**DERS 1: Nesne Anatomisi - Referans Türlü Değişkenler**

- Günümüzde nesne tabanlı programlama birçok modern dil tarafından desteklenir haldedir. Yazılım geliştirme süreçlerini oldukça kısaltan ve sistematik hale getiren bir tekniktir.

- Gerçek hayatı programlama için simüle eden nesneleri baz alan bir programlama tekniğidir. Her şey bir nesnedir. İnsan, personel, ürün, satış, araba...

- Gerçek bir sistem, nesnel parçalara ayrılır ve bu parçalar(nesneler) arasında ilişkiler kurulur.

- Nesneler kendi aralarında haberleşebilir.

**Nesnenin Anatomisi**

- OOP'nin en küçük esas parçası nesne/obje/object'tir.

- Nesneler içerisinde veri tutabilecekleri alanlar barındırırlar. Biz bu alanlara field diyeceğiz.

- Nesneler içerisinde field'lardaki değerleri işleyebilmesi için fonksiyonlar mevcuttur. Bu fonksiyonlara ileride metotlar, propertyler veya indexer'lar diyeceğiz.

- Tüm nesneler aslında bir sınıf modellemesinin örneğidir.

**Nesne Modellemesi**

- Nesnelerin oluşturulabilmesi için öncelikle modellenmesi gerekmektedir. Nesne modeli class ile gerçekleştirilir.

- Car adında bir class oluşturalım. Oluşturulan bu Car modeli sayesinde bizler istediğimiz araba verilerini taşıyan nesneler üretebiliriz. Audi, Nissan, Volvo hepsi birbirinden farklı birer Car nesnesidir. Daha doğrusu class Car adlı nesne modelinden üretilen nesnelerdir.

- Nesneler referans türlü değişkenlerdir.

**Referans Türlü Değişkenler**

- Kod yazarken Stack'teki değişkenlere isimleri üzerinden erişebilmekteyiz. Örneğin a dediğimizde 5 değerini elde edebilmekteyiz.

Lakin Heap'e erişim hakkımız bulunmamaktadır.

- Heap'teki nesnelere biz direkt erişemesekte Stack'teki referanslar erişebilmektedir. Biz de, Stack'te Heap'teki nesneleri işaret eden referanslar tanımlarız. Haliyle Stack'teki referans'a erişebilir ve dolaylı olarak o referans aracılığıyla Heap'teki nesneye erişmiş oluruz.

Stack Heap

(Nesne1) r1 --r1 referansı-> Nesne 1 //r1 ile Nesne 2'yi referans ediyorum.

(Nesne2) r2 --r2 referansı-> Nesne 2

(Nesne3) r3 --r3 referansı-> Nesne 3

Stack'teki değişkenler Heap'teki değerlere işaretleme yapıyor, referans ediyor.

**Class Kavramı**

- OOP'de bir object oluşturabilmek için öncelikle o objenin modellenmesi, tanımlanması gerekmektedir.

- Bir objenin modelini, tanımını oluşturabilmek için class yapısı kullanılır.

Model -> (Field, Property, Indexer, Metotlar) --class--> object

- Class'larda nesnelerdeki ortak alan tanımları yapılır.

Class -----------> Object1 //Class içerisindeki member'lar class'tan üretilen tüm nesnelerde ortak olarak bulunur.

Object2 //buradan da Adi ve YasHesapla'ya ulaşılabilir.

Adi(Field) Object3 //buradan da...

YasHesapla(Metot)

**Sınıf Oluşturma**

class [isim]

{

}

- class'lar bir referans türüdür.

- class'lar namespace içerisinde, namespace dışarısında ve class içerisinde(nested type) oluşturabilir.

- Aynı amaca hizmet eden özellikleri, sınıfları ve fonksiyonları bir çatı altında toplarken, bu çatıya verdiğimiz isime isim alanı (namespace) denir.

Namespace İçerisinde:

using System;

namespace OOPExamples

{

class Program

{

}

class MyClass

{

}

}

Namespace Dışarısında:

using System;

class MyClass2

{

}

namespace OOPExamples

{

class Program

{

}

class MyClass

{

}

}

- Aynı namespace altındaki class'lar birbirlerine direkt isim üzerinden erişebilirken, farklı namespace altındaki sınıflar birbirlerine namespace ismi üzerinden erişebilir. Namespace ismi belirtilmeyen class'lara ise herkes erişebilir.

**Class İçerisinde Tanımlama:**

class MyClass

{

class MyClass3

{

}

}

- Bir class tanımlamasında tanımlanan yerde(namespace/dışı, class) aynı isimde birden fazla class tanımlanamaz!

**Sınıf ile Nesne Modeli Oluşturma**

class OrnekModel

{

int a; //değişkenler

int b;

public void X() //metorlar

{

Console.WriteLine(a + " " + b);

}

public int Y()

{

return a\*b;

}

}

**Sınıf Modelinden Referans Noktası Oluşturma**

- Bir class tanımlandığında o class adı bir türdür. Haliyle o türü kullanabilmek için direkt olarak class adını kullanmamız yeterlidir.

using System;

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

OrnekModel w; //OrnekModel class'ı türünde bir referans noktası, bir değişken tanımlarız.

//referans noktası alma, referans noktası ayarlama, referans alma.

//şu anda herhangi bir nesneye referans etmiyor. Sadece tanımlama yaptık. İleride detaylı.

}

}

class OrnekModel

{

int a;

int b;

public void X()

{

Console.WriteLine(a + " " + b);

}

public int Y()

{

return a \* b; }}}

- Eğer ki bir referans noktasında herhangi bir nesne referans edilmiyorsa o referans null değere sahiptir.

**Class Members**

**Field Yapılanması**

- Nesne içerisinde veri deopladığımız/tuttuğumuz alanlardır.

- Class içerisindeki değişkenlerdir. Herhangi bir türden olabilir.

using System;

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m1 = new MyClass();

//new keyword'ü ile MyClass'tan bir nesne oluşturabiliriz. Oluşturulan bu MyClass nesnesini

//MyClass türünden m1 referansıyla işaretliyoruz. ='in sağı HEAP'e nesneyi koyacak, ='in solu ise Stack'teki m1 referansıyla

//Heap'teki nesneyi işaretleyecek.

//m1. dediğimiz zaman içerisindeki tüm elemanlara erişebileceğiz.

m1.a = 5;

m1.b = "Comolokko";

MyClass m2 = new MyClass();

m2.a = 25; //m1'in a'sı kendine ait bir alanken, m2'nin a'sı da kendine ait bir alandır.

m2.b = "";

}

}

class MyClass

{

//m1.a hata verir. a public değil.

//int a;

//MyClass'tan üretilen tüm nesnelerde a isimli bir alan olacaktır ve ona integer türünden bir değer atıyabiliyor olacağız.

//public int a; dersek Main'den de erişebiliriz.

public int a;

public string b;

}

}

- Field'lar türüne özgü varsayılan değer alırlar.

class MyClass

{

int a; //Stack'te a'ya 0 değeri atanır. integer'ın varsayılanı 0'dır.

}

- Eğer bir değişken class içerisinde field olarak tanımlanıyorsa default değeri verilir. Yok eğer class'ta değil metot vs. içerisinde (main de dahil)

tanımlanıyorsa default değeri verilmez! Hata verir ve değer atanmasını ister.

**Property Yapılanması**

- Nesne içerisinde özellik/property sağlar. Property esasında bir metottur. Yani programatik/algoritmik kodlarımızı inşa ettiğimiz bir metot. Lakin fiziksel olarak metottan farkı parametre almamakta ve içerisinde get ve set olmak üzere iki adet blok almalarıdır.

public int X() //Bir metot.

{

return 0;

}

Ve bu metodun Property muadili diyebileceğimiz Property:

public int X //Property'de parantez yoktur.

{

get

{

return 0;

//Property'nin değeri okunmak istendiğinde/çağrıldığında get bloğu tetiklenir ve get bloğundan değeri return edilir.

}

set

{

//Property'e atanan veri buradan yakalanır.

}

}

- Property'nin işlevsel açıdan metottan farkı yoktur. Lakin davranışsal olarak nesne üzerinde bir değer okuma ve değer atama işlemlerinde kullanılır.

- Bu bloklar compile neticesinde get ve set isimli metotlar olarak karşımıza çıkmaktadır.

- Yazılımcılar olarak nesnelerimiz içerisindeki field'lara direkt erişilmesini istemeyiz. Dolayısıyla field'lardaki verileri kontrollü bir şekilde dışarıya açmak isteriz. İşte böyle bir durumda diğer dillerdeki gibi metotları kullanabiliriz. Fakat C#'ta hem metotları hem Property'leri kullanabiliriz.

- Developer'lara "Ancak benim kontrolümde buradaki değeri elde edebilirsin." demek lazım. İlgili field'daki değerin hepsi veya bir kısmını talep edene gönderilir. Böylece kontrol sağlanmış olur.

- Sonuç olarak Property yapıları özünde nesne içerisindeki bir field'ın dışarıya kontrollü açılmasını ve kontrollü bir şekilde dışarıdan değer almasını sağlayan yapılardır.

Not: Biz property'lerin bu işlevine Encapsulation(Kapsülleme/sarmalama) diyeceğiz.

**Encapsulation(Kapsülleme/Sarmalama)**

- Bir nesne içerisindeki dataların(field'lardaki verilerin) dışarıya kontrollü bir şekilde açılması ve kontrollü bir şekilde veri almasıdır.

Property İmzaları

- Property yapısı oluşturabilmenin yapılsal olarak birkaç farklı yolu/farkı imzası vardır.

(Full Property, Prop, Auto Property Initializers, Ref Readonly Returns, Computed(Hesaplanmış) Properties, Expression-Bodied Property, Read Only Property

ve C# 9.0'da gelen Init-Only Properties and Init Accessor)

Full Property

- En sade property yapılanmasıdır. İçerisinde get ve set blokları tanımlanmalıdır

[erişim belirleyicisi] [geri dönüş-alış değeri] [property adı]

{

get { } //property'den veri istendiğinde tetiklenir.

set { } //Property'de veri gönderildiğinde tetiklenir. Gönderilen veriyi value keyword'ü ile yakalar.}

- Full propertylerde set bloğu tanımlanmazsa sadece okunabilir(readonly); get bloğu tanımlanmazsa sadece yazılabilir(write only) olacaktır.

using System;

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass myClass = new MyClass();

//myClass. dediğimiz an yasi property'si ingiliz anahtarı biçiminde gelir. Mor-pembe kutular ise metot'tur.

Console.WriteLine(myClass.Yasi); //get bloğu tetiklenir.

myClass.Yasi = 65; //değer atarken de set bloğu tetiklenir.

Console.WriteLine(myClass.Yasi); //yine get bloğu tetiklenir ve yasi = 65 olur.

}

}

class MyClass

{

int yasi;

string b;

//Property hangi türden bir field'ı temsil ediyorsa o türden olmalıdır.

//Genellikle temsil edecekleri field'ın isminin baş harfi büyük olan yazımı ile isimlendirilirler.

public int Yasi

{

get

{

return yasi;

//property üzerinden değer talep edildiğinde bu blok tetiklenir. Yani değer buradan gönderilir.

}

set

{

yasi = value;

}

}}}

Örnek:

class Banka

{

int bakiye;

public int Bakiye

{

get

{

if(bakiye > 0)

return bakiye \* 10 /100;

return 5;

}

set

{

if (value < 10)

bakiye = value; //10 liranın altındaysa kesinti yapma.

else

bakiye = value \* 95 / 100; //mesela burada işlem ücreti almış olabiliriz.

}}}

- Karşı taraf aldığı parayı bilmeli. O sürecin nasıl gerçekleştiğini bilmesine gerek yok.

- Bir property'nin sadece get bloğunu yazıp set'i yazmaksak sadece get'i kullanabiliriz. Aynısı set içinde geçerli. Böylece Property'i readonly ve writeonly yapabiliriz.

**PROP**

- Bir Property her ne kadar encapsulation yapsa da temsil ettiği field'daki dataya hiç müdahele etmeden erişilmesini ve veri atanmasını sağlıyorsa böyle bir durumda kullanılan property imzasıdır.

- Ne gerek var? Ahlaki sebepler. Field'daki değere müdahele olsun olmasın direkt erişim yapılmasını istemiyoruz. Haliyle yine de property kullanacağız.

[erişim belirleyicisi] [geri dönüş-alış değeri] [property adı]{ get; set; }

- Prop property'ler compile edildiklerinde arkaplanda kendi field'larını oluştururlar. Dolayısıyla bir field tanımlamaya gerek yoktur.

class MyClass

{

int yasi; //field

public int Yasi //Full Property

{

get

{

return yasi;

}

set

{

yasi = value;

}

}

}

tek satırda Prop ile aynı işi yapabiliriz.

class MyClass

{

public int Yasi { get; set; }

}

Sonuç olarak;

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass myClass = new MyClass();

//kendi field'ımı tanımlamıza gerek yok. Eğer ki tanımlamak istiyorsak Prop kullanmacağız.

myClass.Yasi = 12; //değer tanımlayıp,

Console.WriteLine(myClass.Yasi); //değer okuyabiliriz.

}

}

class MyClass

{

public int Yasi { get; set; } //compiler seviyesinde bir değer atanacaktır.

}}

**Auto Property Initializers (C# 6.0)**

- Bir property'nin ilk değerini nesne ayağa kaldırılır kaldırılmaz aşağıdaki gibi verebiliriz.

class InsanEntity

{

public string Adi { get; set; } = "Genco";

public string Soyadi { get; set; } = "Star";

public int Yasi { get; set; } = 23;

}

Not:

- Prop property'ler readonly olabilir ama writeonly olamaz.

public int Yasi { get; } = 23; olur.

public int Yasi { set; } = 15; writeonly olamaz.

**Ref Readonly Returns**

- ref only returns, bir sınıf(class) içerisindeki field'ı referansıyla döndürmemizi sağlayan ve bir yandan bu değişkenin değerini readonly yapan özelliktir.

class MyClass

{

//artık MyClass üzerinden Adi property'sine eriştiğimde

string adi = "Alican Cano"; //buradaki değere değil,

public ref readonly string Adi => ref adi; //bu değeri temsil eden adi field'ına erişmiş olacağız.

//bunun getirisi hem bellek optimizasyonu hem de bunu static yapılanmaya çevirdiğimizde daha da efektif bir kodlama yapmamızı sağlayacaktır.

//ileride daha detaylı.

}

**Computed(Hesaplanmış) Property**

- İçerisinde türetilmiş bir bağıntı taşıyan property'dir.

int s1 = 5;

int s2 = 10;

public int Topla

{

get

{

return s1 + s2;

}}

**Expression - Bodied Property**

- Tanımlanan property'de Lambda Expression kullanmamızı sağlayan söz dizimidir.

public string Cinsiyet

{

get

{

return "Erkek";

}

}

aynı ifadeyi;

public string Cinsiyet => "Erkek";

yazabiliririz. Bu şekilde expression-bodied ile imzalanan propertyler read only olarak oluşturulacaktır.

**Init-Only Properties - Init Accessor (C# 9.0)**

- Init-Only Properties, nesnenin sadece ilk yaratılış anında propertylerine değer atamaktadır.

- Böylece iş kuralı gereği run time'da değeri değişmemesi gereken nesneler için bir önlem alınmaktadır. Init-Only Properties, developer açısından süreç esnasında değiştirilmemesi gereken property değerlerinin yanlışlıklı değiştirilmesinin önüne geçmekte ve olası hata ve buglardan yazılımı arındırmaktadır.

- Böyle bir durumda akla direkt Auto Property Initializers gelebilir. Sonraki derslerde görülecek Object Initializer desteği Init-Only'de gelmektedir. Auto property-initializers object initializers'a izin vermemektedir. Lakin bu özelliği Init-Only Properties desteklemekte ve sonrasında readonly özelliği göstermektedir.

**Indexer**

- Nesneye indexer özelliği kazandıran, fıtrat olarak property ile birebir aynı olan elemandır. Indexer özel tanımlı bir property'dir ve sadece class içerisinde tanımlanabilir. Tanımlandığı class'a indexlenebilir özelliği kazandırır. Array işlemlerinde kullandığımız **[ ]** operatörünü tanımlamış olduğumuz bir class'ı diziymiş gibi işlemler yapabilmek içinde kullanabiliriz.

[erişim belirleyicisi] [geri gönüş değeri] this [ parametreler ]

{

get { } -> indexer'dan veri istenildiğinde tetiklenir.

set { } -> indexer'a veri gönderildiğinde tetiklenir. Gönderilen veriyi value keyword'ü ile yakalar.

}

public int this [int a]

{

get

{

return a;

}

set

{

}

}

Çağırırken de;

..main..

{

MyClass myClass = new MyClass();

myClass[5] = 10;

}

Örnek:

public static void Main()

{

Department dprt = new Department();

dprt[0]="Bilgi İşlem";

dprt[1]="Proje Yönetimi";

dprt[2]="Analiz";

dprt[3]="İş Geliştirme";

dprt[4]="Destek Sistemler";

Console.WriteLine(dprt[4]); //Destek Sistemler

}

public class Department

{

public string Name { get; set; }

public int ID { get; set; }

//indexer tanımlaması

private string []names = new string[5];

public string this [int index]

{

get

{

return names[index];

}

set

{

names[index] = value;

}

}

}

- Department class'ımıza indexer kullanarak indexlenebilir özelliği kazandırdık ve [ ] kullanarak tıpkı array kullanıyormuş gibi değer atama ve değer okuma işlemlerini yapabildik.

**Class İçerisinde Tanımlanan Class Sınıf Elemanı mıdır?**

class MyClass

{

int a; //field bir class elemanıdır.

public int MyProperty { get; set; } //property bir class elemanıdır.

public void X() {} //metot bir class elemanıdır.

public int this [int a] //indexer class elemanıdır.

{

get {return 0; }

}

class MyClass2

{

//nested class'lar class elemanı değildir.

//tek farkı, MyClass2'ye erişmek için MyClass.MyClass2 m2 = new MyClass.MyClass2(); yazmamız gerek. Bu kodun türü MyClass2'dir.

}

}

**Class Elemanlarına Açıklama Satırı Nasıl Eklenir?**

Random random = new Random();

//Random'un üzerine geldiğimizde sınıfa dair bir açıklama görürüz.

//Represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet certain statistical requirements for randomness.

//metotlarında overload'larında açıklamalar mevcuttur.

Bu açıklamayı kendi sınıflarımıza eklemek için;

MyClass myClass = new MyClass();

//myClass. yazdığımızda açıklamaların eklendiğini kontrol edebiliriz.

/// <summary>

/// Bu bir örnek class'tır.

/// </summary>

class MyClass

{

int a;

/// <summary>

/// Bu bir örnek property'dir.

/// </summary>

public int MyProperty { get; set; }

/// <summary>

/// Bu bir örnek metot'tur.

/// </summary>

public void X() {}

/// <summary>

/// Bu bir örnek metot overload'dur.

/// </summary>

/// <param name ="a">a da parametresidir.</param>

public void X(int a) {}

public int this [int a]

{

get {return 0; }

}

class MyClass2

{

}

}

**this Keyword'ü Nedir? İşlevleri Nelerdir?**

- 1) Sınıfın nesnesini temsil eder.

- 2) Aynı isimde Field ile Metot parametrelerini ayırmak için kullanılır.

- 3) Bir contructer'dan bir başka contructer'u çağırmak için kullanılır.

**1)** Sınıfın nesnesini temsil eder.

MyClass m1 = new MyClass();

MyClass m2 = new MyClass();

//m1.X m1 üzerinden X'i çağırdığımızda kullanlan this m1 nesnesini temsil eder.

//m2.X m2 üzerinden X'i çağırdığımızda kullanlan this m2 nesnesini temsil eder.

class MyClass

{

//this keyword'ü sadece member'ların içerisinde çağırılmalıdır.

public void X()

{

//ilgili sınıftan oluşturulan nesneyi, sınıf modellemesinde temsil etmeyi sağlayan keyword'dür.

//this

//bir class'ın içerisinde this keyword'ünü görüyorsak bilmeliyiz ki o class'tan o anda üretilmiş olan nesne neyse o nesneyi temsil eden bir keyword'dür.

//kullanım amacı değişebilir.

}}

İleride daha detaylı.

**2)** Aynı isimde Field ile Metot parametrelerini ayırmak için kullanılır.

class MyClass

{

int a;

public void X(int a)

{

this.a //field'daki a'yı temsil eder. Nesnenin member'ı olan a gelir.

a //parametre olan a'yı temsil eder.

}

}

- this keyword'ü ilgili class yapılanmasının o anki nesnesine karşılık gelir.

- this kullanmak zorunda değiliz.

class MyClass

{

int a;

public void X()

{

a //bu şekilde kullanımda compiler seviyesinde zaten otomatik this kullanılmaktadır.

//sadece parametre ve field ismi aynı ise ayırt etmek için this kullanmak zorundayız.

}

}

**Not:** Typescript denilen Javascript'te this aynı anlama gelir. Fakat o dilde kullanmak zorunludur. C#'da this zorunlu değildir. Sınıf modelinin içerisinde nesneyi temsil edecek this'i kullanmasak da olur.

**3)** Bir contructer'dan bir başka contructer'u çağırmak için kullanılır.

İleride…

**DERS 2 : Nesne Kavramı Nedir? Nesne Nasıl Oluşturulur? | new Operatörü**

- Nesneler kompleks değerlerdir. Nesneleri modellememizi sağlayan class'lar Complex Type'lardır.

**new Operatörü ile Nesne Üretimi**

- C#'ta nesne üretimi için new operatörü kullanılmaktadır.

new Type() //Type -> nesnesini talep ettiğimiz class

//() ise constructor method

HEAP -> (Type object) //Heap'e direkt ulaşamayız.

Type x = new Type();

//Stack'e geldik. Type türünde x isminde bir referans noktası tanımladık ve bu değişkenle Heap'teki (Type object) nesnesini //işaretlemiş olduk.

- Böylece x üzerinden ilgili nesneye erişebilmiş olduk.

- Artık = bir assign operatörü değil referans etme - işaretleme operatörü oldu.

Normalde nesneyi böyle oluşturuyorduk fakat,

**Target-Typed New Expressions (C# 9.0)**

- Nesne oluşum sürecinde, oluşturulacak olannesne eğer ki direkt bir referansa atılıyorsa eğer burda hangi nesnenin oluşturulduğu

referans sayesinde bilinebilmektedir. Dolayısıyla ilgili nesnenin oluşturulması için;

Type x = new Type(); semantiğinden ziyade,

Type x = new(); şeklinde oluşturmamızı sağlayan yeni bir özelliktir.

**Referans Nesne İlişkisi**

**Referans Nedir?**

- RAM'in Stack bölgesinde tanımlanan ve Heap bölgesindeki nesneleri işaretleyen/referans eden değişkenlerdir/noktalardır.

STACK

MyClass m

MyInterface m

MyAbstractClass m

Ya da C# 9.0'da gelen record özelliği ile referanslar oluşturabilmekteyiz.

- Oluşturduğumuz referans noktası illa ki bir class olacak demek değildir, class türevleri-akrabaları olabilecek farklı yapılanmalarda referans noktaları oluşturup

çalışmamıza devam edebiliriz. Fakat Heap'teki nesneler sadece class'lardan oluşturulabilir. Ne Interface ne de Abstract Class'lardan nesne oluşturamayız.(ileride)

- Referans'lar illa ki bir nesne referans etmek zorunda değildir!

- Eğer ki bir referans herhangi bir nesne işaretlemiyorsa null değerini alır.

**Stack - Heap İlişkisi**

MyClass m = new MyClass();

//compiler'dan new keyword'ü ile nesne talep edilmiş, talep edilen MyClass türündeymiş, üretilen bu nesneyi MyClass türünden m isimli referansla işaretlemişiz.

STACK HEAP

m -------------> \_\_nesne\_\_ //m, heap'i referans ediyor.

Örnek:

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m = new(); //buradaki MyProperty çalıştırılırken m'nin referans ettiği nesne üzerindeki MyProperty'i çalıştıracaktır.

m.MyProperty = 10; //nesne üzerinde ilgili property 10 değerine sahip.

MyClass m2 = null; //nıll olan yani nesnesi olmayan referanslar üzerinden herhangi bir member'ı çağırıp

m2.MyProperty = 20; //işlemeye çalıştığımızda bu durumda null reference hatası verecektir.

}

}

class MyClass

{

public int MyProperty { get; set; }

public void X()

{ }

}

}

**Referanssız Nesneler**

- Bir nesne oluşturulduğu an herhangi bir referansla işaretlenmezse eğer Heap'e yerleştirilir. Lakin bu nesneye tarafımızca bir daha erişemeyiz.

Haliyle ilgili nesneyle aramızdaki tek diyalog oluşturma anıdır.

new MyClass();

- Bu tip nesneler sistemde/memory'de lüzumsuz yer kaplayacağından dolayı bir süre sonra Garbage Collector dediğimiz çöp toplayıcı tarafından temizlenecektir.

HEAP'te referanssız nesneleri imha etmekten sorumlu bir yapılanmadır. Manyaktır.

new MyClass().MyProperty = 10; //sadece tanımlama anında . ile ilgili metot, field, property'lerine erişebiliriz sonra da daha kullanamayız. Sonra da imha edilecek zaten.

new MyClass()

{

MyProperty = 10,

MyProperty2 = 30,

MyProperty3 = 40,

}; //aşağıdaki gibi referanssız nesneye de ilk değer atayabiliriz.

**Object Initializer ile Nesne Oluşturma Esnasında Propertylere İlk Değer Atama**

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m = new MyClass()

{

a = 5;

MyProperty = 10,

MyProperty2 = 30,

MyProperty3 = 40

//içerindeki propertyler verilen değerlerle oluşturulacaktır.

};

m.MyProperty = 20;

//sonradan değer atayabiliriz.

MyClass m2 = new MyClass();

//bu nesne oluşturulurken içlerindeki property'lerin değerleri default olarak verilecektir.

m2.MyProperty = 10;

//ama sonradan da değer atabiliriz tabi.

}

}

class MyClass

{

public int a;

public int MyProperty { get; set; }

public int MyProperty2 { get; set; }

public int MyProperty3 { get; set; }

}

}

- Object Initializer'da sadece field ve property'lere ilk değer atanabilir. Metotlara bunu yapamayız.

**Nesne Kopyalama Davranışları | Shallow Copy | Deep Copy**

**Shallow Copy**

- Var olan bir nesnenin, değerin, değerin, referansının kopyalanmasıdır.

Shallow copy neticesinde eldeki değer çoğaltılmaz, sadece birden fazla işaretlenmiş olur.

Örnek:

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m1 = new MyClass();

//HEAP'teki ilgili nesne STACK'teki MyClass türündeki m1 tarafından işaretlendi.

MyClass m2 = m1;

// m2 ile m1 referansı işaretlendi. Yani m2, m1'in işaretlemiş olduğu nesneyi işaretler. m1'den de m2'den de aynı nesneyi elde edebiliriz.

MyClass m3 = m2;

// m3, m2'yi, m2'de m1'in gördüğü nesneyi referanslıyor. Yani m3'te aynı nesneyi işaretliyor.

MyClass m4 = new MyClass(); //bu da kendi oluşturduğu nesneyi referansıyla işaretlemiş.

}

}

class MyClass

{

}

}

Örnek:

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m1 = null; //başta bir nesneyi işaret etmez.

MyClass m2 = new MyClass(); //m2 kendi nesnesini işaret ediyor.

MyClass m3 = m2; //m3'te m2'nin nesnesini işaret ediyor.

m1 = m3; //önceden tanımladığımız m1 referansımız m3'ün gördüğü nesneyi yani m2 ile aynı nesneyi işaretliyecek.

//referansların işaretlediği nesneleri bu şekilde değiştirebilmekteyiz.

}

}

class MyClass

{

}

}

Örnek:

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m1 = new MyClass(); //m1 referansı nesnesini işaretledi.

MyClass m2 = new MyClass(); //m2 referansı da kendi nesnesini işaretledi.

MyClass m3 = m1; //m3 referansı oluşturulup m1'in işaretlediği nesneyi işaretleyecek.

m1 = m2; //m1 önceden işaretlemiş olduğu nesneden kopup, artık m2'nin işaretlediği nesneyi işaretliyor.

m2 = m1; //m2, m1'in işaret ettiği nesneyi işaret edecek. Ki zaten üst satırda m1 artık m2'nin işaretlediği nesneyi işaretliyordu.

//Önceki referans kopup yenisi oluşacak ve hala işaret etmeye devam edecek.

m1 = m1; //m1 kendisi neye işaret ediyorsa eski referans kopup yenisi oluşacak ve aynı nesneyi referans etmeye devam edecek.

//bir referans birden fazla nesneyi işaretleyemez.

}

}

class MyClass

{

}

}

**Deep Copy**

- Var olan bir nesne, deep copy ile kopyalanıyorsa, ilgili nesne miktarı çoğalır. Aynı özelliklere ve değerlere sahip olan, bellekte farklı bir nesne daha oluşur.

- Referans türlü değişkenlerin/değerlerin default davranışı shallow copy'dir. Değer türlü değişkenlerde default davranış deep copy'dir.

MyClass m = new MyClass();

MyClss m2 = m; //ile shallow copy yapılır.

Nesne Üzerinde Deep Copy yapabilmek syntax ile değil iradeli bir kod yazımı yapmak lazım.

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m1 = new MyClass();

MyClass m2 = m1; //shlllow

MyClass m3 = m1.Clone(); //deep copy

}

}

class MyClass

{

public MyClass Clone()

{

return (MyClass)this.MemberwiseClone();

//MemberwiseClone bir sınıfın içerisinde o sınıftan üretilmiş olan o anki nesneyi klonlamamızı sağlayan ve sonucu object olarak döndüren bir fonksiyondur.

//object olarak tutulmuş MyClass nesnesini MyClass türünde unboxing ederek geriye döndürmek gerekir.

}

}

}

**DERS 3: Derinlemesine Encapsulation ve Record Yapısı**

- Encapsulation, nesnelerimizdeki field'ların kontrollü bir şekilde dışarıya açılmasıdır.

Bir başka deyişle, nesnelerimizi başkalarının yanlış kullanımlarından korumak için kontrolsüz değişime kapatmaktır.

Property'lerden önce bu işlem metotlar ile yapılıyordu.

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m = new MyClass();

m.ASet(15);

Console.WriteLine(m.AGet());

}

}

class MyClass

{

private int a;

public int AGet()

{

return this.a; //a field'ının değerini okumak istiyorsak AGet() fonk. kullan.

}

//MyClass nesnesinde AGet() fonksiyonu tetiklendiğinde this'deki yani o anki nesnedeki a field'ını değerini geri gönder.

public void ASet(int value)

{ this.a = value; }

//dış dünyadan a field'ına bir değer vermek istiyorsak ASet() kullan.

}

}

- Artık;

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

MyClass m = new MyClass();

m.A = 123;

Console.WriteLine(m.A); //otomatik olarak get set durumunu ayarlar.

}

}

class MyClass

{

int a;

public int A

{

get { return a; }

set { a = value; }

}

}}

- Property'ler ile daha kolayca halledebiliyoruz.

**Record Nedir? OOP'de ki Yeri Neresidir?**

Ön Hazırlık: Init Only Properties

- C#'da herhangi bir nesnenin propertylerine ilk değerlerinin verilmesi ve sonraki süreçte bu değerlerin değiştirilmemesini garanti altına almamızı sağlayan Init Only Properties özelliği gelmiştir.

- Bu özellik sayesinde nesnenin sadece ilk yaratılış anında propertylerine değer atanmakta ve böylece iş kuralları gereği runtime'da değeri değişmemesi gereken nesneler için ideal bir önlem alınmaktadır.

Property'nin değerinin değiştirilmemesi ? Bunun için readonly propertyler yani getter-only (sadece get işlemi yapan) propert'ler yok mu?

class Book

{

public string Name { get; } = "Mahalle Kahvesi"

public string Author { get; }

public Book()

{

Authorization = "Saik Fait Asabıyanık";

}

}

- Doğru, yukarıdaki örnekte olduğu gibi tanımlanan propertyler sadece get olduklarından dolayı ya tanımlandıkları anda ya da sadece constructr'dan değer alabilmektedir.

- Yeni gelen Init Only Properties özelliği ile mevcudiyetteki getter-only properties arasında Object Initializer işlevselliği farkı vardır.

Book book = new Book

{

Author = "Şiirlerim",

Name = "Cemal Süreya"

};

- Değer atamaya çalıştığımızda hata veriyorsa bunlar getter-only properties yani eski readonly yapılanmasıdır.

- C# 9.0 ile gelen Init Only Properties özelliği ile hata vermez.

**Init Only Properties Tanımlama**

- init keyword'ü ile;

class Book

{

//get-init bir araya geldiğinde bu property readonly olmakta ve ilk değerlerini constructor

//yahut auto property initializers üzerinden(alt satırdaki) alabileceği gibi init sayesinde

public string Name { get; init; } = "Mahalle Kahvesi"; //bu property artık Init Only demiş olduk. Hem readonly hem de Object Initializer edilirken değeri atanabilir.

public string Author { get; init; }

public Book()

{

Authorization = "Saik Fait Asabıyanık";

}}

Artık hata vermez;

Book book = new Book

{

Author = "Şiirlerim", //hem ilk değerleri verdik.

Name = "Cemal Süreya" //hem de readonly yaptık.

};

Book.Name = "Veli Konak"; //readonly olduğu için bu satır hata verir.

Console.WriteLine(book.Author); //istediğimiz gibi okuyabiliriz.

Console.WriteLine(book.Name);

- init, get keyword'ü olmaksızın kullanılamaz. Ayrıca yapısı gereği set bloğu da kullanılamaz.

- Ayrıca getter-only-properties ile çalışmaktansa readonly bir field üzerinde işlem yapmamız gerekiyorsa eğer aşağıdaki gibi 'init' bizlere eşlik edebilmektedir.

class Book

{

private readonly string name;

private readonly string author;

public string Name

{

get => name;

init => name = value;

}

public string Author

{

get => author;

init => author = value;

}

}

- Böylece C# yeniliklerinden olan init-only-properties sayesinde, nesnelerin ilk yaratılış esnasında salt okunabilir değerlerini vermek için Object Initializer'ı bloklamadan kullanabilmekteyiz.

Örnek:

class MyClass

{

readonly int a;

public int A

{

get { return a; }

set { a = value } //hata. a readonly. set çalışmaz.

}

}

fakat nesne oluşturulurken nesne oluşum sürecinde değer atamak istersek init olarak bildirdiğimizde init bloğu hem set görevi görür hemde bir kereliğine mahsus

oluşturma anında ilgili field'a değeri atar.

MyClass my = new MyClass

{

A = 5

};

class MyClass

{

readonly int a;

public int A

{

get { return a; }

init { a = value }

}

}

**Records Nedir?**

- C# ile gelen Init Only Properties özelliği nesne üretim esnasının dışında değişmez değerler oluşturulması için constructor ve auto property initializers yapısının yanında object initializer yapısının kullanılabilir olmasını sağlıyordu.

- Bir nesnenin sadece tek bir property'si readonly/sabit/değişmez ise biz bunu Init Only Properties özelliği ile sağlıyoruz.

- Eğer bir nesneyi bütünsel olarak değişmez yapmak istiyorsak o zaman daha fazlasına yani Records türüne ihtiyacımız olacaktır. Gelen nesne tamamen, değişmeyen değerlerden ibaret olacaksa Record yapılanması kullanılır. Bir property'den ziyade nesnenin genel değişmezliğini sağlar.

- Böylece bu obje-nesne, artık değeri değişmeyeceğinden dolayı esasında objeden ziyade bir değer gözüyle bakılan bir yapıya dönüşmektedir.

- Buradan yola çıkarak record'ları, içerisinde data barındıran lightweight(hafif) class'lar olarak değerlendirebiliriz.

- Record'lar bir class'tır. Fakat, class'lara istinaden objeden ziyade içerisinde bulunan dataları sabitleyerek nesneden ziyade verilerini/datalarını ön plana çıkarmaktadır.

|Class| ---> new ----> instance(nesne) ----> x //x ve y referanslar

| | ---> new ----> instance(nesne) ----> y

|Record|---> new ----> instance(nesne) ----> a = 5, b=10 ---> x //x ve y referanslar

|prop a|---> new ----> instance(nesne) ----> a = 5, b=10 ---> y

|prop b|

- Class'larda verilsel olarak nesne ön plandadır ve bir farklı referansa sahip olan nesne farklı değer olarak algılanmaktadır.

- Dolayısıyla Equals(x,y) karşılaştırması yanlış olarak döner.

- Record'lar ise verisel olarak değeri ön planda tutar. Sadece, nesnel olarak bu veriler bir objede tutulmakta lakin değiştirilememektedir.

- Haliyle farklı objelerde de olsa veriler(property değerleri) aynı olduğu sürece Equals(x,y) önermesi doğru olacaktır.

- Her iki türde de veriler objede tutulmakta lakin record'lar class'lara nazaran, bir objeden ziyade topyekün veri imajında olacak şekilde özel bir davranış sergiler.

**Record Tanımlama**

Prototip:

record [Name]

{

}

- Aşağıdaki gibi tanımlanan record'lara Norminal Records denmektedir.

public record Book

{

public string Name { get; init; }

public string Author { get; init; }

}

Örnek:

record MyRecord

{

public int MyProperty { get; init; } //aynı class'lar gibi property

public void X() { } //fonksiyon

public MyRecord() { } //constructor

~MyRecord() { } //deconstructor

static MyRecord() { } //statik constructor kullanabiliriz.

}

Örnek:

using System;

MyClass m1 = new MyClass()

{

MyProperty = 5

};

MyClass m2 = new MyClass()

{

MyProperty = 5

};

MyRecord m3 = new MyRecord()

{

MyProperty = 5

};

MyRecord m4 = new MyRecord()

{

MyProperty = 5

};

Console.WriteLine(m1.Equals(m2)); //nesne ön planda False döner.

Console.WriteLine(m3.Equals(m4)); //değer ön planda True döner.

record MyRecord

{

public int MyProperty { get; set; } //init olarak tanımlasak daha record'un amacına hizmet eder.

}

class MyClass

{

public int MyProperty { get; set; }

}

Örnek:

- Record'lar değiştirilemez objeler oluşturmamızı sağlamaktadır. Bu değiştirilemez objeler clss'lar ile gerçekleştirilemez mi?

- Görüldüğü üzere içinde init olan(yani readonly) property'ler barındıran bir class tasarlanabilir.

public class Employee

{

public string Name { get; init; }

public string Surname { get; init; }

public int Position { get; init; }

}

bu class'tan nesne ürettiğimizde object initializer'da ilgili property'lere değerlerini set edebliriz.

Employee employee1 = new Employee

{

Name = "Ali Joe",

Surname = "Jonathanoğlu",

Position = 1

}

- Bu nesnenin süreçte herhangi bir property değerini değiştirmek istediğimizde bunu gerçekleştirebilmek için yeni bir Employee nesnesi üretmemiz ve değişikliğin yapılacağı property dışında diğer property'leri bu nesneden eşleştirmemiz gerekecektir.

Mesela Position'u 2 yapmak istersek;

Employee employee2 = new Employee //isimli yeni bir nesne oluşturup değişmeyecek olan propery'lerin değerlerini önceki nesneden alıp

{

Name = employee1.Name,

Surname = employee1.Surname,

Position = 2 //değişmek istediğimizi manuel olarak değişiyoruz.

}

- Bu yöntem adil sayıdan fazla property'e sahip olunduğu zaman can sıkıcı olabilir!

- Halbuki bu senaryoda record kullansaydık, bu tarz kopyalama durumlarda "with Expression"dan faydalanabilmekte ve istenilen özellik hızlıca değiştirilebilmektedir.

**With Expression**

- İmmutable türlerde çalışırken nesne üzerinde değişiklik yapabilmek için ilgili nesneyi ya çoğaltmamız/klonlamamız(deep copy) ve üzerinde değişiklik yapmamız gerekmekte ya da manuel yeni bir nesne üretip mevcur nesnedeki değerleri, değişikliği yansıtılacak şekilde aktarmamız gerekmektedir.

- Örnek olarak aşağıda bu tarz durumlara istinaden With function çözümü verilebilir.

Employee employee1 = new Employee

{

Name = "Ali Joe",

Surname = "Jonathanoğlu",

Position = 1

};

Employee employee2 = employee.With(2);

public class Employee

{

public string Name { get; init; }

public string Surname { get; init; }

public int? Position { get; init; }

public Employee With(int position)

{

return new Employee

{

Name = this.Name,

Surname = this.Surname,

Position = position

};

}

}

- Y da class değilde record kullanarak elimizdeki record'u direkt kopyalamızı sağlayan with expression'u ile sorunu çözebiliriz.

Employee employee1 = new Employee

{

Name = "Ali Joe",

Surname = "Jonathanoğlu",

Position = 1

};

Employee employee2 = employee1 with { Position = 2 };

Employee employee3 = employee1 with { Name = "Hilmi", Position = 3 };

Employee employee4 = employee2 with { Name = "Rıfkı", Surname = "Potter", Position = 4 };

Console.WriteLine(employee1.Position); //1

Console.WriteLine(employee2.Position); //2

Console.WriteLine(employee3.Surname); //Jonathanoglu

Console.WriteLine(employee4.Surname); //Potter

public record Employee

{

public string Name { get; init; }

public string Surname { get; init; }

public int? Position { get; init; }

}

**DERS 4: Özel Sınıf Elemanları - Constructor Metot Nedir?**

- Constructor bir nesne üretimi sürecinde ilk tetiklenen metottur.

new MyClass (); -> parantez, Constructor metot demektir.

- Constructor, nesne oluşturma sürecinde tetiklenmek zorundadır.

Davranış Modeli

new MyClass ();

MyClass Object

constructor() //sembolik

{

int a = 5;

int b = 15;

int c = 25;

}

int a;

int b;

int c;

- Bu nesne oluşturulurken constructor tetikleneceğinden dolayı ilgili nesne içerisindeki field'ların ilk değerleri verilmiş olacaktır.

- Görüldüğü üzere, constructor işlevsel açıdan nesnelerin yaratılma aşamalarında içerisindeki field'lara ilk değerlerini atamakla yükümlü olabilmektedir.

- Bu nesne oluşturulurken constructor tetikleneceğinden dolayı ilgili nesne içerisindeki field'ların ilk değerleri verilmiş olacaktır.

Önce bu nesneye hafızada bir yer ayrılır bu ayrım yapıldıktan sonra constructor tetiklenir.

**Constructor Metot Oluşturma**

- Constructor, özel bir sınıf elemanıdır.

- Özel olsa da fıtrat olarak bir metottur. Ufak farklılıklarla..

Constructor'ların;

- Metot adı sınıf adıyla aynı olmalıdır!(özel sınıf elemanlarının dışında hiçbir member sınıf adıyla aynı olamaz.)

- Geri dönüş değeri olmaz/belirtilmez.

- Erişim belirleyicisi public olmalıdır.(private olduğu durum ayrıca incelenecektir.)

class MyClass

{

public MyClass() //constructor metot.

{

}

public void X()

{

}

}

Örnek:

new MyClass(); //ilk olarak constructor metot tetiklenecektir.

new MyClass();

new MyClass();

new MyClass();

new MyClass();

class MyClass

{

public MyClass() //constructor metot.

{

Console.WriteLine("Bir adet myclass nesnesi oluşturulmuştur.");

}

public void X()

{

}

}

Çıktı:

Bir adet myclass nesnesi oluşturulmuştur.

Bir adet myclass nesnesi oluşturulmuştur.

Bir adet myclass nesnesi oluşturulmuştur.

Bir adet myclass nesnesi oluşturulmuştur.

Bir adet myclass nesnesi oluşturulmuştur.

Not: C# 9.0 da gelen target type özelliğini hatırlarsak.

MyClass m = new(); //ile nesneyi üretebiliyorduk.

**Default Constructor**

class MyClass //bu sınıf içerisinde herhangi bir constructor tanımlanmamış olabilir.

{

}

new MyClass(); //lakin nesne üretirken de constructor tetiklenmektedir. Her sınıfın içerisinde tanımlamasak dahi default bir constructor mevcuttur.

- Eğer ki bir class'a constructor eklersek default constructor'u ezmiş oluruz.

**Parametreli Constructor'lar**

new MyClass(5); //ona göre nesne oluşturmak lazım.

MyClass m = new(15);

class MyClass

{

public MyClass(int a) //artık parametre alıyor.

{

Console.WriteLine("Bir adet myclass nesnesi oluşturulmuştur." + a);

}

public void X()

{

}

}

Çıktı:

Bir adet myclass nesnesi olusturulmustur.5

Bir adet myclass nesnesi olusturulmustur.15

- Bir sınıf tanımlamasında birden fazla constructor tanımlayabiliriz. Yani constructor'lar overload olabilen fonksiyonlardır.

Overload: Bir sınıf içerisinde aynı isimde birden fazla member olamaz. Eğer ki aynı isimde birden fazla metot oluşturacaksak bu metotlar, overloading(çoklu yüklenme) yapılması lazım. İsimler aynı fakat parametrelerin sayıları yahut türleri veya yerleri değişmiş olmalıdır.

new MyClass();

new MyClass(5);

new MyClass("Ali");

new MyClass("alican",25);

MyClass m = new();

MyClass m1 = new(15);

MyClass m2 = new("cemal");

MyClass m3 = new("comolok",15);

class MyClass //alayı overload

{

public MyClass()

{

}

public MyClass(int a)

{

}

public MyClass(string a)

{

}

public MyClass(string a, int b)

{

}

}

Not: Default constructor public'tir.

Private Constructor?

new MyClass(); //nesne üretiminde hata alırız. Sınıfın dışında nesne üretilemez.

class MyClass

{

private MyClass() //ya da sadece MyClass() yazarsak zaten private olur. Erişimi bu sınıfla sınırladı.

{

}

void X()

{

new MyClass(); //class'ın içinde constructor'u çağırıp nesne üretebilirim.

}

}

- Bir sınıfın constructor metot'unu private yaparsak o sınıfın constructor metoduna erişilemeyeceği için nesne üretimini engellemiş oluruz.

- Buna ihtiyacımız olduğu durumlar olacaktır. (İleride Singleton Design Pattern'larda) Nesne oluşumunu dışardan engelleyip içeriden nesne oluşum sürecini

kontrol etmek istediğimiz durumlar örnek verilebilir.

**This Keyword'ü ile Constructor'lar Arası Geçiş**

this keyword: O sınıfın o anki nesnesini temsil eder ve this. dediğimiz an o sınıfın tüm memberlarını getirir.

Constructor'lar ile olan 2. Kullanımında ise;

new MyClass(5);

class MyClass

{

public MyClass()

{

Console.WriteLine("1.Constructor.");

}

public MyClass(int a) : this() //bunu tetikledik ama aynı anda bir üsttekine de ihtiyacımız var. : this() ile bu nesnenin diğer constructor'larını bana getir demiş olduk.

{

Console.WriteLine($"2.Constructor : a = {a}");

}

//Eğerki nesne üretiminde public MyClass(int a) contructor'u tetikleniyorsa bu contructor işlemini görmeden önce this()'deki "şu" contructor'u tetikle,

//onu tamamladıktan sonra ilkini işleme al demiş olduk.

}

Çıktı:

1.Constructor.

2.Constructor : a = 5

Örnek:

new MyClass(5,10);

class MyClass

{

public MyClass()

{

Console.WriteLine("1. Constructor.");

}

public MyClass(int a) : this() //bunu tetikledik ama aynı anda bir üsttekine de ihtiyacımız var. : this() ile bu nesnenin diğer constructor'larını bana getir demiş olduk.

{

Console.WriteLine($"2. Constructor : a = {a}");

}

//Eğerki nesne üretiminde public MyClass(int a) contructor'u tetikleniyorsa bu contructor işlemini görmeden önce this()'deki "şu" contructor'u tetikle,

//onu tamamladıktan sonra ilkini işleme al demiş olduk.

public MyClass(int a, int b) : this(a)

{

Console.WriteLine($"2. Constructor : a = {a} | b = {b}");

//Eğer ki bu constructor tetikleniyorsa, öncelikle tek parametreli olan constructor'a git a'yı tetikle. a'ya tetiklendiği parametredeki değeri ver.

//istersek manuel'de değer verebiliriz.Bu şekilde bir üstteki constructor'a git. onun içindeki işlemleri yap. oradan da en baştakine git.

//bu şekilde silsile halinde contructor'ları tetikleyebiliriz.

}

}

Çıktı:

1. Constructor.

2. Constructor : a = 5

2. Constructor : a = 5 | b = 10

Not: this keyword'ü ile atlamalar yaparken değer vermemiz gerektiğinde ait olduğu syntax'da constructor'dan gelen parametreyi kullanabiliriz. Ya da manueal değer verebiliriz.

public MyClass(int a, int b) : this(a) ya da

public MyClass(int a, int b) : this(10) gibi.

Bunun dışında farklı bir field'ın ya da property'nin değerini burada çağıramayız.

int x; //ya da public int x; farketmez.

.

.

public MyClass(int a, int b) : this(x) //hata verir. Buradan erişilmez.

{

...

}

Not: İleride Custom Exception sınıfları oluştururken bolca kullanılır.

Örnek:

class Program

{

static void Main(string[] args)

{

Product product = new Product("Laptop");

Console.WriteLine(product.Name);

Product product2 = new Product("Laptop", 1500);

Console.WriteLine(product2.Name);

Console.WriteLine(product2.UnitPrice);

Product product3 = new Product("Laptop", 1500, "Asus Laptop 4GB RAM");

Console.WriteLine(product3.Name);

Console.WriteLine(product3.UnitPrice);

Console.WriteLine(product3.Description);

}

}

class Product

{

public string Name { get; set; }

public decimal UnitPrice { get; set; }

public string Description { get; set; }

public Product(string name)

{

this.Name = name;

}

public Product(string name, decimal unitPrice) : this(name)

{

this.UnitPrice = unitPrice;

}

public Product(string name, decimal unitPrice, string description) : this(name, unitPrice)

{

this.Description = description;

}

}

**Record'larda Constructor'lar**

- Aynı kurallar record için de geçerlidir.

new MyRecord(10);

record MyRecord

{

public MyRecord()

{

Console.WriteLine("İlk record.");

}

public MyRecord(int a) : this()

{

Console.WriteLine($"İkinci record + a = {a}");

}

}

Çıktı:

Ilk record.

Ikinci record + a = 10

**DERS 5: Destructor/Finalizer Functions ve Deconstruct Metodu**

- Bir class'tan üretilmiş olan nesne imha edilirken otomatik çağrılan metottur.

- C# programlama dilinde Destructor sadece class yapılanmasında kullanılabilir ve bir class sadece bir Destructor içerebilir.

object

constructor() //sembolik

{

CW(".....");

}

Destructor ile imha edeceğiz.

Bir Nesne Hangi Şartlarda - Kim Tarafından İmha Edilir?

- İlgili nesne herhangi bir referans tarafından işaretlenmemelidir.

- Nesnenin oluşturulduğu ve kullanıldığı scope sona ermiş olmalıdır.

- Yani ilgili nesneye bir daha erişilemez hale gelinmelidir.

- Uygulamada lüzumsuz olan nesneleri toplamakiçin Garbage Collector isimli bir mekanizma devreye girer.

- GC, C#'da bellek optimizasyonunu üstlenen bir yapılanmadır.

- Geliştiricilerin bu mekanizmaya müdahele etmesi pek önerilmez.

**Destructor Tanımlama Kuralları**

class MyClass

{

~MyClass(){ //sadece 1 tane destructor tanımlanabilir. Her özel sınıf elemanlarında olduğu gibi destructor'da sınıf //ismiyle aynı bir fonksiyondur.

//işlemler.. //tanımlayabilmek için ~ kullanılır.

}

}

- İşte bu fonksiyon, bu sınıftan üretilen nesne imha edilirken otomatik olarak son kez "selametle" demek için (yani son işlemleri yapabilmek için) tetiklenecek olan fonksiyondur.

namespace OOPExamples

{

class Program

{

static void Main(string[] args)

{

X();

GC.Collect(); //garbage collector devreye sokulmuş oldu.

Console.ReadLine();

}

static void X()

{

MyClass m = new MyClass();

}

}

class MyClass

{

public MyClass()

{

Console.WriteLine("Nesne üretilmiştir.");

}

~MyClass()

{

Console.WriteLine("Nesne imha ediliyor.");}}}

Çıktı:

Nesne üretilmistir.

Nesne imha ediliyor.

Örnek:

int sayi = 100;

while(sayi >= 1)

{

new MyClass2(sayi--);

}

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

GC.Collect();

Console.ReadLine();

class MyClass2

{

int no;

public MyClass2(int no)

{

this.no = no; //soldaki class içindeki sağdaki parametredeki no.

Console.WriteLine($"{no}. nesne oluşturulmuştur");

}

~MyClass2()

{

Console.WriteLine($"{no}. nesne imha edilmiştir.");

}

}

Çıktı:

100. nesne olusturulmustur

99. nesne olusturulmustur

98. nesne olusturulmustur

...

2. nesne olusturulmustur

1. nesne olusturulmustur

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2. nesne imha edilmistir.

3. nesne imha edilmistir.

4. nesne imha edilmistir.

...

99. nesne imha edilmistir.

100. nesne imha edilmistir.

**Deconstruct Metodu Nedir?**

- Bir sınıf içerisinde "Deconstruct" ismiyle tanınan metot compiler tarafından özel olarak algılanmakta ve sınıfın nesnesi üzerinden geriye hızlıca bir özet maiyetinde

Tuple bir değer döndürmemizi sağlamaktadır.

A ------------> var(x,y) = A;

object

X=3

Y=5

Z=6

class MyClass

{

public string Name { get; set; }

public string Name { get; set; }

//prototip

public void Deconstruct(out string a, out string b)

{

a=x;

b=y;

}

}

Örnek:

Person person = new Person

{

Name = "Mahmut",

Age = 15

};

var (x,y) = person; //person'u al x ve y değişkenlerine ayır.

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public void Deconstruct(out string name, out int age)

//public olması gerekli. geriye değer döndürmemekte. out olarak aldığı parametreleri Tuple olarak döndürecektir.

//Compiler bu metodun bir yapılanma olduğunu isminden anlayacaktır.

{

name = Name;

age = Age ;

}

}

**Static Constructor**

Güncelleme: Bir sınıftan nesne üretilirken "constructor"a nazaran ilk tetiklenen metot "static constructor"'dır.

Fakat, belirli bir duruma istinaden.

class A

constructor()

{

//ilgili sınıftan her nesne üretildiğinde tetiklenir.

}

static constructor()

{

//ilgili sınıftan ilk nesne üretilirken tetiklenen fonksiyondur.

}

Örnek:

new MyClass(); //ilk nesne üretiminde static constructor tetiklenir. Sonra MyClass constructor'u tetiklenir.

new MyClass(); //ikinci nesne üretiminde ve sonrakilerde MyClass constructor'u tetiklenir

class MyClass

{

public MyClass()

{

Console.WriteLine("MyClass constructor'ı tetiklenmiştir.");

}

static MyClass() //static constructor'da geri dönüş değeri ve erişim belirleyicisi belirlenmez.

{

//overloading yapılmaz.

//Bir sınıfın içerisinde sadece bir tane tanımlanabilir.

//Parametre almaz.

//İsmi sınıf ismiyle aynı olmalı.

Console.WriteLine("MyClass static constructor'ı tetiklenmiştir.");

}

}

Çıktı:

MyClass static constructor'i tetiklenmistir.

MyClass constructor'i tetiklenmistir.

MyClass constructor'i tetiklenmistir.

- Static constructor'ın tetiklenebilmesi için illa ilk nesne üretimi yapılmasına gerek yoktur. İlgili sınıf içerisinde herhangi bir static yapılanmanında tetiklenmesi static constructor tetiklenmesini sağlayacaktır.

**DERS 6: Positional Record Nedir?**

- Norminal Record'lar Object Initializer'lar ile ilk değerleri verilerek üretebilen readonly datalardı.

- Positional Record'lar ise esasında Record'lar içerisinde tanımlama yapabildiğimiz constructor ve deconstructor

kullanımlarını daha da özelleştirerek kullanılmasını sağlar.

record MyRecord

{

public MyRecord()

{

}

public void Deconstruct()

{

}

}

yapmaktansa;

record MyRecord(string name, string surname)

{

}

tanımladığımızda yukarı semantik arka planda name ve surname propertyleri tanımlayacak ve bu propertylerin özellikleri de init olacaktır.

public class Program

{

static void Main(string[] args)

{

MyRecord m = new MyRecord("ali", "veli");

var (n, s) = m;

}

}

record MyRecord(string name, string surname) //record name(... , ...) bu semantik positional record özelliğidir.

{

//(string name, string surname) parametrelerinin karşılığı olarak compiler seviyesinde propertyler otomatik oluşturulacaktır.

//bu propertyler oluşturulurken init olacak şekilde oluşturulur.

}

- Bir tek Record yapılanmasına özeldir.

- Bir Record üzerinde constructor ve deconstructor yapılanmasını hızlı bir şekilde oluşturmamızı sağlayan bir semantik yapılanmadır.

- Positional Record kullanırken parametrelerin karşılıkları olan propertyleri manuel olarak oluşturmak zorunda değiliz.

**Peki ayrıca Constructor tanımlayabilir miyiz?**

public class Program

{

static void Main(string[] args)

{

MyRecord m = new MyRecord(); //boş constructor kullanıyorsak ve bu record'da positional record varsa

var (n, s) = m;

}

}

record MyRecord(string name, string surname) //positional record'un bize getirdiği constructor'u tetiklemek her daim şart.

{

public MyRecord() : this ("ali", "veli.") //bunlara buradan değerlerini göndermek zorundayız.

{ }}

**Peki ayrıca Property oluşturabilir miyiz?**

- Evet ama pek tercih edilmez. Property'leri kendi oluşturmuş olduğumuz isimlerde kullanabiliriz.

public class Program

{

static void Main(string[] args)

{

MyRecord m = new MyRecord("ali", "veli");

Console.WriteLine(m.Name + " " + m.Surname);

var (n, s) = m;

}

}

record MyRecord(string name, string surname) //fakat bunu yapınca Positional Record'un oluşturmuş olduğu diğer property'lerde duracaktır.

{

public string Name => name;

public string Surname => surname;

}

**DERS 7: Inheritance(Kalıtım) Nedir?**

- OOP'nin en önemli özelliğidir.

- Üretilenn nesneler farklı nesnelere özelliklerini aktarabilmekte ve böylece hiyerarşik bir düzenleme yapılabilmektedir.

- Bir programcı açısından bu özellik; aynı aile grubundan gelen nesnelerin ya da yatayda eşit seviyede olan tüm olguların benzer özelliklerini tekrar tekrar her birinde tanımlamaktansa bir üst sınıfta tanımlanmasını ve her bir sınıfın bu özellikleri üst sınıftan kalıtımsal olarak almasını sağlamaktır.

- Böylece hem kod maliyeti düşmekte, hem de mimarisal tasarım açısından avantaj sağlanmaktadır.

-Canlı-

-insan- -hayvan-

erkek kadın -kedi -inek -tavuk

-montofon

-angus

-sığır

Örnek: Sexist bir örnek vermek gerekirse;

Erkek Kadın

Adı Adı

Soyadı Soyadı

Yaş Yaş

Sakal Makyaj

Bıyık Ruj

ikisinin de ortak özellikleri var. Bunları tekrar tekrar yazmaktansa ortak bir çatı altında, bir üst sınıf altında tanımlamak daha iyi gibi sanki.

- Nesne tabanlı programlamada benzer/aynı olgudaki nesnelerin aynı olan memberları/özellikleri/içerikleri eğer ki bu şekilde her sınıf içinde tekrarlanmışsa bu aykırı bir durumdur. Aynı olguda olan sınıfların tekrar eden memberları bir başka sınıfta toplansın ve oradan ilgili sınıflara kalıtımsal olarak aktarılsın, diye düşünebiliriz.

İnsan //ortak özellikleri insan sınıfında tanımla.

Adı

Soyadı

Yaş

Erkek Kadın //kalıtım kullanarak bu özellikleri Erkek ve Kadın'a gönder.

Sakal Makyaj

//genellenemeyen diğerlerin olmayan ve sadece o sınıfa ait olan özellikler direkt ilgili sınıfta tanımlanmalıdır.

Bıyık Ruj

- Erkek ve Kadın sınıfları İnsan sınıfından kalıtım alırsa/türetilirse/miras alırsa insan sınıfındaki tüm memberlar (erişimine izin verilen/miras olarak aktarılmasına izin verilen memberlar) miras olarak aktarılacaktır.

Kalıtımın Kullanılmadığı Durumlarda Kodumuz;

class Opel

{

public string Marka {get; set;}

public string Model {get; set;}

public int KM {get; set;}

}

Opel

Marka =

Model =

KM =

class Mercedes

{

public string Marka {get; set;}

public string Model {get; set;}

public int KM {get; set;}

}

Mercedes

Marka =

Model =

KM =

class Fiat

{

public string Marka {get; set;}

public string Model {get; set;}

public int KM {get; set;}

}

Fiat

Marka =

Model =

KM =

- Çok lüzumsuz, fazlalık kod yazıldı. Tekrara düşüldü.

Kalıtımın Kullanılmadığı Durumlarda Kodumuz;

class Araba

{

public string Marka {get; set;}

public string Model {get; set;}

public int KM {get; set;}

}

class Opel

{

}

Opel

Marka =

Model =

KM =

class Mercedes

{

}

Mercedes

Marka =

Model =

KM =

class Fiat

{

}

Fiat

Marka =

Model =

KM =

- Araba sınıfından kalıtım aldığımız kod daha sade ve pratik oldu.

- Kalıtım gelişigüzel tasarlanmamalıdır. Ortak olguda olan nesneleri temsil edecek olan bir üst ve daha evrensel nitelikte olgu olmalıdır. Opel, Mercedes, Fiat ortak olgudur. Yani üçüde bir arabadır. Haliyle bunların daha evrensel üst niteliği Araba olarak nitelendirilebilir.

- Kalıtım operasyonunda, kalıtım veren sınıfın erişilebilen tüm memberları kalıtım alan sınıfa kalıtsal olarak aktarılacaktır.

- OOP'de kalıtım özünde nesnelerin birbirlerinden türemesini sağlayan bir özelliktir. Bu özellik yanında birçok özellik ve stratejik yapılanma getirmektedir.

virtual, polimorfizm, abstraction, interface .......

**C# Programlama Dilinde Hangi Yapılar Kalıtım Alabilirler?**

- C# programlama dilinde kalıtım sınıflara özel bir niteliktir.

- Yani bir sınıf sadece bir sınıftan kalıtım alabilir.

- record'lar özünde sınıftır ve kalıtım alabilmektedir. Lakin sadece kendi aralarında. Kalıtım alabildikleri tek istisnai sınıf ise Object sınıfıdır.(ileride) Abstract class, interface ve struct gibi yapıların da kendilerine göre kalıtımsal operasyonları mevcuttur.

**C#'da kalıtım Nasıl Alınır?**

: Operatörü kullanılmaktadır.

- Kalıtımsal tüm ilişkiler : operatörü tarafından yapılmaktadır.

class Araba

{

public string Marka {get; set;}

public string Model {get; set;}

public int KM {get; set;}

}

class Opel: Araba //soldaki sağdakinden kalıtım alsın yani Opel sınıfı, araba sınıfından kalıtım alsın demiş oluyoruz.

{

}

\_\_\_\_\_1\_\_\_\_\_ : \_\_\_\_\_2\_\_\_\_\_\_ -> : operatörü 2 sınıfındaki tüm erişilebilir memberları, 1. sınıfa kalıtımsal olarak aktarmaktadır.

- Kalıtım, operasyonel olarak gerçekleştirildikten sonra compiler seviyesinde member aktarımı sağlanır.

class Araba

{

public string Marka {get; set;}

public string Model {get; set;}

private int KM {get; set;} //bu sadece class Araba içerisinde kullanılabilir. Opel'e aktarılmaz.

}

class Opel: Araba //soldaki sağdakinden kalıtım alsın yani Opel sınıfı, araba sınıfından kalıtım alsın demiş oluyoruz.

{

}

- Yani artık Opel sınıfından bir nesne ürettiğimizde içerisinde Marka ve Model property'leri kalıtımsal olarak aktarıldığı için erişilebilir olacaktır.

Örnek:

namespace OOPExamples

{

public class Program

{

static void Main(string[] args)

{

Muhasebeci muhasebeci = new Muhasebeci();

Mudur mudur = new Mudur();

//Personel'den erişilebilenler

muhasebeci.Adi

muhasebeci.Soyadi

muhasebeci.Medenihal

//Muhasebeci'den gelen

muhasebeci.Musavir

mudur.Musavir //hata, ilişki yok.

yazilimci.KullandigiDiller

}

}

class Personel

{

public string Adi { get; set; }

public string Soyadi { get; set; }

public string Medenihal { get; set; }

}

class Muhasebeci : Personel

{

public bool Musavir { get; set; }

}

class Yazilimci : Personel

{

public string[] KullandigiDiller { get; set; }

}

class Mudur : Personel

{

}

}

**Base Class ve Derived Class Nedir?**

- Kalıtım veren sınıfa Base/Paren Class denir. Kalıtım alan sınıfa Derived/Child Class denir.

class A

{

}

class B:A //Base Class:A, Derived Class: B

{

}

class C:B //Base Class:B, Derived Class: C

{

}

class D:C //Base Class:C, Derived Class: D

{

}

- A, B2nin olduğu gibi bir yandan da C ve D'nin de Base Class'ı mıdır? Hayır. Bir sınıfın sadece tek bir Base Class'ı olabilir. Tüm atalar tüm torunların Base Class'ı

değildir.

- Bir sınıfın Base Class'ın direkt türediği sınıftır.

- Bir class'ın birden fazla Derived Class'ı olabilir.

Örnek:

class A

{

}

class B:A //Base Class:A, Derived Class: B

{

}

class C:A //Base Class:A, Derived Class: C

{

}

class D:C //Base Class:C, Derived Class: D

{

}

- C hem derived hem de base class.

**Kalıtımın Altın Kuralı:**

- C# programlama dilinde bir class'ın sadece tek bir class'tan türetilmesine izin verilmektedir. Aynı anda birden fazla class'tan türeme işlemi gerçekleştirilemez.

class BuyukBaba

{

}

class Baba : BuyukBaba //Anne ile Baba, BuyukBaba'nın derived class'larıdır.

{

}

class Anne : BuyukBaba

{

}

class Ogul : Baba, Anne //hata verir.

{

}

Not:

class Y : X, Z,W //hata verir.

{

}

//İleride birden fazla kalıtım tanımlamasının yapılabildiğini göreceğiz. Lakin orada da göreceğiz ki Z ve W bir sınıf olmayacak.(interface konusu)

Örnek:

class BuyukBaba

{

}

class Baba : BuyukBaba //Anne ile Baba, BuyukBaba'nın derived class'larıdır.

{

}

class Anne : Baba

{

}

class Ogul : Anne //tarzı bir çözümle Ogul hem Anne'den hem Baba'dan kalıtım alabilir.

{

}

**Kalıtımda Nesne Üretim Sırası**

- Bir sınıftan nesne üretimi yapılırken kalıtım aldığı üst sınıflar varsa eğer önce o sınıflardan sırayla nesne üretilir.

A -> B -> C

new C(); //dediğimizde önce A'dan sonra B'den en son C'den nesne üretilir. HEAP'te A, B ve C için üretilen 3 tane nesne olacaktır.

Örnek:

new B(); //önce A'dan bir nesne oluşturulur. Sonra istenilen B nesnesi oluşturulur. HEAP -> A ve B

class A

{

}

class B:A

{

}

Örnek:

new D(); //Ne kadar Ata varsa herbirinden nesne oluşturulur.

class A //önce A'dan

{

}

class B:A //sonra B'den

{

}

class C:B //sonra C'den

{

}

class D:C //en son D'den nesne üretilir.

{

}

Örnek:

namespace OOPExamples

{

public class Program

{

static void Main(string[] args)

{

new D();

}

}

class A //önce A'dan

{

public A()

{

Console.WriteLine($"{nameof(A)} nesnesi oluşturulmuştur.");

}

}

class B : A //sonra B'den

{

public B()

{

Console.WriteLine($"{nameof(B)} nesnesi oluşturulmuştur.");

}

}

class C : B //sonra C'den

{

public C()

{

Console.WriteLine($"{nameof(C)} nesnesi oluşturulmuştur.");

}

}

class D : C //en son D'den nesne üretilir.

{

public D()

{

Console.WriteLine($"{nameof(D)} nesnesi oluşturulmuştur.");

}

}

}

Çıktı:

A nesnesi olusturulmustur.

B nesnesi olusturulmustur.

C nesnesi olusturulmustur.

D nesnesi olusturulmustur.

**Bir Sınıftan Base Class Constructor'una Ulaşım**

- Madem ki, herhangi bir sınıftan nesne üretimi gerçekleştirildikten öncelikle base class'ından nesne üretiliyor, bu demektir ki önce base class'ın constructor'ı tetikleniyor.

- Haliyle bizler nesne üretimi esnasında base class'ta üretilecek olan nesnenin istediğimiz constructor'larını tetikleyebilmeli yahut varsa parametre, bu değerleri verebilmeliyiz.

- İşte bunun için base keyword'ünü kullanmaktayız.

class Program

{

static void Main(string[] args)

{

new MyClass2(); //buradaki operasyon, MyClass2'den nesne oluşturulurken,

}

}

class MyClass //ilk buradan bir nesne oluşturacak.

{

public MyClass(int a) //bu constructor'u tetiklemek lazım.

{

//fakat buradaki constructor parametreli, bu parametreye ne değer verileceğini compiler bilemez.

}

}

class MyClass2 : MyClass //o yüzden burada hata alırız.

{

//o değeri buradan göndermek lazım. Bunu yapabilmek için de base keyword'ünü kullanırız ve base class'ın constructorlarına //erişiriz.

}

düzeltirsek;

class Program

{

static void Main(string[] args)

{

new MyClass2();

}

}

class MyClass

{

public MyClass(int a)

{

}

}

class MyClass2 : MyClass

{

public MyClass2() : base (5) //hata ortadan kalktı.

{

//Myclass2'den bu constructor'u kullanarak bir nesne oluşturuyorsam öncelikle base class'ın nesnesini oluşturacağından haliyle //bu işlemde (5) değerini

//MyClass constructor'undaki (int a) 'ya alacağından base constructor'u tetikleyerek o nesneyi oluşturup sonra MyClass2'nin //nesnesini oluşturacak.

}

public MyClass2(int a) : base (a) //ya da base(5)

{

//burada da aynı hatayı almamak için base eklememiz gerekli. Bu constructor'u kullanarak nesne oluşturmaya kalkarsak, bu //constructor ilgili base class

//ilgili base class'ın parametre alan constructor'una hangi değeri vereceğini bilemediğinden dolayı burada da o değeri bizim //öndermemiz lazım.

}

}

- Eğer ki base class'ın constructor'u sadece parametre alan constructorise derived class'larda o constructor'a bir değer göndermek zorundayız. Bunu da base keyword'ü ile sağlayabiliriz.

Base Class'ta birden fazla constructor olması durumunda (overload);

class Program

{

static void Main(string[] args)

{

new MyClass2();

}

}

class MyClass

{

public MyClass(int a)

{

}

public MyClass() //eğer ki base class'ta boş parametreli bir constructor overload'umuz olsaydı

{

}

}

class MyClass2 : MyClass

{

public MyClass2()

//derived class'ta base ile bir bildirimde bulunmamıza gerek yoktu. Çünkü varsayılan olarak kalıtımsal durumda boş parametreli

//constructor (base class'taki) tetiklenir.

{

}

public MyClass2(int a) : base (a)

{

}

}

Örnek:

- Bir class'ın constructor'ının yanında: base (...) keywordünü kullanırsak eğer o class'ın base class'ının tüm constructorlarını bize getirecektir. Haliyle ilgili sınıftan bir nesne üretilirken base class'tan nesne üretimi esnasında hangi constructor'un tetikleneceğini bu şekilde belirleyebiliriz.

class Program

{

static void Main(string[] args)

{

new MyClass2();

}

}

class MyClass

{

//base class'ımızda birden fazla constructor var.

public MyClass(int a)

{

}

public MyClass(string a)

{

}

public MyClass(int a, string b) //bu constructor tetiklenecektir.

{

}

public MyClass()

{

}

}

class MyClass2 : MyClass

{

//biz de derived class'ta bu class'tan nesne oluşturulurken, base class'taki bu constructorlardan herhangi birinin tetiklenip ona göre nesnesini

//oluşturulmasını istiyorsam eğer.

public MyClass2() : base(5,"Rammstein") //işte öyle bir durumda constructor'a göre base'deki istediğimiz bir consttuctor'u seçebiliriz.

{

//base(5,"Rammstein") böyle bir base belirlersek.

}

public MyClass2(int a) : base (a)

{

} }

**base Keyword ile this Keyword farkları;**

- this, bir sınıftaki constructor'lar arasında geçiş yapmamızı sağlar.

- base, bir sınıfın base class'ının constructor'larından hangisinin tetikleneceğini belirlememizi ve varsa parametrelerinin değerlerinin dervied class'tan

verilmesini sağlar.

Sembolik olarak this;

constructor(): this(3) //buradaki this tek parametreli olan bir alttaki constructor'a yönlendirir.

{ }

constructor(int a): this(a,5) //bu this bir alttakine

{ }

constructor(int a, int b): this(a) //bu this ise bir yukarıdakine yönlendirmiş.

{ }

constructor(string c): this() //bu this ise en üsttekine yönlendirmiş.

{ }

Sembolik olarak base;

Base Class

constructor()

{ }

constructor(int a)

{ }

Derived Class

constructor(): base() //base'deki boş parametreli constructor'a git.

{ }

constructor(int a) : base(a) //base'deki parametreli constructor'a git.

{ }

- This, ilgili sınıfta o anki nesnenin memberlarına erişebilmemizi sağlıyorsa, aynı şekilde base de base class'daki memberlara erişebilmemizi sağlar.

Base Class

int a;

public int MyProperty{ get; set; }

public void X()

{

}

public void Y()

{

}

Derived Class

int b;

public int MyProperty2{ get; set; }

private void Z()

{

//this bu sınıf içerisindeki member'lara erişmemizi sağlar.

this.b = 5;

this.MyProperty2 = 10;

base.a = 15; //a erişilebilir değil. hata verir. default olarak private.

this.MyProperty2 = 20;

base.X(); //base ise base class'taki member'lara erişmemizi sağlar.

base.Y(); //erişilebilir değil.

}

Not: Erişim belirleyicilere dikkat etmek şartıyla, C#'da memberların başına this veya base koymadan da işlemler yapabilmekteyiz. Fakat constructor'lar ile kullanımda şarttır.

Örnek:

class A

{

int a;

public int b;

public int MyProperty { get; set; }

}

class B : A

{

int c;

public void X()

{

base.MyProperty = 1;

MyProperty = 10; //base yazmadan da bu şekilde kullanabiliriz. Çünkü artık A'dan gelen bir member. Compiler otomatikman başına base koyacaktır.

b = 1; //aynı şekilde base koymaya gerek yok.

c = 2; //yine aynı şekilde compiler otomatik olarak this'leyecektir.

}

}

**Nesnelerdeki ToString, Equals, GetHashCode ve GetType Metotları nereden gelmektedir?**

**Nesnelerin Atası/Adem'i Object Türü**

- C# programlama dilinde tüm sınıflar Object sınıfından türetilir. (ileride delegate konusunda bu cümle değişebilir.)

Object yazıp sağ tıklayıp Go To Definition dersek;

Object sınıfının Memberlarını görebiliriz.

namespace System

{

//

// Summary:

// Supports all classes in the .NET class hierarchy and provides low-level services

// to derived classes. This is the ultimate base class of all .NET classes; it is

// the root of the type hierarchy.

public class Object

{

public Object(); //constructor

~Object(); //destructor

//members

//public olanlar tüm sınıf nesnelerinde otomatik olarak erişilebilir.

public static bool Equals(Object? objA, Object? objB);

public static bool ReferenceEquals(Object? objA, Object? objB);

public virtual bool Equals(Object? obj);

public virtual int GetHashCode();

public Type GetType();

public virtual string? ToString();

protected Object MemberwiseClone();

}

}

class Canli

{

public int Yasi {get; set;}

}

class Insan: Canli

{

}

- Bir sınıf kalıtım almazsa default olarak Object sınıfından türetilir.

- Herhangi bir sınıfttan kalıtım alıyorsa, bir sınıfın aynı anda birden fazla sınıftan kalıtım almama presinsibinden yol çıkarak bir yandan da Object sınıfından türemeyecek sadece kalıtım aldığı sınıftan türeyecektir.

- Tabi burada kalıtım veren sınıf herhangi bir sınıftan türemiyorsa eğer en nihayetinde Object'ten türeyeceği için dolaylı yoldan insan sınıfı da Object'ten kalıtım almış olacaktır.

**İsim Saklama (Name Hiding)**

- Kalıtım durumlarında atalardaki herhangi bir member ile aynı isimder member'a sahip olan nesneler olabilmektedir.

class A

{

public int X {get; set;}

}

class B : A

{

public int X {get; set;}

}

B b = new B();

b.X -> Nereden geldi bu member? A'mı B'mi İşte bunun adı Name Hiding'dir.

- Bu tasarımda base class'tak/atalardaki X member'ına B üzerinden erişmek mümkün değildir! Yani base'deki üye gizlenmiştir.

- Derleriycii hata vermez ama uyarı verir.

Warning CS0108 'B.X' hides inherited member 'A.X'. Use the new keyword if hiding was intended.

- Görüldüğü üzere B sınıfının base class'taki X member'ını gizleyeceğini ifade etmekte ve gizlenmesi için ise new keyword'ünün kullanılması gerektiği söylenmektedir.

Eskiden name hiding durumlarında alt sınıfın içerisindeki üyeyi new ile işaretliyorduk(bilinçli bir şekilde yapıyorum). Günümüzde buna ihtiyaç yoktur.

Yine de Name Hiding durumlarında new operatörünün kullanımını incelersek:

class A

{

public int X {get; set;}

}

class B : A

{

public new int X {get; set;}

}

**Record'larda Kalıtım**

- Record'lar sade ve sadece Record'lardan kalıtım alabilmektedir.

- Class'lardan kalıtım alamazlar yahut veremezler.

- Kalıtımın tüm temel kuralları geçerlidir.

- Bir record aynı anda birden fazla record'dan kalıtım alamaz.

- Record'lar da temelde class oldukları için üretilir üretilmez otomatik olarak Object'ten türerler.

- base ve this keyword'leri aynı amaçlar kullanabiliriz.

- Name Hiding söz konusu olabilmektedir.

record MyRecord

{

}

record MyRecord2 : MyRecord

{

}

Örnek: Kalıtım Hiyerarşisi Örneği

using System;

namespace TestInheritance

{

public class Person //Person adında bir class tanımladık. name adında bir field'ı ve outName adında metodu var.

{

public string name;

public void outName()

{

Console.Write("Name: {0}", this.name);

}

}

public class Staff : Person //Staff adında Person'dan kalıtım alan bir sınıf tanımladık.

{

public int salary; //kendisinin de field ve

public void outStaff() //çıktı metodu da var.

{

outName(); //metodun içinden Person'daki outName metodunu çağırdık. this.outName();'de diyebilirdik.

Console.Write("\tSalary: ${0}", this.salary);

}

}

public class Student : Person //Person'dan kalıtım alan Student sınıfı oluşturduk.

{

public byte grade = 1; //grade adında yeni bir field,

public void outStudent() //çıktı vermesi için bu metot oluşturuldu.

{

outName();

Console.WriteLine("\tGrade: {0}", this.grade);

}

}

public class Teacher : Staff //Teacher isimli Staff'tan kalıtım alan bir sınıf oluştur.

{ //ayrıca Person'dan da kalıtsal özellikleri alacaktır.

public int teachTime; //field

public void outTeacher() //metot

{

outStaff();

Console.WriteLine("\tTeach Time: {0}hrs", this.teachTime);

}

}

public class Cleaner : Staff //Staff'tan kalıtım alan bir başka sınıf oluşturuldu.

{

public string cleanArea; //field

public void outCleaner() //output metot

{

outStaff();

Console.WriteLine("\tClean Area: {0}", this.cleanArea);

}

}

class Program //main metodu buraya yazalım.

{

static void Main(string[] args)

{

Teacher t = new Teacher(); //Teacher sınıfından bir nesne oluşturalım.

t.name = "Ben"; //değerleri ver

t.salary = 3000;

t.teachTime = 500;

t.outTeacher(); //ve metodu çağır. bunu yaparken kalıtım gereği outName ve outStaff'da çağırılır.

Cleaner c = new Cleaner(); //Cleaner sınıfından bir nesne.

c.name = "David";

c.salary = 1500;

c.cleanArea = "Play Ground";

c.outCleaner(); //ve metodu çağır. bunu yaparken outName ve outStaff'da çağırılır.

Student s = new Student(); //son olarak Student sınıfından s nesnesi oluştur.

s.name = "Steven";

s.grade = 3;

s.outStudent(); //'ı çağır ve bunu yaparken outName'i de tetikle.

Console.Read();

}

}

}

Çıktı:

Name: Ben Salary: $3000 Teach Time: 500hrs

Name: David Salary: $1500 Clean Area: Play Ground

Name: Steven Grade: 3

Örnek: Metodlar ve Constructor ile Kalıtım

namespace TestInheritance2

{

public class Pet //Pet adlı bir sınıf oluştur.

{

private string name; //name adlı private bir string field oluştur.

public Pet() : this("None") //default constructor'u this("None") ile tanımla. this("None") demek Pet("None") demektir. Bununla public Pet(string name) constructor'unu çağırmış oluyoruz.

{

//eğer ki bu default constructor silinirse,

//Error CS7036 There is no argument given that corresponds to the required formal parameter 'name' of 'Pet.Pet(string)' hatası alınacaktır.

//Kalıtım almış derived class Dog yaratıldığında, base class'ın default constructor'u Pet() otomatik olarak çağırılacaktır.

//Eğerki Pet sıfında tanımlanan başka constructor yoksa default constructor otomatik olarak yaratılacaktır.

//Çünkü halihazırda tek parametreli public Pet(string name) constructor'u olduğundan default constructor üretilemez ve hata alınır.

}

public Pet(string name) //tek parametreli bir constructor tanımla.

{

this.setName(name); //setName metodunu çağır ve name'in değerini ata.

}

public void setName(string name) //setName adlı bir metot tanımla.

{

this.name = name; //parametredeki değeri field'a ata.

}

public void outName() //outName adlı bir metot tanımlayıp private name field'ının değerini çıktı olarak ver.

{

Console.Write("Name: {0}", this.name);

}

}

public class Dog : Pet //Dog isimli Pet sınıfından kalıtım alan bir sınıf oluştur.

{

public bool isBark; //isBark isimli boolean bir field tanımla.

public Dog(bool isBark) //Dog sınıfında tek parametreli bir constructor tanımla ve parametreden gelen değeri field'a ata.

{

this.isBark = isBark;

}

public void outDog()

{

this.outName(); //Pet sınıfından kalıtım almış outName() metodunu çağır.

Console.WriteLine("\tBark: {0}", this.isBark); //isBark'ı çıktı al.

}

}

class Program

{

static void Main(string[] args)

{

Dog d = new Dog(true); //Dog sınıfından d nesnesi oluştur ve true değerini tek bool parametreli constructor'a gönder.

d.outDog(); //outDog() metodunu tetikle.

d.setName("Rocky"); //kalıtım ile gelen setName'i çağır ve tek string parametreli string constructor ile name'e "Rocky"'i ata.

d.isBark = false; //isBark'a false'ı ata.

d.outDog(); //tekrar outDog'u çağır.

Console.Read();

}

}

}

Çıktı:

Name: None Bark: True

Name: Rocky Bark: False

**DERS 8: Sanal Yapılar (Virtual & Override)**

- Bir nesne üzerinde var olan tüm memberların tamamı derleme zamanında belirgindir.

- Yani, derleme aşamasında hangi nesne üzerinden hangi metotların çağırılabileceği bilinmektedir.

class MyClass

{

public int MyProperty {get;set;}

public void MyMethod()

{

}

}

--> Compiler time

--> Bu sınıftan bir nesne üreteceğim zaman bu sınıfın içerisinde hangi metotların, propertylerin olacağını biliyoruz.

Buna derleme zamanında belirginlik diyoruz.

- Sanal yapılar ile derleme zamanında kesin bilinen bu bilgi runtime'da (çalışma zamanında) belirlenebilmektedir.

Yani ilgili nesnenin hangi metodu kullanacağı bilgisi runtime'da kararlaştırılır.

**Sanal Yapılar Nedir?**

- Sanal yapılar, bir sınıf içerisinde bildirilmiş olan ve o sınıftan türeyen alt sınıflarda da tekrar bildirilebilen yapılardır.

Bu yapılar metot ya da property olabilir.

Not: Name Hiding ile bir sınıftaki herhangi bir member'ı ondan türeyen torunlarda oluşturabiliyoruz ve buradaki

yaşanan çakışmaya da Name Hiding diyoruz.

Not: Lakin sanal yapılarda olay bu şekilde değildir. Bir sınıfta bildirilen sanal yapı(metot ya da property) bu sınıftan

türeyen torunlarında ezilebilmekte yani devre dışı bırakılıp yeniden oluşturulabilmektedir.

- Bir sınıfta tasarlanmış sanal yapının işlevinin iptal edilip edilmeme durumuna göre tanımlandığı sınıftan mı yoksa

bu sınıfın torunlarından mı çağrılacağının belirlenmesi runtime'da gerçekleşecektir.

A Object

Member //bu ve

Member //bu memberlar sanal olmadığı için derleme zamanı hangi nesneden çağırılabilecekleri bilinmektedir.

Sanal Member //Lakin bu member sanal olduğu için kendisinden miras alan torunlarında ezilip ezilmeme/iptal edilip

//yeniden yazılma durumuna göre ya A Object'ten ya da B Object'ten çağrılacaktır.

//eğer ki bu sanal member ezilmezse/iptal edilmezse B object'ten miras edilecek lakin tanımlandığı

//A Object'inden çağrılacaktır.

//Yok eğer ezilirse ve yeniden yazılırsa, artık bu işlemin yapıldığı torunun bir member'ı olcaktır ve

//B Object'inden çağrılacaktır. Fiziksel bir member olacaktır.

---> interits

B Object

Member

Member

Sanal Member

- İşte bu davranışın kararı ancak Run Time'da anlaşılacağı için Run Time'da bunların kararı verilmektedir.

- Metot ya da property fark etmez! Bir sanal member'ın hangi türe ait olduğunun bu şekilde runtime'da belirlenmesine

Geç Bağlama(Late Binding) denir!

**Sanal Yapılar ne için kullanılır?**

- Bir sınıfta tanımlanmış olan herhangi bir member'ın kendisinden türeyen alt sınıflarda Name Hiding durumuna

düşmemeksizin ezilip/yeniden yazılıp yapılandırılması için kullanılır.

Peki bu zorunlu mudur? Yani bir sanal yapı illa ki kendisinden türeyen torunlarda ezilmek/yeniden yazılmak zorunda mıdır?

- Tabi ki de hayır! Bir member sanal yapıldı diye illa ki kendisinden türeyen alt sınıflarda ezilmek zorunda değildir.

Ama ezilmek istenirse de Name Hiding'e bulaşmadan direkt ilgili sınıfın bir member'ı olacak şekilde çalışması sağlanmış olur.

**virtual Keyword'ü**

- Bir sınıfta sanal yapı oluşturabilmek için ilgili member'ın (metot ya da property) imzasını virtual keyword'ü ile

işaretlemek yeterlidir.

public virtual ya da virtual public

class MyClass

{

public void MyMethod() //normal metot, MyClass'tan üretilen bütün class'larda miras olarak aktarılabilir.

{

}

}

Sanal metot;

class MyClass

{

public virtual void MyMethod() //MyClass'tan üretilen bütün class'larda miras olarak aktarılabilir.

{

//ek olarak aktarılan bu sınıflardan herhangi birinde ezilebilir yani iptal edilip yeniden yazılabilir.

}

}

Normal property;

class MyClass

{

public int MyProperty(get; set;)

{

}

}

Sanal property;

class MyClass

{

virtual public int MyProperty(get; set;)

{

}

}

**Override Keyword'ü**

- Bir class'ta virtual ile işaretlenerek sanal hale getirilmiş bir member(metot ya da property), bu class'tan

miras alan torunlarında ezilmek isteniyorsa eğer ilgili class'ta imzası override ile işaretlenmiş bir vaziyette tekrardan

aynı member oluşturulur.

class MyClass

{

public virtual void MyMethod()

{

}

}

Class MyClass2 : MyClass

{

public override void MyMethod() //MyMethod'u MyClass2 nesnesi üzerinden çağırırsak MyClass2'den gelecektir.

{

//MyClass'tan gelmeyecektir. Artık buraya ait. Ezildi, yenilendi.

}

}

- Aynı durum property'ler için geçerli.

- Base class'ta ya da atalarda virtual ile işaretlenerek sanallaştırılmış herhangi bir member torunlarda illa ki override

ile ezilmek zorunda değildir.

- Ama bir torun class base class'ta ki herhangi bir member'ı override edecekse eğer o member'ın kesinlikle virtual ile

işaretlenmiş olması gerekmektedir.

- virtual ile işaretlenmemiş bir metot kesinlikle override edilemez.

Not: overide keyword'ü ileride Abstract Class'ların implementasyonunda da kullanacağız. (ileride)

Örnek:

namespace TestInheritance2

{

class Program

{

static void Main(string[] args)

{

Terlik t = new Terlik();

t.Bilgi();

Kalem k = new Kalem();

k.Bilgi();

}

}

class Obje

{

virtual public void Bilgi()

{

Console.WriteLine("Ben bir objeyim...");

}

}

class Terlik : Obje

{

public override void Bilgi()

{

Console.WriteLine("Ben bir terliğim.");

}

}

class Kalem : Obje

{

public override void Bilgi()

{

Console.WriteLine("Ben bir kalemim.");

}

}

}

Çıktı:

Ben bir terligim.

Ben bir kalemim.

**İkiden Çok Hiyerarşik Kalıtım Durumlarında Override Durumu**

- Bir class'ta virtual işaretlenmiş herhangi bir member illa ki direkt o class'tan türeyen 1. dereceden class'lar da

override edilmek mecburiyetinde değildir.

İhtiyaca göre alt seviyeden torunlardan herhangi birinde override edilebilir.

A Object

virtual

--> interits

B Object

//B'yi atladık ve C'de override ettik.

--> interits

C Object

override

--> interits -->\*

Peki C'nin altındaki torunlarda override edilebilir mi?

- İlgili soyut member'ın en son override edildiği Object'ten sonra geçerli olduğu kritiği önemlidir.

Ve o object'ten sonra hiyerarşik olarak türetilen sınıflar varsa onlarda da override işlemi gerçekleştirilebilir.

\*-->

D Object

--> interits

E Object

everride

--> interits

- A'daki virtual member C'de override edildiği zaman devamında ister D ister E'de de override edilebilir.

- C'de override ettik bundan sonraki davranış D'ye E'ye de burada override ettiğimiz şekilde gönderilecektir.

Fakat E'de tekrar override edersek bundan sonra varsa diğer sınıflarda E'deki yeni davranışı ile devam edecektir.

Örnek:

namespace TestInheritance2

{

class Program

{

static void Main(string[] args)

{

Maymun m = new Maymun();

m.Konus();

Inek i = new Inek();

i.Konus();

}

}

class Memeliler

{

public virtual void Konus()

{

Console.WriteLine("Ben konuşmuyorum.");

}

}

class Maymun : Memeliler

{

public override void Konus()

{

Console.WriteLine("Ben maymunum.");

}

}

class Inek : Maymun

{

public override void Konus()

{

Console.WriteLine("Ben ineğim.");

}

}

}

Çıktı:

Ben maymunum.

Ben inegim.

Örnek:

namespace TestInheritance2

{

class Program

{

static void Main(string[] args)

{

Ucgen u = new Ucgen(3, 4);

Console.WriteLine(u.AlanHesapla());

Dortgen d = new Dortgen(3, 4);

Console.WriteLine(d.AlanHesapla());

DikDortgen dik = new DikDortgen(3, 4);

Console.WriteLine(dik.AlanHesapla());

}

}

class Sekil

{

protected int \_boy;

protected int \_en;

//protected ile işaretlenmiş herhangi bir member sadece ilgili sınıft yahut o sınıftan kalıtım almış olan

//sınıfların içerisinde erişilebilir. Fakat nesne üzerinden erişilemez.

public Sekil(int boy, int en)

{

this.\_boy = boy;

this.\_en = en;

}

virtual public int AlanHesapla()

{

return 0;

}

}

class Ucgen : Sekil

{

public Ucgen(int boy, int en) : base(boy,en)

{

}

public override int AlanHesapla()

{

return \_boy\*\_en /2;

}

}

class Dortgen : Sekil

{

public Dortgen(int boy, int en) : base(boy, en)

{

}

public override int AlanHesapla()

{

return \_boy \* \_en;

}

}

class DikDortgen : Sekil

{

public DikDortgen(int boy, int en) : base(boy, en)

{

}

public override int AlanHesapla()

{

return \_boy \* \_en;

}}}

Çıktı:

6

12

12

Dipnot: Static yapılar sanal olarak bildirilemezler.

Örnek:

using System;

namespace TestVirtualOverride

{

public class A

{

public int i = 1;

public int j = 2;

public virtual void test()

{

Console.WriteLine("A class");

}

}

public class B : A

{

new public int j = 3; // Add new here

public override void test() // Add new here

{

Console.WriteLine("B class");

}

}

class Program

{

static void Main(string[] args)

{

A a = new A();

Console.WriteLine("i={0}, j={1}", a.i, a.j); // Output i=1, j=2

a.test(); // Output A class

B b = new B();

Console.WriteLine("i={0}, j={1}", b.i, b.j); // Output i=1, j=3

b.test(); // Output B class

Console.Read();

}

}

}

Çıktı:

i=1, j=2

A class

i=1, j=3

B class

**DERS 9: Polimorfizm(Çok Biçimlilik) Nedir?**

- İki ya da daha fazla nesnenin aynı tür sınıf tarafından karşılanabilmesi/referans edilmesidir.

Bir başka deyişle bir nesnenin birden fazla türdeki referans tarafından işaretlenebilmesi polimorfizm'dir.

- Polimorfizm, bir nesnenin kalıtımsal olarak ilişkisi olan diğer nesnelerin referanslarıyla işaretlenebilmesini sağlar.

- Polimorfizm, OOP tasarımlarında geliştirilen koda daha manevrasal bir şekilde nitelik kazandıran ve yaklaşım sergilememizi sağlayan bir özelliktir.

- Programlamadaki temel prensip olan sol/sağ prensibi (lafın gelişi) aşıp, eldeki nesnenin birden fazla referansla işaretlenebilmesini sağlar.

A a = new A();

ancak işaretlenebilir 'A' türünden bir nesne

'A' türünden

bir referansla

aşıp, eldeki nesnenin birden fazla referansla işaretlenebilmesini sağlar.

- Bir nesnenin, birden fazla referansla işaretlenmesi, o nesnenin, birden fazla türün davranışlarını gösterebilmesini sağlar.

Not: Yani bir olguyu çoklu(poli) form (morfizmos) olarak tarif edebilmektir.

Örnek:

'Kuş' cinsinden olan tüm hayvanların kendi türlerinin dışında bir yandan 'Kuş' olarak tarif edilmeleri çok biçimliliktir.

Peki kuş olmayan bir hayvana 'Kuş' diyemiyoruz! Misal 'Maymun' bir 'Kuş' değildir! Buradan şöyle bir sonuca varabilir miyiz?

Ortak atadan gelen, kalıtımsal olarak 'Kuş'tan türeyen tüm hayvanlar kendi türleri yahut 'Kuş' türü ile referans edilebilirler.

Buradan anlıyoruz ki, yazılımsal açıdan çok biçimliliğin söz konusu olabilmesi için teknik olarak 'Kalıtım' olması gerekmektedir.

Örnek:

Elimizdeki herhangi bir byte türünde veriyi ister byte ister byte'dan büyük olan herhangi bir türde tutmak çok biçimliliktir.

byte 100

byte value = 100

int value = 100

ya da

string "hilmi"

object value = "hilmi"

string value = "hilmi"

Yukarıda görüldüğü üzere "hilmi" değerini ister 'object' türüyle referans edebilir istersekte 'string' türüyle işaretleyebiliriz.

İşte burada 'object' ve 'string'in aynı veriyi işaretleyebilmesi bir çok biçimlilik-polimorfizm durumudur.

- Nesne Tabanlı Programlama'da bir nesneyi kendi türünün referansıyla birlikte farklı türdeki referanslarla işaretleyerek Polimorfizm'i uygulayabilmekteyiz.

Peki bunu nasıl yapıyoruz? Normal şartlarda bir nesne kendi sınıfının referansı dışında başka bir sınıfın referansıyla işaretlenemez!

- Bir nesnenin bir başka nesne ile işaretlenebilmesi/referans edilmesi için kesinlikle arada kalıtımsal bir ilişki olması gerekmektedir.

Yani bir başka deyişla Nesne Tabanlı Programlama'da Polimorfizm aralarında kalıtımsal ilişki olan sınıflarda uygulanabilir. Aksi mümkün değildir!

Örnek:

namespace deneme

{

class Program

{

static void Main(string[] args)

{

MyClass m = new MyClass();

MyClass2 m = new MyClass(); //arada kalıtımsal ilişki yoksa hata verir.

//böyle bir durumda MyClass2, MyClass'ı referans edebilecekse Ata olan MyClass2 olmalı.

}

}

class MyClass : MyClass2

{

}

class MyClass2

{

}

}

Polimorfizm Kalıtım İlişkisi

- Bir nesneyi, kendi türünün dışındaki bir tür ile/referansla işaretleyebilmek için kesinlikle ilgili nesne, o türden türemiş olması gerekmektedir.

class A

{ }

class B

{ }

B b = new B(); //bugüne kadar bu prensibi izledik.

STACK HEAP

(B) b B object

(A) a B object //şimdilik mümkün değil

A a = new B(); //şu anki durumda hata verir.

class A

{ }

class B : A // bu şekilde B sınıfı A sınıfından türediği takdirde artık B sınıfı bir yandan da A sınıfı olacağı için B nesnesi A türde a referansı tarafından işaretlenebilecektir.

{ }

STACK HEAP

(B) b B object

(A) a B object //sorun yok.

Örnek:

namespace deneme

{

class Program

{

static void Main(string[] args)

{

A a = new B();

B b = new B();

C c = new B(); //...buradaki polimorfizm geçerli olacaktır.

}

}

class A : C //C'den B nesnesi oluşturmak istiyorsak,bir sınıf sadece tek bir sınıftan türeyebilir ilkesi ile,

//class B : A, C yazamayacağımızdan dolayı, B'yi C'den türetebilmek istiyorsak B'nin atalarından birini C'den türetmemiz gerek.

{

//Yani A'yı C'den türetirsek A, C olacak gibi düşünelim B'de A olduğu için dolaylı yoldan B'de C niteliğini kanacak ve ...

}

class B : A //A'dan B nesnesi oluşturmak istiyorsak B, A'dan kalıtım almalı.

{

}

class C

{

}

}

Örnek:

class Insan

{

}

class Erkek : Insan

{

}

class Kadın : Insan

{

}

Erkek e = new Erkek();

//işte Kadın ve Erkek nesnelerinin Insan referansıyla işaretlenebilmesi polimorfizm'dir.

Insan i = new Erkek();

Insan i = new Kadin();

**Peki Polimorfizm Bir Nesne Yöneliminde Neye Yarar?**

class A

{

public void X() {}

}

class B : A

{

public void Y() {}

}

class C : B

{

public void Z() {}

}

new C(); //HEAP -> C Object [ X function Y function Z function ]

C c = new C(); //C nesnesini c referansı ile işaretlersem C'deki bütün elemanlara erişebilirim.

**Polimorfizm Türleri (Genel Kültür)**

1)Statik Polimorfizm

2)Dinamik Polimorfizm

**Statik Polimorfizm**

- Static, ileride göreceğimiz bir kavramdır. Şimdilik sadece Polimorfizm çevresinde Static Polimorfizm'i değerlendireceğiz.

- Static polimorfizm; derleme zamanında sergilenen polimorfizm'dir. Hangi fonksiyonun çağrılacağına derlem zamanında karar verilir.

- C#'da static polimorfizm denilince akla Metot Overloading terimi akla gelmelidir.

Not: Metot Overloading; aynı isimde birbirinden farklı imzalara sahip olan metotların tanımlanmasıdır. Ya da başka deyişle bir isme birden fazla farklı türde metot yüklemektir. Haliyle burada bir metodun birden fazla formunun olması polimorfizm'ken bunlardan kullanılacak olanın derleme zamanında bilinmesi statik polimorfizm olarak nitelendirilmektedir.

class Program

{

static void Main(string[] args)

{

Matematik m = new Matematik();

m.Topla(4, 6); //hangi metodun kullanılacağı, şu anda yazdığım anda derleme zamanında karar verildi.

}

}

class Matematik

{

public long Topla(int s1, int s2) => s1 + s2;

public long Topla(int s1, int s2, int s3) => s1 + s2 + s3;

public long Topla(int s1, int s2, int s3 , int s4) => s1 + s2 + s3 + s4;

}

**Dinamik Polimorfizm**

- Çalışma zamanında sergilenen polimorfizm'dir. Yani hangi fonksiyonun çalışacağına runtime'da karar verilir.

- C#'da dinamik polimorfizm deyince akla Metot Override gelmektedir.

namespace deneme

{

class Program

{

static void Main(string[] args)

{

Arac a = new Arac();

a.Calistir(); //derleme süreci

Taksi t = new Taksi();

t.Calistir(); //derleme süreci

Arac arac = new Taksi();

arac.Calistir(); //Araç sınıfdaki Calistir()'mi Taksi sınıfındaki Calistir()'mi

//tetiklenecek buna runtime'da karar verilir.

}

}

class Arac

{

public virtual void Calistir() => Console.WriteLine("Araç çalıştı.");

}

class Taksi : Arac

{

public override void Calistir() => Console.WriteLine("Taksi çalıştı.");

}

}

**Polimorfizm Durumlarında Tür Dönüşümleri**

- Polimorfizm, OOP'de bir nesnenin kalıtımsal açıdan ataları olan referanslar tarafından işaretlenebilmesidir. Haliyle ilgili nesne, bu ataları olan referans türlerine göre dönüştürülebilmektedir.

namespace deneme

{

class Program

{

static void Main(string[] args)

{

A a = new C(); //bu nesne A referansında tutulan/işaret edilen/referans edilen bir C türünden nesnedir.

//Haliyle ihtiyaç doğrultusunda a referansı üzerinden diğer kalıtımsal ilişkide olduğu referanslara yahut

//kendi referansına dönüştürülebilmektedir.

C c = (C)a; //misal olarakiburada görüldüğü üzere A türünden olan a referansındaki, özünde C türünden

//nesne kendi türünden bir referansla işaretlenmiştir. (cast operatörü ile) (bilinçli dönüşüm)

C c2 = new C();

A a2 = c2; //bilinçsiz dönüşüm yapılıyor. A a2 = (A) c2; bilinçli dönüşüm yazmamıza gerek yok.

//Bu durumun tersi de geçerlidir. Yani ilgili nesne kendi türünden kalıtımsal olarak ataları olan diğer

//türlere cast edilebilmektedir.

}

}

class A

{

public string X { get; set; }

}

class B :A

{

public string Y { get; set; }

}

class C : B

{

public string Z { get; set; }

}

}

Not: Polimorfizm durumlarında kalıtımsal açıdan üst bir referans ile işaretlenebilmiş herhangi bir nesneyi kendi türünden işaretleyebilmek için Cast operatörünü kullanabilmekteyiz. Buradan anlıyoruz ki, object türünde gerçekleştirilen UnBoxing durumu esasında object türü ile gerçekleştirilebilen Polimorfizm'in bir sonucudur.

Hatırlatma: Unboxing

Unboxing object tipinin direk (explicit) olarak değer tipine (value type) dönüştürülme işlemi olarak tanımlanabilir.

int i = 123; // değer tipi (value type)

object o = i; // boxing

int j = (int)o; // unboxing

ya da

object a = new A();

A \_a = (A)a;

**Polimorfizm Durumlarında Tür Dönüşümleri - Devam**

- Polimorfizm durumlarında tür dönüşümünü gerçekleştirebilmek için Cast ya da as operatörleri kullanılır.

Cast Kullanımı

A-->B-->C-->D

- Üst türden alt türe kalıtımsal ilişkide dönüşüm sağlar.

A a = new C(); //A, C'nin atası olduğundan A referansı ile C nesnesini tutabiliriz.

C c = (C)a; //A referansının içerisindeki C nesnesini, C referansı ile tekrardan elde edebilmek için cast operatörünü kullanırız.

- Eğer kalıtımsal ilişki olmayan herhangi bir türe dönüştürülmeye çalışılırsa derleyici hatası verir.

- Yok eğer kalıtımsal ilişkide olup fiziksel nesnenin hiyerarşik altında olan bir türe dönüştürülmeye çalışılırsa runtime hatası verilecektir.

A a = new C(); //A referansının içinde C türünden bir nesne var.

D d = (D) a; //C türünden olan bir nesneyi alt sınıf olan D türünden bir referansla almaya çalıştığımızda derleme sürecinde hata vermez.

//derleyip çalıştırdığımızda A referansının içerisindeki nesne D değil ve işaretlenebilir bir referansta değil; System.InvalidCastException hatası verir.

"Unable to cast object of type 'polimorfizm.C' to type 'polimorfizm.D'

- Tersine olarak, kalıtımsal ilişkide alt türden üst türe cast operatörü ile de bir dönüşüm sağlamaktadır.

C c = new C();

A a = c;

**as Kullanımı:**

- Cast gibi kalıtımsal ilişki olan türler arasında referans dönüşümü yapabilmemizi sağlayan operatördür.

A a = new C();

C c = a as C; //a'nın içerisindeki nesneyi bana C türünde ver.

- Dönüşüm esnasında hiyerarşik olarak tüm türlere dönüşüm sağlar. Lakin kalıtımsal ilişki yoksa derleyici hatası verir.

- Ya da kalıtımsal ilişkide olup fiziksel nesnenin türünden daha alt hiyerarşide olan nesnelere dönüştürülmeye çalışıldığında

Polimorfizm mantığı gereği ilgili referans o nesneyi karşılayamacağından runtime hatası vermeyecek! geriye null dönecektir.

A a = new C() { X = "1", Y = "2", Z = "3" };

D c = a as D;

**is Operatörü**

- is operatörü kalıtımsal ilişkiye sahip nesnelerin Polimorfizm özelliğine nazaran fiziksel olarak hangi türde olduğunu veren bir operatördür.

A a = new C() { X = "1", Y = "2", Z = "3" };

CW(a is B); //True

CW(a is C); //True

CW(a is A); //True

CW(a is D); //False, D kalıtımsal ilişkide olsa da olmasa da False döner. Olsa, C'nin altı olacağından False döner. Olmazsa zaten False.

- Çok biçimlilik uygulanmış bir nesnenin ihtiyaç doğrultusunda(uygun olan) farklı bir türe dönüştürülebilmesi için işi garantiye almak adına

önce is kontrolü ardından Cast ya da as operasyonu sağlanması uygundur.

A a = new C();

if (a is D) //a, D midir?

{

D d = (D)a; //eğer D ise a'yı D'ye cast et. Runtime'da da hata vermez. Çünkü tetikleniyorsa zaten a bir D'dir. İçinde d referansı, D nesnesi barındırıyordur.

}

else if (a is C)

{

...

}

Örnek:

using System;

class Person //Person adlı base class tanımla ve içinde whoAmI() isim virtual metot oluştur.

{

public virtual void whoAmI()

{

Console.WriteLine("I'm a person.");

}

}

class Man : Person //Man isimli bir derived class oluştur, Person'dan kalıtım al.

{

public override void whoAmI() //whoAmI() override metodu tanımla.

{

Console.WriteLine("I'm a man.");

}

}

class Woman : Person //Woman isimli bir derived class oluştur, Person'dan kalıtım al.

{

public override void whoAmI() //whoAmI() override metodu tanımla.

{

Console.WriteLine("I'm a woman.");

}

}

public class TestPerson

{

static void Main()

{

Person a = new Man(); //Man türünde bir nesne oluştur ve Person türündeki a referansı ile işaretle.

a.whoAmI(); //Person sınıfındaki string yerine Man class'ındaki string'i output ver. Çünkü override ile ezdik.

Person b = new Woman(); //Aynı şekilde.

b.whoAmI();

Console.Read();

}

}

Çıktı:

I'm a man.

I'm a woman.

Örnek:

using System;

namespace MyFirstProgram

{

class Program

{

static void Main(string[] args)

{

//nesnelerimizi oluşturup referanslayalım.

Car car = new Car();

Bicycle bicycle = new Bicycle();

Boat boat = new Boat();

//Vehicle sınıfı türünde vehicles isimli bir array oluşturup, yukarıda oluşturduğumuz nesneleri elemanları olarak atayalım.

Vehicle[] vehicles = { car, bicycle, boat }; //bunu yazabilmek için Vehicle sınıfından kalıtım almaları şart!

//foreach ile array'i itere edelim. sırasıyla ilgili nesnenin sınıfına gidip ilgili metodu tetikleyecektir.

foreach (Vehicle vehicle in vehicles)

{

vehicle.Go();

}

Console.ReadKey();

}

}

class Vehicle

{

public virtual void Go()

{

//nasılsa override edeceğimizden yazmaya gerek yok.

}

}

class Car : Vehicle

{

public override void Go() //[0] index olan Car sınıfının Go() metodu tetiklenip aşağıdaki CW'i çıktı verecek.

{

Console.WriteLine("The car is moving!");

}

}

class Bicycle : Vehicle

{

public override void Go() //[1] index olan Bicycle sınıfının Go() metodu tetiklenip aşağıdaki CW'i çıktı verecek.

{

Console.WriteLine("The bicycle is moving!");

}

}

class Boat : Vehicle //[2] index olan Boat sınıfının Go() metodu tetiklenip aşağıdaki CW'i çıktı verecek.

{

public override void Go()

{

Console.WriteLine("The boat is moving!");

}

}

}

Çıktı:

The car is moving!

The bicycle is moving!

The boat is moving!

**DERS 10: Nesneler Arası İlişki Türleri**

**Association - Aggregation - Composition**

- Nesneler arasında terminolojik olarak nitelendirilebilir ilişki türleri mavcuttur. Bu ilişkiler; kalıtım, referans yahut soyutlama gibi durumların getisi olan mantıksal izahatlerdir.

is - a ilişkisi

has - a ilişkisi

can - do ilişkisi

nesneler kendi aralarında bu 3 şekilde somut ilişki kurabilmektedir.

**is - a ilişkisi**

- Tamamıyla kalıtımla alakalıdır. C#'da : operatörü ile gerçekleştirilir.

\_\_\_\_\_\_A\_\_\_\_\_\_\_ is a \_\_\_\_\_\_\_B\_\_\_\_\_\_

A bir B'dir.

class Araba

{

}

class Opel : Araba //Opel bir arabadır.

{

}

**has - a ilişkisi**

\_\_\_\_\_Opel\_\_\_\_\_ has a \_\_\_\_Motor\_\_\_\_

- Bir sınıfın başka bir sınıfın nesnesine dair sahiplik ifadesinde bulunan ilişkidir. Bir yandan kompozisyon/composition ilişkisi de denmektedir.

class Araba

{

}

class Opel : Araba // Opel bir arabadır. is a

{

Motor motor; // Opel'in bir Motor'u vardır. has a

}

class Motor

{

}

**can - do İlişkisi**

- Sonraki derslerimizde göreceğimiz interface yapılanmasının getirisi olan bir ilişki türüdür.

- can do ikişkisini anlayabilmek için interface bilmek gerekir.

- Kısaca interface; bir sınıfın imzasıdır. Yani bir sınıfın içerisinde olacak olan tüm member'ların şablonunu/arayüzünü oluşturduğumuz bir kontrattır. Herhangi bir interface'i uygulayan class o interface içerisinde tanımlanmış member imzalarını kendisinde oluşturmak zorundadır. Aksi taktirde compiler hata verecektir. İleride daha detaylı.

- Interface'ler içlerindeki member'ların imzalarını class'lara uygulattırdıklarından dolayı o interface'ler ilgili nesnelerin yapabilecekleri kabiliyetleri göstermektedir.

- Yani can do ilişkisi bir nesnenin davranışlarını/kabiliyetlerini belirtmektedir.

- Bu davranış/kabiliyetler interface içerisinde tanımlanmaktadır.

interface IAraba

{

void Gazla();

void Frenle();

}

class Opel : IAraba //demek ki bu interface'i uygulayan sınıflar gazlayabilir ya da frenleyebilir.

{

public void Gazla()

{ }

public void Frenle()

{ }

}

**Association Nedir?**

- Associationi sınıflar arasındaki bağlantının zayıf biçimine verilen addır.

- Bu bağlantı oldukça gevşektir. Yani, sınıflar kendi aralarında ilişkilidir lakin birbirlerinden de bağımsızdırlar.

- Parça bütün ilişkisi yoktur.

- Otobüsteki yolcular ile otobüs arasındaki ilişki Association'dur. Hepsi yanı yöne gitmektedir, lakin bir yolcu indirinde bu durum otobüsün ve diğer

yolcuların durumunu değiştirmez.

Aggregation ve Composition Nedir?

- Nesneleri birleştirip daha büyük bir nesne yapmaya verilen isimlerdir.

- Her ikisi de birleştirilmiş nesnelerden bütünsel nesneler yapma durumlarını ifade eder.

- Her ikisinde de Association'da olmayan parça-bütün ilişkisi söz konusudur.

- Her ikisinde de sahiplik ilişkisi(has - a) vardır.

İkisinin arasındaki fark ise;

- Sahip olunan nesnenin, sahip olan nesneden bağımsız bir şekilde var olabilmesi durumudur. (Aggregation)

Sahip olunan nesnenin, sahip olan nesneden bağımsız bir şekilde var olamaması durumudur. (Composition)

Örnek: Bir araba düşünelim. Bu arabanın tekeri ile vitesi arasındaki ilişki;

- Bu araba teker olmadan olmaz lakin teker araba olmadan da tek başına var olabilir. (Aggregation)

- Bu araba vites olmadan da olmaz, Ama vites araba olmadan bir anlam ifade etmez. (Composition)

Örnek: Bir kitap sayfalardan ve kapaktan meydana gelmektedir.

- Kitabın herhangi bir sayfasının yırtılması işlevselliğini kaybettirir. Lakin her sayfa kendi başına var olabilir. (Aggregation)

- Kapak ise kitapla bir anlam kazanır. Kitabın dışında bağımsız bir şekilde var olmasının izahı yoktur. (Composition)

Örnek: İnşa edilen duvarda; tuğlalar, çimento ve usta arası ilişki.

- Duvarla tuğla arasındaki ilişki. Duvar, tuğlalardan örülmektedir. Tuğlasız duvar olmaz. Fakat tuğla duvar örme dışında farklı noktalarda da kullanılabilir.

(Aggregation)

- Duvarla çimento arasındaki ilişki. Duvar, çimentosuz olmaz. Ama çimento tek başına bir işe yaramaz.(kabul edelim)

(Composition)

- Duvarla usta arasındaki ilişki. Duvar ile usta arasında parça - bütün ilişkisi yoktur. Duvarı herhangi bir usta örebilir. Haliyle duvar ile usta arasında Association ilişkisi vardır.

**DERS 11: Static Keyword**

- Erişim belirleyicisinden hemen sonra gelmesi tercih edilir.

Eğer metot static değilse bu şekilde kullanıyorduk:

namespace Ornek

{

class Program

{

public static void main (string[] args)

{

Musteri M1 = new Musteri();

M1.SelamVer();

}

}

public class Musteri

{

public void SelamVer()

{

}

}

}

Eğer metodumuz/değişkenimiz static olarak tanımlanmışsa yukarıdaki gibi o class'tan bir obje oluşturup referans örneği

üzerinden erişmek zorunda değiliz.

namespace Ornek

{

class Program

{

public static void main (string[] args)

{

Musteri.SelamVer(); //static ise class ismi üzerinden metoda erişebiliriz.

}

}

public class Musteri

{

public static void SelamVer()

{

}

}

}

**Static Metot Kullanımı**

- Console'da bir class'tır. Console. yazdığmızda WriteLine metodunu görebiliriz. Class ismiyle direkt olarak

erişebildiğimize göre WriteLine'da Console sınıfı içerisinde static olarak tanımlanmış bir metot'tur.

ReadLine'da aynı şekilde.

- Aynı şekilde aşağıdaki örnekte de static keyword'ü ile obje oluşturmadan direkt isimle erişebiliriz.

Örnek:

namespace OOPExamples

{

class Program

{

public static void Main(string[] args)

{

Personel.isimSoyisimYazdir();

Personel.sayilariTopla(10, 20);

}

}

class Personel

{

public static void isimSoyisimYazdir()

{

System.Console.WriteLine("Ali Veli");

}

public static void sayilariTopla(int sayi1, int sayi2)

{

Console.WriteLine($"İki sayinin toplamı {sayi1 + sayi2}.");

}

}

}

Çıktı:

Ali Veli

Iki sayinin toplami 30.

**Static Değişken Kullanımı**

Örnek:

namespace OOPExamples

{

class Program

{

public static void Main(string[] args)

{

Calisan C1 = new Calisan();

C1.calisanID = 1; //bu üç değişkene static değil.

C1.isim = "Ali"; //bu yüzden obje oluşturarak referans üzerinden erişip

C1.soyisim = "Veli"; //işlem yaptık.

Calisan.maas = 2500; //fakat aynı metotlarda olduğu gibi static değişkenlere de

//class ismi üzerinden erişebildik.

Console.WriteLine($" Çalışan ID: {C1.calisanID} \n İsim: {C1.isim} \n Soyisim {C1.soyisim} \n Maaş: {Calisan.maas}");

}

}

public class Calisan

{

public int calisanID;

public string isim;

public string soyisim;

public static int maas;

}

}

Çıktı:

Çalisan ID: 1

Isim: Ali

Soyisim Veli

Maas: 2500

- Aynı şekilde Calisan sınıfı içerisindeki değişkenleri Property olarak da tanımlayabilirdik.

public class Calisan

{

public int calisanID { get; set; }

public string isim { get; set; }

public string soyisim { get; set; }

public static int maas { get; set; }

}

Çıktı değişmeyecektir. Sadece C1. veya Calisan. dediğimizde değişken ikonu yerine ingiliz anahtarı

olan Property ikonu gelecektir.

**Static Constructor/Yapıcı Metot Kullanımı**

Static yapıcı metotlar:

- Erişim belirleyicisi kullanamaz. (!)

- Parametre alamaz. (!)

- Sınıfa ait tüm yapıcılardan önce çalışır.

- Kaç nesne örneği oluşturulursa oluşturulsun sadece 1 kere çalışır. (Önemli!)

- Bir sınıf sadece bir static yapıcı metot içerebilir.

- Static yapıcı metot ya ilk nesne örneği oluşturulduğunda ya da ilk static sınıf üyesi çağrılmadan

hemen önce yürütülür.

Örnek:

namespace OOPExamples

{

class Program

{

public static void Main(string[] args)

{

Kullanici K1 = new Kullanici(1, "Ali", "Bayram");

//class'tan bir nesne ürettiğimiz zaman ilk yapıcı metoda gider.

//Ve ilk olarak static yapıcı metot çalışır. Parametreli constructor'a gitmez.

//maas'a 2500'ü atar sonra diğer parametreli yapıcı metot'a girer ve işlemlerini yapar.

// Kaç nesne örneği oluşturulursa oluşturulsun sadece 1 kere çalışır. (Önemli!)

Kullanici K2 = new Kullanici(2, "Uğur", "Mahmut");

//bu sefer static metot çalışmayacaktır.

//ancak standart yapıcı metot her obje oluşturulduğunda çalışır.

K1.bilgileriGoster();

K1.zamYap(700);

K2.bilgileriGoster();

K2.zamYap(700);

}

}

public class Kullanici

{

private int kullaniciID { get; set; } //sadece bu class içerisinden erişilebilsin.

private string isim { get; set; }

private string soyisim { get; set; }

private static int maas { get; set; }

static Kullanici() //public erişim belirleyicisi yerine static yazarsak static constructor olur.

{

maas = 2500; //maas property'si static olmasaydı erişemezdik.

}

public Kullanici(int kullaniciid, string isim, string soyisim) //Kullanici sınıfının yapıcı metodu

{

kullaniciID = kullaniciid;

isim = isim;

soyisim = soyisim;

}

public void bilgileriGoster()

{

Console.WriteLine($" Kullanıcı ID: {kullaniciID} \n İsim: {isim} \n Soyisim {soyisim} \n Maaş: {maas}");

}

public void zamYap(int zamMiktari)

{

Console.WriteLine("Kullaniciya zam yapılıyor...");

Console.WriteLine("Şuanki maaşı: " + (maas + zamMiktari));

}

}

}

Çıktı:

Kullanici ID: 1

Isim:

Soyisim

Maas: 2500

Kullaniciya zam yapiliyor...

Suanki maasi: 3200

Kullanici ID: 2

Isim:

Soyisim

Maas: 2500

Kullaniciya zam yapiliyor...

Suanki maasi: 3200

**Static Class Kullanımı**

- Static class'lardan obje oluşturulamaz.

- Static class'ların içinde static olmayan bir şey tanımlanamaz.

- Static class'lardan kalıtım yapılamaz.

- Static class'lar diğer class'larda kullanılacak bir grup alakalı metot'ları gruplamak açısından faydalıdır. Static class'lara System.Math class'ı güzel bir örnektir.

namespace System -> Class ismi Math

Bize bir çok matematiksel metot sağlar.

double result = Math.Cos(angle); // Math class'ında bulunan Cos()metodu bize cosinus değerini verir.

Örnek:

namespace OOPExamples

{

class Program

{

public static void Main(string[] args)

{

/\*

Musteri M1 = new Musteri(); //hata verir.

\*/

}

}

public static class Musteri

{

/\*

public int id; //hata. static değil.

\*/

public static int id; //hata ortadan kalktı.

/\*

public void yazdir() //hata

{

}

\*/

public static void yazdir() //hata ortadan kalkar.

{

}

}

}

- Static class'lardan kalıtım yapılamaz.

public static class Musteri

{

public static void yazdir()

{

}

}

public class Personel : Musteri

{

//Error CS0709 'Personel': cannot derive from static class 'Musteri'

}

ya da

public static class Musteri

{

public static void yazdir()

{

}

}

public static class Personel : Musteri

{

//Error CS0713 Static class 'Personel' cannot derive from type 'Musteri'. Static classes must derive from object.

}

ya da

public class Musteri

{

public static void yazdir()

{

}

}

public static class Personel : Musteri

{

//yapılamıyor işte zorlamamak lazım.

}

**ArrayList Tekrar**

- Bir koleksiyon türüdür. İçerisinde yardımcı metotlar barındırır. İstediğimiz türde değişkeni tutmamızı sağlayan bir sınıftır.

Normal bir array tanımlarsak, (primitive-ilkel dizi)

public static void Main(string[] args)

{

string[] isimler = { "Enes", "Ahmet", "Yusuf", "Ercan" };

foreach (string a in isimler)

{

Console.WriteLine(a);

}

}

Bu şekilde elemanları ekranda görebiliriz.

Aynı örneği ArrayList ile yaparsak;

using System.Collections; //kütüphanesini eklemek şart.

public static void Main(string[] args)

{

ArrayList koleksiyon = new ArrayList();

//ArrayList sınıf olduğu ve içerisinde hazır metotlar barındırdığı için çok daha verimli ve hızlı bir kullanımdır.

//koleksiyon. dediğimiz an bir sürü metot'a erişim sağlayabildiğimizi görürüz.

koleksiyon.Add("Enes"); //değer eklemek için Add

koleksiyon.Add("Ali");

koleksiyon.Add("Yusuf");

koleksiyon.Add("Mehmet");

koleksiyon.Remove("Yusuf"); //değer kaldırmak için Remove

foreach (var a in koleksiyon)

{

Console.WriteLine(a);

}

}

Aynı çıktıyı alırız.

Örnek: Metot kullanımları.

namespace OOPExamples

{

class Program

{

public static void Main(string[] args)

{

ArrayList koleksiyon = new ArrayList();

//ArrayList sınıf olduğu ve içerisinde hazır metotlar barındırdığı için çok daha verimli ve hızlı bir kullanımdır.

//koleksiyon. dediğimiz an bir sürü metot'a erişim sağlayabildiğimizi görürüz.

koleksiyon.Add("Enes"); //koleksiyonun sonuna değer eklemek için Add

koleksiyon.Add("Ali");

koleksiyon.Add("Yusuf");

koleksiyon.Add("Mehmet");

//koleksiyon.Remove("Yusuf"); //herhangi bir değer kaldırmak için Remove

//koleksiyon.Clear(); //dizinin tüm elemanlarını sil

Console.WriteLine("Var mı? " + koleksiyon.Contains("Yusuf")); //Mehmet diye bir değer var mı? True.

Console.WriteLine("Var mı? " + koleksiyon.Contains("Yasin")); //Yasin yok. False.

Console.WriteLine("Kişinin index no: " + koleksiyon.IndexOf("Ali")); //Ali kaçıncı index'te?

koleksiyon.Insert(3, "Sercan"); //3. index'e Sercan'ı ekle.

Console.WriteLine();

koleksiyon.Reverse(); //diziyi ters çevirir.

foreach (var a in koleksiyon)

{

Console.WriteLine(a);

}

Console.WriteLine($"Dizinin eleman sayısı: {koleksiyon.Count}"); //Count property'si ile eleman sayısını..

}

}

}

Örnek:

public static void Main(string[] args)

{

ArrayList rakamlar = new ArrayList();

rakamlar.Add(4);

rakamlar.Add(2);

rakamlar.Add(66);

rakamlar.Add(6);

rakamlar.Add(41);

rakamlar.Sort(); // diziyi küçükten büyüğe sıralar. 2-4-6-41-66 yazılır alt alta.

foreach(int a in rakamlar)

{

Console.WriteLine(a);

}

}

Örnek: Static Fields Örneği

namespace StaticFields

{

public class Student //Student adında public bir class oluşturalım.

{

public string name;

public int score;

}

public class MyClass //MyClass adında bir public class oluşturalım.

{

const int MAX = 50; //MAX adlı integer constant bir değer oluşturalım.

public static int students; //student isimli, sınıftaki öğrenci sayısını tutacak bir static integer field tanımlayalım.

public static void Main() //Main metodunda sadece class içerisinde tanımlanabilen bir static metot olduğuna dikkar edelim.

{

Student s1 = new Student(); //Student'tan bir nesne oluştur.

s1.name = "Tom"; //değerleri ata

s1.score = 90;

MyClass.students++; //static students değerini bir arttır. Class ismiyle eriştik dikkat.

Student s2 = new Student(); //Student'tan bir başka nesne oluştur.

s2.name = "Peter";

s2.score = 80;

students++; //üstteki ile aynı ifade çünkü students aynı sınıfta olduğundan MyClass'ı yazmamıza gerek yok.

Student s3 = new Student();

s3.name = "Sam";

s3.score = 70;

StaticFields.MyClass.students++; //yine aynı ifade ama böyle bir kullanım da var.

Console.WriteLine("Students in the class: {0}", students); //çıktılarımızı alalım.

Console.WriteLine("Maximum students allowed: {0}", MyClass.MAX); //const özünde bir static'tir. Yani class ismiyle de çağrılabilir.

Console.Read();

}

}

}

Çıktı:

Students in the class: 3

Maximum students allowed: 50

Örnek:

namespace StaticMethods

{

public class Circle //cirle isimli bir public class tanımla

{

const double PI = 3.14; //Constant double PI sayısı

public double radius;

public static void Main()

{

Circle c = new Circle(); //Circle'dan nesne oluştur.

c.radius = 1.0; //yarıçapı 1.0 yap.

Console.WriteLine("Non-Static Method Area: {0}", c.getArea()); //nesne üzerinden çağırabiliriz.

Console.WriteLine("Static Method Area: {0}", getArea(10.0)); //aynı class'ta olduğundan direkt çğaırabiliriz.

Console.Read();

}

public static double getArea(double r)

{

//return getArea(); //hata verir static olmayan bir metot nesne referanssız çağrılamaz.

return PI \* r \* r;

}

public double getArea() //statik olmayan metot

{

//return getArea(this.radius); //burada hata vermez. Aynı class'ta iken static metot çağrılabilir. Aynı çıktı alınır. Ama alttaki satıra erişemeyiz. Tek bir return olabilir.

return PI \* radius \* radius;

}

}

}

Çıktı:

Non-Static Method Area: 3.14

Static Method Area: 314

Örnek: Static Class'lar

namespace StaticClass

{

public static class Area //Alan isimli static bir class tanımladık. static bir class sadece static member'lara sahip olabilir.

{

const double PI = 3.1416; //const bir static'dir.

public static double Rectangle(double length, double width) //static Rectangle metodu ile dikdörtgenin alanını hesaplayan bir metot.

{

return length \* width;

}

public static double Triangle(double baseline, double height) //aynısı üçgen için,

{

return baseline \* height / 2.0;

}

public static double Square(double side) //kare için,

{

return Rectangle(side, side);

}

public static double Circle(double radius) //daire için.

{

return PI \* radius \* radius;

}

}

public class TestArea

{

public static void Main()

{

Console.WriteLine("Rectangle: length=10 width=10 Area={0}", Area.Rectangle(10, 10));

Console.WriteLine("Triangle: base=10 height=10 Area={0}", Area.Triangle(10, 10));

Console.WriteLine("Square: side=10 Area={0}", Area.Square(10));

Console.WriteLine("Circle: radius=10 Area={0}", Area.Circle(10));

Console.Read();

}

}

}

Çıktı:

Rectangle: length=10 width=10 Area=100

Triangle: base=10 height=10 Area=50

Square: side=10 Area=100

Circle: radius=10 Area=314,16

Örnek:

using System;

namespace StaticConstructor

{

public class A

{

static A()

{

Console.WriteLine("Static constructor");

}

public A()

{

Console.WriteLine("Instance constructor");

}

}

public class TestA

{

public static void Main()

{

Console.WriteLine("Before creating a1");

A a1 = new A();

Console.WriteLine("After creating a1");

A a2 = new A();

Console.WriteLine("After creating a2");

Console.Read();

}

}

}

Çıktı:

Before creating a1

Static constructor

Instance constructor

After creating a1

Instance constructor

After creating a2

**DERS 12: Access Modifiers - Erişim Belirleyicileri**

**Public ve Private Kardeşler (Naruto ile Sasuke)**

namespace AccessModifiers

{

public class Musteri

{

public string isim; //public hiçbir kısıtlaması yok. Her yerden erişilebilip kullanılabilir.

public string soyisim;

public int maasDegeri;

public string cinsiyet;

private int yas;

public void musteriBilgileriGoster()

{

Console.WriteLine($" İsmi: {isim} \n Soyisim: {soyisim} \n Maaşı: {maasDegeri} \n Cinsiyet: {cinsiyet} \n Yaşı: {yas}");

}

private void IsimSoyisimYazdir(string musteriisim, string musterisoyisim)

{

Console.WriteLine($"Musterinin ismi {musteriisim} ve soyismi {musterisoyisim}");

}

public Musteri()

{

Console.WriteLine("Müşteri sınıfının yapıcı metodu tetiklendi.");

yas++; //class içerisinden erşililebilir.

IsimSoyisimYazdir("Ali", "Muhtar"); //class içerisinden erşililebilir.

}

}

public class Program

{

public static void Main()

{

Musteri musteri1 = new Musteri();

musteri1.isim = "Enes";

musteri1.soyisim = "Bayram";

musteri1.maasDegeri = 2500;

musteri1.musteriBilgileriGoster(); //private metot değil erişilebilir.

//musteri1.yas = 21; //Hata. private olduğundan başka class'tan erişilemez. Değişken, field ya da metot; private ise dış dünyaya kapalıdır.

//Sadece bulundukları class {} scope'undan erişilebilirler.

}

}

}

**Internal Kullanımı**

Örnek:

namespace AccessModifiers

{

class Kisi //class'ların default'u ise internal'dır.

{

//string Ad; //diğer yazımı private Ad;

internal string Ad; //aynı assembly'de yani dll ve exe dosyasındaki tüm her yerde ulaşılabilir demektir. Başka assembly'den ulaşılamaz.

//başka bir projede ise private olarak davranır.

}

public class Program

{

public static void Main()

{

Kisi kisi = new Kisi();

kisi.Ad = "Ali";

Console.WriteLine(kisi.Ad);

}

}

}

**Protected ve Protected Internal**

Örnek: Protected

namespace AccessModifiers

{

class A : B

{

public void yaz()

{

//buradan x'e ulaşabiliriz.

//protected sadece türetilmiş class'larda kullanılabilir.

//burada A, B'den türediği için kullanabiliriz.

x = 5;

}

}

class B

{

protected int x;

}

public class Program

{

public static void Main()

{

//buradan x'e ulaşamayız.

A nesne = new A();

nesne.x = 8; //HATA

B nesne2 = new B();

nesne2.x = 5;//hata

} }}

**Protected Internal**

//öncelikle protected erişim belirleyicisi taşıdığı için derived class'larda kullanılabilir.

//Aynı solution içerisinde fakat farklı bir proje ya da Namespace (isim alanı) içerisinde olan bir sınıf başka bir sınıfı miras almış

//(inheritance) ve ilgili alan Protected Internal ile tanımlanmış ise bu alana ulaşabiliriz.

- Protected internal olarak tanımlanmış değer, tanımlandığı class’ın içinden ve ondan türetilen sınıfların içinden erişilebilir durumdadır.

Türetilen sınıfın aynı program içinde olmaması sorun teşkil etmez.

Örnek:

using System;

namespace Tutlane

{

class User

{

protected internal string Name; //aynı assembly içerisindeki herhangi bir yerden ya da

protected internal string Location; //başka bir assembly'deki herhangi bir türetilmiş class'tan erişilebilir.

protected internal int Age;

protected internal void GetUserDetails()

{

Console.WriteLine("Name: {0}", Name);

Console.WriteLine("Location: {0}", Location);

Console.WriteLine("Age: {0}", Age);

}

}

class Program

{

static void Main(string[] args)

{

User u = new User(); //User class'tan üretilmiş nesneler üzerinden elemanlara erişebiliyoruz.

u.Name = "Suresh Dasari";

u.Name = "Suresh Dasari";

u.Location = "Hyderabad";

u.Age = 32;

u.GetUserDetails();

Console.WriteLine("\nPress Enter Key to Exit..");

Console.ReadLine();

}

}

}

**Private Protected**

- Private Protected’a gelirsek eğer; bu erişim belirleyicisiyle işaretlenen elemanlar, protected oldukları için ilgili sınıftan türeyen

sınıflardan erişilebilirken, private oldukları için sade ve sadece aynı assembly(namespace) içerisinden erişilebilecektirler.

Yani private burada farklı bir namespaceler arasındaki protected elemanlara olan erişimi engellemektedir.

- farklı bir Class Library’de bulunan sınıf içerisindeki Private Protected erişim belirleyicili elemanımız o sınıftan kalıtım alan

sınıfımız içerisinde Private özellik göstermiş bulunmaktadır. Halbuki Class Library içerisindeki farklı bir sınıf içinde bu işlemi

yapsak aynı assembly içerisinde olduklarından dolayı ilgili elemana protected olduğundan dolayı erişilebilecektir.

**Struct Yapısı**

struct MyStruct //değer tiplidir. Stack'te bulunur.

{

//şeklinde tanımlanır.

//16 byte'ın altındaki verilerde kullanılır.

//veriler net olmalıdır.

//default constructor yoktur.

public MyStruct

{ // hata verir }

//ama parametreli constructor tanımlanabilir.

public int y;

public MyStruct (int x)

{ // hata ortadan kalkar.

y = x;

}

~MyStruct() //Hata. Destructor bir nesne Heap'ten silinmeden önce çalışan metot'dur.

//Struct'un Heap ile alakası yok.

public int a; //struct'ta değişken oluşturabiliriz,

public static int b;

public int x = 2; //hata alırız. Ama değer veremeyiz.

}

class MyClass //referans tiplidir. Heap'tedir.

{

//16 byte'ın üstündeki verilerde kullanılır.

//field, metot, property'lerin hepsinin kullanımı Struct ile benzerdir.

public int x = 0; //Class'ta değişken tanımlayabilir ve direkt değer verebiliriz.

}

- Stack'te büyük veriler taşınamadığı için (overflow hatası) Heap var olmuştur.

Main'den ulaşmak için;

static void Main()

{

//class'ta nesne oluşturup değişkene ulaşıp değiştirip yazdırdık.

MyClass nesne = new Myclass();

nesne.x = 10; //class'ta değer ataması yapmasaydık default olarak 0 atardı.

CW(nesne.x);

//struct'ta direkt isim verip çağırabiliriz.

MyStruct isim;

isim.a = 15; //eğer ki burada değer ataması yapmasaydık hata alırdık.

CW(isim.a);

//değer vermeden çağırma yapmak istiyorsak

CW(MyStruct.b); //MyStruct üzerinden çağırabiliriz.

//b static bir değişken olduğundan ilk varsayılan değerini alır.

//ve ekrana 0 yazılır.

//static yapmadan çözüm.

MyStruct isim = new MyStruct();

CW(isim.a); //0 yazdığını görürüz.

ya da

MyStruct isim2 = new MyStruct(10); // ile parametreli Struct constructor'una değer gönderip işlem yapabilir ve

CW(isim2.y); //10 yazdığını görebiliriz.}

Örnek:

namespace StructYapisi

{

class Program

{

static void Main()

{

MyClass nesne = new MyClass();

nesne.x = 50; //yapmamıza rağmen ekrana 30 yazdırır.

Degistir(nesne);

CW(nesne.x); //referans tipli olduğu için,

}

static void Degistir(MyClass a) //buraya geldiğinde direkt orjinali gelecektir.

{

a.x = 30; //ve burada 30'a eşitlenecektir.

}

}

class MyClass

{

public int x;

}

}

fakat Struct olsaydı,

namespace StructYapisi

{

class Program

{

static void Main()

{

MyStruct isim = new MyStruct(); //Struct'lar değer tipli olduğu için,

isim.b = 50; //orjinali 50 olduğu için,

Degistir(isim); //değer kopyalanacaktır.\*

CW(isim.b); //ekrana 50 yazdırılacaktır.

}

static void Degistir(MyStruct a)

{

a.b = 30; //\*kopya 30 olacaktır.

}

}

struct MyStruct

{

public int b;

}

}

Örnek:

using System;

struct Circle //struct Circle tanımla.

{

public double radius; //radius ilk değerini veremeyiz.

const double PI = 3.14; //fakat const ve static field'lara ilk değer atanabilir.

public void area() //alanın çıktısını hesapla ve output ver

{

Console.WriteLine("Area: {0}", radius \* radius \* PI);

}

}

class Program

{

static void Main()

{

Circle c; //new operatörü olmadan Circle c oluşturabiliriz.

c.radius = 10; //c'yi kullanarak radius'a değer atadık.

c.area(); //c nesnesinin tüm field'larına ilk değer atandığı için metodu çağırabiliriz.

Console.Read();

}

}

Çıktı:

Area : 314

Not: Struct, protected yapılarla kullanılamaz çünkü struct member'lar kalıtım'da kullanamaz. (aynı şekilde private protected ve protected internal)

Not: C#'ta Struct member'lar default olarak private'dır.

**DERS 13: Abstract Class ve Interfaces**

- **Abstract sınıf** içerisinde abstract ile işaretlenen metotlar veya propertyler, bu sınıftan kalıtım alan her sınıfta uygulanmak zorundadır.

- Abstract metot ya da propertylerin gövdeleri tanımlandıkları class içinde yazılmazlar.

- Sadece imza dediğimiz ger dönüş tipleri, isimleri ve geri dönüş belirleyicileri tanımlanır.

- Gövdeleri kalıtım alan sınıfta override edilerek yazılır.

- Bir Abstract class içerisinde Abstract olmayan elemanlar da olabilir. Ama içerisinde Abstract elemanlar olan bir class Abstract olmak zorundadır.

- Abstract elemanlar private olamazlar.

- Abstract class'lardan nesne yaratılmaz. Ama stack'teki referans noktası alınabilir.

- Field'lar abstract olamaz.

Örnek:

namespace AbstractClass

{

class Program

{

static void Main()

{

}

}

//

abstract class MyClass

{

public abstract void X(); //gövde yok sadece imza.

abstract public int MyProperty { get; set; } //gövde yok.

/\*

\* abstract olmayan elemanların kullanımı

public void X()

{

}

public int MyProperty { get; set; }

private int myVar;

public int MyProperty2

{

get { return myVar; }

set { myVar = value; }

}

\*/

}

}

Örnek:

- Ornek Abstract sınıfı bütün derived sınıflara (kendinden kalıtım alan bütün sınıflara) bir imza niteliği tşaıyan bir sınıftır.

namespace AbstractClass

{

class Program

{

static void Main()

{

// Ornek ornek = new Ornek(); hata verir. Abstact ile Interface'lerden nesne üretilmez.

Ornek ornek = new Calisma(); //Polimorfizm. ornek referansıyla Calisma nesnesi tuttuk.

//Calisma, Ornek'ten miras alıyor. Yani her Calisma bir Ornek'tir. Ama her Ornek bir Calisma değildir.

//Her Calisma bir Ornek ise eğer her Ornek referansı bir çalışmayı referans edebilir.

//Abstract referans noktası üzerinden Abstract class'ın dervied class'larından herhangi birisinin nesnesi üzerinden

//çalışmamızı gerçekleştirebiliyoruz.

//ornek. dediğimizde elemanlara X, Y, Z (govdeleri Calisma'da bulunduğu için) Calisma nesnesi üzerinden erişebilir ve çalıştırabiliriz. (override ile)

//W elemanı da kalıtım ile geleceğinden aynı nesne üzerinden çalıştırılacaktır. (override'sız)

}

}

//Nesne Oluşturulmama Sebebi: Eğer ki Abstract sınıflardan nesne oluşturulsaydı, kullanım amacı genişlerdi ama tek bir //amaca dönük bir yapı olmazdı.

//Abstract class'lar içerisinde Abstract olarak işaretlenmiş yapılar bu sınıftan kalıtım alan yapılara zorunlu uygulatılacağından

//dolayı Abstract sınıflardan nesne oluşturulması engellenmiştir.

//Abstract sınıflar başka sınıflara kalıtım vereceklerdir ve buradaki imzaları o sınıflar kullanacaktır.

abstract class Ornek

{

abstract public void X(); //public abstract da olabilir.

abstract public int MyProperty { get; set; }

abstract public bool Z(); //sonradan Z() ekleyelim.\*

public void W() { } //normal metot'ta bir sorun yok. Zorunlu uygulanması şart değil.

}

class Calisma : Ornek //Ornek sınıfı içerisindeki member'ları kullanmazsak hata alırız.

{

//Ctrl + . 'ya basıp Implement Abstract Class dersek;

public override void X()

{

throw new NotImplementedException();

}

public override int MyProperty { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

public override bool Z() //\*öneri ile otomatik tamamlayacaktır.

{

throw new NotImplementedException();

}

}

}

Örnek:

abstract class Animal

{

public abstract void animalSound(); // Gövdesi olmayan bir Abstract Metot

public void sleep() // Normal metot

{

Console.WriteLine("Zzz");

}

}

class Pig : Animal //Animal Abstracy Class'ından kalıtım alan Pig derived class'ı

{

public override void animalSound() // animalSound() fonksiyonunu override ile ezdik.

{

Console.WriteLine("The pig says: wee wee");// animalSound() gövdesini burada doldurduk.

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Pig nesnesi oluşturduk.

myPig.animalSound(); // Override edilmiş Abstract Metodunu çağırdık.

myPig.sleep(); // Normal metodu çağırdık.

//ya da polimorfizm kullanarak aşağıdaki gibi yapabilirdik.

/\*

Animal myAnimal = new Pig(); // Pig nesnesini oluşturup base class olan Animal türünde myAnimal referansı ile işaretledik.

myAnimal.animalSound(); // Override edilmiş Abstract Metodunu çağırdık.

myAnimal.sleep(); // Normal metodu çağırdık.

\*/

}

}

Çıktı:

The pig says: wee wee

Zzz

Örnek:

public abstract class A //boş bir abstract class tanımlayabiliriz.

{

}

public abstract class B //önerilmese de içinde abstract member barındırmayan bir Abstract class da tanımlayabiliriz.

{

public int i;

public void method1()

{

Console.WriteLine("method1");

}

}

public abstract class C //B class'ı ile abstract method2() dışında aynı bir C abstract class tanımladık.

{

public int i;

public void method1()

{

Console.WriteLine("method1");

}

public abstract void method2(); //gövde yok

}

public abstract class D : C //C class'ından kalıtım alan D class'ı tanımladık ve içerisine abstract method3()'ü ekledik.

{

public abstract void method3(); //gövde yok

}

public class E : D //bu seferde D'den kalıtım alan E normal class'ı tanımladık. Ve method2 ile method3'ü override ettik. Ve gövdelerini yazdık.

{

public override void method2() //method2 ile method3'ü implement etmiş olduk.

{

Console.WriteLine("method2");

}

public override void method3()

{

Console.WriteLine("method3");

}

}

public class TestAbstract

{

static void Main()

{

E e = new E(); //E'den nesne oluşturduk. ve sırasıyla normal method1'i ve abstract override metotlar olan method2 ve method3'ü çağırdık.

e.method1();

e.method2();

e.method3();

//B b = new B(); //hata verir. Abstract class B'den nesne üretilemez.

Console.Read();

}

}

**Interface - Arayüz Yapısı**

- Interface kendisinden kalıtım alacak derived class'lara (o class'lar içerisinde bulunacak elemanlara) kendi içerisinde

o elemanların imzasını barındıran bir yapıdır. O elemanlara bir şablon bir tasarım, arayüz, arabirim özelliği gösterir.

- Nesne oluşturulamaz! Doğal olarak constructor yapısı da yoktur.

- Ama referans noktası alabiliyoruz.

namespace AbstractClass

{

class Program

{

static void Main()

{

}

}

class Ornek : IOrnek, IOrnek2

{

//bir sınıfın birden fazla interface'den kalıtım alması mümkündür.

}

interface IOrnek //Interface yapılarında isimlendirmede başta I ile kullanmak tercih edilir. Zorunluluk değildir. Visual Studio'da hazır interface'ler de I ile başlar.

{

//bireysel çalışmalardan ziyade toplu ve design patternlarda İmza-Şablon özelliğinden dolayı kullanılır.

}

interface IOrnek2

{

}

}

- Abstract class'lar içerisinde kendisinden kalıtım alan sınıfta yazılması zorunlu olmayacak yapılar da tanımlayabilirdik.

Fakat, interface de sadece imza özelliği olduğundan dolayı kendisinde herhangi bir yapının gövdesi tanımlanamaz.

- Interface içerisinde imzası tanımlanmış yapıların gövdeleri interface'den kalıtım alan class'lar oluşturulur.

- Interface içerisinde static yapı olamaz.

- imzalarda public, private, protected cs. gibi erişim belirleyicileri olmaz.

Örnek:

namespace InterfaceOrnek

{

class Program

{

static void Main()

{

}

}

class Ornek : IOrnek

{

int OrnekProperty { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

int IOrnek.OrnekProperty { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

//ampulden implement interface diyelim.

public int X()

{

throw new NotImplementedException();

}

public void Y()

{

throw new NotImplementedException();

}

}

interface IOrnek

{

int X();

void Y();

int OrnekProperty { get; set; }

}

}

**Name Hiding Durumu - Implement interface'de aynı isimde metot kullanırsak;**

- Implement interface uygulanan elemanlara ilgili sınıfın nesnesi üzerinden erişebilirken diğer durumda erişemiyoruz.

Örnek:

namespace InterfaceOrnek

{

class Program

{

static void Main()

{

}

}

class Ornek : IA, IB

{

public void m() //Implement interface ile public erişim belirleyicisi ile gelir.

{

throw new NotImplementedException();

}

public void n()

{

throw new NotImplementedException();

}

public void x() //ortak olmayan memberların hepsi fakat ortak olanın sadece biri gelmiş.

{ //bu x()'in hangisinden geldiğini bilmediğimiz için Name Hiding dediğimiz istenmeyen durumla karşılaşırız.

throw new NotImplementedException();

}

public void y()

{

throw new NotImplementedException();

}

public void z()

{

throw new NotImplementedException();

}

}

interface IA

{

void x();

void y();

void z();

}

interface IB

{

void x();

void m();

void n();

}

}

Bunu çözmek için, ampüle tıklayıp Implement interface explicitly'i seçmemiz gerekli.

namespace InterfaceOrnek

{

class Program

{

static void Main()

{

Ornek o = new Ornek();

//o.yazdığımızda Implement interface ile memberlara nesne referansı üzerinden erişebiliriz.

//Implement interface explicitly ile Name Hiding'den kurtuluruz ama memberlara erişemeyiz.

IA a = new Ornek(); //polimorfizm ile bu sorun çözülebilir.

//a. yazarsak ilgili memberlara erişebiliriz.

IB b = new Ornek();

//b. yazarsak ilgili memberlara erişebiliriz.

}

}

class Ornek : IA, IB

{

void IB.m() //bu yöntemde ise private şeklinde gelir.

{

throw new NotImplementedException();

}

void IB.n()

{

throw new NotImplementedException();

}

void IA.x() //artık hangisinden

{

throw new NotImplementedException();

}

void IB.x() //geldiğini görebiliyoruz.

{

throw new NotImplementedException();

}

void IA.y()

{

throw new NotImplementedException();

}

void IA.z()

{

throw new NotImplementedException();

}

}

interface IA

{

void x();

void y();

void z();

}

interface IB

{

void x();

void m();

void n();

}

}

Örnek: Sınıf ve Interface'lerin Beraber Kalıtım Alma Durumu

namespace InterfaceOrnek

{

class Program

{

static void Main()

{

}

}

class C //eğer ki bir sınıftan da kalıtım alsın isteniyorsa,

{

...

}

class Ornek : C, IA, IB //önce o sınıf yazılır ardından virgülle gelenlerin mecburen interface olması gerekir.

{

...

}

interface IA

{

...

}

interface IB

{

...

}

}

- C# 8.0 ile interface içerisinde tanımlanan imzalarda artık member'lar Access Modifiers alabilmekte ve gövde oluşturulabilmektedir. Ayrıca static member'lar da kullanılabilmektedir.

Örnek:

interface IAnimal

{

void animalSound(); // interface method (gövdesi yok)

}

class Pig : IAnimal // Pig class'ı IAnimal interface'ini "implement" ediyor. Interface'lerde bu ifadeyi kullanıyoruz.

{

public void animalSound()

{

// animalSound() metodunun gövdesi burada yazılır.

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Pig nesnesi oluşturulur ve metot çağrılır.

myPig.animalSound();

}

}

Çıktı:

The pig says: wee wee

Örnek: Çoklu Interface Implementasyonu

interface IFirstInterface

{

void myMethod(); // interface metodu

}

interface ISecondInterface

{

void myOtherMethod(); // interface metodu

}

// birden fazla Interface'i implement ettik.

class DemoClass : IFirstInterface, ISecondInterface

{

public void myMethod()

{

Console.WriteLine("Some text..");

}

public void myOtherMethod()

{

Console.WriteLine("Some other text...");

}

}

class Program

{

static void Main(string[] args)

{

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

Çıktı:

Some text..

Some other text...

Örnek:

using System;

using System.Security.Claims;

interface A //method1 imzasıyla bir A interface'i tanımladık.

{

void method1();

}

interface B : A //A interface'ini implement eden ve içinde method2 bulunan bir B interface'i tanımladık.

{

void method2();

}

public abstract class C : B //C isimli interface B'yi implement eden bir abstract class oluşturduk.

{

public abstract void method1(); //public abstract method1 ile interface member olan method1'i realize ettik. (public şart)

public virtual void method2() // public virtual method2() ile method2'yi realize ettik.

{

Console.WriteLine("Method2 in class C"); //ve gövdesini doldurduk çünkü abstract değil.

}

}

public class D : C //C class'ından kalıtım alan D sınıfı oluşturduk.

{

public override void method1()

{

Console.WriteLine("Method1 in class D"); //override ile method1'i ezdik ve gövdesini doldurduk.

}

public override void method2() //eğer ki override demezsek Class C'deki virtual metoduna gider.

{

Console.WriteLine("Method2 in class D"); //override ile method1'i ezdik ve gövdesini doldurduk.

}

}

public class TestInterface

{

static void Main()

{

B b = new D(); //B interface'inin referansı b ile D nesnesi oluşturup işaretledik.

b.method1();//public override void method1() e gitti yazdırdı.

b.method2();//public override void method2() e gitti yazdırdı.

Console.Read();

}

}

**DERS 14: Enums**

Örnek:

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Person person = new Person();

person.school = "Primary School";

if(person.school.Equals("Primary School"))

{

Console.WriteLine("XYZ");

}

else if(person.school.Equals("High School"))

{

Console.WriteLine("ABC");

}

else if (person.school.Equals("University"))

{

Console.WriteLine("123");

}

}

}

class Person

{

public string school;

}

}

Çıktı:

XYZ

Kodu değiştirip, class'a default constructor eklersek;

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Person person = new Person();

if(person.school.Equals("Primary School"))

{

Console.WriteLine("XYZ");

}

else if(person.school.Equals("High School"))

{

Console.WriteLine("ABC");

}

else if (person.school.Equals("University"))

{

Console.WriteLine("123");

}

}

}

class Person

{

public Person() //Default Constructor oluşturalım.

{

school = "High School"; //nesne oluşturulduğunda ilk bunu çalıştıracak.

}

public string school;

}

}

Çıktı:

ABC

Kodun okunabilirliğini arttırmak adına rakamlar ile çalışırsak;

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Person person = new Person();

person.school = 2;

if(person.school.Equals(0))

{

Console.WriteLine("XYZ");

}

else if(person.school.Equals(1))

{

Console.WriteLine("ABC");

}

else if (person.school.Equals(2))

{

Console.WriteLine("123");

}

}

}

class Person

{

public Person()

{

school = 0;

}

public int school;

}

}

Çıktı:

123

fakat burada şöyle bir sorun çıktı, artık neyin ne olduğunu bilmiyoruz.

Çözüm olarak string ve sayıları ilişkilendirebilen bir yapıya ihtiyacımız var. Burada Enum'lar devreye giriyor.

Sayıları anlamlı şekilde isimlendirerek bizim daha efektif kullanmamızı sağlar.

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Person person = new Person();

person.school = Schools.PrimarySchool;

if(person.school.Equals(Schools.PrimarySchool))

{

Console.WriteLine("XYZ");

}

else if(person.school.Equals(Schools.HighSchool))

{

Console.WriteLine("ABC");

}

else if (person.school.Equals(Schools.University))

{

Console.WriteLine("123");

}

}

}

enum Schools

{

//PrimarySchool,HighSchool,University

//default başlangıç deperleri 0-1-2

//PrimarySchool=1,HighSchool,University

//dediğimiz anda 1-2-3 olacaktır.

//hepsine farklı değer de verebiliriz.

//PrimarySchool=1,HighSchool=15,University=25

PrimarySchool = 1, HighSchool = 2, University = 3

}

class Person

{

public Person()

{

school = Schools.HighSchool; //Schools. dediğimiz anda üç elemanda gelir. Burada varsayılan değerimizi default constructor ile HighSchool seçmiş olduk.

}

public Schools school; //enum Schools türünden değişken oluşturduk.

}

}

Çıktı:

XYZ

Hem daha okunabilir hem daha şık oldu.

Peki PrimarySchool = 1, HighSchool = 2, University = 3'deki int değerlerini nasıl elde ederiz?

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Console.WriteLine((int)Schools.PrimarySchool); //casting yaparak enum'un değerini alabiliriz. Ekrana 1 yazdırır.

Console.WriteLine((int)Schools.PrimarySchool == 1); //true yazacaktır. if karşılaştırmalarında kullanabiliriz.

}

}

enum Schools

{

PrimarySchool = 1, HighSchool = 2, University = 3

}

}

Çeşitlendirirsek;

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Console.WriteLine((int)Schools.HighSchool); // ekrana 2 yazdırır.

}

}

enum Schools

{

PrimarySchool = 1, HighSchool, University

}

}

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Console.WriteLine((int)Schools.HighSchool); // ekrana 42 yazdırır.

}

}

enum Schools

{

PrimarySchool = 4, HighSchool = 42, University = 555

}

}

Enum. dediğimiz an birçok metoda erişebiliriz.

Enum metotları type ile kullanılır. Yani direkt enum adını yazamıyoruz. Bunu da typeof ile veriyoruz. Type of tipini döndüren bir operatördür.

Console.WriteLine(typeof(int)); //ekrana System.Int32 yazdırır.

Console.WriteLine(typeof(Program)); //class adımız ile kullanırsak MyNamespace.Program yazdırır.

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Console.WriteLine(Enum.Parse(typeof(Schools),"PrimarySchool"));

//Console.WriteLine(Enum.Parse(typeof(Schools),"5")); aynı çıktıyı alırız. Ekrana PrimarySchool yazar.

//Console.WriteLine(Enum.Parse(typeof(Schools), "PrimaryxSchool")); hata verir.

//Console.WriteLine(Enum.Parse(typeof(Schools), "18")); //olmayan bir index yazarsak (string formatta) ekrana 18 yazar.

}

}

enum Schools

{

PrimarySchool = 4, HighSchool = 42, University = 555

}

}

Bir diğer metodumuz GetName()

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

Console.WriteLine(Enum.GetName(typeof(Schools),4)); //Ekrana PrimarySchool yazdırır.

}

}

enum Schools

{

PrimarySchool = 4, HighSchool = 42, University = 555

}

}

Tüm değerleri almak için;

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

string[] schools = Enum.GetNames(typeof(Schools)); //bir string dizisi oluşturup GetNames ile isimleri diziye atayalım.

foreach(var item in schools)

{

Console.WriteLine(item);

}

}

}

enum Schools

{

PrimarySchool = 4, HighSchool = 42, University = 555

}

}

Çıktı:

PrimarySchool

HighSchool

University

Ya da değerleri almak için GetValues'i kullanabiliriz. GetValues'ün dönüş tipi Array'dir.

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

foreach(var item in Enum.GetNames(typeof(Schools))) //şeklinde yazabiliriz.

{

Console.WriteLine(item);

}

}

}

enum Schools

{

PrimarySchool = 4, HighSchool = 42, University = 555

}

}

Çıktı:

PrimarySchool

HighSchool

University

Örnek:

class Program

{

enum Level

{

Low,

Medium,

High

}

static void Main(string[] args)

{

Level myVar = Level.Medium;

Console.WriteLine(myVar);

}

}

Çıktı:

Medium

Örnek:

enum Months

{

January, // 0

February, // 1

March, // 2

April, // 3

May, // 4

June, // 5

July // 6

}

static void Main(string[] args)

{

int myNum = (int) Months.April;

Console.WriteLine(myNum);

}

Çıktı:

3

Örnek:

enum Level

{

Low,

Medium,

High

}

static void Main(string[] args)

{

Level myVar = Level.Medium;

switch(myVar)

{

case Level.Low:

Console.WriteLine("Low level");

break;

case Level.Medium:

Console.WriteLine("Medium level");

break;

case Level.High:

Console.WriteLine("High level");

break;

}

}

Çıktı:

Medium level

**DERS 15: Generic Yapılar**

**List - Listeler**

- List'de bir class'tır. Dizilerin kısıtlamalarından bizi kurtarır.

- using System.Collections.Generic; dahil etmemiz lazım.

List <type> isim = new List <type> () {...}

Örnek:

namespace MyNamespace

{

class Program

{

public static void Main(string[] args)

{

int[] sayilar = new int[3]; //normal Array

sayilar[0] = 2;

sayilar[1] = -1;

sayilar[2] = 6;

//sayilar[3] = 7; yazarsak hata alırız. Çünkü dizinin sınırını aştık.

//int[] sayilar2 = new int[3] {2, 6, 7, 9 }; hata.

List<int> numbers = new List<int>() { 2, 6, 7, 8, -15 }; //List'lerde sınırlama yok. Daima ekleme yapabiliriz.

numbers.Add(19);

numbers.Add(-7);

numbers.Add(30);

foreach (var item in numbers) //değerleri foreach ile yazdırabiliriz.

{

Console.WriteLine(item);

}

}

}

}

**Generic Yapıları**

- Generic'ler tasarlandığımız interface, class, metod yada parametrelerin (argümanların) belirli bir tip için değil bir şablon yapısına uyan her tip için çalışmasını sağlayan bir yapıdır.

- Yazılım parçacıkları içerisinde tekrar kullanılabilir kod yazmamıza yardımcı olarak kod tekrarını önler.

- Kaliteli ve daha yönetilebilir kod yazmamıza olanak sağlar.

- Çalışma zamanında (run time) gereksiz Cast ve Boxing-Unboxing kullanmasını önlediğinden efektif performans sağlar.

- Derleme zamanında (compile time) (type safe) tip güvenli değişken kullanılmasını zorlayarak çalışma zamanında oluşabilecek tip dönüşüm hatalarını önler.

- Programcıya kod üzerinde daha güçlü esnek bir kontrol sağlar.

**Generic Koleksiyonlar (Generics Collections)**

- List en çok kullanılan koleksiyon tiplerinden biridir.

- Generic koleksiyonlar System.Collections.Generics namespace’i sayesinde kullanılabilir.

- Bu namespace içirisinde çeşitli parametreleştirilmiş koleksiyon/konteynır sınıfları barındırmaktadır.

- Bunları kullanmak için sadece koleksiyonun tipini belirtip parametre olarak geçmek yeterlidir.

Örnek:

List<int> myList = new List<int>(); //<> içerisinde int tipini prametre olrak girip generic List türünde bir nesne oluşturuyoruz.

myList.Add(3);

myList.Add(4);

// myList.Add(5.0); yorumu kaldırırsak hata verir. Çünkü sadece int tipinde değer kabul eder. Böylece derlee anında tip güvenliği sağlanmış olur.

int total = 0;

foreach(int val in myList)

{

total = total + val;

}

Console.WriteLine("Toplam : {0}", total);

Çıktı:

Toplam : 7

**Dictionary Sınıfının Temel Yapısı**

- Bir koleksiyon sınıfıdır.

- Standart diziler sabit boyutludur; programlama aşamasında dizinin boyutu belirtilir ve programın çalışması sırasında değiştirilemez. Dictionary ise değişken boyutludur. Eleman ekleme ve çıkarma durumuna göre boyutu dinamik olarak değişmektedir.

-- Standart dizilere eklenen elemanlar, belleğe sıralı bir şekilde yerleştirilmektedir. Sıfırdan başlanarak her bir elemana birer indeks değeri verilip, elemanlara o indeksler aracılığıyla erişmemiz sağlanmaktaydı. Koleksiyon sınıflarından biri olan ArrayList içinde aynı durum söz konusu. ArrayList’e eklenen her bir elemana indeks numarasıyla erişebilmekteyiz.

- Dictionary koleksiyonunda ise Anahtar(Key) ve Değer(Value) olmak üzere iki kavram karşımıza çıkmakta. Konuyu daha anlaşılır kılmak açısından; standart dizilere eklediğimiz elemanları Değer, o elemanlara erişmek için kullandığımız indeksleri de Anahtar olarak düşünebilirsiniz.

- Her bir Değerin farklı bir Anahtarı olmalıdır yani koleksiyon içerisinde yer alan Anahtarlar birbirinden farklı olmalıdır.

- Dictionary sınıfından bir nesne oluştururken, anahtar ve değerin veri tiplerini belirtmemiz gerekmekte.

Aşağıdaki kod satırında, Key\_Tipi yerine Anahtarın veri tipini, Value\_Tipi yerine de Değerin verini tipini belirtmeliyiz.

Dictionary<Key\_Tipi, Value\_Tipi> Referans\_Adi = new Dictionary<Key\_Tipi, Value\_Tipi>();

Örnek: Dictionary ile Öğrenci numarası girildiğinde ilgili öğrencinin adı-soyadı bilgilerini ekrana yazdır.

Dictionary<int, string> Ogrenci = new Dictionary<int, string>();

//öğrenci numarasını Anahtar, ad-soyad bilgisini de Değer olarak tutalım. Bu yüzden Anahtar int, Değer de string olsun.

Ogrenci.Add(134, "Tolga Demirer"); //buraya gelir Anahtar 134'ün Değerini yani Tolga Demirer'i ekrana yazdırcaktır.

Ogrenci.Add(158, "Ümit Özkan");

Ogrenci.Add(115, "Kadir Aydemir");

Ogrenci.Add(174, "Cemal Çiftçi");

Console.Write("Öğrenci No Giriniz:");

int No = int.Parse(Console.ReadLine()); //klavyeden girilen numarada bir öğrenci koleksiyon içerisinde mevcut ise ad-soyad bilgileri ekrana yazdırılacak.

try

{

Console.WriteLine(Ogrenci[No]); //134 girersek Ogrenci[134] 'e gidecek

}

catch

{

Console.WriteLine("Öğrenci Bulunamadı."); //olmayan bir No girersekte burası tetiklenecektir.

}

Detaylı Dictionary bilgisi için: https://www.srdrylmz.com/c-dictionary-sinifi/

**Generic Sınıflar**

- Şu ana kadar oluşturduğumuz sınıflarda, sınıf içerisindeki değişkenlerin, metotların ve parametrelerin veri tiplerini onları tanımlarken belirttik.

public class Example

{

public int example\_1; //example\_1 değişkeninin veri tipini ve

public void example\_2(int parameter) { } //example\_2 metodunun parametresinin ver tipini de int olarak belirttik.

}

- Artık example\_1 değişkenine int haricinde bir değer atamamız veya example\_2 metoduna int'den farklı bir parametre atamak mümkün değildir.

- Generic yapıyı kullanığımız takdirde sınıf içerisindeki değişkenlerin, parametrelerin ve metotların geri dönüş tipleirni o sınıftan bir nesne oluştururken belirtebilmekteyiz.

public class Example<T>

{

public T example\_1;

public void example\_2(T parameter) { }

}

- Artık Example sınıfından nesne oluştururken T (farklı bir harf ya da isim de verilebilir) yerine hangi veri tipini yazarsak; sınıf içerisindeki değişken ve parametrenin

tipi o şekilde olacaktır.

class Program

{

static void Main(string[] args)

{

Example<int> exp\_1 = new Example<int>(); //exp\_1 nesnesinde example\_1 değişkeni ve example\_2 metodunun parametresi int tipinde olacaktır.

exp\_1.example\_1 = 18;

exp\_1.example\_2(128);

Example<string> exp\_2 = new Example<string>(); //exp\_2 nesnesinde ise example\_1 değişkeni ve example\_2 metodunun parametresi string tipinde olacaktır.

exp\_2.example\_1 = "Serdar YILMAZ";

exp\_2.example\_2("C# - Generic Sınıflar");

}

}

- Yani generic'ler sayesinde bir sınıfın elemanlarının veri tiplerini ihtiyaç doğrultusunda yeni bir sınıf tanımlamaya gerek kalmadan değiştirebilmekteyiz.

Example<int> exp\_1 = new Example<int>(); //bu nesne oluşturulduğunda

public class Example<int> //class bu şekli alacaktır.

{

public int example\_1;

public void example\_2(int parameter) { }

}

aynı şekilde,

Example<string> exp\_2 = new Example<string>(); //nesnesi oluşturulduğunda

public class Example<string> //class bu şekli alacaktır.

{

public string example\_1;

public void example\_2(string parameter) { }

}

**Generic Metotlar**

- Bazen bir sınıf içerisindeki metotların sadece birkaçını Generic olarak kullanmak isteyebiliriz.

- Böylesi bir durumda sınıfı Generic yapmak yerine sadece ilgili metotları Generic yapmak çok daha mantıklı olacaktır.

public class Example

{

public int example\_1(int parameter) //bu ve

{

return parameter;

}

public bool example\_2(bool parameter) //bu metotların parametre ve geri dönüş tipleri bellidir. Bu yüzden farklı türden bir değer almaları ve geri döndürmeleri mümkün değildir.

{

return parameter;

}

public T example\_3<T>(T parameter) //ancak example\_3 metodu Generic olarak tanımlandığı için parametre ve geri dönüş tipi çağrılmdan önce belirtilmelidir.

{

return parameter;

}

public string example\_4<T>(T parameter) //example\_4 metodunun ise geri dönüş tipi string olup, parametresinin veri tipi çağrılmadan önce belirtilmelidir.

{

return parameter.ToString();

}

}

class Program

{

static void Main(string[] args)

{

Example exp\_1 = new Example();

Console.WriteLine(exp\_1.example\_1(18));

Console.WriteLine(exp\_1.example\_2(true));

// Metodun parametre ve geri dönüş tipi string olacaktır.

Console.WriteLine(exp\_1.example\_3<string>("Serdar YILMAZ"));

// Metodun parametre ve geri dönüş tipi double olacaktır.

Console.WriteLine(exp\_1.example\_3<double>(3.14));

// Metot int tipinde parametre alacaktır.

Console.WriteLine(exp\_1.example\_4<int>(15));

// Metot string tipinde parametre alacaktır.

Console.WriteLine(exp\_1.example\_4<string>("Serdar YILMAZ"));

}

}

**Generic Interfaces - Arayüzler**

- Tıpkı sınıflarda olduğu gibi arayüzleri de Generic olarak tanımlayabiliriz.

Örnek:

public interface IExample<T> //tek fark Generic sınıflarda veri tipini o sınıftan bir nesne oluştururken belirtmekteyiz.

{

T example\_1(T input);

void example\_2(T input1, T input2);

}

//Generic arayüzlerde ise veri tipini, o arayüzü bir sınıfa implement ederken belirtmekteyiz.

public class ExampleClass\_1 : IExample<int> //veri tipini ExampleClass\_1'e implement ederken belirttik.

{

public int example\_1(int input) { return input; } //her şey int oldu

public void example\_2(int input1, int input2) { }

}

public class ExampleClass\_2 : IExample<string> //aynı şekilde

{

public string example\_1(string input) { return input; } //her yeri stringledik mübarek.

public void example\_2(string input1, string input2) { }

}

**Arayüzlerde Generic Metot Bildirimi**

- Arayüz içerisindeki metotların tamamı Generic değilse; Arayüz yerine sadece ilgili metotları Generic yapmak daha mantıklı olacaktır, tıpkı Generic metotlarda anlattığımız gibi.

public interface IExample

{

void example\_1(int input);

T example\_2<T>(T input);

}

public class ExampleClass\_1 : IExample

{

public void example\_1(int input) { }

public T example\_2<T>(T input) { return input; }

}

- IExample arayüzü Generic olmadığı için herhangi veri tipi belirtmeden ExampleClass\_1 sınıfına implement edebildik.

- example\_2 metodu Generic olduğu için veri tipi metot çağrılırken girilecektir.

class Program

{

static void Main(string[] args)

{

ExampleClass\_1 example = new ExampleClass\_1();

example.example\_1(18);

example.example\_2<string>("Serdar YILMAZ"); //veri tipini metot çağırırken string olarak girdik.

}

}

**Statik Generic Sınıflar**

- Statik sınıflar ile statik olmayan sınıfların Generic yapılması noktasında arada herhangi bir fark bulunmuyor.

- Statik sınıfların metot ve değişkenlerine nesne oluşturmadan erişebildiğimiz için, veri tipini sınıf adını yazdıktan hemen sonra belirtiyoruz.

public static class Example<T>

{

public static T example\_1(T input) { return input; }

}

class Program

{

static void Main(string[] args)

{

Example<string>.example\_1("Serdar YILMAZ");

}

}

**Generic’lerde Birden Fazla Veri Tipinin Kullanılması**

- Generic ifadelere birden fazla veri tipi gönderebiliriz.

public class Example<T, Y, C>

{

public T example\_1;

public void example\_2(Y parameter1, C parameter2) { }

}

class Program

{

static void Main(string[] args)

{

Example<int, string, bool> example = new Example<int, string, bool>();

example.example\_1 = 18;

example.example\_2("Serdar YILMAZ", true);

}

}

**Generic Kısıtlar**

**Değer ve Referans Tip Kısıtı**

public class ExampleClass<T>

{

public T example\_1 { get; set; }

public T example\_2(T parameter)

{

return parameter;

}

}

- ExampleClass Generic sınıfına herhangi bir kısıt uygulanmadığı için T yerine int, double, float gibi değer tipleri gönderebileceği gibi;

string, object, array gibi referans tipleri de gönderilebilir.

- Değer Tipleri: “int”, “long”, “float”, “double”, “decimal”, “char”, “bool”, “byte”, “short”, “struct”, “enum”

Referans Tipleri: “string”, “object”, “class”, “interface”, “array”, “delegate”, “pointer”

class Program

{

static void Main(string[] args)

{

ExampleClass<int> obj1 = new ExampleClass<int>();

obj1.example\_1 = 18;

obj1.example\_2(128);

ExampleClass<string> obj2 = new ExampleClass<string>();

obj2.example\_1 = "Serdar YILMAZ";

obj2.example\_2("C# Doküman");

}

}

- Oluşturduğumuz ExampleClass Generic sınıfına sadece referans tip veya sadece değer tip gönderilmesini isteyebiliriz.

Böylesi bir durumda Generic sınıfımıza bir kısıt koymamız gerekmekte.

- Sadece referans tip gönderilmesi için;

public class ExampleClass<T> where T:class

{

public T example\_1 { get; set; }

public T example\_2(T parameter)

{

return parameter;

}

}

- ExampleClass Generic sınıfına, “where T:class” kısıtı sayesinde sadece referans tipleri gönderilebilir.

Değer tipli bir tür gönderildiği takdirde hata oluşacak ve proje derlenmeyecektir.

ExampleClass<string> obj1 = new ExampleClass<string>(); // Geçerli tip

ExampleClass<object> obj2 = new ExampleClass<object>(); // Geçerli tip

ExampleClass<int> obj3 = new ExampleClass<int>(); // HATA! Geçersiz tip

ExampleClass<bool> obj4 = new ExampleClass<bool>(); // HATA! Geçersiz tip

- Sadece değer tip gönderilmesi için;

public class ExampleClass<T> where T : struct

{

public T example\_1 { get; set; }

public T example\_2(T parameter)

{

return parameter;

}

}

- ExampleClass Generic sınıfına, “where T : struct” kısıtı sayesinde sadece değer tipleri gönderilebilir.

Referans tipli bir tür gönderildiği takdirde hata oluşacak ve proje derlenmeyecektir.

ExampleClass<int> obj1 = new ExampleClass<int>(); // Geçerli tip

ExampleClass<bool> obj2 = new ExampleClass<bool>(); // Geçerli tip

ExampleClass<string> obj3 = new ExampleClass<string>(); // HATA! Geçersiz tip

ExampleClass<object> obj4 = new ExampleClass<object>(); // HATA! Geçersiz tip

**Base Class Kısıtı**

- T parametresi herhangi bir class veya o class'tan türeyen sınıfları refere edebilsin koşulu koyabiliriz.

class MyClass<T> where T:A //T tipi A veya A'dan türeyen sınıfları temsil edebilir.

{

T t;

public void X()

{ }

}

class A

{ }

class B:A

{ }

class C:B

{ }

class D

{ }

Main...

{

MyClass<A> m1 = new MyClass<A>(); //A veya A'dan kalıtım alan sınıfları refere edebilir.

MyClass<B> m2 = new MyClass<B>();

MyClass<C> m3 = new MyClass<C>();

MyClass<D> m3 = new MyClass<D>(); //HATA. D, A'dan kalıtım almıyor.

}

**new() Kısıtı**

-Generic sınıfa gönderilen veri tipini, sınıf içerisinde new’liyorsak, yani o tipten yeni bir nesne oluşturuyorsak; Generic sınıfa “new()“ kısıtını uygulamamız gerekmektedir.

Eğer Visual Studio ile geliştirme yapıyorsanız zaten IDE sizi new() kısıtını uygulamaya zorlayacaktır.

- T obj = new T(); 'de verilen tip bir nesne üretilemeyen bir sınıf olabilir(static), constructor'u private olan ya da parametre alan bir sınıf olabilir, interface olabilir, veyahut bunun

gibi nesne üretmemizi engelleyen bir tip olabilir. where T : new() ile bana bunun nesne üretilebilir bir sınıf olduğunun garantisini ver, sadee bu olduğunda nesne üret demiş oluyoruz.

T'den bir nesne üretilebilir sınıf verilecektir.

public class ExampleClass<T> where T : new()

{

public T createObject()

{

T obj = new T();

return obj;

}

}

- ExampleClass Generic sınıfında gönderilen veri tipi createObject metodunda new’lenerek yeni bir nesne oluşturulmak istendiğinden new() kısıtı uygulanmıştır.

Artık ExampleClass Generic sınıfına sadece nesne oluşturulabilen yani new() lenebilen türler gönderilebilir.

**Arayüz-Interface Kısıtı**

- Generic sınıfa sadece belirttiğimiz arayüzleri implement alan türlerin gönderilmesini istiyorsak arayüz kısıtını uygularız.

public interface IExample1 { }

public interface IExample2 { }

public class ExampleClass1 : IExample1

{

public string example\_1 { get; set; }

}

public class ExampleClass2 : IExample1

{

public string example\_1 { get; set; }

}

public class ExampleClass3 : IExample2

{

public string example\_1 { get; set; }

}

- IExample1 ve IExample2 isminde iki arayüz oluşturduk. IExample1 arayüzünü ExampleClass1 ve ExampleClass2‘ye implement ettik.

IExample2 arayüzünü de ExampleClass3 sınıfına implement ettik.

public class GenericClass<T> where T : IExample1

{

public T example\_1 { get; set; }

public T example\_2(T parameter) { return parameter; }

}

- Generic sınıfımıza “where T : IExample1” kısıtını eklediğimiz takdirde; Generic sınıfımıza veri türü olarak sadece IExample1 arayüzünü implement alan ExampleClass1 ve ExampleClass2

gönderilebilecektir. ExampleClass3 sınıfı IExample1 arayüzünü implement almadığı için GenericClass’a veri tipi olarak gönderilmek istendiğinde hata alınacaktır.

Arayüz kısıtı sayesinde Generic sınıflara sadece belli arayüzleri implement alan sınıfların gönderilmesini sağlayabiliriz.

**Birden Fazla Kısıt Ekleme**

- Yukarıda anlatmış olduğumuz kısıtları birlikte de kullanabiliriz.

public interface IExample1 { }

public class ExampleClass<T> where T : class, IExample1 , new()

{

public T createObject()

{

T obj = new T();

return obj;

}

}

- ExampleClass Generic sınıfına sadece referans tipli olan, IExample1 arayüzünü implement alan ve new’lenebilen veri türleri gönderilebilir.

- new() kısıtı her zaman en son da yer almalıdır.

**Generic Metotlara Kısıtların Eklenmesi**

- Generic kısıtları tıpkı sınıflarda kullandığımız gibi generic metotlarda da kullanabiliriz.

public class ExampleClass

{

public T createObject<T>() where T : class, IExample1, new()

{

T obj = new T();

return obj;

}

}

- createObject Generic metoduna sadece referans tipli olan, IExample1 arayüzünü implement alan ve new’lenebilen veri türleri gönderilebilir.

**DERS 16: C# Ekstra Makaleler - Bilgiler**

**C# params Nedir? Kullanımı**

https://www.yusufsezer.com.tr/csharp-params/

**const c#**

https://www.tasarimkodlama.com/csharp-programlama/c-const-kullanimi/

**Genişletme Metotları - Extension Methods**

https://www.kazimcesur.com/c-extension-methods/

**Partial Class - Partial Methods**

https://www.hikmetokumus.com/makale/23/csharp-partial-class-nedir-

https://www.gencayyildiz.com/blog/c-partial-method-yapilari/

**Sealed Keyword**

https://www.hikmetokumus.com/makale/152/csharp-ile-sealed-anahtar-kelimesinin-kullanimi

**Dispose Method**

https://www.yusufsezer.com.tr/csharp-idisposable/

**C#’ta Anonymous Type**

https://www.gencayyildiz.com/blog/cta-anonymous-type/

http://www.kazimcesur.com/anonymous-types-anonim-tipler/

**Non-Generic Koleksiyonlar**

https://www.gencayyildiz.com/blog/cta-non-generic-koleksiyon-yapilari-system-collections/

**Iterasyon Kavramı ve Yield Anahtar Kelimesi**

https://www.gencayyildiz.com/blog/cta-iterator-kavrami-ve-yield-anahtar-sozcugu/

**IEnumerable ve IEnumerator Arayüzleri**

https://medium.com/@srhtayata/c-ienumerable-ve-ienumerator-aray%C3%BCzleri-9f3b90f81629

https://www.gencayyildiz.com/blog/cta-ienumerable-ve-ienumerator-interfaceleri-nedir-ve-nasil-kullanilir/

**C# Delegate(Temsilci) ve Event(Olay) Kullanımı**

https://www.gencayyildiz.com/blog/cta-delegatetemsilci-ve-eventolay-kullanimi/

**C#’ta Öznitelik(Attribute) Oluşturma**

https://www.gencayyildiz.com/blog/cta-oznitelikattribute-olusturma/

**Reflection**

https://www.gencayyildiz.com/blog/cta-system-reflection-kutuphanesi/

https://www.ismailgursoy.com.tr/reflection-yansima/