**C# nədir**

C# Microsoft tərəfindən .Net texnologiyası üçün inkşaf etdirilən modern bir tam obyekt yönümlü proqramlaşdırma(OOP) dilidir və C ailəsinə daxildir. C# datatype sensitivity’dir.

**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

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

Var keyword’ü ilə variable’a value mənimsədildiyi zaman compile olaraq datatype müəyyən edilir və biz həmin dəyişənə 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’ın 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 edtiyimiz 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 collection’un 1-1 elementlərinə 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 dəyişiıməzdir 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).Biz hərhansısa bir code block’nu 2 və ya daha çox 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.

**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).

**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.

**Parametr və arqument**

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

**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 inistance 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 inistance 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 inistance 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

**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 inistance 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 inistance 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 inistance 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 inistance 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 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 Middle ware hissə.

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

Web server UI’ı yayımlayan hissədir.

Controller class’larının sonu controller ilə bitməlidir.

Service class’larının sonu service ilə bitməlidir.

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

app.MapControllerRoute(

name: “default”,

pattern:”{controller}/{action}/{id?}”

);

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 , ContentResult , JsonResult da qaytara bilərik lakin hər 2’si də IActionResult interface’dən implement edilər deyə ümumi olaraq IActionResult type olaraq return edəcəklər.

**View()**

ViewResult qaytarmaq üçün ilk başda Views adlı folderimiz olmalıdır. Və bu folder daxilində hər bir Controller’in adına uyğun olaraq folder yaradılır. Və həmin Controller folderinin daxilində isə Actionun adına uyğun olaraq Razor View(cshtml file) yaradılır.

Biz Controller’in method’u daxilində return View() yazanda reflection vasitəsi ilə arxa tərəfdə controllerin adına uyğun olaraq Views folderi daxilində Controller’in adına uyğun olaraq View folderi 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)**

@ işarəsi ilə biz hər hansısa c# elementinə müraciət edə bilərik və ya @{} block’u daxilində istədiyimiz kodları yaza bilərik.

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

**View start**

Əgər eyni layout bir neçə səhifədə istifadə edilirsə parametr olaraq səhifə layout qeyd olunmasın, 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 View start istifadə edildiyini başa düşür

**ViewBag , ViewData, TempData**

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

ViewData[“key”] = value; vasitəsi ilə Controller’də hər hənsası dəyər set edib, cshtml file’da @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.

**wwwroot**

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

**Models**

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

**AppDbContext**

DbContext üçün Solution daxilində DataAccess adında yeni project(library) yaradırıq və onun daxilində Contexts adında folder yaradıb AppDbContext class’ı burada saxlayırıq.Bu səfər AppDbContext class’ın constructorunu 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.

Appsettings.json vasitəsi ilə biz ConnectionString olaraq verdiyimiz dəyəri saxlaya bilərik.

“ConnectionStrings”: {

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

}

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

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

**@Model**

Controller’imiz constructor ilə AppDbContext object’i qəbul edir. Bizim return zamanı View daxilində göndərdiyimiz dəyəri @model object vasitəsi ilə qəbul edirik və cshtml file daxilində @Model yazaraq istifadə edirik

**View Import**

Views daxilində View Import file vasitəsi ilə biz using’ləri saxlayıb bütün cshtml file’larına aid edirik.

**Area**

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

Area yaratmaq üçün Start up project’ə sağ click eləyib Scaffolded item’ə clickləyirik və açılan pəncərədən MVC Area seçilir.Bizə verdiyi MapControllerRoute’u gəlib Midleware’də yuxarıya yazırıq.

Daha sonra Area Controller daxilində DashboardController yaradılır. Və annotation olaraq [Area(“AreaNAme”)] qeyd edilir.

**View Models**

View Models’in istifadə olunmasının səbəbi odur ki, biz View’a eyni vaxtda bir neçə Data göndərməli ola bilərik. View Models folderi yaradırıq. Göndərəcəyimiz dataları View Models’in daxilində property şəklində saxlayacayıq.

**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

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

Sredlər istifadə edilərsə string istifadə etmək daha düzgündür

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

Datatype? Nullable deməkdir

“” {} ’

