide to write to a PDF instead of a text file

So this class definitely breaks the Single Responsibility Principle. Let's refactor it. If one class needs to be responsible for one thing only, we probably need some more classes:

1. A class that reads the list of people from a database:

```
public class DatabaseReader : IReader<Person>
{
    private readonly string _connectionString;
    public DatabaseReader(string connectionString)
    {
        _connectionString = connectionString;
    }

    public IEnumerable<Person> Read()
    {
        var people = new List<Person>();
        using (var connection = new SqlConnection(_connectionString))
        {
            using (var command = new SqlCommand("select * from People", connection))
            {
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        people.Add(
                            new Person(
                                reader["Name"] as string,
                                reader["LastName"] as string,
                                (int)reader["Username"]));
                    }
                }
            }
        }
        return people;
    }
}
```

This class is responsible only for reading the list of people from the database. The only reason to change it could have is if the way of reading would change - for example, the "Name" column in the database would be changed to "FirstName".

`

2. The class that builds a text from a list of people:

```
public class PeopleTextFormatter : IPeopleTextFormatter
{
    public string BuildText(IEnumerable<Person> people)
    {
        return string.Join("\n", people.Select(person => person.ToString()));
    }
}
```

Again, this class would only have one reason to change - if the way how the text is formatted would be changed, for example if we decided to use ";" instead of a new line as a separator between a particular person's information.

3. The class that writes to a file:

```
public class TextWriter : IWriter
{
    private readonly string _filePath;

    public TextWriter(string resultFilePath)
    {
        _filePath = resultFilePath;
    }

    public void Write(string text)
    {
        File.WriteAllText(_filePath, text);
    }
}
```

The same thing here - this class has only one reason to change, for example if we decided to write to another file format than a text file (in this case it might be a better idea to just create a new implementation of the IWriter interface).

All those classes are small, cohesive, and clean. It is much easier to read them and understand what exactly they do.

Now all that's left is to use those classes in the PeopleInformationPrinter class:

`

```
public class PeopleInformationPrinter
{
    private readonly IReader<Person> _reader;
    private readonly IPeopleTextFormatter _peopleTextFormatter;
    private readonly IWriter _writer;

    public void Print()
    {
        var people = _reader.Read();
        var text = _peopleTextFormatter.BuildText(people);
        _writer.Write(text);
    }
}
```

Some may argue "but hey, this class still does three things! It still reads from the reader, it still asks the TextFormatter to build the text, and it still writes to a Writer!". Well, not exactly. This class **only** orchestrates work of other classes - in this case with IReader, IPeopleTextFormatter, and IWriter interfaces. They do the actual work - connecting to the database, writing to the file - and this class **only** works as their manager. It has only one reason to change - if the flow of this process changes, for example, if there is a new requirement to somehow filter the data after reading it from the database, but before sending it to the TextFormatter.

Why is the Single Responsibility Principle important and why should we care to follow it?

- A class responsible for one thing only is **smaller, more cohesive, and more readable**
- Such code is **reusable** - in the example above, the original class would not be likely to be used in any other context. After the refactoring it is easy to imagine plenty of other usages for the DatabaseReader or the TextWriter
- Such code is much **easier to maintain**, as it is much easier to introduce changes and fixes to a class that only does one thing
- Overall, the development speed will be faster and the number of bugs will be smaller

Let's summarize. "S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should be responsible for only one thing. Sometimes the alternative definition is used: that a class should have no more than one reason to change.

`

Tip: other interview questions on this topic:

- "**How to refactor a class that is known to be breaking the SRP?**"
  One should identify the different responsibilities and move each of them to separate classes. Then the interactions between those classes should be defined, ideally by one class depending on an interface that the other class implements.

`

15. What is the Open-Closed Principle?

Brief summary: "O" in the SOLID principles stands for Open-Closed Principle (sometimes referred to as the OCP). This principle states that modules, classes, and functions should be opened for extension, but closed for modification.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"O" in the SOLID principles stands for Open-Closed Principle (sometimes referred to as the OCP). This principle states that modules, classes, and functions should be **opened for extension, but closed for modification**. In other words - we should design the code in a way that if a change is required, we can implement it by adding new code instead of modifying the existing one.

It might be a bit hard to understand what exactly it means from a **practical** point of view, but don't worry - we will take a closer look at this principle in a moment. But first, let's understand what is the reasoning behind this principle, and why it is so important. Have you ever heard the expression "The only constant in life is change"? Those are the words of Ancient Greek philosopher Heraclitus, and they are surprisingly fitting to the modern problem of software development. When designing an application, we must always keep in mind that whatever the business requirements are at the moment, they are very likely to change in the future. When they do, we want to be able to:

- Introduce the changes quickly and easily
- Don't break any existing functionality

Following the Open-Closed Principle helps us to achieve it. According to this rule, the code should be:

- **Opened** - that means, it can be extended so new functionality can be added
- **Closed** - that means, we shouldn't be forced to modify the existing code to introduce this piece of functionality

Why is it so important to avoid modifications of the existing code?

- Firstly, it can lead to bugs. Before introducing a code change we have an application that **works**. Modifying a class can make it no longer true. Of course, we should have tests that ensure that everything works fine, but no tests are perfect.
- Secondly, changing the behavior of a class can surprise other developers - what if they need the class the way it was? It might not be such a big

`

> problem in small projects, but the bigger the project, the bigger the impact of such change might be (not to mention projects that are publicly accessible and can be used by people all around the world). Whenever possible, we want to keep backward compatibility.

Modifying the existing code is particularly dangerous when it affects base types. It's sometimes very hard to anticipate the impact on all the derived types.

Following the Opened-Closed principle mitigates the risk when introducing new functionality. When no changes are done to the existing code, we are sure it still works. The project is more stable and less error-prone.

All right. I hope you now see the benefits of following the Open-Closed Principle. Let's see an example of this principle being broken, and how such a code can be fixed.

`

```
public class Circle
{
    public double Radius { get; }

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

public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}

public class AreaCalculator
{
    public double Calculate(Circle circle)
    {
        return Math.PI * circle.Radius * circle.Radius;
    }

    public double Calculate(Triangle triangle)
    {
        return triangle.Base * triangle.Height / 2.0;
    }
}
```

This code looks pretty simple. We have two classes representing shapes and an AreaCalculator class that, well, calculates areas. This design **breaks** the Open-Closed Principle. If you are not sure why, imagine what would happen if there was a new business requirement - for example, to start supporting Rectangles and Squares. We would have to add new Rectangle and Square classes (this is fine - we would be **adding** new classes) but we would also have to **modify** the AreaCalculator class. Each time a new shape is introduced, we would need to modify the AreaCalculator class:

`

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}

public class AreaCalculator
{
    public double Calculate(Circle circle)
    {
        return Math.PI * circle.Radius * circle.Radius;
    }

    public double Calculate(Triangle triangle)
    {
        return triangle.Base * triangle.Height / 2.0;
    }

    public double Calculate(Square square)
    {
        return square.Side * square.Side;
    }
}
```

Let's refactor this code. Instead of having area calculation logic in the AreaCalculator class, let's move it to where it belongs - to each of the shapes. To keep backward compatibility, let's leave the AreaCalculator class, but only as a proxy to call the methods from each of the shape classes:

`

```
public interface IShape
{
    double CalculateArea();
}

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

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

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

```
public class Triangle : IShape
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }

    public double CalculateArea()
    {
        return Base * Height / 2.0;
    }
}
```

`

```
public class Square : IShape
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }

    public double CalculateArea()
    {
        return Side * Side;
    }
}
```

```
public class AreaCalculator
{
    public double Calculate(IShape shape)
    {
        return shape.CalculateArea();
    }
}
```

That looks better. Now, if a new shape is needed, the only thing we will have to do will be adding a new class implementing the IShape interface. The AreaCalculator class will not be affected, as it now depends on an abstract interface instead of a concrete class.

Let's consider one more example, that will point out some of the limitations of the Opened-Closed Principle. Imagine we are creating a system for an ice cream parlor. They sell some kinds of ice cream (like vanilla, chocolate, strawberry). They noticed that a huge amount of time is wasted on clients who can't decide which type of ice cream they want. That's why they asked you to create a mechanism that will randomly pick the ice cream for the client. Let's see this code:

`

```
public enum IceCreamType
{
    Vanilla,
    Chocolate,
    Strawberry
}

public class IceCream
{
    public IceCreamType IceCreamType { get; }
    public string[] Ingredients { get; }

    public IceCream(IceCreamType iceCreamType, string[] ingredients)
    {
        IceCreamType = iceCreamType;
        Ingredients = ingredients;
    }

    public override string ToString()
    {
        return IceCreamType.ToString();
    }
}

public class RandomIceCreamGenerator
{
    private Random _random = new Random();

    public IceCream Generate()
    {
        var randomType = GetRandomIceCreamType();
        switch (randomType)
        {
            case IceCreamType.Vanilla:
                return new IceCream(IceCreamType.Vanilla,
                    new[] { "Cream", "Sugar", "Vanilla" });
            case IceCreamType.Chocolate:
                return new IceCream(IceCreamType.Chocolate,
                    new[] { "Cream", "Sugar", "Chocolate" });
            case IceCreamType.Strawberry:
                return new IceCream(IceCreamType.Strawberry,
                    new[] { "Sugar", "Strawberry", "Coconut Cream" });
            default:
                throw new ArgumentException(
                    $"Invalid type of ice cream: {randomType}");
        }
```

You probably know what the problem is - what if a new type of ice cream is introduced? We will have to modify the RandomIceCreamGenerator. That's not right. You might have also noticed that not only the Open-Closed Principle is violated here, but also the Single Responsibility Principle. This class has two

`

responsibilities - picking a random type of ice cream and creating the ice cream basing on the type. Let's create a factory whose only responsibility will be to create the ice cream. You can learn more about Factory Method design pattern in the "What is the Factory Method design pattern?" lecture.

```
public class IceCreamFactory : IIceCreamFactory
{
    public IceCream Create(IceCreamType iceCreamType)
    {
        switch (iceCreamType)
        {
            case IceCreamType.Vanilla:
                return new IceCream(IceCreamType.Vanilla,
                    new[] { "Cream", "Sugar", "Vanilla" });
            case IceCreamType.Chocolate:
                return new IceCream(IceCreamType.Chocolate,
                    new[] { "Cream", "Sugar", "Chocolate" });
            case IceCreamType.Strawberry:
                return new IceCream(IceCreamType.Strawberry,
                    new[] { "Sugar", "Strawberry", "Coconut Cream" });
            default:
                throw new ArgumentException(
                    $"Invalid type of ice cream: {iceCreamType}");
        }
    }
}
```

`

Let's use it in the RandomIceCreamGenerator:

```
public class RandomIceCreamGenerator
{
    private Random _random = new Random();
    private readonly IIceCreamFactory _iceCreamFactory;

    public RandomIceCreamGenerator(IIceCreamFactory iceCreamFactory)
    {
        _iceCreamFactory = iceCreamFactory;
    }

    public IceCream Generate()
    {
        var randomType = GetRandomIceCreamType();
        return _iceCreamFactory.Create(randomType);
    }

    private IceCreamType GetRandomIceCreamType()
    {
        var values = Enum.GetValues(typeof(IceCreamType));

        return (IceCreamType)values.GetValue(_random.Next(values.Length));
    }
}
```

Great. Now this class will not be affected when a new type of ice cream is introduced… but the IceCreamFactory will be! We will have to add another case to the switch. That's actually one of the limitations we encounter when following the Open-Closed Principle. When adding new classes instead of modifying the existing ones, we still need some kind of toggle mechanism to switch between the original and extended behavior. The best we can do is to keep the code implementing such a toggle mechanism in one place - for example a factory class. This way, such change will be simple and will have a small impact on the application as a whole.

The other limitation is that we can't always predict every possible change, and sometimes we will simply be forced to modify the existing code to make it meet the business requirements. We should try to predict the most likely changes that may be needed, but we can't always do it perfectly. But, trying to predict **everything** is also a bad thing. Preparing the code to be modified in every way possible often leads to overcomplication and introducing premature abstraction. As with all things, moderation is recommended. Sometimes it is better to introduce a small modification in the existing code than to spend days on designing advanced mechanisms and abstractions and find out later that we actually never needed them.

`

Of course, bug fixing is another case when modification of the code is simply needed.

Let's summarize. The Open-Closed Principle states that the code should be opened for extension, but closed for modification. That means, when new business requirements are implemented, we should be able to do so by adding new classes instead of modifying existing ones. Following this principle makes the changes easier to be introduced, and mitigates the risk of causing bugs. There are reasonable scenarios when simple code modification is still required - mostly to implement a toggle mechanism between original and extended classes and bug fixing.

Tip: other interview questions on this topic:

- "**What are the good reasons to modify a class, disregarding the Open-Closed Principle?**"
  Firstly, for bug fixing. Secondly, sometimes sticking to the OCP might be an "overkill" - when it generates huge amounts of super-abstract code that brings more complexity than the OCP reduces.

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