**Declarative and Imperative paradigms**

A great C# example of declarative vs. imperative programming is LINQ. With imperative programming, you tell the compiler what you want to happen, step by step. For example, instead of using ready methods we create those methods ourselves🡪

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

With imperative programming, we'd step through this, and decide what we want:



With declarative programming, on the other hand, you write code that describes what you want, but not necessarily how to get it (declare your desired results, but not the step-by-step):



**Programming Paradigms**

Paradigm can also be termed as a method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming languages or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.







**.NET core and CLR(common language runtime)**

.NET is a free, cross-platform, open source developer platform for building many different types of applications. Developer platform- languages and libraries that we use. .NET or .NET core is a framework that has all the namespaces (System and others). ASP.NET CORE is a web framework that has all the needed namespaces. We can write .NET in C#, F# or VB.

.Net core is a new version of .Net framework. There are some limitations with the .NET Framework. For example, it only runs on the Windows platform. .NET core is an open-source framework. Our dll files are our libraries. In .NET framework there are few namespaces but they are gigantic. In contrast, in .NET core we have many but small namespaces. Dll files can be class libraries interfaces and etc.

The components of .NET are CLR, Garbage Collector, JIT Compiler and base class library.

CLS (Common Language Specification) As we know that .NET supports 63 programming languages and also we know that each and every language has their own syntax for writing code and one language does not support the syntax of another language. That means .NET has features of **common syntax** which are supported by all 63 programming languages.

CTS (Common Type System) As we know that .NET supports 63 programming languages and also we know that each and every language has their own data type system and One language does not support data type of another language. That means .NET has features of Common Data type which is supported by all 63 programming languages.

CLR is the run-time environment in the .NET. Language specific compiler compiles the source code into the MSIL(Microsoft Intermediate Language) which is also known as the CIL(Common Intermediate Language) or IL(Intermediate Language) along with its metadata. Metadata includes all the types, actual implementation of each function of the program

Now CLR comes into existence.



CLR invokes JIT compiler which then takes our bytecode (CIL) and converts that into machine code. This process is known ad JIT(just in time compilation).

Bytecode or MSIL is compiled by the compiler.

Why do we need to have the CIL? The runtime environment and development environment can be very different. Once our code is compiled and turned into MIL, depending on the runtime environment (whether you have Windows 10 or xp or completely another OS like MacOS) just-in-time compiler complies the best optimized code for that runtime environment.

**JIT**

When the execution code encounters a method call, for example, JITcompiler function is called. The JIT compiler is responsible for compiling IL code into native instructions. First, JIT compiler searches the defining assembly’s metadata for the method’s IL. JIT compiler next verifies and compiles the IL code into CPU instructions. The native instructions are saved in a dynamically allocated block of memory so that if this method is called again, the already translated native code can be used again, resulting in a better perfomance.

**Dynamic types**

C# 4.0 (.NET 4.5) introduced a new type called dynamic that avoids compile-time type checking. A dynamic type escapes type checking at compile-time; instead, it resolves type at run time.

dynamic a = 3;

a = "hello";

The last one is gonna be taken as a type which is a string in our case.

Int a =3; is a static type

The difference between generic types and dynamic types is that generic

types are resolved at compile time however dynamic types are decided at runtime.

Note that with dynamic types implicit cast is implemented but with object types explicit cast is required.

**TimeSpan**

This .NET type represents a length of time. There are 2 ways to create a timespan. One is to use the ctor, the other is to use the static methods of the Timespan object.

TimeSpan time1 = new TimeSpan(1,1,1);

Console.WriteLine(time1);

TimeSpan time2 = TimeSpan.FromMinutes(1);

Console.WriteLine(time2);



We can also use the properties of the TimeSpan object🡪

TimeSpan time = TimeSpan.FromMinutes(1);

Console.WriteLine(time.Minutes); 🡪 the output is 1

TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Minutes);

Console.WriteLine(time.TotalMinutes);



Minutes property just takes the number of minutes but total takes the entire timespan and calculates the minutes.

We can also add another time span to our existing timespan🡪

TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Add(TimeSpan.FromMinutes(1)).Minutes);



We can also subtacrt Timespan.

TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Subtract(TimeSpan.FromMinutes(1)).Minutes);



**LinkedList**

LinkedList consists of nodes. Each node has a piece of data and reference to the next node. The last node in the LinkedList always points to null.



public class Node

{

int \_data;

public Node next;

public Node(int data)

{

\_data = data;

next = null;

}

public void Print()

{

if (next!=null)

{

Console.Write($"{\_data}-->");

next.Print();

}

else

{

Console.Write($"{\_data}-->");

}

}

public void AddToEnd(Node newNode)

{

if (next != null)

{

next.AddToEnd(newNode);

}

else

{

next = newNode;

}

}

}

Node node = new Node(1);

node.AddToEnd(new Node(3));

node.Print();

**Constructor**

List<int> x = new List<int>();

Here we initialize parameterless constructor.

List<int> y = new List<int> {1,3};

Here we don’t need a parameterless constructor.

**Method Signature**

It includes only the number and the types of parameters. Method signature in c# doesn’t include the return type only the delegate signature includes the return type. Methods cannot be overloaded based on their return type. They must have a unique name, unique parameter types, or pass their arguments differently (e.g. using out or ref).

Public int GetName(bool isGraduated)

**S.O.L.I.D. Principles**

SOLID is one of the most popular sets of design principles in object-oriented software development.

Single Responsibility Principle - Robert C. Martin describes it as: A class should have one, and only one, reason to change. Meaning that one class should have just one responsibility and not more. Responsibility here is a reason.



For example, here Employee class has 3 reasons to change. It can change if the CalcPay method fails, ReportHours or WriteEmployee. So It violated the SRP.



For instance, here this Program class has just one responsibility which is to control the flow of the application. And all the other responsibilities such as Capturing Person’s first name and last name or Messages(StandardMessage) are handled by other classes so it is their responsibility. So we should separate everything. It is not a problem to have many classes. Each of them has their own responsibility. Of any of the classes that we create if one of them has a scroll then we have done something wrong because they gotta be short. Only Main( ) method ,of course, can be a little bit bigger.

This principle aims to separate behaviours so that if bugs arise as a result of your change, it won’t affect other unrelated behaviours.

Open-Closed Principle - The Open-Closed Principle (OCP) states that software entities (classes, modules, methods, etc.) should be open for extension, but closed for modification. For example, let’s say that we have different models which are just classes: Person, ManagerModel, ExecutiveModel, and also we have different accounts for these models: Accounts, ManagerAccounts, ExecutiveAccounts. So to implement OCP we need to create an IApplicantModel which will be inherited by Person, ManagerModel, ExecutiveModel. And for accounts we will also create an interface (Accounts, ManagerAccounts, ExecutiveAccounts)🡪





Accounts is a class for normal employees (that are just employees not managers or etc.).



This is for normal employees.



This model is for Managers.







So in the end, if we we wanted to add a new model, for instance, TechnicianModel, then we would create a TechnicianModel class that would inherit from IApplicantModel and we would create Accounts for this Model (TechnicianAccounts) which would be somewhat different. And IAccounts for TechnicianAccounts would point to TechnicianAccounts.

This principle aims to extend a Class’s behaviour without changing the existing behaviour of that Class.

Liskov Substitution Principle- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.



For example, here we have an Ostrich class that inherits from Bird class. It doesn’t make sense beacause ostriches cannot fly so LSP is broken. To comply with LSP, we can use OCP. So we can create an interface for FlyingBirds and implement this interface for other birds.

So the goal of this principle is basically providing a proper way for inheritance.

Interface Segregation Principle - Clients should not be forced to depend on methods that they do not use. So if we have multiple method declarations in our interface then our class that implements this interface will be forced to implement them. According to this principle, we need to create other interfaces for other tasks (separate interfaces).

Dependency Inversion Principle - High-level modules should not depend on low-level modules. Both should depend on the abstraction (interfaces). - Abstractions should not depend on details. Details should depend on abstractions.

Bacically, we just should get rid of new keyword or any low module in a high module so that the high module doesn’t depend on that.

Here we can use dependency injection to implement DI.



Here for instcance, even though we used IA in B still we have new A( ). So it breaks DI and in order to make DI we need Dependency injection.



In Dependency injection we just pass that IA as as argument to the constructor which then assigns that a to our private \_a. Nonetheless, at some point in our application we are gonna have to new up this class.

Dependency injection is one of the implementations of DIP.

One of the the benefits of implementing dependency inversion principle is that in unit testing when we test one class it will new up low modules if we have them there and it can take a lot of time or space in memory to new them up. However, if we implement this then we can create mock classes for those interfaces and that’s it.

**Concurrency and Parallelism**

Concurrency- Making progress on more that one task –seemingly at the same time. But this is actually happining one at a time, meaning that firt we do a little bit of task 1 or thread 1 then we go to task 2 or thread 2 then again we go back to task 1 or thread1 to do a little bit of that. It is called context switching.





Paralellism is exactly what we think of this, doing multiple things at the same time independently.

Concurrency- our computer takes once core and seperates it into threads. Then these threads are called one at a time.

In c# when we use thread classes, it doesn’t depend on us as to whether it is going to be concurrency or parallelism. Our computer decides it.



On a single-processor computer, **a thread scheduler** performs time-slicing — rapidly switching execution between each of the active threads. Under Windows, a time-slice is typically in the tens-of-milliseconds region — much larger than the CPU overhead in actually switching context between one thread and another (which is typically in the few-microseconds region).

On a multi-processor computer, multithreading is implemented with a mixture of time-slicing and genuine concurrency, where different threads run code simultaneously on different CPUs. It’s almost certain there will still be some time-slicing, because of the operating system’s need to service its own threads — as well as those of other applications.

**Synchronous vs Asynchronous**

Synchronous or Synchronized means **sequential** "connected", or "dependent" in some way. In other words, two synchronous tasks must be aware of one another, and one task must execute in some way that is dependent on the other, such as wait to start until the other task has completed.  
Asynchronous means they are totally independent and neither one must consider the other in any way, either in the initiation or in execution.

**Async and Await (TPL)**

The async keyword only enables the await keyword (and manages the method results).

Async and await are nothing but just markers that wait until the execution of the operation is done, only then you can continue. It is like using Task.Run(()=>…).Wait(). It is just a marker to tell the app from where to continue to read the code after completing the task.

When the await operator is applied to the operand that represents an already completed operation, it returns the result of the operation immediately without suspension of the enclosing method. The await operator doesn't block the thread that evaluates the async method. When the await operator suspends the enclosing async method, the control returns to the caller of the method.







When our program gets to He(x,y) method it goes inside that method and when it sees await keyword it checks if the result is already there, if it is not ready yet, then it goes back to the main conrtol which is the main thread and continues its work and when the result is ready it is going to continue from where it left off which is the line 3. So it continues from there and completes the method.



Here for example when we get to BolWaterAsync it starts that method goes inside 🡪



When it prints our “Start the kettle” and “waiting for the kettle” then when it gets to await it goes back to the main controller which was BoilWaterAsync and continues from there. So “take the cups out” and “put tean in cups” are printed out then we wait for boildwater to complete and then we can continue but again where when it gets to await keyword it goes back to the main control. Another thread is going to be designated for the process.

Await is just a checkpoint for the state machine.

Async keyword just tells the compiler to generate the state machine and this keyword will be in the il code so only compiler understands it not CLR or JIT.

Async methods can return Task<T>, Task, or void. In almost all cases, you want to return Task<T> or Task, and return void only when you have to.

TPL or Task Parallel library is just a set of software APIs that make it easier for developers to work with paralellism or concurrency.

Why return Task<T> or Task? Because they’re awaitable, and void is not. So if you have an async method returning Task<T> or Task, then you can pass the result to await. With a void method, you don’t have anything to pass to await. So you cannot await a void method. That’s why it is always better to use Task or Task of Tresult.

Async methods returning Task or void do not have a return value. Async methods returning Task<T> must return a value of type T:

public async Task<IViewComponentResult> Invoke()

{

ICollection<Product> products = await \_context.Products.Take(8).ToListAsync();

return View(products);

}

**Nameof( ) expression**

A nameof expression produces the name of a variable, type, or member as the string constant. It is always better to use nameof() expression instead of using a string name to find a property,method,action and etc. because when using nameof () expression it gives us a compile time error. Bu when using normal strings it doesn’t. For example, in RedirectToAction( ) method we can pass the name as a string 🡪



However, it is not a good practice. Because if later on we change the name then we will not know because it doesn’t give a compile time error. Once we run the app it will start looking for the Index action and when it doesn’t find it, it will give a run time error. Nevertheless, by using nameof( ) expression we make this better by having a compile time checker.



It says that Index doesn’t exist in the current context. So we go and fix this.



The output is x. it only outputs the names of variables, properties, methods 🡪



The output is Add.



The output is Count.

However we gotta be careful with this expression because it returns the value as it is in the nameof parenthesis. For instance, here it is an error.



The name of the assembly is “Allup Template” but it will return Allup\_Template resulting in an error. So here we are gonna have to use a normal string for the argument.

**Generics**

In c#, generic means not specific, to a particular data type. For instance, when we create a class with an int field. That field is static. But if it is a generic class. 🡪

Class A<T>where T:struct{

Public T x;

}

x is gonna be generic.

**Querystring**

The QueryString collection is used to retrieve the variable values in the HTTP query string. The HTTP query string is specified by the values following the question mark (?), like this:

<a href= "test.asp?txt=this is a query string test">

Query strings are also generated by form submission, or by a user typing a query into the address bar of the browser.

**The Singleton design pattern**

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code. In other words, a singleton is a class that allows only a single instance of itself to be created and usually gives simple access to that instance.



The constructor is private so that we can’t initialize the constructor with the parenthesis. The single object of this class is only initialized here in itself and it is readonly so that it doesn’t change.





They both call the default parameterless constructor which is private and it is not available. So we can’t create an instance of the class A. We can only access the single instance through GetA( ) method which is a public static method.

**Serialization**

Serialization is the process of converting an object in memory into a stream of bytes so that it can either be able to be sent over the network or be stored in a persistent storage. Deserialization is exactly the opposite – Fetch a stream of bytes from network or persistence storage and convert it back to the object with the same state.

We can convert objects to XML, JSON.

**The immutability of strings**

**Passing by reference vs Passing by value**

In the example shown below, the result is 4 because it references that integer. If there wasn’t the “ref” keyword then it would be 3.



As we know in c# strings are reference types but they act as value types. Here is why🡪



Here the result is “hey” which means the string x didn’t change despite the change method because the string that is passed as an argument in the change method is a brand new object in the memory so when we exit that method we leave that string as well. Although if we use the ref keyword then we can reference that string 🡪



The result is “changed”.



Here the result is unchanged because strings are immutable. Once we changed y (y=”w”) it allocates memory for that string in the heap and y will reference a new value “w” in the heap so x will remain unchaged. Therefore we can’t change even the characters of a string because they can’t be changed 🡪



**Managed code vs Unmanaged Code**

Managed is an environment where you have automatic memory management, garbage collection, type safety and etc. Unmanaged code is everything else. So for example, .NET is a managed environment and C/C++ is unmanaged. Type safety is that the casting has to be correct. For example, you can’t cast the following string to an int “hey”. It will throw InvalidCastException, so it the CLR allowed this happen then it wouldn’t be type safe.

CLR (common language runtime) in .NET offers services like garbage collection, run-time type checking, and reference checking. So we can think of it as, “My code is managed by the CLR”

**Binary Search**



**Expression**

Expressions always return a value. Loops such as while, for loops and etc. also if’s, switches are all statements.

Normally what our c# compiler does when it encounter a delegate is 🡪



It converts it into some code 🡪





Also for example, with IEnumerables the same method is also applied🡪



But with expressions, the compiler doesn’t actually convert that into code, it converts that into an expression 🡪





Our compiler takes that apart (dismantles it) 🡪



So the right part of our expression is 5 and it is a constant expression.





NodeType is an ExpressionType type which is an enum 🡪





And the left side of our expression, which is “I” 🡪







It doesn’t have a value! But it has a name.

Now the GreaterThan expression combines the two expressions into a single expression tree so it is called a binary expression

The expression tree is a binary tree 🡪



3 + ((5+9)\*2)







Lastly, we create a lambda expression. The first parameter is the top node (greaterThan Expression) and the second parameter is the parameters (Params[]) 🡪



Now these expressions are all stored in the heap cuz they are all objects. Now we can compile the lambdaExpression and return a delegate that represents the lambda expression 🡪



EntityFramework will use this data to convert this data to a sequal statement.

**IEnumerable and GetEnumerator**

Foreach works only for types that that have GetEnumerator meaning that types that implement IEnumerable interface or types that have matching methods because here you don’t have to implemeny any interface. If we didn’t have foreach then this is what we would have to do every time 🡪



So since array implements IEnumerable which has GetEnumerator method that returns IEnumerator we can get the array’s iEnumerator by using GetEnumerator() method of the array.

IEnumerator uses yield return combination for GetEnemarator() method meaning that only when it needs it will call the GetEnumerator method again and continue from where it left off.

**IEnumerator** has the method MoveNext which returns a boolean value telling whether there is a next element or not and the next element is IEnumerator.Current.

**IQuerable vs IEnumerable**

First of all, IQuerable<> implements IEnumerable. Bot Querable and Enumerable are just static classes that only have extension methods.

Differences: Enumerable is for **generic** collections such as lists, arrays. However, Querable class is for the collections of entities of the database.

Querable is a static class that has extension methods for IQuerables and Enumerable is also a static class that has extension methods for IEnumerables.

IQuerable is a collection of queries.



If we print people then 🡪



For where( ), select( ) and etc. methods in Enumerable the second parameter is always the delegate which means it will be converted into code.

But in Querable (for entity collections of the database), the second parameter is an expression which means they all will be individual objects so that we can analyze them and take them separately.

Lists and other System.Collections.Generic collections implement generic IEnumerable<T>. But our DbSets extend IQuerable. So IQuerables are Entity Collections and IEnumerables are just System.Collections.Generic’s collections.

The reason as to why DbSets are actually IQuerables is because after our code is compiled and run, EF will have to write some query for us. If they were IEnumerables which means it would use Enumerable class for the extension method and it would use just a normal delegate as a second parameter which means our lambda expression would get converted into a method. Here the EF will not be able to take parameters that it needs to write a query separately because all the data is lost.

However, if they are IQuerables then it will use the Querable class which means the second parameter is the expression which means all the data will be able to be taken separately, **the compiler generates code that will create objects in the heap which will represent our lambda expressions then we can reason about that data at runtime instead of having some generated code from the c# compiler(when we use just Func delegates)** and EF will be able to write a query for us.



In the first expression, EF will make a filtered query (select \* from dbo.Categories where id>5) then it will convert it into IEnumerable.

In the second expression, EF will also query the database but this time all the entities will be takes (select \* from dbo.Categories). Then our c# app will capture entity collection then it will convert it into IEnumerable and filter it.

IQuerable is inside System.Linq but IEnumerable is inside System.Collections. So when we write linq queries it means we create some IQuerables.

**Heap vs Stack**

Value types: enums, structs, number types (ints,byte …).

Reference types: classes, strings, arrays



Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM. Values types in classes are stored in the heap because they are in a class which is a reference type.

All objects on the managed heap contain two members: Type object pointer and Sync block index.

Every time a function declares a new local variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

**Type object pointer** is the reference to the type of the object.



As you can see these two instances of the same type (Cow) b betsy and g georgy have this “type object pointer” object that points to the same object in the heap which is just the Type object. Since both of their types are Cow, their “type object pointer” member is the same.

**Sync block index** is for thread synchronization. When we use lock keyword so that the code is used by one thread the sync block index changes.

All value types derive from ValueType class which is itself derived from System.Object. 🡺





**Static (properties,fields)**

Static fields, methods or properties are bound to the class itself not an instance of the class. So when we access a static member of a class we always reference the class itself rahter than through an instance.

**Compiler vs Interpreter**

Compiler - It takes an entire program as an input, Display all errors after compilation, all at the same time.

Interpreter- It takes a single line of coding as an input, Displays all errors of each line one by one.

**Out keyword**

If we want to return multiple outputs from a function, we use “out” keyword. In other words, we create variables then we reference to these variables from a function and calculate something and then we put the results in them. So we return 2 values or more for instance.





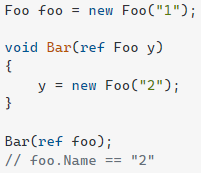
We can simplify it 🡪



Here we declare those variables right inside our Method parenthesis.

**Ref keyword for reference types**

When we pass reference types as arguments, their reference is passed. But say we want to modify this reference then we won’t be able to affect the actual reference type variable (caller of the method). To actually modify the reference of the reference type outside of its scope we need to use ref keywork🡺



So the above foo variable will also reference this newly created Foo(“2”) object. But if we pass without ref then foo will again reference the same Foo(“1”) object. So when we create a new reference we need to use ref keyword. This way we change what foo points to.

**Ref and out cannot be used with Asynchronous method**

Internal State machine (IAsyncStateMachine): Every time we have an asynchronous method, the compiler actually turns the method into an internal state object. Internal state class is designed in build time and instantiated in run time by the CLR, it implements the IAsyncStateMachine interface. This class is responsible for keeping the state of your method during the life cycle of the asynchronous operation, it encapsulates all the variables of your method as fields, splits your code into sections that are executed as the state machine transitions between states, so that the thread can leave the method and when it comes back the state is intact.

Unfortunately in internal state class we can't store the address of an out or ref parameter, basically CLR has no safe way to deal with the address of an object. As consequence the compiler forbid the use of Ref and out parameters in asynchronous method.

Workaround for out in async method would be to use tuples to return multiple values.

**Operator overloading (static|compile-time polymorphism)**

So here we are just telling that whenever you see a plus operator in the left side of a box objext then do this 🡪



The box that is a parameter is actually our box to which the plus operator is assigned to.

 The result is 1

We can also have a custom operator overloading for adding 2 boxes 🡪



Box1 is the left and box2 is the right operand.

 the result is 8

**Note that operator overloading methods must be static methods!**

**Unary Operator**

A unary operator, in C#, is an operator that takes a single operand in an expression or a statement. The unary operators in C# are +, -,!, ~, ++, -- and the cast operator.

 **the result is 3**

int x = 0;

Console.WriteLine(x++); //0 it first prints it then increments

Console.WriteLine(++x); //2 increments it then prints

 the output is zero. Because it goes from left to right.

 the result is 1. Because it first incremented y then x copied that value.

**Struct**

Structs are value types. They cannot inherit a class but they can implement inetrfaces. Int32, Boolean,Single(float) are readonly structs.

**Delegate vs Events**

In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class. Event is a delegate reference with 2 restrictions

1. You cannot invoke the delegate reference directly as opposed to just delegates where you have the power to invoke it and assign something to it such as null.
2. You cannot assign to it directly

So let’s say that we have an Action delegate 🡪



So in the main class, we can invoke it ass well as assign something to it 🡪





As a result, it can crash our program. So in order to prevent it we can use events. They will give you a compile-time error if you try to either invoke the delegate or assign something to it. You can only subscribe and unsubscribe something.

Only in the publisher class you can invoke the event.

EventHandler is a delegate that takes 2 arguments. Object sender and EvenrtArgs. So when we invoke this delegate we gotta pass 2 arguments.

**Tuple**

Tuples are used to return multiple values from a method without using ref or out parameters.





We can also name parameters in the method itself so that it is distinct. 🡪



There is also a Tuple class that does pretty much the same thing 🡪



However here we access elements by item1, item2 and etc.

**ConvertTo and Parse**

Parse and Convert ToInt32 are two methods to convert a string to an integer. The main difference between int Parse and Convert ToInt32 in C# is that passing a null value to int Parse will throw an ArgumentNullException while passing a null value to Convert ToInt32 will give zero.

**Using statement and IDisposable implementors**

Normally, when our app ends, garbage collector will free up the memory from objects. However, there are some objects that implement IDisposable (files and etc) and we ourselves need to get rid of them. Some IDisposables: filestreams, readers such as sql, xmlreader, SqlConnection and etc.

Using statement is used to create a scope for a resource and once the scope ends it is going to dispose of that object (database and etc.)

If the type implements IDisposable, it automatically disposes that type.

public class SomeDisposableType : IDisposable

{

...implmentation details...

}

If it inherits from IDisposable then it will have Dispose() method to clean up

SomeDisposableType t = new SomeDisposableType();

try {

OperateOnType(t);

}

finally {

if (t != null) {

((IDisposable)t).Dispose();

}

}

This is the same as🡪

using (SomeDisposableType u = new SomeDisposableType()) {

OperateOnType(u);

}

Using calls Dispose() after the using-block is left, even if the code throws an exception. So you usually use using for classes that require cleaning up after them. In our example above(database) it will close our database and SqlDataReader as well.

If an error occurred in the middle of the using statement, then it throws an exception and finally block is activated to dispose of all objects.

**.NET standard**

There are various implementations of .NET. Each implementation allows .NET code to execute in different places—Linux, macOS, Windows, iOS, Android, and many more. Our class library targets .NET standard.

.NET Standard is a formal specification of the APIs that are common across all these .NET implementations(Xamarin, .NET framework). .NET Standard allows libraries to build against the agreed on set of common APIs, ensuring they can be used in any .NET applications—mobile, desktop, IoT, web, or anywhere you write .NET code. .NET Standard is just a set of requirements (APIs,libraries).

.Net Standard is an "interface" that both the FULL .NET Framework, the .NET Core Framework, and Xamarin iOS/Android and Unity implement.

NET 5 and all future versions will always support .NET Standard 2.1 and earlier.

.NET standard is just a contract with empty classes and methods (that throw null). It is a specification that all .NET apps must have specific libraries or apis. .NET standard comes with plenty of namespaces in which there are classes that are empty. They just tell that you must have it! **We don’t run against .NET standard assembly we compile against this assembly!**

Each implementation of the managed framework has its own set of Base Class Libraries. The Base Class Library (BCL) contains classes such as exception handling, strings, array, XML, I/O, networking, and collections. .NET Standard is a specification for implementing the BCL. Since a .NET implementation is required to follow this standard, application developers will not have to worry about different versions of the BCL for each managed framework implementation.

So in the past when we wanted to switch to another platform, for example from .NET framework to Xamarin then we had the following problem🡺 the BCLs that contain basic classes for string manipulations, I/O, array and etc. were different in all of the platforms. They all had their own base class libraries. So we needed to change our BCL to the one that we were switching to. Microsoft solved this problem by creating .NET standard library and all the platforms will reference this specification (interface, contract). .NET standard is a set of apis that all .NET implementations must adhere to. So every .NET platform must implement this contract.

Framework Class Libraries (FCL) such as WPF, WCF, and ASP.NET are not part of the BCL, and therefore are not included in .NET Standard.

The relationship between .NET Standard and a .NET implementation is the same as between the HTML specification and a browser. The second is an implementation of the first.

Hence, the .NET Framework, Xamarin, and .NET Core each implement .NET Standard for the BCL in their managed framework. Since the computer industry will continue to introduce new hardware and operating systems, there will be new managed frameworks for .NET. This standard allows application developers to know that there will be a consistent set of APIs(libraries) that they can rely on. Each .NET version has an associated version of the .NET Standard.

**BCL and FCL**

Base class library contains all the common classes that are used in .NET apps. The Base Class Library (BCL) is literally that, the base. It contains basic, fundamental types like System.String and System.DateTime.

The Framework Class Library (FCL) is the wider library that contains the totality: ASP.NET, WinForms, the XML stack, ADO.NET and more. You could say that the FCL includes the BCL.

**Data types**

1. Int16 (aka short): A signed integer with 16 bits (2 bytes) of space available.
2. Int32 (aka int): A signed integer with 32 bits (4 bytes) of space available.
3. Int64 (aka long): A signed integer with 64 bits (8 bytes) of space available.
4. Single (aka float): A 32-bit floating point number.
5. Double (aka double): A 64-bit floating-point number.
6. Decimal (aka decimal): A 128-bit floating-point number with a higher precision and a smaller range than Single or Double.



Here the result is 0. Because in (y / x) y and x are converted into Int32 and when we divide 2 by 3 it is 0 so the multiplication is also 0.

Double is better at perfomance than decimal but decimal takes more time to run however it is better at precision.

**Logical vs Conditional operatos**

The single & operator evaluates both operands even if the left-hand operand evaluates to false, so that the operation result is false regardless of the value of the right-hand operand. In the following example, the right-hand operand of the & operator is a method call, which is performed regardless of the value of the left-hand operand:



But when we use double && (conditional operators) then the second operand is not evaluated if the first operand is false.

Unlike the Boolean logical operators "&" and "|," which always evaluate both the operands, conditional logical operators execute the second operand only if necessary. As a result, conditional logical operators are faster than Boolean logical operators and are often preferred. The execution using the conditional logical operators is called as “short-circuit” or “lazy” evaluation.

**Equals**

When we want to compare two objects by default their references are compared to one another but let’s say that we want to compare them by their values. So then we need to override Equals method 🡪



**GetHashCode() method (HashTables)**

GetHashCode returns an Int32, which has “only” about 4.2 billion possible values, and there’s potentially an infinity of different objects, so some of them are bound to have the same hash code.

Two objects that have the same hash doesn’t mean that they are equal. It works the other way: two objects that are equal have the same hash code (or at least they should, if Equals and GetHashCode are correctly implemented).

GetHashCode mostly exists for one purpose: to serve as a hash function when the object is used as a key in a hash table.

GetHashCode for value ints returns their values.

A hash code is not a permanent value so do not serialize, store the hash values in databases etc. Do not test for equality of hash codes to determine whether two objects are equal.

Let’s say that we have an object of A and another object of the same type and they have a property X 🡪





Let us say that for Type A if the values of X are the same in both of the instances then they are equal. So when we use dictionaries it will not use the values for X for comparison but it will use the reference comparison which in our case we don’t want. So what we can do is to override Equals method but since the object’s Equals is a non-generic method we will implement IEqutable to get a better perfomance with Equals 🡪



After this we will be warned that we now need to override Equals of System.Object because if the casting is not successful then it will call the Equals of the base class

Once we override that as well one we are gonna also need to override GetHashCode method It is because the framework requires that two objects that are the same must have the same hashcode. If you override the equals method to do a special comparison of two objects and the two objects are considered the same by the method, then the hash code of the two objects must also be the same. (Dictionaries and Hashtables rely on this principle). But if we don’t override GetHashCode then it will use the default HashCode which we don’t want and as a result the same considered objects will be added to the hashtable and we don’t want it.

So our entire class🡪



Now when the values of X are equal then they will be deemed equal and in our dictionary System.ArgumentException will be thrown🡪



So here what happens is for a1 hashcode is gonna be fetched and for a2 hashcode is gonna be fetched. If their hashcodes are different then they are gonna be different and Equals method is not gonna get called but if their hashcode is the same then The Equals method is gonna get called to see if they are the same or not if they are the same then the ArgumentException is gonna be thrown at us. if they are different then the hash function is gonna be decided on the object’s memory address because it can be decided on their hash codes since the have the same hash code.

If we don't override Equals then the default behavior is that references of the objects are compared. The same applies to hashcode - the default implmentation is typically based on a memory address of the reference. Because you did override Equals it means the correct behavior is to compare whatever you implemented on Equals and not the references, so you should do the same for the hashcode.

Clients of our class will expect the hashcode to have similar logic to the equals method, for example linq methods which use a IEqualityComparer first compares the hashcodes and only if they're equal they'll compare the Equals() method which might be more expensive to run, if we didn't implement hashcode, equal object will probably have different hashcodes (because they have different memory address) and will be determined wrongly as not equal (Equals() won't even hit).

Dictionary<TKey, TValue> class is the most commonly used hash table implementation. HashSet<T> is also based on a hash table, as the name implies. In dictionaries, keys are unique but values can be the same and if the values are eqaul then it is collision.

|  |  |
| --- | --- |
| Hashtable | Dictionary |
| A Hashtable is a non-generic collection. | A Dictionary is a generic collection. |
| Hashtable is defined under System.Collections namespace. | Dictionary is defined under System.Collections.Generic namespace. |
| In Hashtable, you can store key/value pairs of the same type or of the different type. | In Dictionary, you can store key/value pairs of same type. |
| In Hashtable, there is no need to specify the type of the key and value. | In Dictionary, you must specify the type of key and value. |
| The data retrieval is slower than Dictionary due to boxing/ unboxing. | The data retrieval is faster than Hashtable due to no boxing/ unboxing. |
| In Hashtable, if you try to access a key that doesn’t present in the given Hashtable, then it will give null values. | In Dictionary, if you try to access a key that doesn’t present in the given Dictionary, then it will KeyNotFound exception. |

**Indexers**

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.



We can access the array through x or instance itself 🡪



**Private set and init keywords**

Private set is just to encapsulate the setter so that setting is only available within the class. 🡪



So outside of the class we are not able to set a value for Y.

Getter also can be private. But with “init” keyword from c# 9 version we can only initialize values but it restricts us from reassinging a value to that variable.





So we can only initialize it through a constructor or a collection-initializer syntax.



We are not able to reassign Y for b.

**Const vs readonly, static**

Conts are compile-time constants however readonly constants are run-time constants.

You want to use const when you have a variable whose value will not change, ever, during the time your application is being used. Further, any variable declared as const will also, implicitly, be declared static.

Only primitive or "built-in" C# types (e.g. int, string, double) are allowed to be declared const. Therefore, you cannot write either of these



A static member (variable, method, etc) belongs to the type of an object rather than to an instance of that type.

A readonly field is one where assignment to that field can only occur as part of the declaration of the class or in a constructor. This means that a readonly variable can have different values for different constructors in the same class.

We use const when the value is always contant from the beginning such as pi, 1m in sm is 100 and it never changes.

We use readonly when the value is not in our hands yet but it will come soon and it can be assigned through a dependency injection and only once! so they both can’t be changed!

A readonly field can be initialized either at the time of declaration or within the constructor of the same class. We can also change the value of a Readonly at runtime or assign a value to it at runtime only in the constructor! 🡪



We can use const keyword to a local variable 🡪



But we can’t use readonly for local variables because it is not able to assign anything to it at runtime. It is impossible to assign something to a local variable at runtime.

When you define a constant, the constant is kept in the assmebly’s metadata that it was defined in. So once we build that assembly then the constant is gonna be written in the metadata and other modules that reference that constant are going to reference that metadata.

If we change the value of the constant we also need to rebuild the assembly that it was defined in. Otherwise we will use the old value.

The compiler saves the constant’s value in the assembly’s metadata. This means that a constant can be defined only for the primitive type like boolean, char, byte and so on

This is also the same for method default values.

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields.

**Var keyword**

**var** is a keyword, it is used to declare an implicit type variable, that specifies the type of a variable based on initial value. The compiler does the job of assigning the type for us but!!! There is a huge but. It can only appear within a local variable declaration so that the compiler can tell what the type is so it can’t be used in method parameters 🡺



Also the variable has to be initialized as soon as the var is used so that the compiler can tell what the type is 🡪



Var keyword can only be used for local vairbales!

**== conditional operator and Equals**

== is used to compare the references of objects meaning they they compare whether they are referencing the same object or not. It is similar to Object.ReferenceEquals(obj1,obj2) method.

== and System.Object.ReferenceEquals are the same.But Equals checks the content and it will throw a NullReference exception if the object before the dot (to which “this” keyword is tied to) is null 🡺



It will throw the NullReference exception. However, ReferenceEquals will return a boolean value (false in this case).

On the other hand, Equals compares the content. However for strings all of the methods above will compare strings by their content so even Object.ReferenceEquals(obj1,obj2) method will be used to compare their content.

However, when it comes to user defined types then we need to override Equals from System.Object and make that explicity. Otherwise the default is reference comparison for both == operator and Equals method.

When comparing values using ReferenceEquals, if objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method. This means that even if both objA and objB represent the same instance of a value type, the ReferenceEquals method nevertheless returns false.

**Bitwise operations and [Flags] attribuet**

Logical operators are used to manupulate bitwise numbers let’s say that we have 3 rights read, write and delete and we represent them by the folowing binary numbers 🡪



| operator is used to denote the combination for example if we want to denote that user can read and write then read| write will do the combination. We can check with & operator.

We can use Convert.ToString() method to get the 2 base number.

However, we can use enums for better legibility. We use [Flags] attribute. They indicate that an enumeration can be treated as a bit field; that is, a set of flags. 🡪

Or we can use the method



It will be false. If we don’t use flags attribute 🡪





Then it will print out 5 meaning that it will combine those binary numbers and return the decimal representation but we don’t want that we want to represent it by strings so in this case we use [Flags] attribute and as a result we will get a string representation of the read and delete combination 🡪



When we have plenty of information for example let’s say we have to store 365 days in the enum so in this case since to represent 365 is going to take a lot of binary numbers we can use bitArray 🡪 let’s represent 001 which is the read right.



**IClonable, cloning reference types**

As we know when we assign a reference type to a reference type it copies the address and it references the same object in the heap so when we try to change something they both get changed.

In order to be able to clone an object we need to implement IClonable interface for that class 🡪



And in the method we can call the MemberwiseClone() method and it returns a shallow copy of the object.

**Garbage Collector**

GC works on managed heap, which is nothing but a block of memory to store objects, when garbage collection process is put in motion, it checks for dead objects and the objects which are no longer used, then it compacts the space of live object and tries to free more memory. Basically, heap is managed by different 'Generations', it stores and handles long-lived and short-lived objects, see the below generations of Heap:

1. 0 Generation (Zero): This generation holds short-lived objects, e.g., Temporary objects. GC initiates garbage collection process frequently in this generation.
2. 1 Generation (One): This generation is the buffer between short-lived and long-lived objects.
3. 2 Generation (Two): This generation holds long-lived objects like a static and global variable, that needs to be persisted for a certain amount of time. Objects which are not collected in generation Zero, are then moved to generation 1, such objects are known as survivors, similarly objects which are not collected in generation One, are then moved to generation 2 and from there onwards objects remain in the same generation.

When we declare a new variable it is stored in generation 0, unless the size of the item is larger that 85,000 bytes (these objects go to a specific heap called LOH large object heap a.k.a generation 3) . If there is no space in generation 0 then garbace collector starts working and placing long lived objects to generation 2 and while doing this if it notices that generation 2 is full then it does the same thing (moves objects to generation 3).

How GC decides if objects are live? GC checks the below information to check if the object is live:

1. It collects all handles of an object that are allocated by user code or by CLR
2. Keeps track of static objects, as they are referenced to some other objects
3. Use stack provided by stack walker and JIT

When GC gets triggered? There are no specific timings for GC to get triggered, GC automatically starts operation on the following conditions:

1. When virtual memory is running out of space.
2. When allocated memory is suppressed acceptable threshold (when GC found if the survival rate (living objects) is high, then it increases the threshold allocation).
3. When we call GC.Collect() method explicitly, as GC runs continuously, we actually do not need to call this method.

When our main flow goes out of one scope what will happen is our GC will check if still the object references that if yes then it will mark it if no then it will free up memory from that object. Marked objects are sent to the next generation. For GC a thread is designated.



Garbage collection occurs when one of the following conditions is true:

* The system has low physical memory.
* The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
* The GC.Collect method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

The garbage collector's optimizing engine determines the best time to perform a collection based on the allocations being made. When the garbage collector performs a collection, it releases the memory for objects that are no longer being used by the application.

It determines which objects are no longer being used by examining the application's roots. An application's roots include static fields, local variables on a thread's stack, CPU registers, GC handles, and the finalize queue.

Each root either refers to an object on the managed heap or is set to null. Using this list, the garbage collector creates a graph that contains all the objects that are reachable from the roots. Objects that are not in the graph are unreachable from the application's roots. The garbage collector considers unreachable objects garbage and releases the memory allocated for them. During a collection, the garbage collector examines the managed heap, looking for the blocks of address space occupied by unreachable objects. As it discovers each unreachable object, it uses a memory-copying function to compact the reachable objects in memory, freeing up the blocks of address spaces allocated to unreachable objects.

Once the memory for the reachable objects has been compacted, the garbage collector makes the necessary pointer corrections so that the application's roots point to the objects in their new locations. It also positions the managed heap's pointer after the last reachable object.

**Finalizer and Destructor**

C# really doesn't have a "true" destructor. The syntax resembles a C++ destructor, but it really is a finalizer. You wrote it correctly in the first part of your example:

~ClassName() { }

The above is syntactic sugar for a Finalize function. It ensures that the finalizers in the base are guaranteed to run, but is otherwise identical to overriding the Finalize function. This means that when you write the destructor syntax, you're really writing the finalizer. According to Microsoft, the finalizer refers to the function that the garbage collector calls when it collects (Finalize), while the destructor is your bit of code that executes as a result (the syntactic sugar that becomes Finalize). They are so close to being the same thing that Microsoft should have never made the distinction. Microsoft's use of the C++'s "destructor" term is misleading, because in C++ it is executed on the same thread as soon as the object is deleted or popped off the stack, while in C# it is executed on a separate thread at another time.

If a type implements the IDisposable interface, you should definitely call the Dispose method (either explicitly or by a using block).

What happens if i don't call dispose()? If you don't, the destructor (finalizer) is responsible for freeing the resources; however, it has some drawbacks:

It is not deterministic: finalizers are executed by the GC on a dedicated thread. The GC decides when to run them. If a reference is kept to the object (eg. in main app window), it is possible that the finalizer will not be executed until you exit the application.

Overhead: unless the finalizer is suppressed, the GC has some todo with the objects to destroy.

Dangerous: if a finalizer throws an exception, it is considered fatal and will crash the whole application.

**Abstract class vs Interface**

Abstract classes are half-defined base classes. In other words, they are normal classes but they have some methods whose implementation may vary depending upon the derived class but all of them will have that method no matter what! It is just gonna be somewhat different. For example we have a Customer class and all customers can make an enquiry and this is gonna be the same for every customer so it is not abstract but let’s say we have another method GetDiscount whose implementation may vary depending on the customer so then this method has to be abstract if, of course, all customers have that method 🡪



Here the name prop is common and we will just set the name through the instances of derived classes. However, the GetDiscount method is gonna change depending on the customer so it is marked as abstract and so it is implicitly a virtual method but forceable one.



VahidCustomer is gonna have Name prop, Request Method, and overridden GetDiscount method.

The problem with abstract classes is that if one of the customers doesn’t have a GetDiscount method then we will be forced to inherit that one the one that we don’t have so it is the violation of Liskov Substitution principle then we can create another abstract class but we can’t inherit from multiple classes.

But let’s say that some customers do not have GetDiscount method so then it would’t be rational to use abstract class here because of the GetDiscount method. It is just not common for all the derived classes. Here we are violating the Liskov substitution principle.

In this case we would create an interface IDiscountable and customers who have discounts will be able to implement the interface.

Interfaces are implicitly public. We should only define the signatures of methods in interfaces. However, we can also have methods that have some logic but they are not gonna be able to be implemented because they will be accessible only through interfaces. They are for the manifestation of the interface.

Interfaces can implement other interfaces but they cannot inherit from anything. Only classes have the ability to inherit. Even structs can’t inherit they can only implement.

The reason as to why fields cannot be abstract is because in fields you don’t have any implementation. You can just assign something to a field or change the value and that’s it. However, in properties you can have a different logic for your getter or setter. That’s why it is not reasonable to put fields as abstract. So we can’t use abstract keyword for fields in abstract classes, and also we can’t use fields in interfaces.

We can implement multiple interfaces but we can inherit only from one class. Abstract classes can have non-abstract methods. We implement interfaces but we inherit from classes!

**Base class**

Suppose we have a base class that has no parameterless constructor. This base class has one parameterized constuctor 🡪



All the sub/derived classes must send this x parameter to the base class 🡪



If the base class has one parameterless constructor, then it is our choice whether we want to use parameterized or parameterless constuctor of the base class. If we send, then the parameterized constructor of the base class will get activated. 🡺





Here the parameterless constructor will get activated



But here the parameterized one will be called.

**Arrays**

Array - The fundamental concept here is that the elements of an array are stored in contigous memory locations, if a[0] is at a memory location 2000, then a[1] is guaranteed to be at location 2002 [ assuming 2 bytes of storage ]. It is this gurantee that makes array access constant time. In general, if I know the base address of array, i.e., a[0], I would get the address of an arbitrary element in array e.g., a[i] by simply doing address of **a[0] + i \* size**; a[1] = 2000 + 1 \* 2; = 2002. So, array look up / access is O(1). This sort of math can be applied to any dimensional array, because the elements would be in a continuous memory location. Note however, that if you take linked list, there is no guarantee that elements would be stored in this form. And that is why, linked list access is not constant time. It is dependent on the size of the linked list. So, linked list look up / access is O(N).

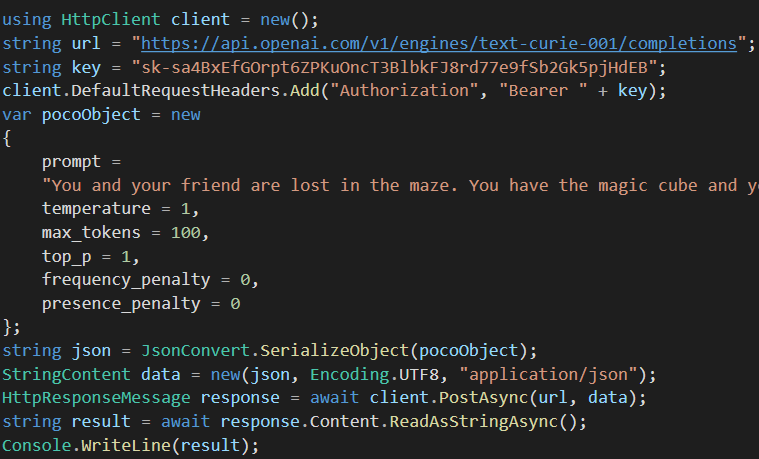
**Framework vs Library**

We depend on a framework but we do not depend on a library. Libraries do not have any structures. Libraries leave it to us as to how we are going to use it. But in a framework, usually there is a certain structure that we need to follow. We can also change libraries and make them however we want. Bootstrap is a css library. React is a javascript library. Angular is a framework.

**Assembly**

An assembly is a unit of reuse, versioning, and security. It allows you to partition your types and resources into separate files so that you, and consumers of your assembly, get to determine which files to package together and deploy.

**Sending a request (HttpClient)**



StringContent provides HTTP content based on a string. To send an object to web api server we use StringContent to add format to HTTP content, for example to add Customer object as json to server.

**Overhead members (Type object pointer and Sync block index)**

The Sync Bloc Index (a.k.a Object Header Word) field is being used for several different purposes:

1. Thread Synchronization – using the CLR Monitor Mechanism with the lock keyword.

2. Storing the object's Hash Code – used in a hash-based collection, such as: Dictionary.

3. Garbage Collector algorithm – used in Mark phase.

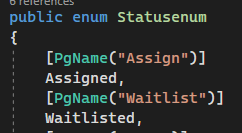
4. Garbage Collector Finalization mechanism.

In garbage collector algorithm, it is used in the following way 🡺 if any variable has a reference to the object then the object is marked by switching sync block index to 1.

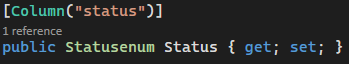
Type object pointer is just an object that points to the original type.

**Enum types for PostgreSql**

When using enum types for Database, keep in mind that if we later decide to remove one of the values from the enum, we won’t be able to do so. Also the order is also important for enum types since they communicate only with numbers. In postgres, if we want to declare a custom enum type we do the following 🡺







We also need to specify the custom enum type in the static constructor

of the DbContext so that it can map the enum type🡺



**Contra-variant and Covariant generic type arguments**

Contra-variant- (**in** keyword) Meaning that the generic type parameter can change from a class to a class derived from it.

Covariant- (**out** keyword) Meaning that the generic type parameter can change from a class to its base classes. 🡺

Public delegate Tresult Func<in T, out TResult> (T arg);

Here T is marked as contra-variant, TResult as Covariant 🡺

Func<Object, ArgumentException> fn1 = null;

We can cast it to another Func type, where generic type parameters are different 🡺

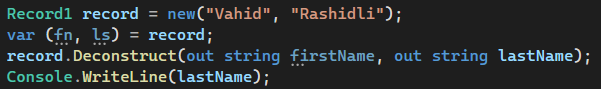
Func<String, Exception> fn2 = fn1;

**Records**

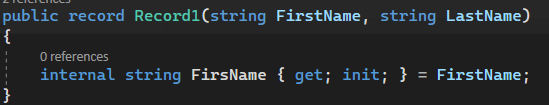
Records are reference types. **Records are immutable meaning that their values are set only once and cannot be changed 🡺**

1. ToString() method for records is overitten. Now it prints the signature of the record 🡺The result is 
2. Equals method is also overritten for records. So normally with reference types it checks their reference by default but with records their values not references are compared by default.

We can also deconstruct any record based upon the same pattern as it was declared. We can deconstruct it with the help of tuple conception or by just using the Desconstruct method. So the above record can be deconstructed as below 🡺



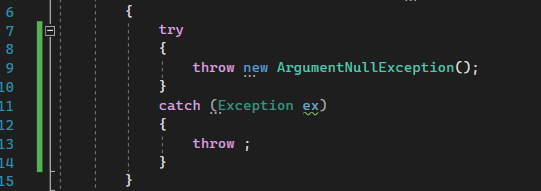
If we want to change something for any prop (maybe make prop internal)🡺



We use records where the data can’t be changed.

**Throw vs Throw ex**

“throw ex” resets the stack trace (so your errors would appear to originate from HandleException) “throw” doesn't - the original offender would be preserved.





But with throw ex 🡺



So throw ex just resets the stack trace.

throw ex just ignores all the previous hierarchy and resets stack trace with line/method where throw ex is written.

**Memory Barrier**

The compiler or JIT might re-order the memory instructions for optimization.

The processor can also re-order memory instructions for performance through different mechanisms including caching, load speculation, and delaying store operations.

It should be noted that the JIT compiler will, as part of an optimization strategy, change the order of the reads and writes in a way that does not change the meaning and eventual output of the program. This is illustrated in the code snippet given below.

x = 0;  
x = 1;

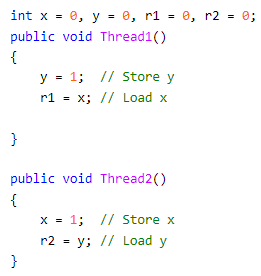
The above snippet of code can be changed to the following—while preserving the program’s original semantics.

x = 1;

x = 1;  
y = x;

The above code snippet can be changed to the following—again while preserving the original semantics of the program.

x = 1;  
y = 1;



If the methods run on different threads, there are a few possibilities for the values of r1 and r2:

1. Thread1 finishes before Thread 2 starts, so that r1 = 0 and r2 = 1.
2. Thread 2 finishes before Thread 1 starts, so that r2 = 0 and r1 = 1.
3. Both Thread1 and Thread2 interleave, so that either one of the following will happen:
   1. r1 = 1 and r2 = 1 or
   2. r1 = 0 and r2 = 1 or
   3. r1 = 1 and r2 = 0

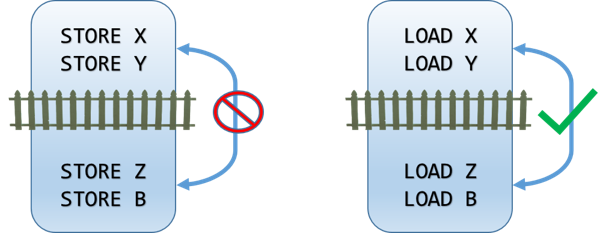
Memory barriers (aka memory fences) are a way to tell the compiler, JIT and the processor to restrict the ordering of memory instructions.

There are different kinds of memory barriers:

1. **Store memory barrier** (or write memory barrier)

A store memory barrier ensures that no STORE operation can move across the barrier.

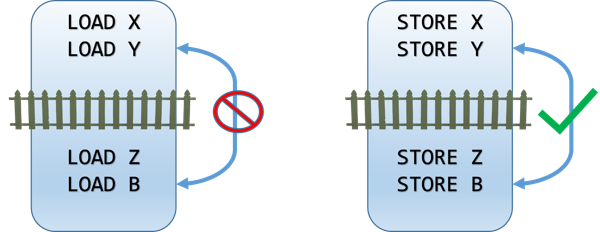
All the STORE operations that appear before the memory barrier will appear to happen before all the STORE operations that appear after the memory barrier. The equivalent CPU instruction is SFENCE.

This has no effect whatsoever on LOAD operations.

1. **Load memory barrier** (or read memory barrier)

A load memory barrier ensures that no LOAD operation can move across the barrier. All the LOAD operations that appear before the barrier will appear to happen before all the LOAD operations that appear after the memory barrier. The equivalent CPU instruction is LFENCE.

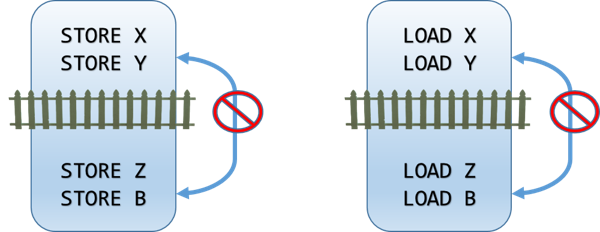
This has no effect whatsoever on STORE operations.



1. **Full memory barrier**

A full memory barrier ensures that no STORE or LOAD operation can move across the barrier.

All the STORE and LOAD operations that appear before the barrier will appear to happen before all the STORE and LOAD operations that appear after the barrier. The equivalent CPU instruction is MFENCE.



A full memory barrier is the strongest and interesting one. At least all of the following generate a full memory barrier implicitly:

* + Thread.MemoryBarrier
  + C# Lock statement
  + Monitor.Enter and Monitor.Exit
  + Task.Start
  + Task.Wait
  + Task continuations
  + Interlocked class mehods
  + Any signaling operation such as ManualResetEvent and AutoResetEvent
  + Thread.Sleep
  + Thread.Join
  + Thread.SpinWait
  + Thread.VolatileWrite
  + Thread.VolatileRead

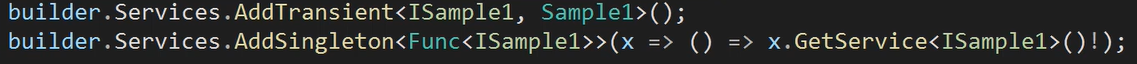
“lock” keyword for example tells the CPU where it may not reorder store commands. In particular, it says all stores before the fence must remain before the fence, and all stores after the fence must remain after the fence. Thus the reordering of commands is not allowed.

When one thread is interacting with the data in the cache, and a second thread tries to read the same data concurrently, the second thread might read an outdated version of the data from the main memory. This is because when the value of a non-volatile object is updated, the change is made in the cache of the executing thread and not in the main memory. However, when the value of a volatile object is updated, not only is the change made in the cache of the executing thread, but this cache is then flushed to the main memory. And when the value of a volatile object is read, the thread refreshes its cache and and reads the updated value.

The **volatile** keyword in C# is used to inform the JIT compiler that the value of the variable should never be cached because it might be changed by the operating system, the hardware, or a concurrently executing thread. The compiler thus avoids using any optimizations on the variable that might lead to data conflicts, i.e. to different threads accessing different values of the variable.

When you mark an object or a variable as volatile, it becomes a candidate for volatile reads and writes. It should be noted that in C# all memory writes are volatile irrespective of whether you are writing data to a volatile or a non-volatile object. However, the ambiguity happens when you are reading data. When you are reading data that is non-volatile, the executing thread may or may not always get the latest value. If the object is volatile, the thread always gets the most up-to-date value.

**Abstract Factory Pattern with dependency injection**



In the above example, we first create ISample service after which we implement a factory pattern which is basically a method that returns ISample each time from the DI container(IScope). X parameter is a func delegate. Then we will just inject this service (Func<ISample>) into the dependency module, and finally we can execute it.



**SynchronizationContext**

SynchronizationContext is a representation of the current environment that our code is running in.

Simply put, SynchronizationContext represents a location "where" code might be executed. Delegates that are passed to its Send or Post method will then be invoked in that location. (Post is the non-blocking / asynchronous version of Send). Every thread can have a SynchronizationContext instance associated with it.

For the 99.9% use case, SynchronizationContext is just a type that provides a virtual Post method, which takes a delegate to be executed asynchronously (there are a variety of other virtual members on SynchronizationContext, but they’re much less used and are irrelevant for this discussion). The base type’s Post literally just calls ThreadPool.QueueUserWorkItem to asynchronously invoke the supplied delegate. However, derived types override Post to enable that delegate to be executed in the most appropriate place and at the most appropriate time.

Windows Forms will install a WindowsFormsSynchronizationContext on the thread on which the first form is created. (This thread is commonly called "the UI thread".) This type of synchronization context invokes the delegates passed to it on exactly that thread. This is very useful since Windows Forms, like many other UI frameworks, only permits manipulation of controls on the same thread on which they were created.

**WindowsFormsSynchronizationContext** (System.Windows.Forms.dll: System.Windows.Forms) Windows Forms apps will create and install a WindowsFormsSynchronizationContext as the current context for any thread that creates UI controls. This SynchronizationContext uses the ISynchronizeInvoke methods on a UI control, which passes the delegates to the underlying Win32 message loop. The context for WindowsFormsSynchronizationContext is a single UI thread.

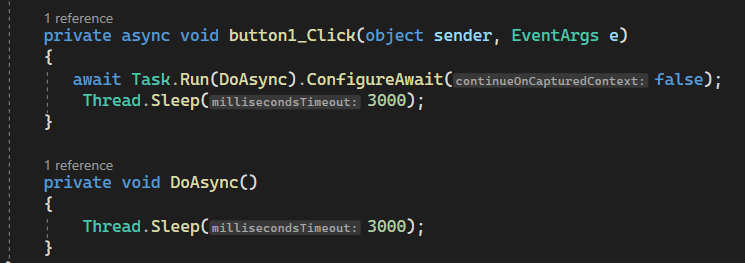
**DispatcherSynchronizationContext** (WindowsBase.dll: System.Windows.Threading) WPF and Silverlight applications use a DispatcherSynchronizationContext, which queues delegates to the UI thread’s Dispatcher with “Normal” priority. This SynchronizationContext is installed as the current context when a thread begins its Dispatcher loop by calling Dispatcher.Run. The context for DispatcherSynchronizationContext is a single UI thread. All delegates queued to the DispatcherSynchronizationContext are executed one at a time by a specific UI thread in the order they were queued. The current implementation creates one DispatcherSynchronizationContext for each top-level window, even if they all share the same underlying Dispatcher.

**Default (ThreadPool) SynchronizationContext** (mscorlib.dll: System.Threading) The default SynchronizationContext is a default-constructed SynchronizationContext object. By convention, if a thread’s current SynchronizationContext is null, then it implicitly has a default SynchronizationContext.

If synchronization context is not null the it means that context is captured.

**ConfigureAwait(bool continueOnCapturedContext)**

If the SynchronizationContext is not install in the enviroment, then when setting ConfigureAwait to false the remaining code will be run on the same context as the task was run on if set to true (default value) then SynchronizationContext will be the intalled one (WindowsFormsSynchronizationContext i.e.). 🡺

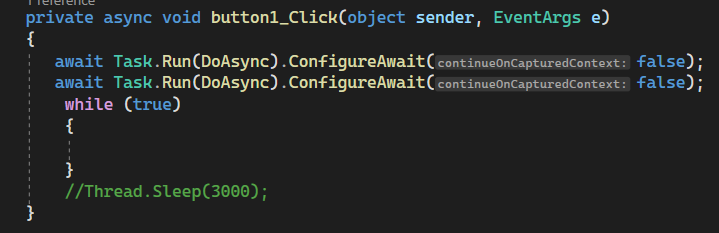


This is a windows forms app which means that the SynchronizationContext is install by the framework so ConfigureAwait will have an effect. Here, before Task.Run the threadId is 1 then inside that doAsync it is in another context, after completion since we set ConfigureAwait to false, the remaining code (Thread.Sleep which can freeze UI thread if run on Default context) will be run on the context of the DoAsync (threadPool context) which is good because we will not freeze the UI thread.

If we don’t set ConfigureAwait (then will use default value which is true) then it will continue on the UI context which means the app will be frozen.

So if we have something heavy after the await keyword that will take sometime to run such as getting something from the web, etc. then we should not continue on the main thread.

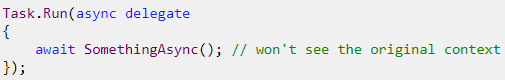
**Is it ok to use ConfigureAwait(false) only on the first await in my method and not on the rest?** In general, no. If the await task.ConfigureAwait(false) involves a task that’s already completed by the time it’s awaited (which is actually incredibly common), then the ConfigureAwait(false) will be meaningless, as the thread continues to execute code in the method after this and still in the same context that was there previously for example🡺



Here the second ConfigureAwait to false is meaningless since it is already in the ThreadPool context.

**Can I use Task.Run to avoid using ConfigureAwait(false)?**

Yes. If you write:



then a ConfigureAwait(false) on that SomethingAsync() call will be a nop, because the delegate passed to Task.Run is going to be executed on a thread pool thread. SynchronizationContext.Current will return null.

Further, Task.Run implicitly uses TaskScheduler.Default, which means querying TaskScheduler.Current inside of the delegate will also return Default. That means the await will exhibit the same behavior regardless of whether ConfigureAwait(false) was used.

**What async/await Does With SynchronizationContext Under the Hood**



It actually first checks the continueOnCaputre value of ConfugureAwait then checks the context if needed.

**No Await inside lock**

We cannot use await inside the lock block.

An await is used where the caller states that they know the call they are about to make may take some time and so prefers to give up the thread rather than block it. When the awaited call completes, the code after the await will execute but this may not be on same thread that called await. Therefore, the compiler cannot guarantee that the Monitor.Exit called on the closing brace of the lock statement will be on the same thread on which Monitor.Enter was called. This is the reason the compiler prevents the construct.

However, we can await something with semaphores 🡺

