**C# nədir**

C#(C Sharp) Microsoft tərəfindən .Net texnologiyası üçün inkşaf etdirilən modern, tam obyekt yönümlü(OOP) və type safety xüsusiyyətinə sahib bir proqramlaşdırma dilidir. C#, C ailəsinə daxildir və Java kimi məşhur dillərə yaxındır. Ilk dəfə yanvar 2000-ci ildə elan edilmiş C#, 1.0 versiyası ilə yanvar 2002-də .Net framework’ün bir parçası olaraq istifadəyə buraxılmışdır. Bu dilin C# adlandırılmasının səbəbi C++’ın bir üst versiyası olmasıdır. C++’ın, C dilinin bir üst versiyası olduğunu bildirmək üçün ++ operatorundan istifadə ediblər və beləliklə dilin C++’ın da bir üst versiyası olduğunu bildirmək üçün C#(C Sharp) yazılaraq bu şəkildə adlandırılmışdır. Hal-hazırda ilk zamanlardan fərqli olaraq C#, platform independence(cross platform da deyilir) prinsipi ilə işləyən dildir. Yəni Windows əməliyyat sistemi ilə yanaşı digər əməliyyat sistemlərində və ya platformlarda da işləmək üçün nəzərdə tutulmuşdur.

**Type safety nədir?**

Bir proqramlaşdırma dilində type safety, dilin type error’lardan qaçmağına kömək edən abstract bir konstruksiyadır. Hər proqramlaşdırma dili gizli type safety səviyyəsinə malikdir. Beləliklə proqram compile olunan zaman compiler type’ları təsdiqləmək üçün type safety konstruksiyasını tətbiq edəcək və dəyişənə yanlış type təyin etməyə çalışsaq,compile time xəta verəcək. Type safety yalnız compile zamanı yox, həm də run time işə salınır.

**Compiler nədir?**

C# compile olunmuş dildir, yəni compiler insan dilinə yaxın olan C# kodlarını maşın tərəfindən icra oluna bilən kodlara(machine-executable code) çevrilməlidir.

C# üçün əsas compiler C# Compiler (csc.exe) olaraq adlandırılır. Bu kompilyator .Net Framework və .Net Core(və onun varisi .Net 5 və daha sonraki versiyaları) bir parçasıdır. C# source kodlarını(.cs file daxilində) MSIL(Microsoft Intermediate Language) kimi bilinən Intermediete Language(IL) code’a çevrilməsindən məsuldur. Kompilyasiya prosesinin qısa icmalına baxsaq,

Biz text editor və ya IDE(Integrated Development Environment) istifadə edərək C# ilə program tərtib edirik. Bu program C# source code adlanır.

C# Compiler’ı(csc.exe) bizim C# soruce kodlarımızı emal edir, syntax təhlilini həyata keçirir və müfaviq olaraq Intermediate Language(IL) kodunu generate edir. Bu kompilyasiya prosesi adlanır.

Intermediate Language(IL) code hər hansı xüsusi aparat və ya əməliyyat sisteminə xas deyil, ara təmsilidir.

Program run olunan zaman .Net mühitində Common Language Runtime(CLR), IL code’nu icraçı avadanlıqa xas maşın dilinə çevirir və gələcəkdə istifadə üçün cach saxlayır. Bu Just-In-Time(JIT) kompilyasiya adlanır.

Bu icmal nətisəcində C#’ın həm Compile olunan həm də run olan zaman JIT kompilyasiya olunan bir dil olmağını təmin edir.

**Solution nədir?**

Solution daxilində bir və ya bir-birləri ilə əlaqəli olan birdən çox project’in saxlandığı workspace, yəni iş sahəsidir.

**Top-Level Statements**

C# 9.0 ilə gəlmiş Top-Level Statements anlayışı bizə əvvəllər Program.cs daxilində yazdığımız main method’u yazmadan kodlarımızı yazıb compile edilə bilməsi üçün şərait yaradır. Yalnız Program.cs daxilində Top-Level Statements’dən istifadə edilə bilər. Compile olduqdan sonra özü Assembly səviyyəsində main method yaradıb, bizim yazdığımız kodları onun daxilinə əlavə edərək işlədir. Compile zamanı class adından Program.cs olduğu başa düşüləcək və main method arxa tərəfdə yaranacaq. Top-Level Statements microservices arxitekturasında kodun inkşaf etdirilməsində sürət qazanmaq üçün istifadə edilir.

**Comment line**

Kritik nöqtələrdə yazılan kodları, başqa developerlərin və ya bir müddət sonra özümüzün daha rahat oxuya bilməsi üçün, comment sətrlərdən istifadə edilir.Comment sətrlər vasitəsi ilə biz yazılan kodların məğzini, yazılma səbəbini açıqlayırıq.Compile zamanı yazılan sətrin comment sətr olduğu anlaşılır və həmin sətr maşın dilinə compile olunmur.

//comment - 1 sətri comment’ə almaq üçün istifadə edilir.

/\*

Çox sətrli comment

\*/ - 1’dən çox sətri comment sətri kimi göstərmək üçün istifadə edilir.

**Region**

Yazılan kodları bir .cs file daxilində kateqorik bir hala gətirmək üçün, yəni qruplaşdırmaq üçün Region operatorundan istifadə edilir.Region istifadə edərək kodlarımızı daha oxunaqlı hala gətiririk.

#region RegionName

Code lines

#endregion

**TODO**

Bir Visual Studio funksiyası vasitəsi ilə biz .cs file daxilində comment sətri yazan zaman hər hansısa bir tapşırıq vermək istəsək və ya xatırladıcı yazmaq istəsək todo keyword’dən istifadə edirik.Yazılışı da bu şəkildədir //todo comment . View pəncərəsində Task List’ə daxil olaraq file daxilindəki todo’ları görə bilərik.

**Variable**

Program daxilində hər hansısa bir datanı ramda saxlamaq və onu lazım olan zaman ramdan çağırmaq üçün variable’dan istifadə edilir. Hər bir variable təyin olunarkən hansı type‘da dəyər saxlaya biləcəyi əvvəlcədən datatype‘lar vasitəsi ilə qeyd edilir. C#, qeyd etdiyimiz kimi type-safety language olduğu üçün hər bir dəyişənə verilən type və ona mənimsədilən dəyər uyğun olmalıdır.

**Datatype**

Datatype‘lar Value type, Reference type və Pointer(unsafed block daxilində) type olaraq ramda saxlanma yerlərinə görə 3 yerə bölünür. Value type variable‘lar dəyəri ilə birlikdə stack yaddaşda, Reference type variable‘larda isə variable’ın referansı stack, dəyəri isə heap yaddaşda saxlanılır.

**Value Data Types**

Value type’lara Integral numeric types, Floating-point numeric types, Boolean və Character daxildir.

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

**Unsigned Integral Numeric Types**

Unsigned yəni işarətsiz value type’lar, işarəli bitlərdən istifadə etmirlər bu səbəbdən işarəli value type’lardan fərqli olaraq daha geniş pozitif dəyər aralığına sahibdirlər.

byte (Yaddaşda tutduğu yer 1 byte) 0 to 255

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

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

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

**Signed Integral Numeric Types**

Signed yəni işarəli value type’lar, dəyərin işarətini bildirmək üçün işarəli bitlərdən istifadə edirlər.

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

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

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

**Floating Point Numeric Types**

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

**High-precision decimal Floating-point Numeric Type**

Decimal typeByte’lardan əmələ gəlmişdir və value type’dır.

Float type variable yaradan zaman dəyərin sonuna F,Double type variable yaradan zaman D və Decimal type variable yaradan zaman isə M yazılmalıdır.

**Character Type**

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

**Boolean Type**

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

**Default Value**

Value type variable’ların default dəyəri 0’dır

Char type variable’ın default dəyəri ‘/0’ dır.

Boolean type variable’ın default dəyəri false’dur

Reference type variable’ın default dəyəri null’dır.

**Nullable**

Verilmiş hər hansısa variable’ın type’na uyğun dəyərlə yanaşı null dəyərə də sahib ola biləcəyini göstərmək üçün variable T? variable = value; şəkilində təyin olunur. Burada T bütün əsas type’ları işarə edir. Əsas type’ların qeyri-müəyyən dəyərini təmsil etmək üçün istifadə edilir.

Nullable<T> variable = value; Nullable type(yəni struct və ya enum type) variable’ların heç bir dəyərə sahib olmadığını bildirən əlavə null bir dəyər saxlaya bilərlər. Sturct’lardan başqa digər type’ları bu şəkildə istifadə edə bilmərik.

Nullable<T> və ya T? İstifadə olunma məqsədi ondan ibarətdir ki, məsələn bizim məlumat bazamızda hər hansısa sütun yalnız true və ya false dəyər qəbul edir. Əgər heç bir data verilməyibsə biz bunu bool dəyişənə mənimsədə bilməyimiz üçün Nullable<bool> və ya bool? Olaraq dəyişənin tipini göstərməliyik.

**?? Null-Coalescing Operator**

?? olaraq bilinən Null-coalescing operator null dəyəri idarə etmək üçün qısa yol təqdim edir. Əgər null dəyərə sahib deyilsə sol operandını qaytarır əks təqdirdə sağ operandını qaytarır.

Məsələn nullable olan “nullableNum” int dəyişən və 2-ci bir “num” int dəyişən verilmişdir. Əgər bizim null dəyər ala bilən “nullableNum” dəyişənimiz null’dan fərqlidirsə 2-ci dəyişənimiz “num”a assign olunacaq əks təqdirdə operatorumuzun sağ operandına assign olacaq.

int? nullableNum = null

int num = nullableNum ?? 5;

**??= Null-Coalescing Assignment Operator**

??= olaraq bilinən Null-coalescing assignment operator dəyişənə qiymət təyin etmək üçün yalnız hal-hazırda null olduqda istifadə olunur.

string name = null;

string defaultName = “cavanshir”;

name??=defaultName;

Bu da o deməkdir ki əgər name variable null olarsa ona defaultName’i mənimsət.

**Var**

Var keyword’ü ilə variable’a value mənimsədildiyi zaman compile zamanı datatype müəyyən edilir. Biz həmin variable’a başqa bir yerdə fərqli datatype’dan bir value mənimsədə bilmərik. Bunu etsək compile time error verəcək. Əlavə olaraq var keyword’ü ilə variable yaranan zaman dəyər mütləq assign edilməlidir ki, compile zamanı hansı datatype’da bir variable olacağı bəlli olsun. Əks təqdirdə compile time error verəcək.

Bu keyword’ün yaradılma məqsədi fərqli dillər arasında dəstəklənməyən datatype dəyişənləri saxlamaq üçün ortaq keyword olsa da, günümüzdə istifadə məqsədi dəyişənin type’nı yazmağa tənbəllik etməkdir. Compiler’a əlavə iş gördürməmək üçün type’nı bildiyimiz dəyişənləri var keyword’ü ilə təyin etməməliyik.

Var ilə object arasındaki fərq odur ki, var keyword’dür, object isə type’dır. var ilə təyin etdiyimiz dəyişən stack yaddaşda saxlanılır, lakin object type dəyişən yaddaşda object şəklində saxlanacaq və daxilində həmin dəyəri saxlayacaq. Boxing etdiyimiz üçün unboxing edərək həmin dəyəri istifadə edə bilərik.

**Dynamic**

Dynamic keyword’ü ilə assign olunmuş variable’a hər dəfə müxtəlif datatype dəyər mənimsədilə bilər. Runtime olaraq datatype mənimsədildiyi üçün var keyword’dən fərqlənir və onun runtime versiyasıdır.

**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 yəni əməliyyata 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(və ya əməliyyat) 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

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

**Ternary Operator**

var variable = (condition)? ValueIfConditionIsTrue: ValueIfConditionIsFalse

Ternary operator vasitəsi ilə biz variable’a if-condition istifadə etmədən, 1 sətr olaraq true və ya false case’ə görə 2 value’dan 1’i mənimsədə bilərik.

**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 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 case’də hər case daxilində müəyyən statement’dən sonra break || return || goto case VALUE keyword’lərindən biri yazılmalıdır. Əks təqdirdə compile time olan “fall-through” error meydana gələcək. "goto case VALUE” ifadəsi bizim case halımız işə düşdükdən sonra müəyyən bir case’ə yönləndirmə etmək üçün istifadə edilir.

Hər hansısa case’dən sonra heç bir şey yazmayıb bir başa olaraq başqa bir case’ə keçə bilərik. Bu da o deməkdir ki əgər bu case doğru olarsa ondan sonra gələn case daxilindəki proses həyata keçiriləcək.

Yuxarıda danışdıqlarımızın hamısına bir misal gətirək.

switch(expression) {

case value1:

case value2:

Statement2;

break;

case value3:

Statement3;

Goto case value4;

case value4:

Statement4;

//yuxarıda sadalanan keyword’lərdən heç biri işlənmədiyi üçün compile zamanı fall-through error’u verəcək.

case value5:

Statement5;

return;

...

case valueN:

StatementN;

break;

default:

StatementDefault;

break;

}

**Switch case expression**

var month = “October”;

var monthCode = month switch {

“January” => 1,

...

“December” => 12,

\_ => -1

};

Switch case expression c# 8.0’dən sonra gəlmişdir. Switch case expression hər hansısa variable’a müxtəlif case’lərdə dəyər mənimsətmək üçün istifadə edilir. Switch case və ya müxtəlif else-if condition’lar yazmaqdansa 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.

int num = 6;

string result = number switch{

<= 0 => “0-dan kiçikdir və ya 0-a bərabərdir.”,

> 0 and <5 => “1 və 4 aralığındadır 1 və 4-ün özü də daxildir.”,

>= 5 and <=10 when number %2 == 0 => “5 və 10 aralığında cüt ədəddir”,

>= 5 and <=10 => “5 və 10 aralığında tək ədəddir”,

\_ => “10-dan böyükdür.”

};

when statement qaydası qalıb

valueCase => expression 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.

**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 hər hansısa array, collection 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. Nəzər yetirmək lazımdır ki for loop’undan daha sürətlidir. C# 8’dən sonra gəlib.

**Break**

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

**Continue**

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**

Return keyword’ü daxilində yazıldığı method’u dəyandırır və özündən sonra gələn code sətrlərini işə salmır. Əgər method void method deyilsə yəni hər hansısa bir return type’ı varsa bu type’a uyğun olaraq dəyər return keyword’ü vasitəsi ilə qaytarılmalıdır.Əgər method void method’dursa müəyyən bir şərt daxilində bu method’u sonlandırmaq üçün return keyword’ü 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. Əsasən value type üçün istifadə edilir 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.

**OOP(Object Oriented Programming)**

OOP, yəni obyekt yönümlü proqramlaşdırma, developerlərin entity’ləri və object’ləri təşkil etməsinə imkan verən müasir proqramlaşdırma dili konseptidir. OOP’in bir sıra mənbələrdə 3 əsas 1 əlavə, bir sıra mənbələrdə isə 4 əsas prinsipin olduğu qeyd olunur.

* Encapsulation
  + Abstraction
* Inherientce
* Polymorphism

**Object**

Object, hər hansısa bir class’dan(type) törəyən və daxilində bir neçə data’nı saxlamaq üçün istifadə olunan obyekt’dir. Əlavə olaraq C#’da Object type da mövcuddur. Object reference type’dır və bütün type’lar Object type’dan törəyiblər yəni bütün class’lar Object class’dan implicitly miras almışdır. Bu səbəbdən object, bütün type dəyərləri özündə saxlaya bilir. Object type dəyişənlər dataları RAM’da object type olaraq saxlayır lakin bununla yanaşı data’ların özünə məxsus olan type’larını da özündə saxlayır. Bu da o deməkdir ki object’in daxilində saxladığı variable’ı öz type’ı ilə təkrar əldə edə bilərik.

**Class**

Class’lar C# typle’larının ən təməlidir. OOP’ni dəstəkləyən proqramlaşdırma dillərində hər hanısa bir object yaratmaq üçün ilk öncə onun strukturunu və davranışı təyin etmək lazımdır, yəni onu modelləşdirmək lazımdır. Struktur və davranış dedikdə, daxilində hər hansısa dəyər(lər) saxlayacaqsa field & property, hər hansısa həyata keçirəcəyi əməliyyat(lar) varsa method, array’lərdə tanış olduğumuz index məntiqi ilə çağrılma olacaqsa indexer başa düşülür. Bunun üçün də Class məntiqindən istifadə edilir. C#’da Class, object’ləri modelləşdirmək üçün istifadə edilən plan(blueprint) və ya şablondur(template). Əlavə olaraq Class’lar yaddaşda(ramda) heç bir yer tutmur. Class OOP’də əsas konseptdir. Hər bir class arxa tərəfdə dilin ən başında duran Object class’dan miras alır. Object’lər, object model olan Class’lardan törəyir, ramda bu object’lər yer tutur. Yaddaşdan asılı olaraq istənilən sayda object yaratmaq olar(instance almaq olar). Class’ın strukturu və davranışı ondan instance alınan bütün object’lərə şamil olunur, lakin hər birinin field & property’ləri fərqli dəyərlər saxlayır.

**Class’ların yaranması**

Yeni Class’lar yaranan zaman Class deklarasiyasından istifadə edilir. Class deklarasiyası header ilə başlayır. Bu header ardıcıllıqla ilk olaraq attribute və modifier, class keyword, Class’ın adı, başqa bir Class’dan miras alma varsa əgər base Class’ın adı, Interface(lər) implement edəcəksə Interface(lər)’in adı yazılır. Artıq bu class’ın yaranması ilə program üçün yeni bir type təyin olunmuş olur. Class’ları 3 şəkildə yaratmaq olar: Namespace daxilində, namespace xaricində və başqa bir Class’ın daxilində(nested Class). Hər biri üçün bilməli olduğumuz vacib ortaq xüsusiyyətlərindən biri eyni namespace daxilində eyni adda başqa bir class’ın yaradıla bilməməsidir. Class yaranan zaman daxilində static olaraq təyin etmədiyimiz member’lər(field, property, constructor, method) hər biri heap yaddaşda saxlanılacaq. Class’dan törəyən object’lər özlərinə type olaraq Class’ın adını götürəcək və new keyword’ü ilə constructor’u işə salınaraq törəyəcək.

**Constructor**

Constructor return type’ı olmayan, yalnız object instance alınan zaman(həmçinin object new’lanan zaman da adlanır) işə düşən method’dur. Class daxilində hər hansısa bir Constructor yazılmadığı halda C# soruce code’larımız compile olan zaman Assembly səviyyəsində class üçün boş Constructor(Parameterless və ya Default Constructor olaraq da bilinən) təyin olunacaq. Əgər class daxilində default və ya parametr qəbul edən Constructor(lar) təyin olunarsa Assembly səviyyəsində boş Constructor təyin olunmayacaq. Constructor’lar həmçinin bir-birlərini də çağıra bilər. Əsasən bu hala object yaranan zaman ona default dəyər(lər) set etmək lazım olarsa rast gəlinir. C# 9 ilə gəlmiş yeni özəllik vasitəsi ilə artıq object instance alınan zaman, əgər reference’ın type’ı ilə instence alınan object’in type’ı eyni olarsa sadəcə new() yazmaq kifayət edir.

C# 9’dan əvvəl => T obj = new T(); C# 9’dan sonra => T obj = new();

**Field**

Field, Class’dan törəyən object(lər)’in daxilində dəyər saxlamaq üçün istifadə olunan, Class’ın scope’ları daxilində təyin olunan variable’ların adıdır. Field’lar təyin olunan zaman heç bir dəyər set olunmazsa öz type’na uyğun olaraq default value avtomatik olaraq set olunar. Bu yalnız Field’lara xas xüsusiyyətdir, method daxilində təyin olunan dəyişənlərdə default value set olunmur.

**Property**

Property’lər isə get və set method’u olan Field’lardır. Property’lərin adları təmsil etdikləri Field’ın adının baş hərfi böyük olan şəkildə yazılır, və təmsil etdikləri Field’ın type’nı alır. Yazılışdaki fərqləri Property’in adından sonra scope daxilində get və set yazılır, gedişata uyğun olaraq get və set üçün də ayrıca scope açılıb hər biri müəyyən əməliyyat yerinə yetirilə bilər. Property’lər compile olan zaman onlar üçün ayrıca get və set method’lar yaranır.

Class’ın hər hansısa Field’ın access modifier’ı private olub onun üçün bir Property yaradılarsa və get olunan zaman geriyə private olan Field’ın dəyərini qaytarıb, set olunan zaman gələn dəyəri value keyword’ü ilə tutub field’a mənimsədərsə bu Property Full Property adlanır.

Full property’lərə misal göstərsək

private int \_num - Field

public int Num {get{return \_num;} set{\_num = value;}} - Property

**Readonly**

Readonly dedikdə hər hansa bir field’a və ya property’ə dəyər assign olunmasının(set olunmasının) qarşısını almaq, yalnızca dəyəri oxuna bilməsini(get oluna bilməsini) təmin etmək məqsədi ilə istifadə olunan bir anlayış başa düşülür. Readonly anlayışı field & property’lərə tətbiq edilir. Readonly field & property’ə yalnız təyin olunduğu yerdə və Constructor daxilində dəyər mənimsədilə bilər(başqa bir yerdə dəyər assign edilən zaman compile time olan readonly errorla qarşılaşırıq). Lakin istənilən yerdə dəyəri oxuna bilər. Readonly field & property təyin olduğu yerdə dəyər assign olunmasa da olar.

Field’ın Readonly olması üçün biz field təyin olunan zaman “readonly” keyword’dən istifadə etməliyik. Property’in Readonly olması üçün isə property təyin olunan zaman sadəcə get method’u yazılır, set method’u təyin edilmir.

**Init only setters**

Property üçün readonly ilə eyni məntiqdə işləyən lakin əlavə bir özəlliyi olan anlayış C# 9 ilə bərabər gəlmiş Init only setters anlayışıdır. Bildiyimiz kimi readonly property’lər get method’u olub set method’u olmayan property’lərdir. Readonly property’lərə yalnız təyin olunduqları yerdə və constructor daxilində dəyər set etmək olur. Əgər property’ə get method’la yanaşı həm də init təyin olunmuşdursa readonly’dən əlavə olaraq object’in initialize olunduğu yerdə də property’ə dəyər set oluna bilər.

**Constant**

Constant dedikdə hər hansısa bir field’ın constant bir dəyərə sahib olması və heç bir situasiyada dəyişilə bilməməsi başa düşülür. Readonly’dən fərqli olaraq yalnız field’lara şamil olunur, və mütləq təyin olunduğu yerdə dəyər assign olunmalıdır. Əks təqdirdə compile time error verir. Const olaraq təyin olunan Field’lar implicitly olaraq static olur.

**Encapsulation**

Class daxilində təyin olunan field’lar kənardan bir başa əl çatan olmamalı, və istənilən value set oluna bilməməlidir. Çünki proqramın gedişatında komandanın digər üzvləri və ya uzun bir müddət sonra özümüz də yanlışlıqla field’a set olunmamalı hər hansısa bir dəyəri set edə bilərik. Bunun qarşısını almaq üçün də field’ların access modifier’ları private olaraq təyin olunmalı, həmin field’lar üçün də property’lər təyin edilməlidir. Beləliklə property’in get və set methodları vasitəsi ilə field’a hansı dəyərin set olunub-oluna bilməyəcəyini gedişata uyğun şəkildə kontrol edə bilərik. Və bu anlayış OOP’in prinsiplərindən biri olan Encapsulation yəni kapsullama adlanır. Qısaca Encapsulation object’in datalarına(field’larına) kənardan kontrollu şəkildə dəyər set olunması və kontrollu şəkildə dataların oxunmasını həyata keçirilməsidir.

**Function & Method**

Function bir sıra ifadələri daxilində saxlayan,müəyyən bir tapşırığı yerinə yetirən kod blokudur. Function’lar geriyə hər hansısa type dəyər qaytara bilər(return edə bilər) və ya void(heç bir dəyər qaytarmayan) ola bilər. 2’dən artıq yerdə təkrarlayacağımız kod sətrlərini DRY(Don’t Repeat Yourself) prinsipini qorumaq üçün function halına çıxarmaq lazımdır. Function’ların istifadə məqsədi isə, təkrarçılığın qarşısını almaq və kodu optimallaşdırmaqdır.

Methodlar, Function’ın Class daxilində təyin olunmuş formasıdır. Method’lar Class member’dir və onun data’ları ilə çalışır. C#’da Method’lar hər zaman bir class’a(type’a) aid edilir.

Method yazılışı bu şəkildə olacaq. İlk olaraq Access modifiers, daha sonra optional olaraq optional modifiers , return type və ya void , method name və sonda method parametr gəlir.

Method təyin olunan zaman hər hansısa access modifier yazılmazsa class daxilində digər bütün class member’lər kimi(constructor xaric) private, struct daxilində isə internal olur.

**Parameter & Argument**

Method təyin edilən zaman onun qəbul etdiyi hər hansısa variable parametr adlanır. Method çağırılan zaman method’un qəbul etdiyi parametr(lər)’ə uyğun olaraq göndərilən variable isə arqument adlanır.

**Method Signature**

Həmçinin method’lar təyin olunan zaman müəyyən bir imzaya sahib olurlar, bu da Method signature adlanır. Method signature’də 4 əsas məqama diqqət edilir. Method’un adı, qəbul etdiyi parametrlərin sayı, parametrlərin type’ı və parametrlərin qəbul edilmə ardıcıllığı. Yadda saxlamaq lazımdır ki, method’un return type’ı, static və ya non-static olması method signature’ə aid deyil.

**Default Value**

Method’a hər hansısa parametr təyin edən zaman biz bu parametr’ə default bir dəyər mənimsədə bilərik. Yəni biz method’u call edən zaman həmin parametrə uyğun olaraq bir arqument göndərməsək default olaraq default value mənimsədiləcək. Əks təqdirdə yeni dəyər set olunacaq və default value override ediləcək.

**Params**

Method yaranan zaman eyni type’dan neçə parametr təyin etməli olduğumuzu bilməyə bilərik. Bu problemin qarışısını almaq üçün params’dan istifadə etməliyik. Sadəcə göndərməli olduğumuz type array yaratmalı və qarşısına param yazmalıyıq. Beləliklə eyni type’dan istənilən qədər arqument göndərə bilərik.Avtomatik olaraq göndərilən arqumentlər bir array halına salınacaq.

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

**Inheritance**

Inheritance miras alma deməkdir. Müəyyən bir class eyni anda yalnız bir class’dan miras ala bilər, istənilən sayda class’a miras verə bilər. “Miras alma” prosesi zamanı derived class təyin olunduğu yerdə adından sonra :baseClassName yazaraq miras ala bilərik. Access Modifier’dan asılı olaraq, base class’dan miras alınan hər bir field & property, method və constructor və bir sözlə desək Class Member’ləri derived class tərəfindən istifadə edilə bilər. Əgər miras alınan base class’ın boş constructor’u yoxdursa derived class daxilində boş constructor və ya onun overload halları yazılmalı, və bu constructor(lar) base class’ın hər hansısa bir constructor’nu çağırmalıdır.

**Override**

Base class’ın hər hansısa bir method’nun body’sini, Derived class daxilində özünə uyğun şəkildə dəyişmək override əməliyyatı adlanır. Həmçinin Base class daxilində təyin olunmuş property’lərin get və set method’larını da override edilə bilən hala salmaq olar. Override ilə bağlı diqqət yetirməli olduğumuz müəyyən məqam mövcuddur. Override zamanı Method Signature qorunmalıdır.

Bir method 3 situasiyada override oluna bilər, override keyword’ü işlənə bilər. Virtual keyword’ü ilə təyin olunmuşdursa, özü override olunmuş bir method’dursa və ya abstract method’dursa. Bu 3 situasiyadan heç olmazsa 1’i ödənmədikdə Override prosesi həyata keçmir, Override keyword yazıla bilmir. Override keyword yazılmadan təyin olunan method 3 şərtdən heç olmasa birini ödəsə belə Override olunmuş sayılmır.

**Overload**

Bir class daxilində eyni adda olan bir neçə method’un yazılmasına Overload deyilir. Method’ların adının eyni olmasına baxmayaraq bu method’lar Method Signature qorunaraq yazılır.

**Polymorphism**

Polymorphism yunan sözü olub, poly - bir çox sayda və morph - form mənasını verir. Polymorphism eyni adda olan bir çox method’un fərqli formlarda istifadə olunmasına (yəni body’lərinin fərqli olmasına) deyilir.   
Polymorphism, Override(runtime, dynamic) və Overload(compile time, static) olaraq 2 yerə bölünür. Əsasən Polymorphism’in izzahı sadəcə regular Class daxilindəki virtual method və ya Abstract class’lar daxilindəki abstract method’ların Override’ı üzərindən olsa da bu kifayət etmir. Polymorphism, Interface’lər daxilindəki body’si olmayan method’ların implement prosesində, operatorların və method’ların overload’ı prosesində baş verir.

**Sealed**

Təyin etdiyimiz Class’dan miras alınmasını istəmiriksə bunun qarşısını almaq üçün Class yaranan zaman Sealed keyword’dən istifadə edilir və Class möhürlənir. Bu səbəbdən Class’dan miras alına bilmir.

Təyin etdiyimiz method’un override olunmamasını istəyiriksə Sealed keyword’dən istifadə edilir və bununla da method override oluna bilmir.

**Indexer**

Array’lərdə və s qarşılaşdığımız index(yəni arr[i]) məntiqini biz custom olaraq özümüz də lazım olarsa yarada bilərik. Bunun üçün bizim class daxilində property məntiqində return type’ı R olan

public RType this[int index]{

get{return collectionObj[index];}

set{collectionObj[index] = value; }

}

indexer’i yazmağımız kifayət edir. Bununla biz bildirmiş oluruq ki həmin class’ın referece’ı qarşısında [index] yazsaq, uyğun olaraq class daxilindəki collection’ın əgər varsa həmin indexdəki dəyərini almış olarıq.

**Writeonly Property**

Writeonly property Readonly property’dən fərqli olaraq sıxlıqla rast gəlmədiyimiz hallarda istifadə edilir. Bu property’lərə yalnız dəyər set oluna bilir, oxumaq mümkün olmur. Əsasən bu property’lər dəyərləri hər hansısa method daxilində oxunmaq və ya işlənmək üçün lazım olduqda təyin olunur.

**Static Class**

Class, program run olan zaman regular class’lardan fərqli olaraq ram’da yaranması üçün onu Static olaraq təyin etmək lazımdır. Static class’lar implicitly sealed’dir(explicitly olaraq sealed keyword’ü ilə təyin etmək istəsək errorla qarşılaşacayıq) yəni Static class’lardan miras almaq olmur. Həmçinin Static class’lar object class’dan başqa heç bir class və ya interface’dən miras ala bilməz. Static class daxilində instance member yəni non-static member’lar təyin etmək olmaz. Static class’ın bütün member’ları static olmalıdır. Static class’dan object instance almaq olmur.

**Static Class Members**

Static keyword’ü ilə təyin olunan class member’lar non-instance member’lardır. Yəni bir başa class’a müraciət etdikdə əlçatandırlar, class’dan alınan instance ilə yox. Proyekt işə düşən zaman ram’da yaranıb öz yerlərini tuturlar. Static member’lar daxilində non-static member’lər çağırıla bilməz və ya istifadə oluna bilməz. Non-Static member’lar daxilində isə Staticmember’lar çağırıla bilər. Static member’lar initonly ola bilməz.

**Static Constructor**

Static constructor program run olduqdan sonra class’a ilk müraciət olunan anda işə düşür, bir də heç bir şəkildə işə düşmür və call oluna bilmir. Static constructor’un overload halı və ya override oluna bilmə xüsusiyyəti yoxdur. Static readonly və ya static init only property’lərə yalnız təyin olunduqları yerdə və Static Constructor daxilində dəyər set olunur. Static Constructor’un əsas məqsədi də elə odur ki, class’a ilk müraciət zamanı və yalnız bir dəfə işə düşdüyü üçün static property’ə dəyər set etsin. Static constructor access modifier’lar ilə işlənə bilməz, parametr qəbul edə bilməz.

**Extension method**

C#’da hal-hazırda var olan hər hansısa class və ya interface’in orjinal mənbə kodlarını dəyişmədən mövcud class və ya interface’ə yeni method’lar əlavə etmək üçün Extension method’lardan istifadə edilir. Extension method’lar vasitəsi ilə biz, tərəfimizdən əlçatan olmayan, müdaxilə edilə bilməyən class və ya interface’lərə də yeni funksionallıqlar əlavə etməyimiz üçün imkan verir. Extension method’lar static method şəklində, static class’lar daxilində təyin olunurlar. Method’un ilk parametri extension method’u tətbiq edəcəyimiz type olmalı və qarşısında this keyword’ü ilə yazılmalıdır. Extension method’lar göstərilən type’ın reference’na tətbiq edilir. MyType myType = new(); myType.MyExtensionMethod(); yazaraq biz MyType tipindən olan object ilə custom olaraq yaratdığımız Extension method’u call edirik. Dediyimiz kimi həmçinin interface’lər üçün də Extension method yazmaq olur. Bu zaman dəyişən tək şey extension method’u call edən zaman, extension method tətbiq olunan interface’dən miras alan hər hansısa class’ın reference’ı istifadə edilir.

**Abstract Class**

Regular class’lardan fərqli olaraq, Abstract class’lardan object instance almaq olmur. Abstract class’lar sealed və ya static ola bilməz. Daxilində regular class’larda olduğu kimi, static və ya non-static member’lar təyin etmək olar. Instance ala bilməməyimizə baxmayaraq constructor da Abstract class daxilində təyin etmək olur. Abstract class’dan miras alan class daxilində və ya Abstract class’ın öz daxilində bu constructor(lar) başqa bir constructor’dan call oluna bilər. Abstract class’ların Regular class’lardan daha bir fərqlənməsi də ondan ibarətdir ki, Abstract method’lar burada təyin olunur. Abstract class’larda miras alma ilə bağlı heç bir fərqlənmə yoxdur.

**Abstract Method**

Abstract method’lar yalnız absract keyword’ü ilə birlikdə, Abstract class’lar və ya Interface’lər daxilində təyin oluna bilirlər. Abstract method’ların body’ləri olmur. Abstract method’ların body’ləri olmadığı üçün derived class daxilində məcburi olaraq override olunmalıdırlar. Abstract class’ların virtual method’lardan fərqləri odur ki, virtual method’lar override olunmasa da olar.

**Anonym Type(Class)**

OOP prinsipi ilə işləyən dillərdə object anlayışı adından da bəlli olduğu kimi geniş istifadə olunur. Bir obyektə aid olan dataları ramda saxlamaq üçün biz hər hansısa bir type object yaratmalı oluruq. Lakin bəzi situasiylarda sadəcə 1 dəfəlik istifadə üçün müəyyən dataları saxlamaq məqsədi ilə bir object’ə ehtiyyac duyacağıq. Bu zaman bizim köməyimizə C# 3.0 ilə tanıdılan Anonym Type çatacaq. Anonym type dedikdə var keyword’ü ilə yaranmış və heç bir type’a aid olmayan, hər biri unique type olan object’lər başa düşülür. Anonym type object’in saxladığı datalar readonly property’lərdir . Anonym type object’lər yalnız method’ların daxilində təyin oluna bilər. Use and throw( istifadə et və at) məntiqi ilə istifadə olunur.

Anonym type object’lər aşağıdaki şəkildə təyin olunur.

var anonymObject = new

{

Prop1 = value,

Prop2 = value

}

Anonym type object yaranan zaman daxilindəki property’lərin type’ı da yadda saxlanır və həmin reference’a yalnız eyni type property’lərlə yaranmış anonym object’lər mənimsədilə bilər. Əks təqdirdə compile time errorla qarşılaşacağıq.

var person = new {Name = “Name”, Surname = “Surname”}

person = new { Name = “Name2”, Surname = “Surname2” , Age = 0}

**Record**

Bildiyimiz kimi class’lardan alınan instance’ların bərabərliyini məntiq operatoru ilə yoxlamağa çalışdığımız zaman onun content’ni deyil, reference’nın ünvanına görə müqayisə aparır. Əgər content’ə görə yoxlanış aparmaq kimi bir ehtiyyacımız olarsa bu dəfə regular class’dan yox C# 9 ilə gəlmiş Record type’dan istifadə edəcəyik. Record’lar immutable və reference type’dır. Record’lar həm regular class’lar kimi, həm də inline olaraq təyin oluna bilirlər.

public record MyRecord1{

datatype Prop1{ get;} & {get;init;} & {get; set;}(əgər immutable’lıq lazımdırsa)

…

datatype PropN

public MyRecord1(datatype parameter1, …, parameterN)

{

Prop1 = parameter1;

…

PropN = parameterN

}

}

public record MyRecord2(datatype Prop1, ..., datatype PropN);

Record vasitəsi ilə alınan instance’ları məntiqi operator vasitəsi ilə müqayisə edilən zaman reference’nın ünvanına görə yox, property’lərinə mənimsədilən dəyərə görə müqayisə edilir. Bu bərabərlik yoxlanışı yalnız eyni type Record’lar arasında olur. Hər iki cür təyin olunan Record’lardan instance alınan zaman mütləq property’lərin dəyəri göndərilməlidir. Əks təqdirdə instance almaq mümkün olmayacaq. Record’lar yanız bir-birindən və object class’dan miras ala bilər, yalnız bir-birləri arasında miras verə bilər. Record’lar static ola bilməz.

**Record Deconstruction**

Record object’lərin saxladığı dəyərləri, yəni property’lərini get method’dan əlavə, Deconstruction edərək də əldə edə bilərik. Deconstruction əməliyyatı 2 və ya daha çox property’ə malik olan Record’lar üçün əlçatandır. Var keyword’ü qarşısında mötərizə açaraq hər bir property üçün variable təyin edib vergüllər vasitəsi ilə ayırırıq. Hər hansısa property dəyəri bizə lazım olmazsa, discard ( \_ ) yazaraq həmin property’nin üzərindən keçə bilərik.

var(variable1,variable2, \_, …, variableN) = recordObject;

**Boxing**

Boxing, hər hansısa type dəyərin object type’a assign olunmasına deyilir. Object type’a çevrilən dəyər object daxilində öz type’ı ilə saxlanılır. Lakin variable çağırılan zaman object type olaraq gəlir. Bu da o deməkdir ki variable’a öz type’ı ilə bağlı heç bir əməliyyat tətbiq edilə bilməz. Object type dəyişənin daxilindəki dəyərə öz type’ı ilə əldə etmək üçün Unboxing etmək lazımdır.

**Unboxing**

Unboxing, boxing olunmuş hər hansısa datanı öz type’na uyğun şəkildə əldə edilməsinə deyilir. Unboxing edə bilmək üçün explicit casting etmək lazımdır. Bunun vasitəsi ilə biz type uyğunluğunu təmin edirik. Yəni casting zamanı saxlanılan datanın orjinal dəyəri nədirsə ona casting edilməlidir. Bunun üçün is və as operatorları ilə müəyyən yoxlanılmalar aparılaraq qayıdan bool dəyərə uyğun olaraq yol izlənilə bilər.

**Casting**

Casting, type’lar arasında dəyişikliklər etmək üçün istifadə edilir. Cast operatoru (T) olaraq işarə olunur. Variable’ın əvvəlində yazılır və mötərizə daxilində hansı type’a cast edəcəyiksə həmin type yazılır. Casting, Implicit casting(Widening) və Explicit casting(Narrowing) olaraq 2 əsasə yerə bölünür.

Implicit casting ilə biz balaca datanı böyük dataya çeviririk və bu əməliyyat avtomatik olaraq compiler vasitəsi ilə həyata keçir çünki burada heç bir data itkisi olmur. Cast operatorundan istifadə etmək məcburiyyətində deyilik.

Explicit casting isə böyük datadan kiçik dataya keçmək üçün və ya əlaqəli olmayan type’lar arasında keçid etmək üçün istifadə edilir. Explicit casting zamanı casting operatorunun istifadə olunması məcburidir. Əgər dəyər target type’a cast oluna bilmirsə hər hansısa data itkisi və ya potensial runtime exception baş verə bilər. Bunun qarşını almaq üçün Is və As keyword’lərlə yoxlanış aparılır. Unboxing zamanı explicit casting’dən istifadə edilir.

**As**

As keyword vasitəsi ilə biz type yoxlanışı apara bilərik. Əsasən casting prosesslərində istifadə edilir. Əgər şərt ödənirsə özünü, ödənmədiyi təqdirdə isə null qaytarır.   
var obj = expression as Type;

**Is**

Is keyword’ü vasitəsi ilə hər hansısa bir referansın və ya variable’ın null olub olmadığını yoxlaya bilərik. “== null” və ya “!= null” ifadəsi ilə eyni güclüdür.  
Həmçinin Is keyword’ü vasitəsi ilə casting üçün olan type yoxlanışını da apara bilərik. Is keyword’ün işlənmə hallarında geriyə true və ya false olaraq bool dəyər qaytarır. Is keyword’ü C# 7 ilə birlikdə gəldi.

**Is Not**

Is not keyword’ü is keyword’ün negative halıdır. C# 9 ilə birlikdə gəldi. is not null və ya is not anyType şəklində yoxlanış aparıla bilər. Geriyə true və ya false olaraq bool dəyər qaytarır.

**Abstraction**

Abstraction hər hansısa bir funksionallığın arxa tərəfdə necə işlədiyi bizə maraqlı deyilsə və bu funksionallığı necə istifadə etməli olduğumuzu biliriksə bu hadisə Abstraction prinsipi adlanır. Abstraction prinsipi Encapsulation prinsipinin alt prinsipi olaraq qələmə verilir. Abstraction dedikdə ağıla ilk olaraq Abstract class düşsə də sadəcə bundan ibarət deyil.

**Interface**

Interface’lər abstraction prinsipini tətbiq etmək üçün vacib modul’dur. Biz Interface’lər daxilində, onu implement edəcək class(lar)’ın məcburi şəkildə override etməsi üçün implicitly abstract method’lar təyin edirik. Interface’lər yalnız interface’lərdən miras ala bilər lakin “cycle compile error” yaratmamaq şərti ilə. Interface’dən object instance almaq olmaz. SOLID prinsiplərindən I-Interface Segregation Principle uyğun şəkildə Interface’lər təyin olunmalıdır.

**Struct**  
C# daxilindəki bütün value type’lar Struct olaraq yaradılmışdır. Struct’lar implicitly sealed olduğu üçün onlardan miras almaq olmur, və Struct’lar yalnız interface’lərdən miras ala bilər. Struct’ların regular class’lardan bir fərqi də odur ki, Struct daxilində boş constructor yazamadan parametrli constructor yazsaq belə Assembly səviyəsində özü yenə də boş constuctor’u əlavə edir. Struct’lar həmçinin immutable’dır. Struct’ların istifadə olunması səbəbi kiçik və az data’lar saxlamaq üçün və qısa davranışlar həyata keçirə bilmək üçün istifadə olunan bir model olması və performans cəhətdən sürətli olmasıdır.

**Required**

C# 11 ilə birlikdə gəlmiş bu keyword field və property’lərə tətbiq edilir. Və object instance alındığı zaman dəyər assign edilməsini tələb edir. Əks təqdirdə compile time error verir. Regular Classlarda, Structlarda və Recordlarda Required field və ya property təyin oluna bilər. Lakin Interfacelərdə bu mümkün deyil.

**Exception**

Biz code yazarkən bilərək və ya bilməyərək müəyyən səhvlərə yol veririk. Bu səhvlər isə müəyyən bug’lara, error’lara gətirib çıxarır. Bu error’lar isə özlüyündə 2 yerə bölünür compiletime və runtime olan error’lar. Compiletime olan error’lar rahatlıqla aradan qaldırıla bilir çünki zatən biz kod yazmağa davam edərkən compiler tərəfindən oxunan kodlar bizə hər hansısa problemlər varsa compile zamanı yerindəcə bildirir. Lakin Runtime olan errorlar yəni Exception’lar isə mütləq catch olunmalıdır. Əks təqdirdə program “partlayacaq”. Müəyyən şərt daxilində biz throw keyword vasitəsi ilə Exception’dan instance alınmış object’i istifadə olunacağı yerdə “partlamasına” səbəb ola bilərik.

if(true){

throw new Exception(“something is running wrong”);

}

Exception’un instance alınması üçün 3 overload hal var. Parametr qəbul etməyən(default,parameterless), string message qəbul edən və həm string message həm də innerException qəbul edən.

Exception exceptionObj = new();

Exception exceptionObj2 = new(“message”);

Exception exceptionObj3 = new (”message”, new Exception());

**Try & Catch**

Exception baş verə biləcək kod blokunu try daxilində yazıb, exception’u isə əgər baş verərsə catch vasitəsi ilə tutub onun üzərində gedişata uyğun bir iş görəcəyik. Try daxilindəki exception baş verdiyi sətrdə program avtomatik olaraq catch hissəyə düşərək oradaki kod blokunu işə salacaq və baş verən exception’ın instace’nı catch mötərizəsi daxilində tutulacaq .

try{

}

catch(Exception ex){

}

Burada exception object’i olan ex vasitəsi ilə biz ex.Message yazaraq exception mesajına, ex.InnerException yazaraq exception’ın daxilində göndərilən exception’a(nullable’dır) və s. müraciət edə bilərik.

**Multiple Catch**

Biz hər exception type’ı üçün fərqli catch əməliyyatı həyata keçirmək istəyiriksə, ardıcıllıq olaraq ən sonuncu derived class ən üstdə, ən başda duran base class isə ən sonda yazılmaqla bir neçə catch bloku yaradılmalıdır. Bir-birindən asılı olmayan exception’lar üçün hər hansısa bir sıralama məcbur deyil.

try{

}

catch(NullReferenceException ex){

}

catch(SystemException ex){

}

catch(Exception ex){

}

**Finally**

Finally keyword vasitəsi ilə biz exception baş versə də, baş verməsə də mütləq işə salınmalı olan kod blokunu təyin edirik.

try{

}

catch(Exception ex){

}

finally{

}

**Collection**

Collection, bir-biri ilə əlaqəli və tək varlıq olaraq əldə edə biləcəyimiz eyni type’dan olan object’lər toplusudur.

**Array**

Eyni type dataları bir object daxilində saxlamaq istəyiriksə ən sadə collection olan Array’dən isitfadə etməliyik. C# da istifadə edəcəyimiz Array’lər fixed-size object’lərdir. Yəni biz əvvəlcədən Array object yaranan zaman məcburi şəkildə length’ni qeyd etməliyik və həmin object’in size’nı bir daha dəyişə bilməyəcəyik. Əgər buna ehtiyyac duyularsa Array class’ın hazır static method’u olan Resize(ref arr, newLength); işə salınmalıdır. Biz Array daxilində method’lar vasitəsi ilə müəyyən axtarışlar edə bilərik. Hazır method’lardan savayı biz for loop istifadə edərək indexer vasitəsi ilə və ya foreach loop vasitəsi ilə array daxilindəki elementlərə müraciət edə bilərik. Value’a görə axtarış sürəti worst case olaraq O(n)’dir. Indexer vasitəsi ilə index’ə görə axtarış sürəti isə O(1)’dir. Value type’larla da array yaratmaq mümkün olsa belə array’lər ümumilikdə reference type’dır.

**Generic**

**Constraint**

Generic type’lar üçün nəzərdə tutulmuş müəyyən hallar ola bilər ki, bizim istəyimizə uyğun olaraq yalnız müəyyən type(lar) göndərmək mümkün olsun. Generic class təyin olunduğu yerdə where: keyword’ü ilə istifadə edilir. Həm class’dır mı, həm hansısa interface’dən miras alır mı, həm de boş construtor’u var mı deyə bir constraint təyin etmək istəsək ardıcıllıq where: class, IAnyInterface,new() şəklində yazılmalıdır. Əlavə olaraq həm də əgər Generic type class 2 type qəbul edirsə, Generic<T,K> burada biz T’in K’dan miras almasını da məcburi hal aldıra bilərik yəni where: T:K

Hashset sürət cəhətdən zəifdir çünki hər dəfə dəyər add olunan zaman gedib digər dataları oxuyub yoxluyur.

**Directory**

Directory class’ı static class’dır.

**CreateDirectory**

Directory class’ın, public static DirectoryInfo CreateDirectory(string path) method’u vasitəsi ilə biz göstərdiyimiz root’da müvafiq adda Folder yarada bilərik. Göndərilən path null olarsa ArgumentNullException, göndərilən path’də Folder adında File mövcuddursa IOException,

Göründüyü kimi geriyə DirectoryInfo object qaytarır. Biz bu object vasitəsi ilə üzərində işlədiyimiz Folder barəsində bir çox məlumatı əl çatan olur.

DirectoryInfo directoryInfo = Directory.CreateDirectory(path);

**Exist**

Hər hansısa Folder yaradan və ya silən zaman ilk öncə o adda Folder’in olub-olmadığını yoxlamaq lazımdır.

public static bool Exist(string path) method’u vasitəsi ilə bu əməliyyat icra edilir. Göndərilən path’də folder’in olmaması halına uyğun boolean dəyər qaytarır.

**Delete**

Hər hansısa Folder’i silmək üçün public static void Delete(string path) method’dan istifadə edirik. Əgər göndərilən path’də Folder yoxdursa o zaman Exception yaranır. Həmçinin əgər silmək istədiyimiz Folder’in daxili boş olmazsa bu zaman başqa bir Exception yaranacaq. Əgər Folderi yenə də silmək istəyiriksə bu zaman Delete method’un overload halını istifadə edəcəyik. public static void Delete(string path, bool recursive) bu method vasitəsi ilə path ilə yanaşı boolean argument true göndərilərsə Folder’in daxilini də təmizləyərək özünü də silir.

**File**

Fileclass’ın, Create(string path) method’u vasitəsi ilə biz göstərdiyimiz root’da müvafiq adda file yarada bilərik. File.Create(path);

**Ado Net**

Database ilə connection qurub data əlavə etmək və ya data üzərində əməliyyat həyata keçirmək üçün istifadə olunan texnologiyalardan biri də Ado Net texnologiyasıdır. Ado Net texonologiyasından istifadə etmək üçün ilk olaraq Database ilə Connection qurmaq lazımdır. Bunun üçün bizə string type ConnectionString lazım olacaq.

**SqlConnection**  
Mssql üçün bu ConnectionString istifadə edilir.

string connectionString = @”Server=serverName; Database=databaseName;Trusted\_Connection=True;”;

Daha sonra Connectionu qurmaq üçün Disposable olan System.Data.SqlClient.SqlConnection type object yaratmaq lazımdır. Open() method’u ilə SqlConnection object’i vasitəsi ilə connection açılmalı və bu connection ilə iş bitdikdən sonra Close() olunmalıdır. Həmçinin Disposable type olduğu üçün sonda Dispose() method’u işə salınmalıdır. Manual olaraq Dispose() və ya Close() method’ları işə salmamaq üçün Using keyword’ü köməyimizə çatır.

using SqlConnection connection3 = new SqlConnection(connectionString);

connection3.Open();

**SqlCommand**

SqlCommand obyekti vasitəsi ilə də biz yazacağımız query’ləri icra edə bilərik. SqlCommand obyekti string type query və SqlConnection type connection qəbul edəcək.

string query = “”;

SqlCommand command = new Command(query,connection);

**ExecuteNonQuery();**

Create,Update və Delete əməliyyatlarını həyata keçirmək üçün command obyektinə ExecuteNonQuery() method’u tətbiq edilir və geriyə dəyişiklik edilən data sətrlərinin sayı olan int type rows qayıdır.

**ExecuteReader();**

Ado Net vasitəsi ilə Select əməliyyatları ExecuReader(); method’u vasitəsi ilə həyata keçirilir. Geriyə SqlDataReader type object qaytarır. Gələn dataları istifadə edə bilmək üçün reader object istifadə edilir. while loop vasitəsi ilə geriyə data qayıdıbsa əgər indexer istifadə edərək gələn sətrdən dataları object type olaraq əldə edirik.

while(reader.HasRows){

int id = (int) reader[0]; və ya (int) reader[“id”];

}

**ExecuteScalar();**

**SQL Injection**

Ado Net istifadəsinin bir çətin tərəfi isə Security hissəsinin də bizim əlimizdə olmasıdır. Əgər düzgün şəkildə query’lər yazmazsaq hər hansısa SQL Injection baş verə bilər. Bu Injection’ın qarşısını almaq üçün parametr’lərdən istifadə edəcəyik. Yəni biz hər hansısa dynamic bir dəyər daxil edib query halına salmaq istəyiriksə, string query = $"SELECT \* FROM Posts WHERE ID = {id}"; əvəzinə string query = "SELECT \* FROM Posts WHERE ID = @id"; yazmalıyıq.  
Bu halda command.Parameters.AddWithValue(parameterName,value); method’u işə salınaraq SQL Injection’ın qarşısı alınır.

**String**

String type dəyişənlər Reference məntiqi ilə biraz fərqli işləyir. Çünki bir başa olaraq String pool ilə çalışdığı üçün eyni dəyərə sahib variablelar hər dəfə yeni obyekt-reference yaratmaqdansa bir dəyərə uyğun bir neçə referans yaradır.

Yəni biz,

string str1 = “My str”;

string str2 = “My str”;

string str3 = str1;

string str4 = String.Empty;

str4 = “My str”;

Burada digər obyektlərdə olduğu kimi str3 ilə str1 eyni referansa sahibdirlərmiş kimi görünsə də, ReferenceEquals methodu bizə hər bir variableın eyni obyekti referans aldıqları məlum olacaq. Bu isə String poolun işləmə prinsipinə görədir. Javada isə bu tam fərqlidir.

string str = $”You can write {CSharpObject} right here”;

str = string.Format(“You can write {index} {sameOrAnOtherIndex}”, textForIndex, textForAnOtherIndex)

Console.WriteLine($”You can write {CSharpObject} right here”);

Console.WriteLine(“You can write {index} {sameOrAnOtherIndex}”, textForIndex, textForAnOtherIndex);

-collection

-exception

-access modifier

-delegate

-expression

-partial