**C# nədir**

C#(C Sharp) Microsoft tərəfindən .Net texnologiyası üçün inkşaf etdirilən modern, tam obyekt yönümlü(OOP) və type-safe bir proqramlaşdırma dilidir. C# C ailəsinə daxildir və Java kimi məşhur dillərə yaxındır. Ilk dəfə 2002-ci ildə çıxmışdır. C# datatype sensitivity dildir.

**Data type ranges (Verilənlərin tiplərinin aralıqları)**

byte (Yaddaşda tutduğu yer 1 byte) 0 to 256

sbyte (Yaddaşda tutduğu yer 1 byte) -128 to 127

short (Yaddaşda tutduğu yer 2 bytes) -32,768 to 32,767

ushort (Yaddaşda tutduğu yer 2 bytes) 0 to 65,535

int (Yaddaşda tutduğu yer 4 bytes) -2,147,483,648 to 2,147,483,647

uint (Yaddaşda tutduğu yer 4 bytes) 0 to 4,294,967,295

long (Yaddaşda tutduğu yer 8 bytes) -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

ulong (Yaddaşda tutduğu yer 8 bytes) 0 to 18,446,744,073,709,551,615

float(Yaddaşda tutduğu yer 4 bytes) vergüldən sonra 6 və ya 7 rəqəm üçün nəzərdə tutulub

double(Yaddaşda tutduğu yer 8 bytes) vergüldən sonra 16 rəqəm üçün nəzərdə tutulub

boolean(Yaddaşda tutduğu yer 1 byte) true və ya false

char (Yaddaşda tutduğu yer 2 bytes) Simvollar və ya onlara uyğun olaraq ASCII kodları

Unsigned– byte,ushort,uint,ulong

Signed– sbyte,short,int,long

Var keyword’ü ilə variable’a value mənimsədildiyi zaman compile olaraq datatype müəyyən edilir və biz həmin variable’a başqa bir yerdə fərqli datatype’dan bir value mənimsədə bilmərik. Bunu etsək compile time error verəcək.

Dynamic keyword’ü ilə variable’a hər dəfə müxtəlif datatype dəyər mənimsədilə bilər. Runtime olaraq datatype mənimsədildiyi üçün var keyword’dən fərqlənir.

Console.ReadLine(); ilə daxil edilən hər bir dəyər string type olaraq qayıdır.

Value type variable’ların default dəyəri 0’dır

Char type variable’ın default dəyəri 0’dır.

String type variable’ın default dəyəri null’dır.

**Arithmetic operators**(Hesab operatorları)

**/** - div bölmə adlanır , tam hissə götürülür

**%** - mod bölmə adlanır , qalıq hissə götürülür

**Increment** 🡪 ++

**Declerement** 🡪 --

int num = 10;

num++ && num-- 🡪 Postfix increment & decrement adlanır. Burada gedən proses özündən sonraki sətrə aid olur. Console.WriteLine(num++); yazarsaq ekrana **10** çıxacaq lakin artıq yaddaşda num **11** olaraq qalacaq və daha sonra num dəyişənini ekrana verərsək **11** kimi çıxacaq

++num && --num 🡪 Prefix increment & decrement adlanır. Burada həmin sətr daxilində artıq proses getmiş olur. Console.WriteLine(++num); yazarsaq ekrana **11** çıxacaq çünki artıq proses həmin sətrdə getmiş olacaq

**Assigment Operators**(Mənimsətmə operatorları)

a += b 🡪 a = a + b

a -= b 🡪 a = a - b

a \*= b 🡪 a = a \* b

a /= b 🡪 a = a / b

a %= b 🡪 a = a % b

**Comparison Operators**(Müqayisə operatorları)

Müqayisə operatorları bizə true və ya false dəyər qaytarır

== 🡪 bərabərdirmi?

!= 🡪bərabərliyin əksi, bərabər deyilmi yəni not equals

> 🡪 böyükdürmü

< 🡪 kiçikdirmi

>= 🡪böyük bərabərdirmi

<= 🡪kiçik bərabərdirmi

Bu operatorlar bizə boolean dəyər qayatırlar. Və əsasən if şərti daxilində və ya loop’larda istifadə olunurlar.

**Logical Operators**(Məntiqi operatorlar)

&& 🡪 and(və) deməkdir. Işləndiyi bütün şərtlər true olduğu halda true dəyər qaytarır

|| 🡪 or(və ya) deməkdir. Işləndiyi şərtlərin ən azı 1-I true olduğu halda true dəyər qaytarır

! 🡪 not(əks) deməkdir. Işləndiyi şərt true olarsa false , false olsarda true dəyər qaytarır.

**Condition**

**If Statements**

If(condition) {

Statement;

}

Statement2;

Bu ifadəyə if-then ifadəsi deyilir. Əgər verilən şərt doğru olarsa if daxilindəki ifadə yerinə yetirilir və daha sonra 2-ci ifadə yerinə yetirilir. Əks təqdirdə sadəcə 2-ci ifadə yerinə yetirilir.

If(condition) {

Statement;

} else{

Statement2;

}

Bu ifadəyə if-then-else ifadəsi deyilir. Əgər verilən şərt doğru olarsa sadəcə if daxilindəki ifadə yerinə yetirilir əks təqdirdə sadəcə 2-ci ifadə yerinə yetirilir.

if(condition) {

Statement;

} else if(condition2){

Statement2;

} else if(condition3){

Statement3;

} else{

StatementN;

}

Else-if ifadəsi müəyyən bir şərt ödənmədikdə verilə biləcək digər şərtə uyğun prosesin işə salınması üçün istifadə edilir

**Switch case**

Bizim təyin etdiyimiz dəyişənin müxtəlif qiymətlərə və ya dəyərlərə bərabər olma halı yoxlanılırsa if statement yerinə switch case’dən istifadə edilir. Əgər bizim dəyişənimizə heç bir case halı uyğun olmazsa default daxilindəki əməliyyatlar işə düşür.

switch(expression) {

case value1:

Statement;

break;

...

case valueN:

StatementN;

break;

default:

StatementDefault;

break;

}

**Switch case expression**

datatype variable = expression switch {

Value1ForExpression => value,

...

ValueNForExpression => value,

\_ => defaultValue

};

Switch case expression hər hansısa dəyişənə switch case’dən daha rahat və qısa yazılış ilə dəyər mənimsətmək üçün istifadə edilir. Default ‘\_’ simvolu ilə işarələnir.

Biz switch case expression vasitəsi ilə həmçinin müəyyən şərtləri aşağıdaki şəkildə yoxlaya bilərik.

Value1ForExpression => value əvəzinə, datatype variable2 when statement => value yazaraq əgər statement bizə true dəyərini qaytararsa variable’a value’nu mənimsədir.Burada variable2 bizim expression kimi göndərdiyimiz dəyərə bərabər olur və statement daxilində istifadə etmək üçündür.

>Value1ForExpression => value və ya <Value1ForExpression=> value bu yazılış isə o deməkdir ki expression olaraq göndərilən dəyər Value1ForExpression expression’dan böyükdürsə və ya kiçikdirsə value’a mənimsədilsin.

**Loop Statements**

Loop(dövr) qarşıya qoyulmuş məsələni müəyyən şərt daxilində müəyyən say qədər yerinə yetirmək üçün istifadə edilir.

**While**

while(condition) {

Statement;

}

Şərtimiz doğru olduğu müddətcə blok daxilindəki ifadə yerinə yetiriləcək.Bu da o deməkdir ki ilk olaraq hər dəfə gedib şərt yoxlanılacaq və true dəyər qayıdarsa prosess yerinə yetiriləcək.Bu səbəbdən While’a həm də ön şərtli dövr operatoru deyilir.

**Do While**

do{

Statement;

}while(condition)

Do while ilə while’ın fərqi odur ki, do while zamanı şərt yoxlanılmadan ilk başda proses 1 dəfə yerinə yetirilir daha sonra şərt yoxlanılır. Bu da o deməkdir ki verilən condition false olsa, while loop’da proses heç vaxt yerinə yetirilməyəcək amma do while’da isə proses 1 dəfə yerinə yetiriləcək. Bu səbəbdən Do While’a həm də son şərtli dövr operatoru deyilir.

**For**

for(dataType variable; condition; process1){

Process2;

}

For loop’u 3 hissəyə bölünür 1-ci hissə dəyişən təyin etmək üçün, 2-ci hissə şərt vermək üçün və sonuncu hissə müəyyən bir əməliyyatı yerinə yetirmək üçün istifadə olunur.Şərt ödəndiyi bütün hallarda For loop’un bloku daxilindəki proses işə düşür.

for(;;){

Process;

}

Bu yazılış infinite for loop adlanır. Müəyyən şərt daxilində loop break olunmazsa sonsuzadək davam edəcək.

**Foreach**

foreach(var item in collectionName) {}

Foreach loop’u vasitəsilə biz hər hansısa array’in, collection’un bir sözlə desək iterasiya oluna bilən hər bir object’ə tətbiq edilə bilər. Foreach vasitəsi ilə object’in hər bir elementinə 1-1 müraciət edəbilərik. Bir sıra situasiyalarda foreach bizim köməyimizə çatır və for loop’undan daha sürətlidir.C# 8’dən sonra gəlib.

**Array**

DataType[] arrName = new DataType[Length]; və ya {value&variable1,...value&variableN}; və ya new DataType[Length] {value&variable1,...value&variableN};

Index’lənmə 0’dan başlayır və sonuncu index’də duran elementin indexi arrName.Length-1 olur bunu həm də ^1 şəklində yaza bilərik..Array’lər resizeable deyildir bu səbəbdən biz ölçünü yalnız array’i yaradan zaman təyin edəbilərik və bir də dəyişə bilmərik. Array daxilindəki elementləri

New DataType[0] yazsaq bu dataType’dan olan array yaddaşda yaranacaq, yer tutacaq lakin daxilində heç bir element saxlamayacaq.

**Keywords**

Break keyword’ü müəyyən bir şərt daxilində dövrü sonlandırmaq üçün və ya switch’dən çıxmaq üçün istifadə olunan bir ifadədir.

Continue keyword’ü vasitəsi ilə biz müəyyən şərt daxilində işləndiyi sətrdən sonra ki əməliyyatlar yerinə yetirilmədən loop’un əvvəlinə qayıtmaq üçün istifadə edilir.

Continue keyword’dən fərqli olaraq break keyword’ü loop’la yanaşı switch daxilində də işlənilir

Return keyword’ü işləndiyi yerdə daxilində yazıldığı methodu dəyandırır və özündən sonra gələn code sətrlərini işə salmır. Əgər method geriyə bir dəyər qaytarırsa return’ün qarışısında o type’dan dəyər yazılmalıdır.

Is null keyword’ü vasitəsi ilə biz == null şərtini yoxlayıb geriyə true(əgər null olarsa) və ya false( əgər null’dan fərqli olarsa) dəyəri alırıq.C# 7’dən sonra gəldi.

Is not null keyword’ü vasitəsi ilə isə is null keyword’ün əksini yerinə yetirmiş olarıq. C# 9’dan sonra gəldi

??=value; keyword’ü vasitəsi ilə biz == null şərtini yoxlayıb əgər geriyə true(əgər null olarsa) dəyəri gələrsə şərt yoxlanılan variable’a uyğun dəyər mənimsətmək üçün istifadə edirik.

Datatype variable = variable2?? value; əgər variable2 null deyilsə value’nu variable’a mənimsətmək üçün bu cür yazılış istifadə edilir.

**Reference & Value type**

Reference type’ların ünvanı(referansları) stack yaddaşda özləri isə heap yaddaşda saxlanılır.Stack yaddaş heap yaddaşdan daha sürətlidir çünki burada datalar ardıcıl yaddaşda saxlanılır.Hər bir şey stack’dən oxunur.

Bütün number type’lar , char və bool type’lar,enumlar yəni struct reference olanların hamısı value type adlanır.

Objectlər,String’lər, array’lər delegate’lər interface’lər yəni Class reference olanların hamısı reference type adlanır.

Reference type dəyişənlərə hər hansısa bir method daxilində dəyişiklik olunarsa method işə düşdükdən sonra heap yaddaşda həmin dəyişəndəki dəyişiklik saxlanılacaq

Reference type variable’ların bərabərliyi yoxlanan zaman onların saxladığı dəyər yox, yaddaşda tutduqları ünvan’ın bərabərliyi yoxlanılır.Yalnız string’lərdə bu fərqli olur bunun səbəbi string type variable’lar immutable’dır yəni dəyişilməzdir və string pool’da yaranan zaman verilən dəyər axtarılır əgər string pool’da həmin dəyərə bərabər hər hansısa string variable varsa gedib həmin ünvana yeni reference mənimsədilir əgər yoxdursa yeni string type variable yaradır.

**String Builder**

String type’lardan fərqli olaraq StringBuilder type mutable type’dır. String type’ların istifadəsi zamanı əgər variable üzərində hər hansısa dəyişiklik aparılarsa öz köhnə dəyəri string pool’da qalır yeni string type variable yaranaraq yeni dəyər ona mənimsədilir. Bu da yaddaşda lazımsız məlumatların saxlanılmasına gətirib çıxartdığı üçün Garbage Collector lazım olmadan işə düşür. Bunun qarşısını almaq üçün StringBuilder type’dan istifadə edilir.

**Ref & Out**

Method daxilində value type’da edilən dəyişiklik ümumi olaraq da(stack yaddaşda) dəyişilməsi üçün ref və out keyword’dan istifadə edilir.Hər zaman value type üçün istifadə edilmiş həmçinin reference type’lar üçün də istifadə edilə bilər.

Method parametr olaraq qəbul etdiyi dəyişənin əvvəlinə ref və ya out yazılır və arqument olaraq göndəriləndə də ref və ya out keyword’ü ilə göndərilir.

**Ref & Out fərqi**

Out keyword’ü istifadə etdikdə variable’a əvvəldə dəyər mənimsədilməsə də olar amma method daxilində mütləq variable’a dəyər mənimsədilməlidir.Ref keyword’dən istifadə zamanı isə mütləq variable’a əvvəldə dəyər mənimsədilməlidir yəni unassign olan dəyər göndərmək olmaz.

**Methods / Functions**

Funksiya sərbəst yazıla bilən, dəyərlər göndərilib call oluna bilən bir code block’dur. Məqsədi isə Don’t Repeat Yourself(DRY principe) prinsipini qorumaqdır. Biz hərhansısa bir code block’nu 2 və ya daha çox dəfə istifadə etsək o zaman həmin code block’nu funciton(method) halına salmaq lazımdır.

Hər hansısa class daxilində yəni hansısa type’a aid yazılan funksiya method adlanır.

Method və funksiya yerləşdiyi yerə və aid olmasına(Type’a) görə fərqlənir.Lakin hər 2’si də eyni işi yerinə yetirir.

**Method’un yazılışı**

accessModifier static&non-static returnType MethodName(parameters) {}

Method’un qəbul etdiyi dəyər parameter, method’a göndərilən dəyər isə argument adlanır.

ReturnType’ı olan method bütün statement’lərdə geriyə dəyər qaytarmalıdır.Əgər returntype’ı yoxdursa geriyə dəyər qaytarmır və yerinə void yazılır.Bu method void method adlanır.

**Method Optional Parameters**

Method’un qəbul elədiyi parametrə default dəyər vermək o deməkdir ki , default dəyəri parametrə set eləsin. Method call olunan zaman əgər default dəyər verilən parametrə hər hansısa bir arqument(dəyər) göndərilməsə default olaraq verilən dəyəri saxlasın və method daxilində istifadə eləsin.Əgər arqument göndərilərsə göndərilən arqumenti set eləsin və method daxilində arqumenti istifadə eləsin,yəni default dəyəri override eləsin.

Məsələn:

Public static void TestMethod(int param1 , int param2 = 5) {}

Biz TestMethod’u call edən zaman 2 arqument də göndərə bilərik 1 arqument də. Əgər 1 arqument göndərsək param2’yə set elədiyimiz default dəyər method daxilində istifadə ediləcək, 2 arqument göndərsək default dəyər override olunacaq və göndərilən arqument istifadə ediləcək.

Method‘un istənilən sayda eyni type‘dan dəyər qəbul etməsi üçün params keyword’dən(optional parameter) istifadə edilir.Göndəriləcək olan dəyərlərin type’na uyğun olaraq həmin type’dan array parametrinin qarşısına params keyword’ü yazırıq.

Məsələn:

Public static void TestMethod2(params dataType[] arr) {}

Biz methodu çağıran zaman istənilən qədər verilən dataType’dan dəyər göndərə bilərik və sonra göndərilən dəyərlər parametr olaraq qəbul olunan array’in daxilinə yığılacaq.Və daxildə array’in elementləri kimi istifadə olunacaq

Optional parametr olaraq verilən parametrlər tələb olunan parametrlərdən sonra yazılmalıdır. Yəni optional parametr’in istifadə ardıcıllığı sonuncudan başlayaraq sağdan sola doğrudur.

**OOP**

**Class və object məntiqi**

Class ramda yer tutmur. Biz bir class yaratdıqda yeni type yaratmış oluruq və bu typların objecti yarana bilər. Class‘dan instance alınaraq object yaranır və bu object Class‘ın adını özünə type olaraq götürür. Class daxilində static olmayan hər şey objectə aid olduğu üçün bir başa heap yaddaşda saxlanılır.

**Anonym Object**

Anonym object dedikdə var keyword‘ü ilə yaranmış objectlər başa düşülür. Var keyword‘ü yazılmasa anonym object yarana bilməz.Biz Anonym object‘ləri class yaratma ehtiyyacımız olmadıqda, hər hansısa objectən yalnız bir dəfə istifadə etməli olduğumuz hallarda yaradırıq.

var objectName = new

{

Field1 = value,

...

FieldN = value

};

**Constructor**

Constructor yalnız object instance alan zaman işə düşən, return type‘ı olmayan method‘dur. Biz Class daxilində yazsaq da, yazmasaq da boş Constructor(parameterless) yəni heç bir dəyər qəbul etməyən Constructor compile zamanı əlavə edilir.Biz Constructor‘u method signature’ə riayət edərək overload edə bilərik.Constructor‘un adı daxilində yazıldığı Class‘ın adı ilə eyni olmalıdır.

Hər hansısa bir Constructor işə düşən zaman, Class daxilindəki hər hansısa başqa bir Constructor‘u da işə salmaq istəsək :this(overload) yazaraq işə sala bilərik.

Adətən constructor‘un constructor‘u çağırması field‘lara default dəyərlər set olunması üçün istifadə edilir.

Hər hansısa bir Constructor işə düşən zaman onun miras aldığı Class‘ın daxilindəki constructor‘u işə salmaq istəsək :base(overload) yazaraq işə sala bilərik.

**Object Oriented Programing(OOP) ‘in 3 əsas prinsipi vardır.**

* Inheritence
* Polymorphism
* Encapsulation

**Inheritence**

Inheritence anlayışı bir class‘ın digər class‘dan “miras” alması deməkdir.Burada miras anlayışı dedikdə, sub class‘ın super class‘daki method‘lara,property‘lərə & field‘lara yəni Class members‘ə müraciət edə bilməsinə, onları istifadə edə bilməsinə icazə verilməsi deməkdir.

Əgər miras alınan Class‘ın boş constructor‘u yoxdursa onun overload halı varsa subClass’dan instance alınan zaman yəni öz constructor‘u işə düşən zaman :base(overload) constructor da işə salınmalıdır.

**Is a & has a**

“Is a” anlayışı o deməkdir ki, əgər “developer is a person” =true, o zaman developer class‘ı person class‘dan miras ala bilər.

“Has a” anlayışı o deməkdir ki, əgər “developer has a -pc” =true, o zaman pc object‘i developer class‘ın property & field‘ı ola bilər.

**Encapsulation**

Proyektimizdə hər hansısa class daxilində bizim hər hansısa object‘imizin hər hansısa Field‘a bir başa müraciət edilə bilməsin, göndərilməməsi gərəkən hər hansısa bir dəyər set olunmasın deyə Encapsulation məntiqindən istifadə edilirərək bizim Field‘ımız kapsullanır.Bunun üçün biz Field‘ımızı private edirik və bununla yanaşı yeni bir Property yaradırıq. Və bizim verəcəyimiz şərtlər daxilində də bu Property‘lərə dəyər set elemək mümkün olmayacaq.

**Set & Get**

Set və Get anlayışı o deməkdir ki, hər hansısa bir object‘in hər hansısa field‘a & property‘ə value assign etmək set etmək deməkdir, object‘in hər hansısa field‘ın & property‘in dəyərini götürmək isə get etmək deməkdir.

This keyword‘ü vasitəsi ilə biz olduğumuz class daxilində onun objectini çağırmış oluruq.Biz class daxilində hər hansısa static olmayan property‘ə & field‘a müraciət etsək arxa tərəfdə this.property & this.field kimi işləyir.

Obj.property – get eləmək deməkdir.

Obj.property = value – set elmək deməkdir.

**Property nədir?**

Property, get və set method‘ları olan Field adlanır. Təyin olunan Field‘in qarşısında { get; set;} yazılır və istədiymiz şəkildə get və set daxilində dəyişikliklər edə bilərik.

Class member‘lərin access modifier‘ı olduğu kimi Property‘lərin get və set method‘larının da access modifier‘ı vardır.

**Readonly**

Biz hər hansısa bir Property‘nin set method‘nu sildiyimiz zaman bu Property readonly Property adlanır.

Biz hər hansısa bir Field yaranan zaman access modifier‘dən sonra readonly keyword‘ü yazsaq bu Field Readonly Field adlanır.

Readonly Property & Field’a yalnızca Constructor daxilində və Property & Field təyin olunan yerdə dəyər set oluna bilər,başqa heç bir yerdə dəyər set oluna bilməz.

**Const**

Biz hər hansısa bir Field yaranan zaman access modifier’dan sonra const keyword‘ü yazsaq bu Field constant olur.Bu da o deməkdir ki biz bu Field‘a yalnızca təyin olunan zaman dəyər set edə bilərik , başqa heç bir yerdə dəyər set oluna bilməz. Yalnızca Field‘lara aid olur

**Polymorphism**

Polymorphism- yunan sözü olub çox biçimlilik deməkdir.

Bildiyimiz kimi miras alınmış class’ın hər-hansısa bir method’un override edilməsi, yəni method’un gövdəsinin özünə uyğun olaraq dəyişdirilməsinə override deyilir, bu halda eyni adda birdən çox method olduğu üçün bu hadisə Polymorphism adlanır. Override ilə yanaşı Overload da Polymorphism anlayışına daxildir. Polymorphism 2 yerə bölünür Dinamik və Statik Polymorphism.Hansı funksiyanın işə düşəcəyinə compile zamanı qərar verilirsə(Method Overloading) bu Statik(Compile Time) Polymorphism adlanır.Hansı funksiyanın işə düşəcəyinə run time zamanı qərar verilirsə bu Dinamik(run time) Polymorphism adlanır.

**Method Signature**

Method signature, method’un adı method’un qəbul etdiyi parametrlərin sayı və method’un qəbul etdiyi parametrlərin type’na deyilir. Eyni class daxilində method signature’ləri eyni olan 2’ci bir method yaratmaq olmaz(compile error verir).

**Parametr və arqument**

Method’un qəbul elədiyi dəyər parametr, method call olunan zaman ona göndərilən dəyər isə arqument adlanır.

**Method Overloading**

Method overloading o deməkdir ki Class daxilində eyni adda olub, Method signature gözlənilərək yaradılan methodlar overload method adlanır.Yəni methodların adı eyni, qəbul etdiyi parametrlər‘in sayı və ya type‘ı fərqli olarsa bu overloading adlanır.

Method overloading static polymorphism adlanır. Və Compile zamanı baş verir.

Overload method‘larda qəbul edilən parametrlərə optional parametr versək method call olunan zaman ən uyğun variant işə düşəcək

Public static void printStr(string str) => Console.WriteLine(str);

Public static void printStr(string str, string str2 = ”default value”) => Console.WriteLine(str);

Public static void printStr(params string[] str) => Console.WriteLine(str);

printStr(“testStr”); method‘u call olunan zaman görünən odur ki hər 3 overload halına da uyğundur, lakin ən uyğun variant sadəcə 1 parametr qəbul edən variantdır, optional parametr olan variantlar yox.

**Struct**

C#’da Reference type’lar class, Value type’lar isə struct’dır. Struct’lar interface’i implement edə bilər, lakin inheritence məntiqi yoxdur

**Access Modifiers**

Bütün Class’ların default Access Modifier’ı internal’dır.

Bütün Class member’ların default Access Modifier’ı privete’dır, yalnız override edilməmiş boş Constructor’dan başqa.

Public – bütün class’lara və class member’lərə aid ola bilər.

Public Access Modifier verilən class və ya member , eyni solution daxilində digər bütün project’lərdən əl çatan olur.

Internal – bütün class’lara və class member’lərə aid ola bilər.

Internal Access Modifier verilən class və ya member , yalnız verildiyi project’də əl çatan olur.

Protected – bütün class member’lərə aid ola bilər.

Protected Access Modifier verilən member , yalnız verildiyi class daxilində və verildiyi class’dan miras alan class’lar daxilində əl çatan olur.

Private – bütün class member’lərə aid ola bilər.

Private Access Modifier verilən member, yalnız verildiyi class’ın daxilində əl çatan olur.

**Generic type**

Eyni işi görən lakin fərqli type‘lar qəbul edən method‘lar və s üçün təkrarın qarşısını almaq üçün istifadə edilir.

**C# DataBase Connection**

C# vasitəsi və DataBase arasında əlaqə qurmağın 2 yolu vardır, ADO(ActiveX Data Object).Net və ORM(Object Relational Mapping).

ORM , ADO .Net’dən daha üstündür. Çünki ADO .Net’də bütün prosesin idarə olunması bizdə olur.Biz SQL’i hard code(sabit kod) olaraq yazırıq və bu səbəbdən Security hissəsini də biz fikirləşməli və qarşısını almalı oluruq, Connection’u açıb bağlamalı oluruq. ORM isə bizə hazır function’lar və hazır security verir.

**ADO .Net**

DataBase ilə Connection qurmaq üçün ConnectionString’ə ehtiyyac var. ConnectionString daxilində server adını, database adını və əgər DataBase local’dırsa Trusted\_Connection = True , yox əgər DataBase remote DataBase’dirsə userid və password gözləyir.

Localhost’a qoşulmaq üçün:

string ConnectionString = “Server = serverName ; Database = dataBaseName ; Trusted\_Connection = True”

Remote olaraq qoşulmaq üçün:

string ConnectionString = “Server = serverName ; Database = dataBaseName ; userid=id;password=password”

SqlClient Package ilə gələn SqlConnection class’dan instance alınır və constructora arqument olaraq ConnectionString göndərilir. Bunun səbəbi SqlConnection object’nə hansı DataBase qoşulacağını bildirməkdir.

Connection’u yaratmaq üçün SqlConnection object’nə .Open() method tətbiq edilir. Yaranan connection close olunmalıdır. Connection’u close etmək üçün .Close() method’u tətbiq edilir.

Connection Class’ı DbConnection class’nı miras alır və DbConnection Class’ı IDisposable interface’ni implement edir.Bu da o deməkdir ki close olunduqdan sonra həmçinin .Dispose() method’u da işə salınmalıdır. Dispose , object’i Garbage Collector’dan silmək üçün istifadə edilir. Biz bu method’ları istifadə etməmək üçün Using daxilində SqlConnection’dan instance alırıq. Using code block’undan sonra həm connection close olur həm də dispose. Lakin Using code block’u daxilində biz connection’u open etməliyik.

Icra olunacaq query’ni string type dəyişən olaraq göstəririk. Bu query’ni icra etmək üçün SqlCommand class’ından instance alırıq və constructora həm icra ediləcək olan query’ni , həm də connection’u göndəririk.

string connectionString = "Server = serverName; Database = dataBaseName;Trusted\_Connection=True";

using(SqlConnection connection = new(connectionString)){

connection.Open();

string query = "SQL Query";

SqlCommand command = new SqlCommand(query, connection);

}

Insert, update və delete əməliyyatlarını icra etmək üçün SqlCommand object’nə .ExecuteNonQuery() method’u tətbiq edilir. Bu method geriyə int type olaraq “Number of row effected” qaytarır. “Number of row effected” bizim insert,update və delete üçün yazılan query’lər icra edildikdən sonra ekrana çıxan ədəddir.

Select əməliyyatlarını icra etmək üçün SqlCommand object’nə .ExecuteReader() method’u tətbiq edilir və geriyə SqlDataReader type dəyər qaytarır. Biz hər hansısa datanın qayıdıb qayıtmamasını yoxlamaq üçün .HasRows() methodundan bu şəkildə reader.HasRows() istifadə edirik və geriyə Boolean dəyər qaytarır.

Əgər .HasRows() method’undan true dəyər qayıdarsa, SqlDataReader object’i while daxilində bu şəkildə while(reader.Read()) .Read() method’na salırıq. Və geriyə hər dəfə true dəyəri qayıtdıqca SqlDataReader class’ın object’i row’ları ifadə edir. row’ları istifadə edə bilmək üçün SqlDataReader class’ın indexer’ini istifadə edirik. Bu indexer int dəyərindən index və ya string dəyərindən column name qəbul edir.Biz istəyə uyğun olaraq bu indexer’lərdən birini istifadə edirik. Geriyə qayıdan cavab isə bizim daxil etdiyimiz dəyərə görə object type datanı qaytarır.Bizə lazım olan type’a cast edib istifadə edə bilərik.

Select əməliyyatı zamanı yalnız 1 data qayıdarsa biz .ExecuteReader() method’u yox, .ExecuteScalar() method’u işə salınır və geriyə 1 object qaytarır.

**SQL Injection**

Biz yazdığımız query daxilində göndərdiyimiz dynamic dəyərə hər hansısa command yazılıb database’ə zərər vurulmasın yəni SQL Injection baş verməsin deyə göndərəcəyimiz dəyəri bu şəkildə göstərəcəyik.

Göndəriləcəy dynamic dəyər’i işarə ediləcək bir parametr yazılır və qarşısına @ işarəsi qoyulur daha sonra command.Parameters.AddWithValue(“@parameter”,dynamicValue); method’u işə salınır.Bu method vasitəsi ilə yuxarıda göndərəcəyimiz dəyəri işarə etdiyimiz parametr’ə dəyər mənimsədilir və string ifadə olaraq başa düşülür bu səbəbdən heç bir şəkildə SQL Injection halı baş verə bilmir.

string query = $"SELECT \* FROM Employee WHERE NAME LIKE '%{name}%'"; əvəzinə

string query = $"SELECT \* FROM Employee WHERE NAME LIKE '%' + @name + '%'"; yazılır və daha sonra command object’inə command.Parameters.AddWithValue("@name", name); yazılır.

**ORM(Object Relational Mapping)**

ORM’i istifadə edə bilmək üçün EntityFrameWorkCore package’ni yükləyirik. Çünki bu package daxilində gələn DbContext class’ı bizə lazımdır.Hansı SQL ilə işləyəcəyiksə(SqlServer) onun özünə aid package’ni yükləməliyik.Migration Command’ları üçün isə EntitiyFrameWorkCore.Tools package’ni yükləyirik.

AppDbContext class’ı yaradırıq və bu class DbContext class’nı miras almalıdır. Miras aldığımız Class’ın .OnConfigure() method’nu override edirik.

OnConfigure method’un qəbul etdiyi parametr vasitəsi ilə .UseSqlServer() method’nu işə salırıq. .UseSqlServer() method’na arqument olaraq ”Server = serverName ; Database = dataBaseName ; Trusted\_Connection = True” string ifadəni göndəririk yəni ConnectionString göndəririk.

CodeFirst işlədiyimiz üçün yazdığımız DataBase adı daha öncədən var olmamış bir DataBase olacaq.Və yaradacağımız hər entity class’ı DataBase daxilində bir table olacaq.Yaratdığımız class’ların mütləq Id propertysi olmalıdır.Property adı classNameId və ya Id olaraq yazılmalıdır və özü avtomatik olaraq PRIMARY KEY və IDENTITY olaraq DataBase’də yaranır. Property’in type’ı int və ya ... olmalıdır.

AppDbContext class’ı daxilində Database’də hansı table’ların olacağını göstərməliyik.Bunun üçün də biz public olaraq DbSet generic type’ından bir property yaradırıq və generic olaraq bizim entity class’ımızı qəbul edir.Property’mizin adı Database’də table’ın adı olacaq.

Migration command’ları Tools package’i vasitəsi ilə istifadə edərək biz Database’mizi yaradırıq.Migration command’lar arxa tərəfdə DbContext Class’dan miras alan class’ı götürür və daxilində DbSet type olan property’lərə görə table yaradıb ona column verir.

Solution daxilində package manager command’ları yaza bilərik ,cmd command’ları üçün isə Projectin daxilində olmalıyıq.

add-migration migrationName – bu command migrations folderi yaradır və daxilində 2 class yaradır. Bu class daxilində table yaratmaq üçün code’lar yer alır.Class daxilindəki up method’u table yaradır və update edir, down method’u isə silir.

remove-migration – sonuncu migration’u silmək üçün istifadə edilir.

Hər hansısa migration Database daxilində icra edilməyibsə silinə bilər.Əgər icra edilibsə ilk öncə dəyişiklik Database daxilində silinməlidir daha sonra isə migration silinməlidir.

update-database – indiyədək yaranmış lakin icra olunmamış bütün migration’ları icra etmək üçün istifadə edilir.

update-database 0 – 0cı migration halına qaytarır.

update-database migrationName – adını göstərdiyimiz migration halına qaytarır.

Add-migration command’dan sonra -Outputdir migrationFolderName yazaraq yaranacaq folder’ə ad verə bilərik

**Cmd commands**

Dotnet ef migrations add migrationName -package manager command ilə eyni işi görür. Əgər error verərsə bunun səbəbi admin olaraq run etmək lazımdır.

Dotnet ef migrations remove - package manager command ilə eyni işi görür

AppDbContext class’dan instance alırıq. Aldığımız object vasitəsi ilə biz SQL sorğularını icra edəcəyik.

AppDbContext context = new();

**Add method**

Add method’u async olmalıdır ,çünki eyni anda bir neçə dəfə Database’ə bu səbəbdən sorğu göndərilə bilər.

await context.DbSetProperty.AddAsync(object); şəklində yazılır lakin bu method object’i Database’ə add etmir, add etmək üçün query hazırlayır. Query’ni icra etmək üçün await context.SaveChangesAsync(); method’u işə salınır.

Əgər add olunacaq object’in type’ı məlum olmazsa biz non-generic və ya generic olaraq method istifadə edə bilərik. Non-generic olaraq await context.AddAsync(object) şəklində də yaza bilərik. Generic olaraq isə await context.AddAsync<T>(object)

Context.AddRangeAsync(params object[]) bu şəkildə istənilən sayda object göndərərək add prosesi həyata keçirilə bilər.

context.Entry(object).State – bu property bizə c# object’inin database ilə əlaqəsini, vəziyyətini qaytarır.

Əgər object izlənmirsə Detached qayıdır.Change tracker bu object’i izləmir sadəcə ramda yaranmış bir objectdir.

Add olunub hələ save olunmayıbsa Added qayıdır.Change tracker artıq object’i izləməyə alıb.Change tracker izlənməyə alıbsa artıq bizim objectimiz database’dəki object olmuş olur

SaveChanges() olduqdan sonra isə Unchanged qayıdır. Bu o deməkdir ki database’dəki object ilə hal-hazırdaki object bərabərdir.

context.Entry(object).State = EntityState.Added və ya Context.Entry<T>(object) = EntityState.Added yazaraq biz object’in state’ni dəyişə bilərik və daha sonra SaveChanges edərək database’ə əlavə edə bilərik.

**Update method**

Eyni anda bir object birdən çox dəfə update ola bilməz bu səbəbdən method’muz async olmayacaq.

context.DbSetProperty.FirstOrDefault(predicate); -bu method vasitəsi ilə biz göndərdiyimiz predicate’ə uyğun olaraq hər hansısa bir objecti update etmək üçün database’dən çağıra bilərik. Object’in state’ni ekrana versək unchanged qayıdacaq çünki bazadan çağırıldığı üçün həmin an Change tracker tərəfindən izlənməyə başlanacaq. Izlənmənin qarşısını almaq üçün çağırıldığı zaman DbSetProperty’dən sonra .AsNoTracking() method’u çağırılmalıdır.Bu proses əsasən çağrılan object’in detail’ni ekrana vermək üçün istifadə edilir.

Əgər database’dən çağırdığımız və Change tracker tərəfindən izlənən object üzərində hər hansısa dəyişiklik edib object’in state’ni yenidən ekrana versək modified qayıdacaq. SaveChanges() method’u çağırılarsa gedib database’də də həmin dəyişiklik icra olunacaq və object’in state’i yenidən unchanged olacaq.

context.Update(object) və ya context.DbSetProperty.Update(object) bu method’lar vasitəsi ilə database’də object’in id’nə sahib olan data modified olunacaq və state’i də modified olaraq göstəriləcək daha sonra SaveChangesAsync() method’u çağırılaraq dəyişikliklər database’də icra ediləcək.

**Delete method**

Context.DbSetProperty.Find(id) – bu method vasitəsi ilə göndərdiyimiz id’ə uyğun olaraq database’dən object qaytarır.

Context.DbSetProerty.Remove(object) yazaraq database’dən göstərdiyimiz object’in id’sinə uyğun data silinəcək. Və state’i deleted olacaq ancaq database’dən silinməyəcək. SaveChangesAsync() method’u çağırıldıqdan sonra dəyişiklik icra olunacaq.

**Select method**

await context.DbSetProperty.ToListAsync() = DbSetProperty’ə uyğun olaraq bütün dataları list olaraq qaytarmaq üçün bu method istifadə edilir. Və Change tracker bu list’i izləyir

**ASP .NET**

ASP.Net ilə 0’dan MVC(Model View Controller) project qurmaq üçün ASP.NET Core Empyt seçilir. Və ya ASP.Net Core Web App də seçilə bilər.

Program.cs daxilində yazılan code’lar 2 yerə bölünür. Service hissə və Middleware hissə.

Web application’un işələməsi üçün 2 server’ə ehtiyyac var. Web server və

Web server UI’ı yayımlayan hissədir.

builder.Services.AddControllerWithViews() – Services hissədə bu method ilə .Net 6 versiyasında və daha yuxarı versiyasında qaldırılan project’lərdə MVC structor’dan istifadə edəcəyimizi bildiririk.

app.MapControllerRoute(

name: “default”,

pattern:”{controller}/{action}/{id?}”

);

Middleware hissədə bu method vasitəsi ilə biz göstəririk ki domain name’dən sonra controller’in adı gəlir daha sonra controller’in method’u gələcək daha sonra isə method’un qəbul etdiyi parameter gələcək.

app.MapControllerRoute(

name: “default”,

pattern:”{controller=defaultControllerName}/{action= defaultActionName }/{id?}”

);

Biz bu şəkildə default olaraq axtarışa sadəcə domain adı yazılarsa, default olaraq hər hansısa root çıxsın deyə controller və action’a default qiymət verib pattern’i göstərərik.

**Controllers**

Controllers folderinin daxilində empty controller’lərimizi yarada bilərik, Controller’lərin adlarının sonu Controller ilə bitməlidir çünki reflection’lar arxa tərəfdə Controllers folderinin daxilində yerləşən sonu Controller ilə bitən cs file’larını oxuyur. Create etdiyimiz Controller class’ları Controller class’dan miras alır(almalıdır).

Controller daxilindəki method IActionResult type dəyişənlər qaytaracaq. ViewResult , PartialViewResult, ContentResult , JsonResult da qaytara bilərik lakin hər biri IActionResult interface’dən implement edirlər deyə ümumi olaraq IActionResult type olaraq return edəcəklər.

**View()**

Controller daxilindəki method & action’lar geriyə hər hansısa ViewResult type View qaytarmalıdır. ViewResult qaytarmaq üçün ilk başda Views adlı folderimiz olmalıdır. Views folder daxilində hər bir Controller’in adına uyğun olaraq folder yaradılır. Və həmin folder’in daxilində isə Controller’in Action’un adına uyğun olaraq Razor View(cshtml file) yaradılır.

Biz Controller’in method’u daxilində View() return etdikdə reflection vasitəsi ilə arxa tərəfdə controller’in adına uyğun olaraq Views folderi daxilində Controller’in adına uyğun olaraq View folder’i axtarırılır və onun da daxilində Controller’in method’nun adına uyğun olaraq Razor View(cshtml) file’ı axtarılır və return edilir. Bu prosesi biz etməyək deyə Controller’in method’na yəni action’un adına sağ click etdikdən sonra Add View düyməsinə click’ləsək özü avtomatik folderləri və view’nu yaradacaq

**Raze View(cshtml)**

Razer View həm html həm də c# kodlarını birlikdə yazıb işlətmək üçün istifadə edilir və cshtml file’ı hesab edilir. Cshtml file daxilində @ işarəsi ilə biz hər hansısa c# elementinə müraciət edə bilərik və ya @{ } block’u daxilində istədiyimiz c# kodları yaza bilərik.

**Razer Layout**

View’larda təkrarın qarşısını almaq üçün layout’dan istifadə edilir. Views folder’in daxilində Shared folderi yaradılır. Bu folder daxilində partial view’lar və layout’lar yer alacaq. Layout əlavə etmək üçün add item’dan Razer Layout seçilir və adının əvvəlində \_ yazılır. Layout istifadə ediləcək view’lar yaradılarkən “use a layout page option’u” seçilir və hansı layout’un istifadə ediləcəksə seçilir.Layout daxilindəki kodlar təkrarlanan kodlardır və bizim təkrarlanmayan kodlarımız isə @RenderBody() vasitəsi ilə işarə edilir ,yaradılan view’ların kodunu bildirir.

**Razer View start**

Əgər eyni layout bir neçə səhifədə istifadə edilirsə, parametr olaraq hər bir cshtml səhifəsində layout qeyd olunmasın deyə, views folderinin daxilində Razer View Start file’ı yaradılır və daxilində parametr olaraq yaratdığımız layout’un adı qeyd edilir. View start istifadə edəcəyiksə yuxarıdaki kimi “use a layout page option’u” seçildikdən sonra layout göstərilmir, boş buraxılır. Özü avtomatik Razer View start istifadə edildiyini başa düşür.

**Razer View Import**

Views folderi daxilində yaratdığımız Razer View Import file vasitəsi ilə View’larda istifadə etdiyimiz təkrarlanan using’ləri saxlayıb bütün View’lara (cshtml file’larına) aid edirik.

**Models**

Model’lərimiz üçün Solution daxilində Core adında yeni project(library) yaradırıq. Library daxilində Enitities adında folder yaradırıq və bütün model’ləri bu folder daxilində saxlayırıq.

**AppDbContext**

DbContext üçün Solution daxilində DataAccess adında yeni project(library) yaradırıq. Library daxilində Contexts adında folder yaradıb AppDbContext class’ı burada saxlayırıq.Bu səfər AppDbContext class’ın constructor’unu overload edəcəyik və DbContextOptions<AppDbContext> type object’i base’ə göndərəcəyik.

Services hissədə builder.Service.AddDbContext<AppDbContext>(opt => opt.UseSqlServer(”Server = serverName ; Database = dataBaseName ; Trusted\_Connection = True”)) deyərək bildiririk ki biz bu context’i istifadə edəcəyik.Lakin biz ConnectionString’i burada verməyimiz düzgün deyil.

Appsettings.json vasitəsi ilə biz ConnectionString olaraq verdiyimiz dəyəri saxlamalıyıq.

“ConnectionStrings”: {

“Default”: ”Server = serverName ; Database = dataBaseName ; Trusted\_Connection = True”

}

Həmin dəyəri Services tərəfdə var connectionString = builder.Configuration[“ConnectionStrings:Default”]; və ya var connectionString = builder.Configuration.GetConnectionString(“Default”); yazaraq götürə bilərik. Builder.Service.AddDbContext<AppDbContext>(opt => {opt.UseSqlServer(connectionString) }); deyərək yuxarıda yazdığımız connection’u security hala salırıq.

**Add Migartion**

WebUI olan project “Set as Startup Project” olaraq set olmalıdır. Migration’umuz isə hansı project’də yaradılıbsa orada add-migration olunmalıdır.Bununla yanaşı Migration’u yarada bilmək üçün startup olararaq set olunan project’ə Core.Design package yüklənməlidir.

**ViewBag , ViewData, TempData**

ViewBag.Key = value; vasitəsi ilə Controller’də və ya cshtml file’da hər hansısa bir dəyər set edib, cshtml file’da və ya Controller’də @ViewBag.Property yazaraq value’nu get edə bilərik.

ViewData[“key”] = value; vasitəsi ilə Controller’də və ya cshtml file’da hər hənsası dəyər set edib, cshtml file’da və ya Controller’də @ViewData[“key”] yazaraq value’nu get edə bilərik.

ViewBag və ViewData bir birinin alternatividir. Yəni biz ViewData ilə verdiymiz data’nı cshtml file’da ViewBag ilə get edə bilərik və ya əksinə

TempData[“key”] = value; ViewData məntiqi ilə işləyir lakin fərqi odur ki, bir səhifədən digərinə yönləndirilən zaman TempData ilə verilən dəyərlər itməyəcək.

Bir səhifədən digərinə yönləndirmək üçün RedirectToAction(“ActionName”); yazılır. Bu zaman bu method’a qədər olan dəyərlər oxunur və yaddaşda saxlanılır digər action vasitəsi ilə view return olan zaman TempData ilə verilən dəyərlər yaddaşda saxlanılır. Lakin bir başa Action’un özünü return edəndə və ya View(nameof(ActionName)) yazıb return edəndə heçbir əvvəlki datanı yaddaşda saxlamır və return olunan Actionun daxilini oxuyur.

**@Model**

Database’dən məlumat gətirmək üçün Controller’imiz constructor injection ilə AppDbContext object’i qəbul edir. Biz action daxilində return View etdiyimiz zamanı arqument olaraq göndərdiyimiz object Model adlanır və cshtml daxilində @model objectType yazaraq qəbul edirik. Daha sonra Cshtml file daxilində isə @Model yazaraq istifadə edirik.

**View Models**

View Models’in istifadə olunmasının səbəbi odur ki, biz View’a eyni vaxtda bir neçə Data göndərməli ola bilərik. Project daxilində View Models folderi yaradırıq. Göndərəcəyimiz dataları View Models’in daxilində property şəklində saxlayacayıq. Bu property’lər IEnumerable type olmalıdır. Çünki biz bu View Models’in object’ni yaradan zaman database’ə eyni vaxtda sorğu gedir bir-birlərinə problem yaratmasınlar deyə bu type vasitəsi ilə generic olaraq verilməlidirlər.

**wwwroot**

Bizə lazım olan static dataları wwwroot folder’in daxilində assets folderi yaradaraq saxlayırıq. Middleware olan hissədə app.UseStaticFiles(); method’u yazılıb bildirilməlidir ki bu static file’lar istifadə olunacaq. Static file’larımızın root’nu göstərəndə isə bütün assets’lərin önünə ~/ artırılmalıdır. Burada tilda(~) işarəsi wwwroot’u göstərir.

**Area**

Area Project daxilində mini project’lər yaratmaq üçün istifadə edilir.

Area yaratmaq üçün Start up project’ə sağ click eləyib Add hissədən Scaffolded item’ə clickləyirik və açılan pəncərədən MVC Area seçib add edirik. Area daxilində özü avtomatik olaraq bizə lazım olan folderləri yaradır.Bizə verdiyi MapControllerRoute’u gəlib Midleware’də digər MapControllerRoute’un yuxarısına yazırıq.

Daha sonra Area Controller daxilində DashboardController yaradılır. Area Controller olduğunu bildirmək üçün attribute olaraq [Area(“AreaName”)] qeyd edilir.

**Tag Helper**

Biz hər-hansısa a tag’i daxilində href’ə root yazmaq əvəzinə tag helper’in köməkliyi ilə attribute’lardan istifadə edirik. asp-area=”areaName” asp-controller=”controllerName” asp-action=”actionName” asp-route-routeName=”route”. Əslində bu qədər uzatmayıb sadəcə asp-action və asp-route istifadə edərək də göstərə bilərik çünki hansı area daxilində və hansı controller ilə səhifədə olduğumuz əvvəlcədən bilinir və biz hansı action’u istifadə edəcəyiksə sadəcə onu bildirərək bizə lazım olan funksiyanı icra edirik.

**Http Get Http Post**

Bütün action’lar default olaraq Http get method’dur. Hər hansısa method’un Http Post olduğunu bildirmək üçün [HttpPost] attribute’dan istifadə edilir.

**Form action**

Əgər get method olan action’un adı, həmin action’un return elədiyi view’nun daxilindəki form tag’i daxilində method=”post” olduğunu mütləq yazıb, action’ı göstərməsək özü avtomatik olaraq həmin adda olan method’un post action’na müracəiət olunmalı olduğunu başa düşür və method’un action’un post olan method’na yönləndirilir. Əgər biz başqa bir action’a yönləndirmək istəsək bu səfər məcbur şəkildə action’u göstərməliyik.

Biz inputa daxil edilən məlumatları input’un name attriubute daxilində yazdığımız adlara uyğun olaraq action’da parametr kimi qəbul edirik.Lakin bunu daha rahat şəkildə edə bilərik. Bu üsul Model Biding adlanır.

**Model Biding**

@model olaraq bizə lazım olan object’i view’da yuxarı hissədə təyin edirik. Input’ların daxilində name və type attribute’larını silib onun yerinə asp-for =“modelPropertyName” yazırıq. asp-for bizə property’nin type’na uyğun olaraq input’a type verir, input’a daxil edilən dəyəri property’ə mənimsədir. Action’umuza da parametr olaraq model kimi təyin etdiyimiz object qəbul edilir.

**ModelState**

ModelState.IsValid method’u bizə input’dan daxil edilən bütün dataların modelimizin bütün annotation’larını ödəyibsə true dəyər qaytarır.False dəyər qaytararsa biz həmin View’nu model arqumenti ilə birlikdə return etsək özü avtomatik olaraq error mesajlarını input’un altında çıxardır. Error mesajlarının çıxarılması üçün span tag daxilində , asp-validation-for attribute’dan istifadə edilir və modelin error qaytara biləcək olan property’sinin adı qeyd edilir. Biz IsValid şərtindən əlavə olaraq müəyyən bir şərt daxilində asp-validation-for’u işə sala bilərik. Bunun üçün ModelState.AddModalError(“PropertyName”,”ErrorMessage”); yazmağımız kifayət edir.

ModelState.AddModalError(“”,”ErrorMessage”); birinci hissəni boş buraxıb errorMessage göstərilərsə asp-validation-for olaraq göstərdiyimiz hissədə yox , asp-validation-summary = “ModelOnly” olaraq yazdığımız hissədə göstərir.Birdən çox mesaj gələcəyi halda bu istifadə edilir və li tag’lərinin içinə yığıb elə ekrana çıxarır.

**Annotations**

Annotation’lar ilə Attribute’ların fərqi odur ki, Annotation’lar DataAnnotations’u using edir.

Modellərimiz üçün Required annotation’dan istifadə edəcəyik. Required(ErrorMessage=”message”) yazaraq biz error zamanı hansı mesajın qaytarılacağını da təyin edə bilərik.

Həmiçin MaxLengt((int)length) verərək maximum neçə simvoldan ibarət dəyər göndərə bilərik onu da qeyd edirik.

**[ValidateAntiForgeryToken]**

Biz bu attribute vasitəsi ilə bildiririk ki form daxilində generasiya olunmuş token uyğunlaşmırsa o zaman bu sorğu qəbul edilmir və action icra edilmir.

**[ActionName(“newActionName”)]**

Biz bu attribute vasitəsi ilə bildiririk ki actioun’un adı göstərdiyimiz yeni ad olacaq.

**File Upload**

Hər hansısa type file qəbul etmək, üçün biz yeni bir model daxilində property’ni IFormFile olaraq göstəririk. Bunun üçün Core’un Http.Features package’ni yükləyirik. IFromFile ilə verdiyimiz property’in sayəsində input’a type verməyə ehtiyyac qalmır özü avtomatik olaraq file qəbul ediləcəyini başa düşür. Lakin seçildiyi halda file’ın null gəlməməsi üçün form attribute olaraq enctype = “multipart/form-data” yazmalıyıq əks təqdirdə null dəyər qayıdacaq çünki form onu göndərmir.Enctype yalnız form’un method’u post method olan zamanı işləyir

IFromFile olaraq göstərdiyimiz property’in :

.FileName vasitəsi ilə adını götürə bilərik.

.Length vasitəsi ilə byte olaraq ölçüsünü götürə bilərik.

.ContentType vasitəsi ilə biz seçilən file’ın typını və uzantısını götürə bilərik.

Əlavə olaraq biz input daxilində attribute olaraq accept = “image/\*” yazaraq bildiririk ki yalnızca file olaraq şəkil file’ları seçilə bilər. Əgər input’a accept attribute’u ilə hansı file type’lara icazə olub olmadığı verilməzsə və bizə yalnızca image type file lazımdırsa biz .ContentType’ı işə salacayıq. Yoxlamalıyıq ki əgər qayıdan ContentType.Contains(“image/”) true olarsa deməli image type file seçilib.

Şəkilləri saxlayacağımız path’i götürürük və sonuna yeni file’ın adını əlavə edirik. FileStream vasitəsi ilə using daxilində yeni bir object yaradırıq və arqument olaraq path’i və FileMode.Create göndəririk. Və deyirik ki bu path’də göstərilən file’ı yaradacayıq.

IFromFile olaraq göstərdiyimiz property’ə .CopyToAsync(fileStreamObject); yazaraq bildiririk ki upload olunan file’ın copy’sini göstərilən path’də yarat.

**IWebHostEnvironment**

Biz path’i static olaraq hər dəfə fərqli olaraq verməyək deyə həmçinin project fərqli fərqli cihazlarda’da qalıdırılanda onun üçün ayrıca path dəyişməsin deyə, IWebHostEnvironment object’ni istifadə edəcəyik. Bu object vasitəsi ilə biz ümumi project haqqında məlumat ala bilirik. wwwroot’un path’ni almaq üçün \_env.WebRootPath; ’dən istifadə edəcəyik.

Path’in .Combine() method’u vasitəsi ilə biz göndərilən string dəyərləri path şəklində birləşdirəcəyik və bütün əməliyyat sistemlərində də düzgün işləyəcək. Yəni backslash və ya slash problemi olmayacaq. Combine() method bizdən params string[] qəbul edir bu səbəbdən istədiyimiz qədər string dəyişən göndərə bilərik.

Guid.NewGuid().ToString() edərək x simvoldan ibarət random bir ifadə ala bilərik. Şəkillərimizin adı eyni olma ehtimalının qarşısını almaq üçün əvvəlinə bu string ifadəni atacayıq.

**Delete File**

Biz hər hansısa path’dəki file’ı silmək üçün ilk öncə o path’də həqiqiətən hər hansısa file’ın olub olmamasını yoxlamalıyıq. Bunu üçün System.IO.File.Exists(“path”) method’dan istifadə edəcəyik. əgər mövcuddursa System.IO.File.Delete(“path”) method’u istifadə edərək göstərilə path’dəki file’ı silirik.

**Session**

Session’u istifadə edəcəyimizi bildirmək üçün app.UseSession() yazılır.

Session’u add eləmək üçün isə aşağıdaki kod bloku yazılır.

builder.Services.AddSession(opt=>

{

opt.IdleTimeout=TimeSpan.FromSecond(second); və ya TimeSpan.FromMinute(minute);

});

Session’lar key-value məntiqi ilə əlavə olunur, eyni key’ə yenə dəyər set olunarsa əvvəlki dəyəri override edərək yeni dəyəri saxlayır.

HttpContext.Session.SetString(key,value); bu method’u yazdığımız action’un view’u işə düşəndə session yaranır.Session yaranan zaman cookie’lər kimi Application daxilində Cookies bölməsində saxlanılır.Cookie’lərdən fərqli olaraq bir başa key-value olaraq saxlanılmır.Və verilən vaxt müddəti bitdikdən sonra web serverdən bu session silinir, lakin web browser’də mövcud olur. Web serverdən bu session silindikdən sonra web browser’də mövcud olmasının heç bir önəmi olmur.

HttpContext.Session.GetString(key); bu method vasitəsi ilə də biz session’a əlavə etdiyimiz key’in qarşılığı olan value’nu method’a key göndərərək əldə edirik.

Browser bağlanıb yenidən session’un yarandığı View’a daxil olmasaq həmin səhifədə əvvəldə yaranan session’u istifadə edə bilmərik. Browser bağlandıqdan sonra web server session’u saxlayır lakin browser’dən session silinir.

**Cookie**

Response.Cookies.Append(key,value,options); Bu method vasitəsi ilə biz Cookie yaradırıq. Və göstərdiyimiz option’lar əsasında formalaşır. Options olaraq new CookieOptions{MaxAge = TimeSpan.FromMinutes(minute)} göstərərək Cookie’in işləmə müddətini bildiririk.

MaxAge yaranan andan etibarən nə qədər müddət mövcud olacaq onu bildirir.Expires isə konkret tarix göstərilir və o tarixə qədər mövcud olacağı bildirilir.

CookieOptions vasitəsi ilə biz göstərilən bütün option’ları qeyd edə bilərik.

Request.Cookies[“key”]; bu method vasitəsi ilə də biz Cookie’ə müraciət edə bilərik.

**Identity**

Identity üçün biz yeni bir model yaradacayıq. Bu modelimiz IdentityUser class’ı extend etməlidir. Bunun üçün bizim Identity.EnitiyFrameworkCore package’nə ehtiyyacımız var. Artıq bizim AppDbContext Class’mız DbContext yox, IdentityDbContext class’ı extend edəcək. Identity üçün yaratdığımız AppUser modeli IdentityDbContext’ə Generic olaraq göndərilir. Generic olaraq göndərdiyimiz üçün gəlib DbSet etməyə ehtiyyac qalmır.Yaradacağımız migration vasitəsi ilə bütün lazımı table’lar yaranacaq.

Services hissəyə gəlib Identity istifadə edəcəyimizi bildirmək üçün builder.Services.AddIdentity<AppUser,IdentityRole>(opt).AddEntityFrameworkStores<AppDbContext>(); yazırıq.

Options vasitəsi ilə biz müəyyən configration’lar yerinə yetirəcəyik. .AddEntityFrameworkStores<AppDbContext> vasitəsi ilə də biz hansı database’dən istifadə edəcəyimizi bildiririk.

builder.Services.AddIdentity<AppUser,IdentityRole>(opt =>

{

opt.Password.RequiredLength = 8;

opt.Password.RequireDigit = true;

opt.Password. RequireNonAlphanumeric = true;

opt.Password.RequireLowercase = true;

opt.Password.RequireUppercase = true;

opt.User.RequireUniqueEmail = true;

opt.Lockout.MaxFailedAccessAttempts = count;

opt.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(minute);

opt.Lockout.AllowedForNewUsers = true;

}).AddEntityFrameworkStores<AppDbContext>();

Daha sonra Middleware hissədə gəlib app.UseAuthentication(); və app.UseAuthorization(); yazaraq bildirməliyik. Ardıcıllıq bu şəkildə olmalıdır və route’dan əvvəl yazılmalıdır.

**Manager<T t>**

CRUD əməliyyatlarını yerinə yetirmək üçün UserManager<AppUser> ‘in \_userManager object’dən istifadə edəcəyik.

SignIn və ya SignOut əməliyyatlarını yertinə yetirmək üçün SignInManager<AppUser> ‘in \_signInManager object’dən istifadə edəcəyik.

Role‘larla bağlı əməliyyatları yerinə yetirmək üçün RoleManager<IdentityRole> ‘in \_roleManager object’dən istifadə edəcəyik.

**Registration**

Registration üçün ViewModel’ə ehtiyyacımız olacaq. Confirm Password’u həyata keçirtmək üçün annotation olaraq Compare(nameof(Password)) yazmaq lazımdır. Bununla yanaşı email’ə annotation olaraq DataType(DataType.EmailAddress) , Password’a da DataType(DataType. Password) annotationları vermək lazımdır.

Registration CRUD əməliyyatlarına daxil olduğu üçün \_userManager object’dən istifadə edəcəyik.

Registration etmək üçün AppUser object’i yaradıb ViewModel’i ona mapping etmək lazımdır, IsActive olaraq da true vermək lazımdır. Daha sonra save zamanı \_userManager.CreateAsync(appUser,password) yazacayıq. Password’u bura göndərməyimizin səbəbi hash’lənmiş şəkildə save olunmasını təmin etmək üçündür. Bizə geriyə identityResult qaytaracaq və bizim options olaraq verdiyimiz şərtlər ödənməsə identityResult.Succeeded bizə false dəyər qaytaracaq.

False dəyər qaytaran zaman həmçinin .Errors property’in daxilində error’lar olacaq bu error’ları for each daxilində ModelState.AddModelError(“”,error.Description) yazaraq view’a göndərəcəyik. Iterasiya olunan collection’umuz identityResult.Errors olacaq. Əgər bizə Succeeded true qayıdarsa bizim .SaveChanges(); yazaraq etdiyimiz əməliyyatı özü avtomatik olaraq həyata keçirəcək.

**Return RedirectToAction(nameof(View)) ilə return View() fərqi**

Hər hansısa View return etdikdə sadəcə View geriyə qaytarır yəni cshtml’i, RedirectToAction edilən method daxilində isə dataları saxlayaraq yönləndirilmə baş verir, View qaytarır yəni method’u çağırır.

RedirectToAction(actionName,ControllerName) kimi overload halı da mövcuddur.

**Login**

Login zamanı qəbul etdiyimiz ViewModel’in yoxlanışını aparmalıyıq. Yəni username və ya email mövcuddur mu? Bunun üçün UserManager object’in .FindByEmailAsync(email) və ya .FindByNameAsync(username) method’nu istifadə edəcəyik əgər mövcuddursa object’i qaytaracaq, mövcud deyilsə null qaytaracaq. Daha sonra SignInManager<AppUser> object’i vasitəsi ilə \_signInManager.PasswordSignInAsync(user,password,rememberMe(bool),lockoutOnFailure(bool));

ViewModel’də boolean olaraq verdiyimiz rememberMe property’ni asp-for vasitəsi ilə göstərildikdə input’a checkbox type verilir.Bunu remember me’ni almaq üçün istifadə edəcəyik. LockoutOnFailure isə səhv yoxlanışlardan sonra lockout olsun yoxsa olmasın bildirmək üçündür.

.PasswordSignInAsync method’u bizə geriyə SignInResult object’i qataracaq. Bu object vasitəsi ilə biz:

IsLockedOut olub-olmamasını yoxlaya bilərik. əgər true dəyər qayıdarsa ModelState.AddModelError vasitəsi ilə error çıxardırıq və return View(loginVM) edirik. Ilk dəfə qeydiyyatdan keçib hələ login olmayan user’lara bu aid edilməyəcək çünki default olaraq options’da AllowedForNewUsers = true olacaq. Aid edilməsi üçün options’da bunu false olaraq göstərmək lazımdır.

signInResult.IsSucceeded vasitəsi ilə də daxil olub olmamasını yoxlayırıq. əgər false dəyər qayıdarsa biz geriyə ModelState.AddModelError vasitəsi ilə error çıxardıb return View(loginVM) edirik.

Və ən sonra isə IsActive olub-olmamasını yoxlayırıq. əgər false dəyər qayıdarsa istifadəçiyə ModelState.AddModelError vasitəsi ilə” not found” error çıxardırıq və return View(loginVM) edirik.

**Lockout**

Lockout hadisəsi, göstərilən say qədər hesab’a yanlış şifrə ilə daxil olmağa çalışanda baş verir. Bu hadisə baş verən zaman müəyyən vaxt ərzində hesaba giriş qadağa olunur və lockout 0’lanır.Həmçinin hesaba uğurla daxil olduqda da lockout 0’lanır.

**Logout**

Logout etmək üçün User.Identity.IsAuthenticated istifadə olunur yəni login olunubsa əgər bizə true dəyəri qayıdacaq və deməli biz logout edə bilərik. \_signInManager.SignOutAsync() method’nu işlədərək signout oluruq. False dəyər qayıdarsa bad request return etməliyik.

**Role**

RoleManager<IdentityRole> ‘in \_roleManager objecti yaradacayıq. \_roleManger.CreateAsync(new IdentityRole(“roleName”)) method’u ilə biz yeni bir role yaradırıq. Biz roleName’ləri enum şəklində saxlasaq daha düzgün olar. Və enum’la göstərdiyimiz roleName’lərə uyğun olaraq avtomatik role’ların yaranması üçün enum’ı foreach daxilində salacayıq. Enum.GetValues(typeof(ourEnumClassName)) yazaraq iterable collection(Enum array) yarada bilərik. Daha əvvəldən roleTable’da olan role’u təkrardan yaratmamaq üçün biz \_roleManager.RoleExistsAsync(role.ToString()) method’dan istifadə edirik və əgər bizə false qayıdarsa \_roleManger.CreateAsync(new IdentityRole(role.ToString())) bu şəkildə Create edəcəyik.

Artıq Role’lara sahib olduğumuz üçün Registration zamanı biz əgər uğurla register olarsa \_userManager.AddToRoleAsync(appUser,roleName) yazaraq yeni role əlavə edəcəyik.

**Authorize**

Authorize attribute vasitəsi ilə biz göstəririk ki bu actionlar yalnız login olan şəxslər tərəfindən əl çatan olur. [Authorize]

[Authorize(Roles = “roleName1,roleNameN”)] – bu şəkildə isə biz göstərdiyimiz role sahib olan user’lərin bu action’lardan istifadə edə biləcəyini bildiririk.

Login olunmamış daxil ola bilməyəcəyimiz səhifələrdən bir başa login view’na yönəltmək üçün Services hissədə

builder.Services.ConfigureApplicationCookie(opt =>{

opt.LoginPath = “/Auth/Login”;

}); yazaraq login olunmalı action’lar istifadə olunmaq istənəndə bir başa login səhifəsinin path’ni yazaraq bildiririk.

**Forget Password**

Forget Password üçün yeni bir ViewModel yaradırıq və property olaraq Email saxlayırıq. Yoxlanışı email’ə görə aparacayıq əgər bu mail’ə sahib user mövcud deyilsə ModelState.AddModelError(“Email”,”email not found”yazıb return View(forgetPasswordVM) yazacayıq.

Biz resetPassword edə bilməyimiz üçün ilk olaraq token yaratmalıyıq. \_userManager.GeneratePasswordResetTokenAsync(user) vasitəsi ilə göndərdiyimiz user üçün token generate etmiş olduq.Generate olunmuş token 24 saat aktiv olaraq qalır. Bu method’un düzgün işləməsi üçün Services hissədə .AddIdentity’in sonuna .AddDefaultTokenProvides() əlavə edilməlidir.

Daha sonra resetPassword edə bilməyimiz üçün link generate edəcəyik.

string link = Url.Action(actionName,controllerName, new{userId , token} ,HttpContext.Request.Schema)

biz object olaraq göndərəcəyimiz value’nu göstəririk. Daxilində userId və token saxlayır.

HttpContext.Request.Schema onu bildirir ki hal-hazırda istifadə olunna protocol’u göstərsin.

Əldə etdiyimiz link üçün action yazırıq və action userId ilə token qəbul edir. Gələn dəyərlərin IsNullOrWhiteSpace olub olmaması yoxlanılır true dəyər qayıdarsa return BadRequest(); yazılır.

Daha sonra gələn Id’ə uyğun olaraq \_userManager.FindByIdAsync(userId) method’u işə salınır.

**One To Many Relation**

One to Many relation yaratmaq üçün Id property’nin üzərinə ForeignKey annotation’u yaza bilərik. Lakin bunun yerinə biz Id property ilə yanaşı həmin object’in özünü də property kimi saxlasaq migration yaranan zaman özü avtomatik olaraq One to Many əlaqəsini yaradır.Sadəcə object’in özünü də saxlasaq OneToMany relation üçün kifayətdir lakin Id bizə lazım ola bilər deyə onu da saxlamaq məsləhətdir.Əgər elə bir hal olsa ki biz object’i və Id’ni saxlamalıyıq lakin foreign key əlaqəsi qurulmasın, o zaman biz object’in üzərinə NotMapped annotation’u yazmalıyıq. NotMapped annotation’u yazılan property migration yaranan zaman əlavə olunmur.

Property olaraq saxladığımız object’in(Parent) daxilində də ICollection<Child> childCollection collection saxlamaq olar.

Yaranan migration daxilində onDelete: ReferentialAction.methodName ilə biz Parent’ə aid olan data’nı silən zaman nə baş verəcəyini göstərə bilərik.

Biz \_context’ə müraciət edən zaman join istifadə etmək üçün, .Include(obj=>obj.Parent).AsEnumerable();

**Repository pattern**

Repository pattern qurmaq üçün ilk öncə DataAcces layer daxilində Interfaces folder’i yaradıb onun daxilində IRepository interface yaratmaq lazımdır. Bu interface Generic type olmalıdır və yalnız class’ları, Entity interface’dən implement alan class’ları,boş constructor’u olan class’ları qəbul etməlidir.

Database vasitəsi ilə həyata keçirilən bütün əməliyyatlar burada yazılmalıdır.

Daha sonra Context folderi daxilində Repository class’ı yaradılmalıdır və IRepository interface’ni implement etməlidir.

Daxilində DbContext object’ni(\_context) və DbSet<T> type object’i(\_table) saxlamalıdır. \_table object’ə\_context.Set<T> assign olunmalıdır. Set method’u bizə göndərdiyimiz generic type’a uyğun olaraq həmin entity’in IQueryable’nı qaytaracaq. ToListAsync(); method’u vasitəsi ilə də biz IQueryable’ı IEnumarable’a çeviririk.

Hər bir entity üçün repository’ə uyğun olaraq interface və class yaradırıq. Hər bir interface IRepository’dən implement edir və generic olaraq həmin entity göndərilir. Hər bir class Repository’dən miras alır , özünün uyğun interface’ni də impelement edir.Generic olaraq da entity göndərilir.

Dependency Injection vasitəsi ilə bu repository’ləri istifadə etmək üçün gəlib services hissədə tanıtmalıyıq, bunun üçün AddScoped istifadə edilir. builder.Services.AddScoped<IRepositoryEntityName,RepositoryEntityName>();

Dependency Injection vasitəsi ilə həmin Entity’in Interface’i constructor vasitəsi ilə qəbul edilir.

**API(Application Programming Interface)**

Solution daxilində 2 folder yaradılır. Src və test folderi.

Src folderi daxilində bizim ASP.Net Core Web Api projectimiz yer alacaq.API projectin Src folderi daxilində yaranması üçün biz location olaraq sonda /src yazmalıyıq.Src folder daxilində Core layeri,DataAccess layeri və Businnes layeri də yer alır.

Core daxilində Interface folderi yaradılır və daxilində IEntity interface’I saxlanılır.

Core daxilində Entities folder yaradılır və daxilində isə bizim Entity’lərimiz yer alır.

DataAccess daxilində Contexts folderi yaradılır və daxilində AppDbContext class’I saxlanılır. AppDbContext class’ı EntityFrameWork package class’ı olan DbContext’i extend edir. Constructor’u DbContextOptions<AppDbContext> options qəbul edir və base constructor’a göndərir.

**Repository pattern**

DataAccess daxilində Repository folder yaradılır və onun da daxilində Interface ilə Implementations folderi yaradılır. Interfaces folderi daxilində IRepository<T> interface’i yaradılır. Bu interface Generic type olmalıdır və yalnız class’ları, Entity interface’dən implement alan class’ları,boş constructor’u olan class’ları qəbul etməlidir. Implementations folderi daxilində Repository<T> class’ı yaradılır və IRepository interface’i implement edir.Biz burada da Constraint’ləri yazmalıyıq.Yaratdığımız Repository class daxilində AppDbContext object’i Db<Set> object’i saxlamalıdır. Burada \_context object’i dependency injection vasitəsi ilə qəbul edilməlidir,Db<Set> object’ə isə \_context.Set<T> assign olunmalıdır. Hər bir Entity üçün ayrıca Repository interface və class yaradılmalıdır. Interface’lər IRepository<T> interface’i impelement edəcək, Repository’lər isə Repository class’dan miras alacaq və özünün interface’ni implement edəcək. Biz burada method’ları IQueryable olaraq saxlayacayıq.

**Business logic**

Businnes layer daxilində Services folderi yaradılır və onun daxilində Interface ilə Implementations folderi yaradılır. Data ötürüləcəyi zaman bütün şərtlər burada yoxlanılır və hazır method halına salınır. API tərəfdə yalnızca bu hazır method’lar çağırılır.

**Controller**

Controller class’ların adı MVC controller’lərin adından fərqli olaraq API controller’lərdə cəmdə olmalıdır. API controller class’ımızda [Route(“api/[controller]”)] və [ApiController] annotation’u yazılır. Yazılan method’un Get və ya Post olduğunu bildirmək üçün [HttpGet] & [HttpPost] attribute’ları istifadə edilir. Route’u bildirmək üçün [HttpGet, Route(“routeName”)] yazılır.

**DTO (Data Transfer Object)**

Biz müştəriyə Entity’in bütün property’lərini qaytarmayaq deyə DTO’dan istifadə edirik və lazım olan dataları bu DTO daxilində saxlayıb geriyə DTO qaytarırıq. Businness layer daxilində DTOs folder yaradılır və bütün DTO’lar burada saxlanılır.

**Auto Mapper**

Auto Mapper vasitəsi ilə biz property’ləri eyni olan 2 object’in dəyərlərini bir-birinə mənimsədə bilərik. Auto Mapper istifadə edə bilmək üçün AutoMapper və AutoMapperDependencyInjection package’lər yüklənməlidir. Daha sonra Auto Mapper istifadə edəcəyimizi bildirmək üçün Services hissədə gəlib bu şəkildə bildirməliyik builder.Services.AddAutoMapper(typeof(MapperClass).Assembly).

AutoMapper’lərimiz Businness layer daxilində Mappers folderində saxlanılır. Yaratdığımız Mapper class’ları Profile class’ı extend etməlidir. Constructor daxilində CreateMap<Object1, Object2>() yazaraq göstəririk ki Object1’i Object2’ə mapp edirik. Hər ikisini də bir-birinə mapp edə bilməz üçün .ReverseMap() method’u əlavə edirik. Mapper’dan istifadə etmək üçün isə IMapper object’i Dependency Injection vasitəsi ilə götürülür

**Fluent API**

Entity’lərimizlə bağlı Database’də yaranacaq table’ın column’larına aid hər hansısa configuration’lar üçün, DataAccess daxilində Configurations folderi yaradılır.Hər entity’ə uyğun olaraq configuration Class yaradılır. Bu class’lar IEntityTypeConfiguration<EntityName> interface’i implement edir. Implement olunan Configure method’u EntityTypeBuilder<EntityName> builder qəbul edir. Biz configuration üçün builderdən istifadə edəcəyik. builder.Property(p=>p.propertyName)

.IsRequired(boolValue)

.HasMaxLength(intValue)

Elədiyimiz configuration’u Entity’ə əlavə etmək üçün AppDbContext class’ı daxilində OnModelCreating method’u override edirik. Method ModelBuilder objecti qəbul edir. modelBuilder.ApplyConfigurationsFromAssembly(typeof(configurationClass).Assembly); method vasitəsi ilə biz configurationClass daxilində olan bütün configurationları əlavə edirik.

**Fluent Validation**

Fluend Validation bizə gələcək olan hər hansısa data’nın müəyyən şərtlərlə yoxlanılması üçün istifadə edilir. Fluent Validation istifadə edəcəyimiz layer’a FluentValidation.AspNetCore package yüklənməlidir, yəni Businnes layer’a. Daha sonra Validators folder yaradılmalıdır və bütün yoxlanmalı olan class’ların adına uyğun folder yaradılmalıdır.Validation üçün Validator class’ı yaradılmalıdır. Validation class’I AbstractValidator<Entity> class’ı extend etməlidir. Daha sonra boş constructor daxilində RuleFor(expression) vasitəsi ilə lazımı configuration’u yazmalıyıq.

RuleFor(e=>e.PropertyName)

.NotEmpity().WithMessage(“CustomMessage”)

.NotNull().WithMessage(“CustomMessage”)

Həmçinin biz xüsusi bir yoxlanış etmək istəsək RuleFor(e=>e.PropertyName).Must(Predicate predicate) vasitəsilə yoxlanış apara bilərik.

Must ilə də .WithMessage(“CustomMessage”) error mesajımızı göstərmək üçün istifadə edilən method’u çağıra bilərik.

builder.Services.AddFluentValidationAutoValidation();

builder.Services.AddFluentValidationClientsideAdapters();

builder.Services.AddValidatorsFromAssembly(typeof(ValidatorClass).Assembly);

kodlarını Service hissədə yazaraq bildiririk ki Validation’dan istifadə edəcəyik.

**Identity**

Identity üçün core layer daxilində Identity folder yaradılır və daxilində AppUser class’ı yaradılır. AppUser class’ı Identity.EntityFrameworkCore package’dən gələn IdentityUser class’dan miras almalıdır. Daha sonra AppDbContext daxilində dəyişiklik edib DbContext class’dan yox, IdentityDbContext<T> class’dan miras alacaq. Identity’dən istifadə etdiyimizi gəlib Services hissədə göstərmək üçün builder.Service.AddIdentity<AppUser,IdentityRole>(opt=>) {

**configurations**

})

.AddDefaultTokenProviders()

.AddEntityFrameworkStores<AppDbContext>(); yazırıq.

**Register**

Register üçün ilkin olaraq RegisterDTO yaradılır və qəbul edilən məlumat olaraq DTO’dan istifadə olunur.Gələn məlumatları yoxlamaq üçün RegisterValidator’dan istifadə edilməlidir.

Businness logic’də IAuthService interface və AuthService class yaradılır və Register əməliyyatı burada yazılır.

AuthService daxilində UserManager<AppUser> object’i(\_userManager) dependency injection vasitəsi ilə götürülür. \_userManager’in .CreateAsync(AppUser,Password) method’u vasitəsi ilə register əməliyyatı yerinə yetiriləcək. Method bizə IdentityResult object’i qaytaracaq əgər identityResult.Succeeded false dəyəri qaytararsa biz string daxilinə identityResult.Errors’dən qayıdan error’ların .Description prorety’sinin dəyərlərini yığacayıq.

Int count =0;

string errors = String.Empty;

foreach(var error in identityResult.Errors) {

errors+= count!=0? $“,{error.Description}” : $“{error.Description}”;

}

əgər identityResult.Succeeded true qayıdarsa özü avtomatik olaraq user yaradacaq və biz \_userManager.AddToRoleAsync(user,role); vasitəsi ilə role yaradacayıq. Bizə IdentityResult qaytaracaq və yuxarıda foreach daxilində elədiklərimizi bu dəfə role üçün edəcəyik

User register olan zaman hər hansısa bir problem çıxarsa custom olaraq yaratdığımız UserCreateException throw edəcəyik.

**Seeder**

Seeder vasitəsi ilə biz program run olan zaman işə düşməli olan bir sıra əməliyyatları yazacayıq.

Contexts folder daxilində AppDbContextInitializer class’ı yaradılır. Dependency Injection vasitəsi ilə UserManeger<AppUser> və RoleManager<IdentityRole> object’ləri götürülür.

**RoleSeed**

RoleSeedAsync method’u yaradılır. Bu method vasitəsi ilə Role’lar yaradılır. Role’ların adını saxlamaq üçün Core layer daxilində Enums folderi yaradılır və Roles enum bu folder daxilində saxlanılır.

foreach(var role in Enum.GetValues(typeOf(Roles)) {

if( await \_roleManger.RoleExistsAsync(role.ToString()){

await \_roleManager.CreateAsync(new() { Name = role.ToString() });

}

}

**UserSeed**

UserSeedAsync method’u yaradılır. Bu method vasitəsi ilə User’lar yaradılır. AppUser object’i yaradılacaq və onun şəxsi məlumatları AppSettings.Json daxilində saxlanılacaq. Hər hansısa məlumatı AppSetttings daxilindən götürmək üçün class’lar daxilində IConfiguration object’dən istifadə edəcəyik və bu object’i Dependency Injection vasitəsi ilə əldə edəcəyik. \_configuration[“key”] yazaraq məlumatı əldə edəcəyik.

\_userManager.CreateAsync(user,password); method’u yazaraq create edəcəyik. \_userManager.AddToRoleAsync(user,role); vasitəsi ilə də user’in role’nu yaradacayıq.

**InitializeSeed**

Project yeni bir mühitdə run olan kimi migration’lar avtomatik update-database command’ı ilə işə düşsün deyə də seed yazmaq olar. Bunun üçün InitializeAsync() method’u yaradılır və method daxilində \_context.Database.MigrateAsync() method’u çağırılır. Lakin bəzən bu method table’ların sayı aşırı çox olan zaman problem yarada bilir bunun üçün EnsureCreatedAsync() method’u localda əvəzlənə bilər lakin product publish olan zaman MigrateAsync() olaraq yazılmalıdır.

Yaradılan Seed’ləri Program.cs daxilində işə salacayıq. Bunun üçün ilk olaraq builder.Services.AddScoped<AppDbContextInitializer>(); şəklində yazmalıyıq.

Using(var scope = app.Services.CreateScope()){} daxilində aşağıdaki prosesləri yerinə yetirməliyik.

var initializer = app. Services.GetRequiredServices<AppDbContextInitializer> vasitəsi ilə servisə tanıtdığımız AppDbContextInitializer object’i çağırırıq. Və ya app.Services yerinə scope.ServiceProvider yazmaq olar.

AppDbContextInitializer class daxilində yaratdığımız bütün method’ları initializer object’i vasitəsi ilə program.cs daxilində işə salırıq.

**JWT(Json Web Token)**

Login olan zaman token generate olunmalıdır. Bunun üçün JwtSecurityToken object’i yaradılır və arqument olaraq string issuer, string audience,IEnumerable<Claim> claims, Datetime notBefore,Datetime expires və sonda SignInCreditional signInCreditional object göndərilir. Jwt token daxilində user’la bağlı dataları saxlayır

Issuer’da tokeni hansı domain’in yaradacağı göstərilir. Audience’də tokeni hansı domain’in qəbul edəcəyi göstərilir. Claims’də payload hissədə göndərilən datalar göstərilir. NotBefore token’in nə zaman yarandığını göstərir. Expires token’in nə zaman deactive olacağını göstərir. SignInCreditional application tərəfindən göndərilən security key’dir və decode zamanı görünmür.

Issuer , Audience və Security Key AppSettings.json daxilində saxlanılacaq.

Claim’lər isə key value məntiqində saxlanılacaq , new Claim(ClaimType.Type,user.Property);

SignInCreditional SecurityKey securityKey və string alqoritm qəbul edir. Biz isə securityKey’i string olaraq saxlamışıq bunu SecurityKey type’a çevirmək üçün SymmetricSecurityKey object bu şəkildə yaradılır new(Encoding.UTF8.GetBytes(securityKey); .Alqoritm olaraq da SecurityAlgorithms.HmacSha256 göndərilir.

Biz yaratdığımız token’i string type olaraq geriyə qaytarmalıyıq. Bunun üçün JwtSecurityTokenHandler object’i yaradılır və bu object’in WriteToken(jwtSecurityToken) method’u işə salınır və geriyə string dəyər qaytarır.

jwtSecurityToken.ValidTo bizə token’in nə vaxta qədər active olduğunu qaytarır

**Extension+**

Hazır type’lara bir-başa müdaxilə edib method əlavə edə bilmədiyimiz üçün Extension’lardan istifadə edilir. Bunun üçün yeni bir static class yaradılır və daxilindəki method’lar da həmçinin static olmalıdır. Yaradılan method’un qəbul etdiyi ilk parametr, this keyword’ü ilə yazılmış həmin type parametr olmalıdır. Daha sonra Extension tətbiq etdiyimiz type object’ə biz yaratdığımız Extension method’u tətbiq edə bilərik.

**Indexer+**

Hər hansısa class daxilində indexer yaratmaq üçün aşağıdaki şəkildə property məntiqində kod yazılır.

public returnType this[int index]

{

get{ return collectionObj[index]; }

set{ collectionObj = value;}

}

**Delegate**

Method’un(function) parametr kimi hər hansısa bir method’a göndərilməsi callback function adlandırılır.

Biz hər hansısa function’ı parametr kimi bir method’a göndərmək istəyiriksə delegate’dən istifadə edirik.Delegate type’ın body’si olmur.

Delegate function’lara type vermək üçün istifadə edilir, hər hansısa bir funcsiton’un geriyə qaytardığı dəyərin type’ı, qəbul etdiyi parametrlərin sayı və type’ı delegate ilə eyni olarsa o zaman function delegate olaraq yaratdığımız type’da olacaq.

Delegate type bu şəkildə yazılır:

accessModifier delegate static&non-static returnType delegateTypeName(parameters);

yuxarıda deyilən şərtləri ödəyən bütün function’lar artıq delegateTypeName type’dan bir function olmuş olur və hər hansısa bir method’a parametr olaraq göndərmək istəsək type olaraq delegateTypeName func yazıb göndərə bilərik

Biz yaratdığımız delegate vasitəsi ilə hər hansı bir function’u bu şəkildə yarada bilərik.

delegateTypeName funcName = delegate(parameters) {

return variable;

};

biz anonym funksiyaları da delegate type olaraq yaza bilərik və qəbul olunan parametr sayəsində göndərilən arqumentin type’nı yazmaq məcburiyyətində qalmırıq.

Məsələn:

Public void method(delegateTypeName func)

Delegate bool TestDelegate(int num);

bool TestMethod1(int num) {return num>0} ilə

bool TestMethod2(int num) => num>0 eyni mənanı kəsb edir.

Biz hər hansısa methoda TestMethod1 və ya TestMethod2’ni TestDelegate type’dan bir function kimi göndərmək yerinə gəlib həmin method’a num => num>0 yazsaq əgər burada TestDelegate type’a görə num’ın int type olduğu bilinir və qarşısına dataType yazılmır.

Method(TestMethod1);

Method(TestMethod2);

Method(num => num>0);

Public delegate R DelegateTypeName<in T, out R>(T variable);

Bu yazılış o deməkdir ki bu method T type qəbul edəcək və R type returnType qəbul edəcək. T type parameter qəbul etsin və R type parameter return etsin.

Action

Biz action vasitəsi ilə delegate type’la yazdığımız methodun eynisi yazırıq lakin fərqləri odur ki, action hazır delegate’ləri istifadə etmək üçündür. Action’un 17 fərqli overload’ı mövcuddur

Delegate int TestDelegate<T>(T str);

TestDelegate<string> Method1 = delegate(string str) {

Console.WriteLine(str);

};

TestDelegate<string> Method2 = str=>Console.WriteLine(str);

Action<string> Method3 = str=>Console.WriteLine(str);

Function

Functionlarda əvvəldə olan qəbul etdiyi type’dır sonda olan isə return type’dır

Function<T , R>

Predicate

Boolean dəyər qaytarır

**Reflection**

Assembly class’ı namespace olaraq reflection’un daxilində yerləsir.

Assemly asmb = Assembly.GetExecutingAssembly();

Bizə hal hazırda olduğumuz proyektin assembly’sini verir yəni .exe file’ı verir.Burada asmb daxilində proyektin adı, version’u və s məlumatlar yerləşir.

Type type = typeof(ClassFullName) bizə type qaytarır.

Type type = obj.GetType() bizə type qaytarır.

Typeof GetType’dan daha üstündür çünki typeof vasitəsi ilə biz birbaşa type adını yazdığımız üçün bunu istifadə edərək gedir type’ı götürür bu səbəbdən static class’ların da type’nı götürə bilirik.

Type type = asmb.GetType(“Class Full Name”);

Bizə type tipindən class’ı(type) qaytarır və daxilində namespace ilə bilikdə adını göndərdiyimiz class’ı saxlayır

Type[] types = asmb.GetTypes();

Bizə Type tipindən bir array qaytarır və daxilində bizim proyektimizdəki class’ları saxlayır.Foreach’ə salıb bir-bir bütün elementlərə müraciət edə bilərik.

foreach(Type type in types) {

Console.WriteLine(type.Name);

}

type.Name bizə class’ların yəni type’ların adını qaytaracaq.

type.FullName bizə namespace ilə birlikdə adını qaytaracaq yəni root’nu.

Type.GetProperty(“Property Name”);

Bizə PropertyInfo tipindən property qaytarır və daxilində həmin property’in adını və dataType’ı saxlayır.

type.GetProperties()

Bizə PropertyInfo tipindən array qaytarır və daxilində həmin class(type) daxilindəki property’lərin adını dataType’ı saxlayır.

PropertyInfo[] properties = type.GetProperties();

foreach(PropertyInfo property in properties) {

Console.WriteLine(property.Name);

}

type.GetMethod(“methodName”)

Bizə həmin class’ın(type) method’nu qaytarır.

type.GetMethods()

Bizə həmin class’ın(type) method’larını qaytarır.

type.Field(“fieldName”)

Bizə həmin class’ın(type) field’nı qaytarır.

type.Fields()

Bizə həmin class’ın(type) field’larını qaytarır.

BURA MƏNƏ XATIRLATICI KİMİ İSTİFADƏ EDİLİR QAYDADAN XARİCDİR.

SUAL:

Const verdiyimiz field ı necə get edə bilərik?

---------------------

Paradigma

Array indexləri ardıcıl yaddaş blokunda saxlayır, bu səbəbdən indexə görə axtarış,çağırma o(1) sürətidir və ən sürətli axtarış array’ə aiddir.

Arrayi valueyə görə axtarış,çağırma sürəti o(n)dir(worst case)

\*Time complexity Big O notation

Anonym function

İnt type‘ında olan hər şey stack yaddaşda saxlanılmır

Proqramlaşdırma dilinin oxunma ardıcıllığı

1.C# intro , 2.C# intro-2 , 2.control structure, 3-Method.Parametr-Argument, 4.Method Overloading. Ref-Out 5.class, 6. Inheritance, Constructor,7. StringBuilder ,8. Encapsulation, Access Modifiers

19.Generic type 30:00

20.Delegate 30:00

10.abstract 40:00

22.Collections 26:47

23.Reflection 40:00

REMOVE INSTERT

28.Ado.Net ,29.Sql Injection

30.Enity framework, 31.Create -ORM,32.Update, Delete, Read 35:00

1.Asp.Net intro, 2.Asp.Net DbContext, 3.Asp.Net Admin Panel

4.Asp.Net View, 5.Asp.Net CRUD(Read), 6.Asp.Net CRUD(Create)

7.Update,Delete, 8.1 File Upload Start, 8.2 FileStream

8.3 FileUpload Extension, 8.4 Delete File, 9.Repository Pattern

10. ASP.Net Identity start, 11 ASP.Net Register, 12.ASP.Net Login

Using directive

String.Empty

Exception global olaraq proqramı dəyandırmır amma error dəyandırır.

Static classın memberləri də static olmalıdır.

Console.WriteLine($”stringIfadə {variable}”)

Console.WriteLine(”stringIfadə {0} ”,variable) – bu yazılış o deməkdir ki biz vergül vasitəsi ilə variable’larımızı göstəririk və index məntiqi ilə {index 0’dan başlayır} onlara müraciət edirik

Thread istifadə edilərsə string istifadə etmək daha düzgündür

IDisposable interface’dən implement edən hər bir class’ın objecti close olunmalıdır.

Datatype? Nullable deməkdir

buider = WebApplication.CreateBuilder(args); bizim mvc application’umuzu build edir.

Builder.Services vasitəsi ilə biz application’umuzda nələri istifadə edəcəyimizi bildiririk.

App.run() method’na qədər bütün middleware’lər yazılır ondan sonra yazılanlar işləmir.

=null!; o deməkdir ki göstərdiyimiz property və ya field null ola bilməz.

Controller class’larının sonu controller ilə, View’ların sonu View ilə , Service’lərin sonu Service ilə bitməlidir.

Layler’ə bax

“” {} ’

builder.Services.AddControllers().AddJsonOptions(x =>

x.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);

bu kod sətri ilə ICollection property’i null qayıtmasın deyə Include vasitəsi ilə yazılmış funksiya run olan zaman cycle error verməsin deyə Program.cs daxilində yazılmalıdır.

Qeyd:

app hissə pipeline adlanır.

Target type object yaranan zaman new ObjectClass() yazmaq əvəzinə new() yazmaq kifayət edir.

IQueryable olan dəyişənə .Include(string entityName) və ya .Include(t=>t.Entity) vasitəsi ilə relation qurulmuş dataların da məlumatı gəlir. Aşağıdaki if şərti vasitəsi ilə params olaraq göndərilən string include arrayi yoxlanılır və include olunur.

if(includes.Length > 0 && includes is not null)

{

foreach(var include in includes)

{

query = query.Include(include);

}

}

modelBuilder.Entity<User>().HasIndex(u => u.UserName).IsUnique(); unique column təyin etmək üçün.

1-2-3-4-5-6-7-8-17-18-19-20-21

\*Override - Bir methodun override oluna bilməsi üçün aşağıdaki 3 şərtdən 1-ni ödəməlidir. abstract olmalı və ya virtual olmalı və ya override olunmuş bir method olması şərtdir.Yəni biz hər hansısa base classdaki methoda yuxarıdaki şərtlərdən heç birini tətbiq etməsək derived class daxilində override keywordünü istifadə edərək base classda hansısa methodun, method signaturesinə uyğun method yaza bilmərik.+

\*Sealed keywordü verilmiş classı möhürlüyür və ondan miras alınmasına icazə vermir.Sealed method isə derived class tərəfindən bu methodun override edilməsinin qarşısını alır.+

\*Static class həm də sealed class olduğu üçün bu classdan miras alına bilməz,object classdan başqa heç bir classdan və ya interfacedən miras ala bilməz.+

\*Interface classlardan miras ala bilməz lakin digər interfacelərdən istənilən qədər(cycle compile error verməmək şərti ilə) miras ala bilər.Classlar isə sırası ilə yalnız 1 classdan və istənilən qədər interfacedən(cycle compile error verməmək şərti ilə). +

\*Virtual keyword’ü verilmiş method vasitəsi ilə derived class, base classın methodunu ovveride edə bilər.Biz hər hansısa propertyni(get və set methodunu) override etmək istəsək virtual keywordü ilə bunu həyata keçirə bilərik+

\*Abstract classdan instance almaq olmur.Abstract class sealed və ya static ola bilməz.+

\*Abstract method yaratmaq üçün, Abstract class daxilində methodun access modifierindən sonra abstract keywordündən istifadə etmək lazımdır.Abstract methodun bodysi olmur.Abstract method yalnız abstract class və interface daxilində yazıla bilər. Bu şəkildə bizim derived classlarımız yaranan zaman məcburi şəkildə abstract methodların implement olunması tələb edilir, Virtual methoddan fərqi budur.

\*Structlar yalnız interfacelərdən miras ala bilər.Structlar sealed class olduğu üçün heçbir class onlardan miras alabilməz

Readonly

Const

Enum

As və digər castinglərin fərqi

Eagerloading və lazyloading

Abstraction

Deciamal və DECİMAL

Javascript

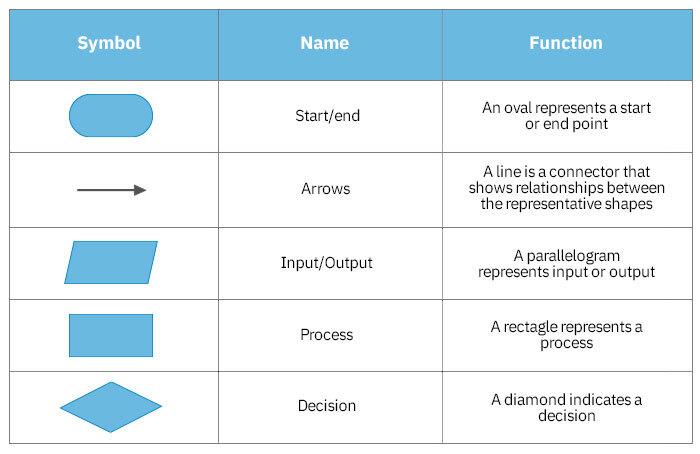
Var let const fərqi

Let və const block scopdurlar yəni hər hansısa bir block daxilində yeni dəyişən yaradılarsa

**SQL**

Normalizasiya – lazımsız dataları aradan qaldırmaq üçün bir dizayn texnikasıdır.

Normalizasiyanın tətbiqi-Normalizasiyanın tətbiqi üçün cədvəl bölünməlidir.Məsələn employee tableda country sütununa həm Ingilis həm English həm də EN yazmaq olar. Lazımsız yerə bu qədər uzatmaqdansa Country cədvəli yaradıb orada statik olaraq dataları saxlaya bilərik və ardı ilə country sütununda foreign key verərək ölkə idləri qeyd edə bilərik



**Ram- Stack və Heap yaddaş**

Ram 2 yerə bölünür stack və heap yaddaş. CSharp daxilində bizim yaratdığımız value type dəyişənlər stackdə saxlanılır. yaratdığımız reference type obyektlər heapdə onların referenceları isə stackda saxlanılır. Bunun səbəbi nomral şərtlərdə stack yaddaşa bir başa accessimiz olduğu halda heap yaddaşa biz bir başa olaraq accessimizi olmamasıdır, bunu bizim yerimizə stack yaddaş edir. Biz heap yaddaşdaki hər hanısa bir objectə müraciət etmək istəsək onun stackdə saxlanılan referenceına müraciət etməliyik.

**Static**

Static member’lər non-instance member’lərdir. Proyekt işə düşən zaman ramda bir başa yaranıb yer tuturlar, ramda mövcud olurlar. Non-static member’lər Static method’lar və s çağırıla bilməz.

**Static Constructor**

Heç vaxt call olunmur. Class‘dan ilk istifadə zamanı işə düşür. Və yalnız bir dəfə işə düşür.Static readonly property’lər init olunduğu yerdə və static constructor daxilində dəyər assing oluna bilər. Overload halı yoxdur.

**Static class**

Static class‘ların bütün class member‘ləri static olmalıdır, ondan instance almaq olmur.

Abstraction principle

Abstraction encapsulation prinsipinin alt prinsipi kimi qələmə verilir.

Abstraction’dan danışılan zaman əsasən Abstract class və ya Interface’lər üzərində danışılır. Lakin Abstraction sadəcə bunlardan ibarət deyil. Arxa tərəfdə necə işlədiyi bizə aid və ya lazım olmayan, necə işlətməli olduğumuzu bilməyimiz bizə kifayət olan hallar abstraction adlanır.

Is true false qaytardığı halda, As null və ya özünü qaytarır

if(person is Programmer) var programmer = (Programmer)person;

var programmer = person as Programmer;

if(programmer is not null) programmer.MethodOfProgrammer