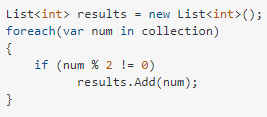
**Declarative and Imperative paradigms**

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:

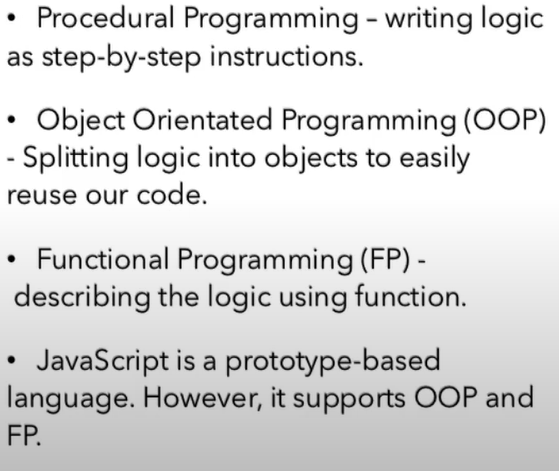


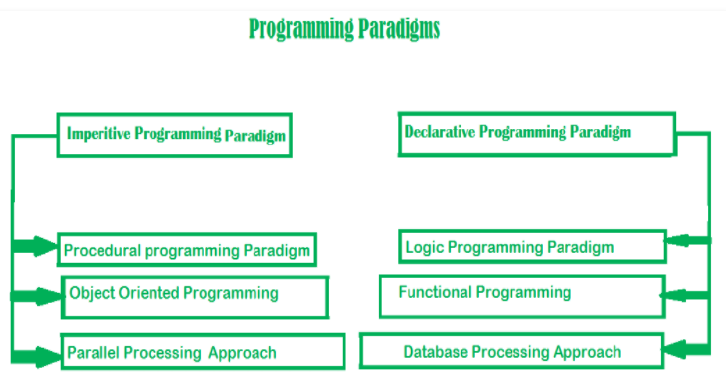
With declarative programming, on the other hand, you write code that describes what you want, but not necessarily how to get it (declare your desired results, but not the step-by-step):

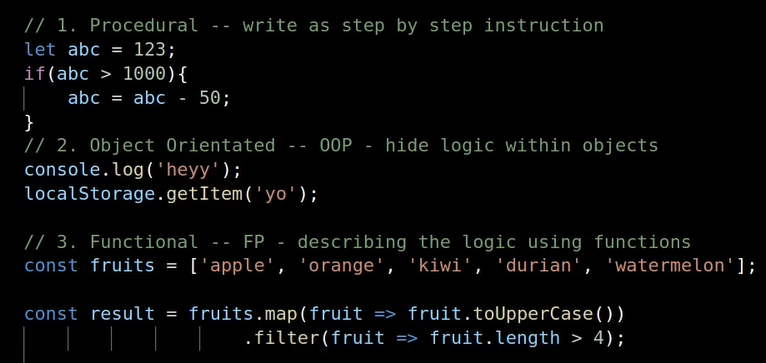


**Programming Paradigms**

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.





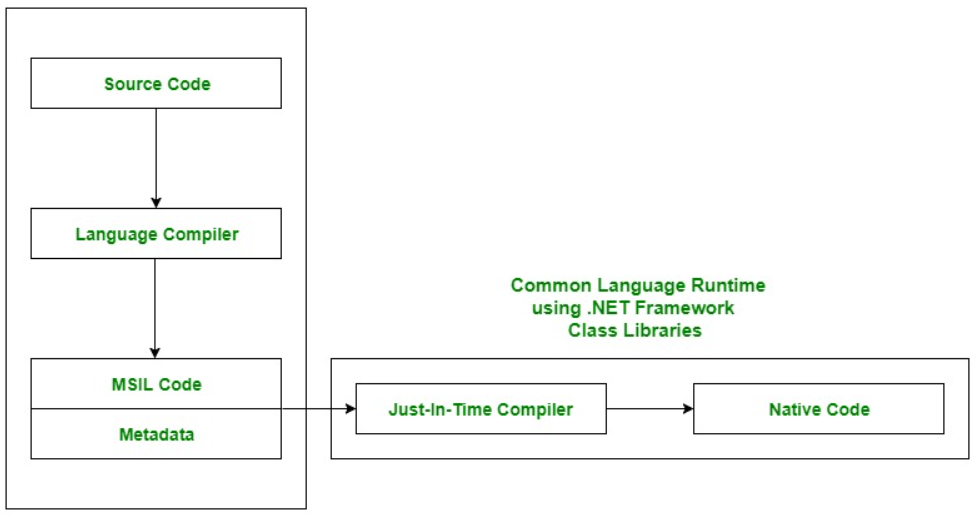


**CLR common language runtime**

.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.

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 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.

**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.

**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.

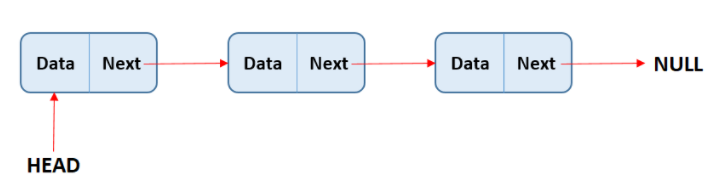
TimeSpan time = new TimeSpan(1,1,1);

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();

**Unary operations**

int x = 0;

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

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

**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 The return type, the name, and parameters.

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.



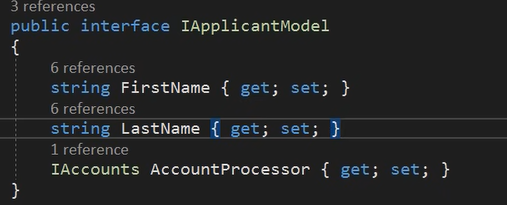
For example, here Employee class has 3 reasons to change. It can change if the CalcPay method fails, ReportHours or WriteEmployee. So It violated the SRP.

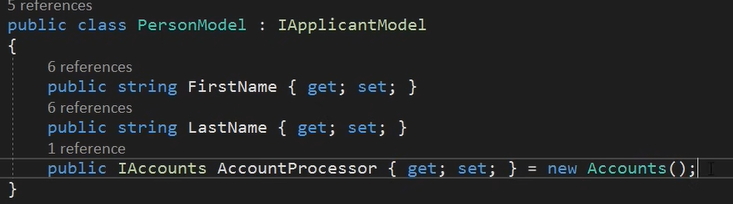


For instance, here this Program class has just one responsibility which is to control the flow of the application. And all the other responsibilities such as Capturing Person’s first name and last name or Messages(StandardMessage) are handled by other classes so it is their responsibility. So we should 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.

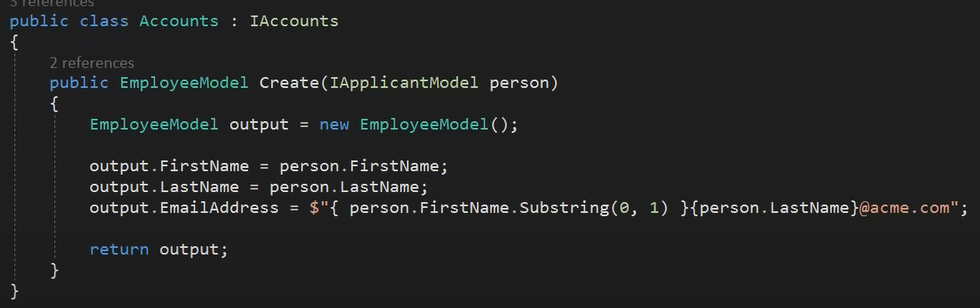
This principle aims to separate behaviours so that if bugs arise as a result of your change, it won’t affect other unrelated behaviours.

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 this models: Accounts, ManagerAccounts, ExecutiveAccounts. So to implement OCP we need to create a IApplicantModel which will be inherited by Person, ManagerModel, ExecutiveModel. And for accounts we will also create an interface (Accounts, ManagerAccounts, ExecutiveAccounts)🡪

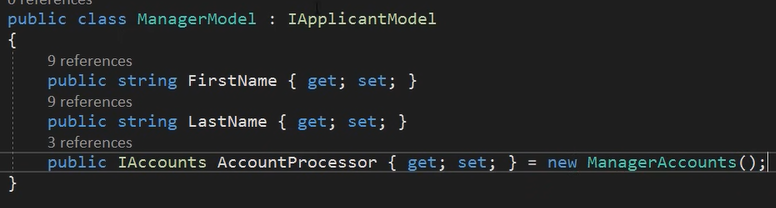




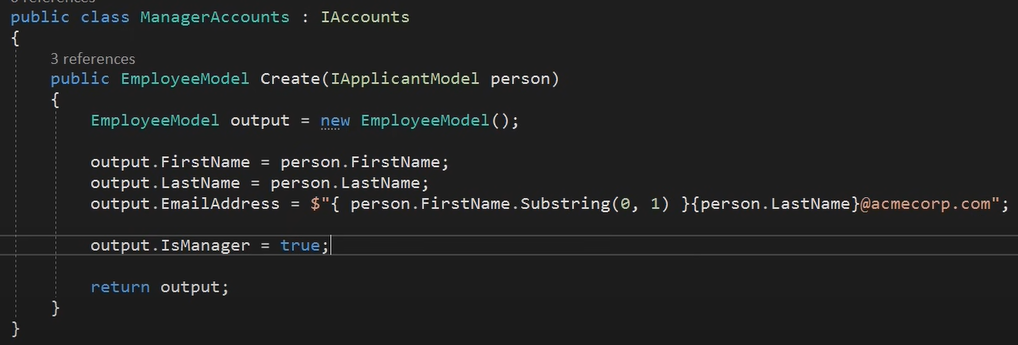
Accounts is a class for normal employees (that are just employees not managers or etc.).

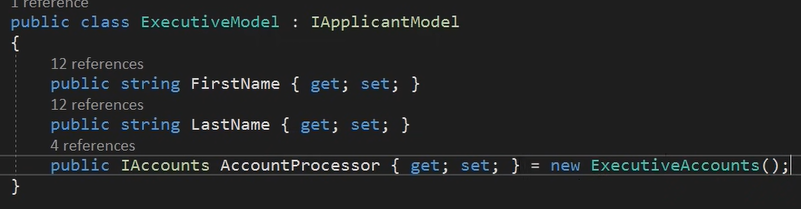


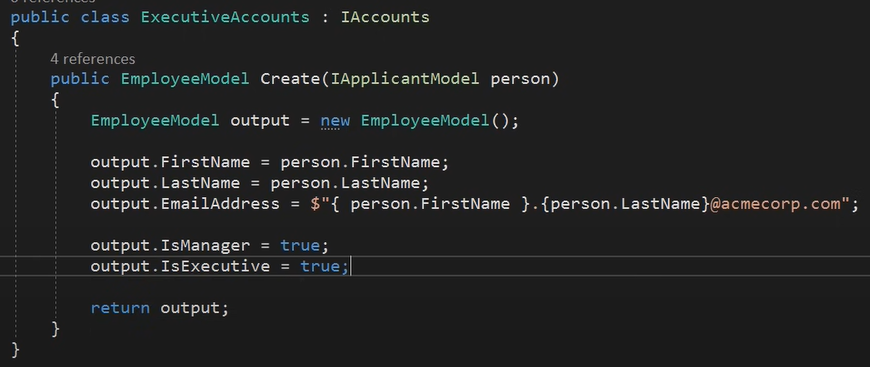
This is for normal employees.



This model is for Managers.



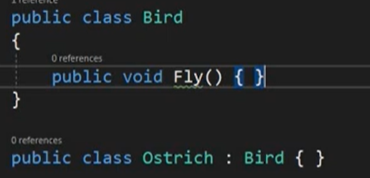




So in the end, if we we wanted to add a new model, for instance, TechnicianModel, then we would create a TechnicianModel class that would inherit from IApplicantModel and we would create Accounts for this Model (TechnicianAccounts) which would be somewhat different. And IAccounts for TechnicianAccounts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

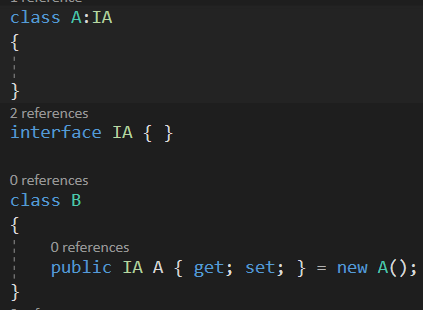
So the goal of this principle is basically providing a proper way for inheritance.

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).

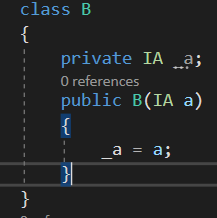
Dependency Inversion Principle - High-level modules should not depend on low-level modules. Both should depend on the abstraction (interfaces). - Abstractions should not depend on details. Details should depend on abstractions.

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.



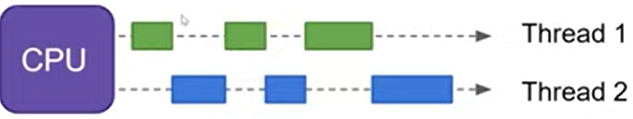
In Dependency injection we just pass that IA as as argument to the constroctor which then assigns that a to our private \_a. Nonetheless, at some point in our application we are gonna have to new up this class.

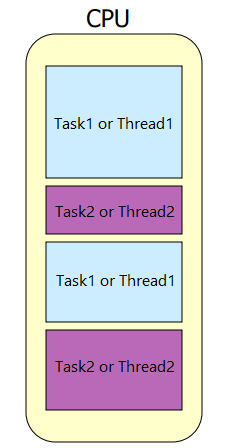
Dependency injection is one of the implementations of DIP.

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

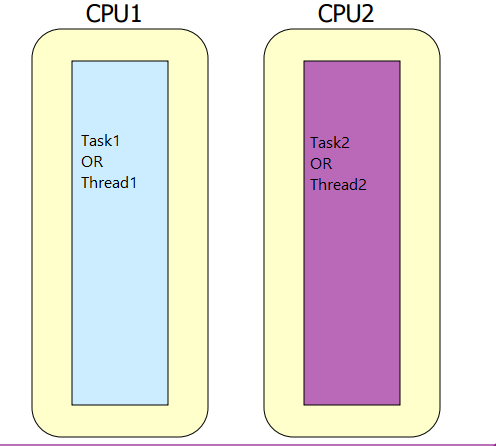




Paralellism is exactly what we think of this, doing multiple things at the same time independently.

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.



**Synchronous vs Asynchronous**

Synchronous or Synchronized means "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**

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

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.

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.

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 🡪

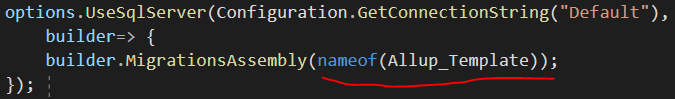


The output is Add.



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**

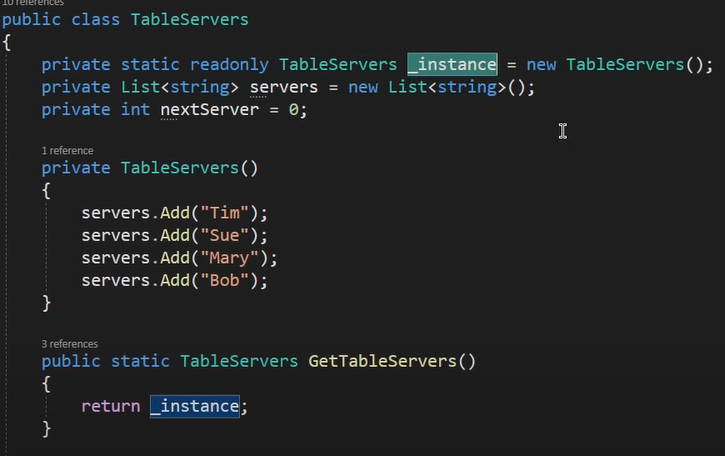
The QueryString collection is used to retrieve the variable values in the HTTP query string. The HTTP query string is specified by the values following the question mark (?), like this:

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

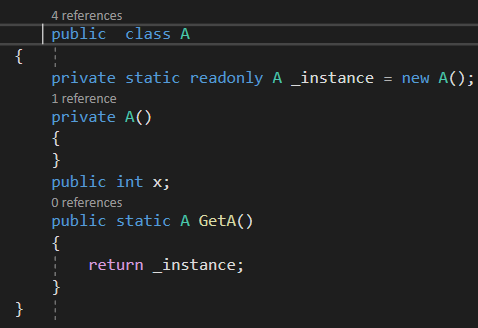
Query strings are also generated by form submission, or by a user typing a query into the address bar of the browser.

**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**

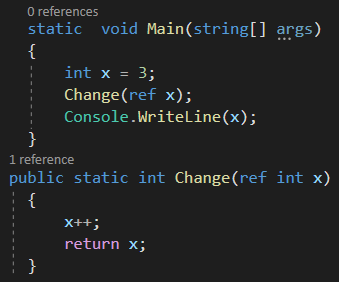
Serialization is the process of converting an object in memory into a stream of bytes so that it can either be able to be sent over the network or be stored in a persistent storage. Deserialization is exactly the opposite – Fetch a stream of bytes from network or persistence storage and convert it back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

We can convert objects to XML, JSON.

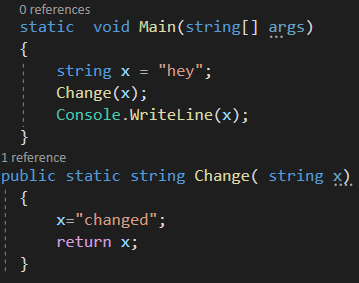
**The immutability of strings**

**Passing by reference vs Passing by value**

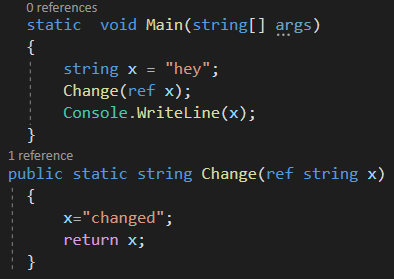
In the example shown below, the result is 4 because it references that integer. If there wasn’t the “ref” keyword then it would be 3.



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 🡪



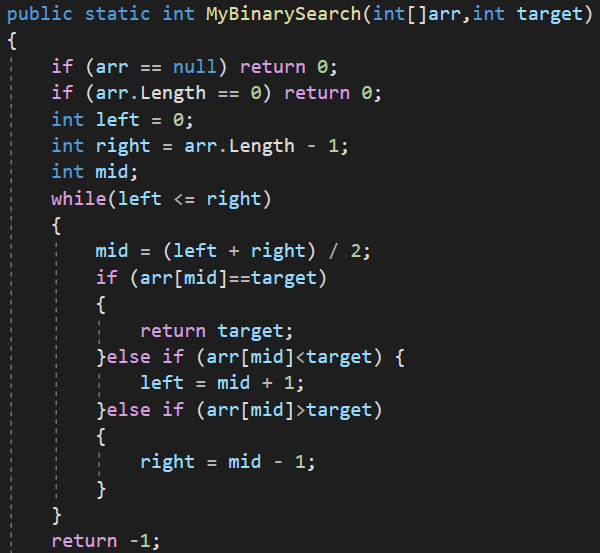
The result is “changed”.

**Managed code vs Unmanaged Code**

Managed is an environment where you have automatic memory management, garbage collectionm type safety and etc. Unmanaged code is everything else. So for example, .NET is a managed environment and C/C++ is unmanaged.

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**

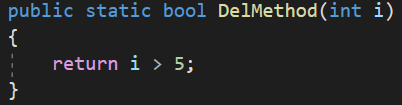
Expressions always return a value. Loops such as while, for loops and etc. also if’s, switches are all statements.

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

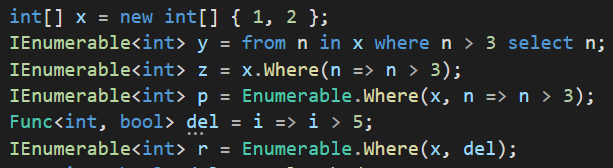


It converts it into some code 🡪

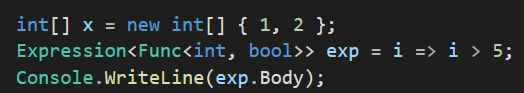




Also for example, with IEnumerables the same method is also applied🡪



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

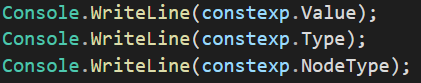


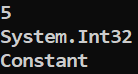


Our compiler takes that apart (dismantles it)🡪



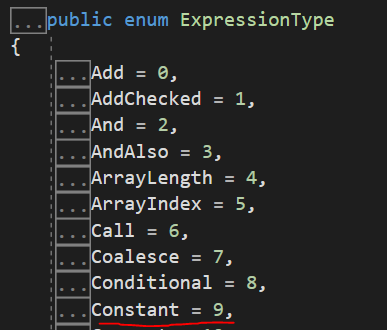
So the right part of our expression is 5 and it is a constant expression.





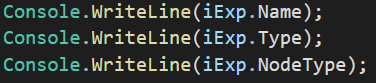
NodeType is an ExpressionType type which is an enum 🡪

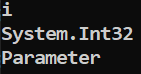




And the left side of our expression, which is “I” 🡪



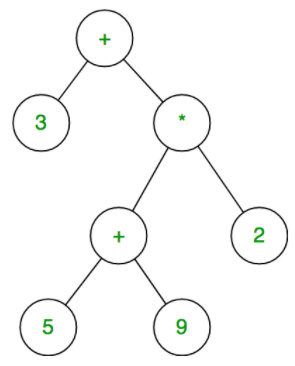




It doesn’t have a value! But it has a name.

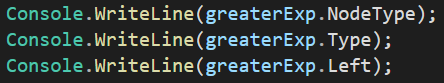
Now the GreaterThan expression combines the two expressions into a single expression tree so it is called a binary expression

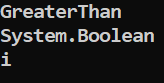
The expression tree is a binary tree 🡪



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.

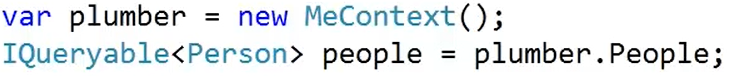
**IQuerable vs IEnumerable**

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

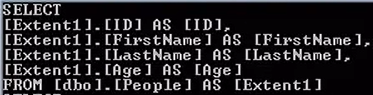
Differences: Enumerable is for collections such as lists also arrays. However, Querable class is for the entities of the database.

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 collections implement IEnumerable. But our DbSets extend IQuerable. So IQuerables are Entity Collections and IEnumerables are just System.Collections’ collections or arrays.

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

