**Object Oriented Programming** - **Erişim Belirleyiciler**

Kendi sınıfınızı yazacaksanız, öncelikle C# dilinde nesnelerin birbirleriyle ilişki kurarken hangi erişim kontrolleri ile çalıştığını bilmeniz önemlidir. Erişim belirleyiciler (access modifiers), sınıflara nerelerden ve ne şekilde erişileceğini belirtir. Anahtar kelimelerin ne işe yaradığı hakkında genel bir tanım yapalım. Bir tipin kendisine veya o tipe ait üyelere (metod, özellik ya da olay) nasıl erişileceğini daha doğrusu, nereden (hangi kod bloğundan) erişelebileceğini belirleyen kelimelere “erişim düzenleyiciler” diyoruz.

**Private:** En küçük erişim düzenleyicisidir. Tipin üyelerinde kullanılır. Üyenin (bu üye metod veya global değişken –alan- olabilir), yalnızca o tipin içerisinden erişilmesine izin verir. Yani üyenin o tipe “özel” olmasını sağlar. Adı üzerinde! Başka bir değişle, private anahtar kelimesi ile tanımlanmış bir üyeye, tip dışından ulaşamazsınız. Ayrıca private üyeler miras yoluyla türetilmiş sınıflara aktarılamazlar ve eğer bir üyenin önüne hiçbir erişim düzenleyici anahtar kelimesi belirtmezseniz, o üye derleyici tarafından private olarak algılanır. Yani private, üyeler için varsayılan erişim düzenleyicisidir.

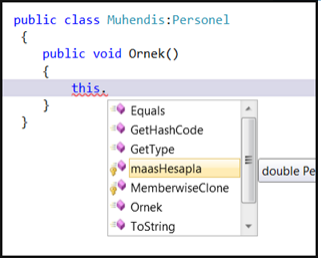
**Public:** En genel ve sınırsız erişim düzenleyicisidir. Hem tip için hem de, tip üyeleri için kullanılabilir. Elbette her ikisi için de farklı anlamları vardır. Önce tipler için konuşalım. Örneğin, bir sınıfın erişim düzenleyicisi public ise bu, o sınıfın, bulunduğu assembly dışından (referans olarak alındığı başka bir assembly’den) her türlü erişilebileceği anlamına gelir. Peki, tipin üyeleri public ise ne olur? Yine adı üzerinde. Public, “genel” demektir. Yani, o üyeye her yerden erişilebilir. Doğal olarak kalıtım yolu ile türetilmiş sınıflara aktarılırlar.

**Protected:** “Korunan” anlamına gelmektedir. Peki, sadece tipin üyelerinde (metod veya alan) kullanılabilir (aynı private gibi). Bu üyeye yalnızca ait olduğu tip içinden ulaşılabilir, private gibi. Peki private’den farkı nedir? İşte tek fark: protected olarak tanımlanmış alan ya da metodlar, miras olarak aktarılabilirler. Yani, “bu üyeye kesinlikle dışardan ulaşılamasın, ama miras bırakılabilsin” diyorsanız, o üye protected olmalı.

Peki, ne zaman, nasıl bir modelde buna ihtiyaç duyarsınız? Örnekle açıklamak gerekirse; “personel” isminde bir sınıfınız var ve bu sınıfın da “maasHesapla” isminde bir metodu var. Metod basitçe, mesai saati ve saat ücretlerini alıp çarpıyor. Şimdi, bu sınıfın temel sınıf olmasına karar verdiniz. Muhendis ve Muhasebeci sınıflarını buradan türeteceksiniz. E onlarda da maasHesapla metodu olacak (yani kalıtım yoluyla aktarılacak – private olamaz). Ama siz, bu sınıfların dışından (yani sınıfın örneğinden) maasHesapla metoduna erişmek istemiyorsunuz (o zaman public olma şansı yok). İşte bu durumda maasHesapla metodu, protected olmalı. Örneklerle bakalım:



Yukarıda mevzu bahis Personel isimli sınıfımı görüyorsunuz.



Burada ise, türetilmiş sınıfım olan Muhendis içinde, “Ornek” isimli metodda, this anahtar kelimesiyle; Personel sınıfından kalıtım aldığım “maasHesapla” metoduna ulaşabiliyorum. Burada bir hatırlatma yapalım; this anahtar sözcüğü, yalnızca o sınıfın (örneğimizde Muhendis sınıfı) üyelerine ulaşmak için kullanılır. Böylece, kabaca “miras bırakılabilen, fakat dışarıdan ulaşılamayan alanlar protected erişim düzenleyicisi ile belirlenir” diyebiliriz.

**Internal:** “Dahili” anlamına gelmektedir. Yalnızca bulunduğu assembly’den erişilebilir. Burada assembly ifadesinden kasıt, projenin kendisidir. Yani, bir kütüphane (.dll) oluşturuyorsanız, internal bir sınıfa sadece o kütüphaneden ulaşabilirsiniz. Bu erişim düzenleyicisi, sınıf üyelerinde de kullanılabilir. Onda da etkisi aynıdır. Bir sınıfın erişim düzenleyicisi belirtilmezse, varsayılan olarak internal kabul edilir.

**Protected Internal:** Yalnızca sınıf üyelerine uygulanır. Kelimelerin arasına “ya da” konulunca anlaşılması daha kolay olacaktır. Erişim yapılan yere göre “internal” ya da “protected” davranır. Doğal olarak assembly dışından erişilmeye çalışıldığında internal, aynı assembly’den erişilmeye çalışıldığında ise proyected davranır.

Şimdi birer cümleyle hepsinden bahsetmek gerekirse;

* **Public:** Her yerden erişilebilir.
* **Private:**Sadece tanımlandığı sınıf içerisinden erişilebilir.
* **Internal:**Sadece bulunduğu projede erişilebilir.
* **Protected:**Sadece tanımlandığı sınıfta ya da o sınıfı miras alan sınıflardan erişilebilir.
* **Protected Internal:**Sadece tanımlandığı sınıfta ya da o sınıfı miras alan sınıflardan erişilebilir. Ayrıca tanımlamanın aynı proje içerisinde olma şartı yoktur. Protected'dan farkı budur.

**Value Types – References Types**

.NET kodunun çalıştırılmasını sağlayan CLR yapısı iki çeşit veri tipini destekler; referans tipi (reference types) vedeğer tipi (value types). **FCL** (Framework Class Library) içerisinde bulunan tiplerin çoğu referans tipi olsa da, programcıların kullandığı tiplerin çoğunu değer tipleri oluşturur. Referans tiplerine her zaman belleğin “Heap” kısmından yer ayrılır ve C#’ın **new** operatörü de bu nesnenin bellek adresini döndürür (Bellek adresi de nesnenin bitlerini kaynak gösterir). Şunu aklınızda tutmalısınız ki, referans tipleri ile çalışırken bazı performans özelliklerine dikkat etmelisiniz. Şu 4 maddeyi göz önünde bulundurun:

1. Yönetilen heap kısmından bellek ayrılmak zorundadır.

2.Heap bölgesine ayrılmış her nesnenin kendisi ile ilişkili ilave üyeleri mevcuttur.

3.Nesnenin içineki diğer bitler her zaman sınıfa ayarlanır. (Alanlar için)

4.Bir nesneye heap bölgesinden kaynak ayırabilmek bazen Garbage Collection’ı ortaya çıkartabilir.

Her tip referans tipi olsaydı, bir uygulamanın performansı fazlasıyla kayba uğrardı. Düşünün ki, her **Int32** tipini kullandığınızda bellekten bir yer tahsis etmek zorundasınız. Ne kadar kötü bir performans alacağınızı farketmişsinizdir. Performans arttırmak için, sıkça tip kullanılan uygulamalarda, CLR “hafif siklet” olarak adlandırdığı değer tiplerini önerir. Değer tipi örnekleri için çoğunlukla belleğin stack bölgesinden yer tahsis edilir (Ayrıca bir referans tipi nesnenin içerisine bir alan olarak gömülebilirler). Bu tipin bir değişkeni, örneğine bir pointer içermez, bu değişken örneğinin kendisinin bir alanını içerir. Değer tipi örnekleri, garbage collector’ın kontrolü altına girmezler ve bu sayede yönetilen heap kısmının üzerindeki yükü azaltır.

Değişkenler, programımızda geçici olarak değer tutar demiştik yani bilgisayarın RAM 'inde tutulurlar. Ram, "**stack**" ve "**heap**" bölümlerinden oluşur. Bir değişkenin tipi, değişken tanımlandığında, ram 'in hangi kısmında tutulacağına göre *değer tipi* ve *referans tipi* olmak üzere iki genel kategoride toplanır. Bu iki tip arasındaki fark, bir değişkenin ne içerdiği ile ilgilidir. Değer tipinde bir değişken 10 ya da 10,05 gibi gerçek bir değer içerirken; referans tipi bir değişken, o değere yapılan referansı içerir.

C#'ın temelinde 13 adet veri tipi yer alır. Bu değer tiplerine primitif yani basit tip(primitive type) de denir. Her zaman tek bir değer içerdikleri için basit olarak nitelendirilmişlerdir. Başka bir deyişle, iki ya da daha farklı değerin biraraya gelmesinden oluşmazlar. C# tip sisteminin temelini oluşturur ve bir programın kullandığı en temel veri tiplerini sağlarlar.

Bu temel tipler aşağıdaki tablolarda görülüyor.

|  |  |  |  |
| --- | --- | --- | --- |
| **Değer Tipleri** | | | |
| **Adı** | **.Net Framework (CTS) Karşılığı** | **Açıklama** | **Aralık** |
| sbyte | System.Byte | 8 bit işaretli tam sayı | [-128, 127] |
| short | System.Int16 | 16 bit işaretli tam sayı | [-32.768, 32.767] |
| int | System.Int32 | 32 bit işaretli tam sayı | [-2.147.483.648, 2.147.483.647] |
| long | System.Int64 | 64 bit işaretli tam sayı | [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807] |
| byte | System.Byte | 8 bit işaretsiz tam sayı | [0, 255] |
| ushort | System.UInt16 | 16 bit işaretsiz tam sayı | [0, 65.535] |
| uint | System.UInt32 | 32 bit işaretsiz tam sayı | [0, 4.294.967.295] |
| ulong | System.UInt64 | 64 bit işaretsiz tam sayı | [0, 18.446.744.073.709.551.615] |
| float | System.Single | 32 bit tek kayan sayı (7 basamak) | ± 1,5\*10-45, ± 3.4\*1038~ |
| double | Sytem.Double | 64 bit çift kayan sayı (15-16 basamak) | ± 5\*10-324, ± 1.7\*10308 ~ |
| decimal | System.Decimal | 128 bit ondalıklı sayı (28-29 basamak) | ± 1,5\*10-28, ± 7,9\*1028~ |
| bool | System.Boolean | 1 bit yer tutar | true ya da false |
| char | System.Char | Karakterleri temsil eder | 16 bit bütün unicode karakterleri |

|  |  |  |
| --- | --- | --- |
| **Referans Tipleri** | | |
| **Adı** | **.Net Framework (CTS) Karşılığı** | **Açıklama** |
| object | System.Object | Bütün veri türlerinin türediği sınıf |
| string | System.String | Unicode karakterlerinden oluşan string |

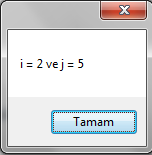
* **Value Type (13) :**Referans tipler string ve object tipler olmak üzere 2 tanedir. Bu tipler dışındaki 13 adet tip ise, değer tipleridir. Değer tipleri ram 'in "stack" bölümünde tutulurlar. Değer tipleri tanımlandığında, tipine göre, ram 'de belirli bir hafıza rezerv edilir. Bu hafıza alanının, basamak tipi "hexadecimal" olan bir adresi oluşur. Bu adreste de değişkenin değeri tutulur. Değişkenin ömrü tamamlandığında, bu alan serbest bırakılır. Başlangıç değerleri olmak zorundadır, "**null**" olamazlar.  Değerlerini üzerilerinde taşırlar. Bu yüzden eğer bir değer tipli değişken, diğer değer tipli değişkene atandığında, değişkenlerin değerleri atanmış demektir. Yani eski değişken de değerini saklar ve iki değişkenin değeri de aynı olur. Bunun örneğini de aşağıda görüyoruz arkadaşlar.

            int i = 2;

            int j = i;

            j = 5;

            MessageBox.Show(string.Format("i = {0} ve j = {1}", i, j));

[](http://3.bp.blogspot.com/-Fgo3wP-aCj4/T5RxOdEEtdI/AAAAAAAAAHM/spjWmZUWo80/s1600/1.png)

* **Reference Type (2) :**Değer tiplerini açıklarken object referans tiptir demiştik. Bundan dolayı class 'lar da referans tiplerdir. Referans tiplerdeki olay ise değişken tanımlandığında, "stack" bölümünde(C++ 'daki pointer 'lar), değişkenin "heap" bölümündeki adresi tutulur. Değişkenin değeri ise "heap" bölümünde tutulur. Bir referans tip tanımlanırken değer almamışsa varsayılan değeri "null" dır. Değeri "null" ise veya "null" değer atanmışsa, o değişkenin değeri yok demektir. Değeri "null" olsa da "stack" bölümünde bir adres tutar. Bu adres , "heap" bölümündeki adresi referans edemeyeceğinden dolayı, "stack" ile "heap" arasında bir bağlantı kurulmamıştır. "Null" bir değişken üzerinde işlem yapılması istenirse, program bize hata verecektir. "Olmayan bir arabamın tekeri" diye bir deyimin hatalı olduğu gibi.

Şimdi bir örnek ile referans tipleri daha yakından inceleyelim.

            Button b1 = new Button();

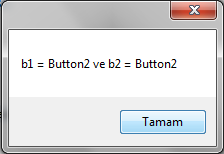
            b1.Text = "Button1";

            Button b2 = b1;

            b2.Text = "Button2";

            MessageBox.Show(string.Format("b1 = {0} ve b2 = {1}", b1.Text, b2.Text));

"new" diyerek ram 'de bir tane "b1" adında "button" nesnesi oluşturdum. Textine de "Button1" yazdım. Daha sonra "new" anahtar sözcüğünü kullanmadan, "yarım instance" alarak, "b2" adında bir "button" nesnesi tanımladım (dikkat ederseniz oluşturdum demedim). "b2" ye de "b1" butonunu atadım.

[](http://4.bp.blogspot.com/-tVWhT0QS0gQ/T5RxSBP2shI/AAAAAAAAAHU/OBsq-2VJb3Q/s1600/2.png)

Bu ekran çıktısını açıklamak gerekirse ram 'de şöyle işler meydana geldi : "b2" butonu için ram 'in "heap" bölümünden bir hafıza rezerv edilmedi. Sadece "stack" bölümünden bir adres verildi. Yani bu adresin "heap" bölümünde bir karşılığı yok. Bu bilgilerin ışığında "Button b2 = b1" bu kod satırında butonların textleri birbirine atamadım. Adresi "null" olan "b2" butonuna, "b1" butonun "heap" bölümündeki adresini atadım. Bu yüzden iki buton da "stack" bölümünde, "heap" bölümündeki aynı adresi refere etti. Yani ortada bir buton var ve iki değişken de bu botunu işaret ediyor. Bu örnekten de anlaşılacağı üzere, referans tipler birbirlerine atandığında, property 'leri değil, adresleri birbirine atanır.

**NOT:** Referans tiplerde eşitleme adres eşitlemesidir.

**NOT:** Referans tiplerden string tipi için "null" olmasıyla boş olması ("") arasında büyük fark vardır. Değeri boş olması durumunda "heap" bölümünde yine bir alan rezerve edilir. String 'in uzunluğuna göre de bu alan genişleyebilir.  
**NOT:** Referans tiplerin ömrü bittiğinde "stack" bölümündeki alan iade edilir. Fakat "heap" bölümündeki alan için "**Garbage Collector**" 'ün (çöp toplayıcı) çalışması beklenir.  
**NOT:** Referans tiplerde "heap" bölümünde ayrılan alanda, "new" anahtar sözcüğü ile oluşturulan class 'ın ve bu class 'ın miras aldığı (inherit ettiği) tüm class 'ların bir instance 'ı (örneği) oluşturulur. Bu örneklerin en alt kısmında da tüm class 'ların atası olan "object" sınıfının adresi bulunmaktadır.

# Değer-Tip & Referans-Tip (Value-Type & reference-Type)

Yüksek seviye dillerin bir iyi tarafı da, bilgisayarın arka planda nasıl çalıştığı konusunda programcının bilgi sahibi olmasına gerek kalmamasıdır. Ama bazen bilmekte de yarar var. Mesela bellekte değişkenlerin nasıl saklandığını bilmek isteyebilirsiniz. Program çalışırken değişken için bellekte değişkenin türüne göre yer ayrılır ve değişkene ihtiyacımız kalmadığında ise belleğin o bölgesi serbest bırakılır.

Bir yöntem çalışmaya başladığı zaman, içerisindeki bütün değişkenler için bellekte yer ayrılır. Bu yere stack denir. Yöntemin kullanımı sona erdiği zaman ise belleğin o bölgesi serbest bırakılır.

Şöyle bir örnek düşünelim:

int A,B;

long C;

string D;

Stack bölgesinde A ve B için 4’er byte, C için 8 byte yer ayrıldı. Peki, D için ne kadar yer lazım? Bu miktar, string türündeki D değişkeninin ne kadar uzun olacağına bağlıdır. Bir string ifade çok kısa da olabilir, çok uzun da. Bu yüzden string değişkenlerin boyutu değişkendir.

Eğer string türü bir değişkene sabit bir ifade atanmışsa, boyut sabit olarak düşünülebilir. Eğer değişkenin içeriği kullanıcıdan istenen bir değer ise, veya dosyadan okunan ya da internetten alınan bir ifade ise boyutunu program çalışana kadar bilemeyiz.

Bu yüzden, string ifadeler, stack bölgesinde saklanmaz. Program çalışırken, string ifade için gereken yer heap adı verilen bölgede oluşturulur. Heap bölgesi, program çalışırken, ayırma ve serbest bırakma işlemleri yapmayı olanaklı kılan genel amaçlı bir bölgedir.

Yalnız stack ve heap bölgeleri hakkında bilinmesi gereken diğer bir gerçek var. Yukarıdaki kod bloğundaki D değişkeninin kendisi stack bölgesinde saklanır, heap bölgesinde bir alana referans içerir. D değişkeninde tutacağımız string ifadenin kendisi ise ilgili referansın gösterdiği heap bölgesindedir. Heap bölgesindeki string ifade programın akışı sırasında küçülebilir ya da büyüyeblir ama stack bölgesindeki referansı barındıran alan sabittir.

Referans derken neyi kastettiğimize biraz bakalım. Belleğin her bir hücresi numaralandırılımıştır ve bu numaralara bellek adresi denir. Program, bir bellek bölgesine erişmek isterse bu numaralardan yararlanır. Bir bellek bölgesine referans içermek demek o bölgenin adresini tutumak demektir. Yapısal programlama ile uğraşmış olanlar bunun işaretçi (pointer) demek olduğunu bilirler. Peki, içinde referans tutan bir değişken stack bölgesinde ne kadar yer kaplar? Günümüz yaygın işletim sistemleri 32-bit olduğuna göre 32 bit yani 4 byte yer kaplar.

Şimdi string ifade tutan bir değişken bellekte ne kadar yer kaplar sorusunun cevabını da bulmuş olduk. Değişkenin kendisi stack bölgesinde 4 byte yer kaplar ama string ifadenin kendisi heap bölgesinde değişken bir yere sahiptir. Kapladığı alan içeridği ifadenin boyutuna göre değişir.

## Değer-Tip & Referans-Tip (Value-Type & Reference-Type)

Yeri gelmişken referans-tip nedir, değer-tip nedir sorusuna da cevap arayalım. C#’ta yapılar değer-tiptirler. Sınıflar ise referans-tip. Bunun ne anlama geldğini anlamak için, biraz açalım.

Aslında bir çok değer tipi daha önce kullandık. Bütün tamsayı türler .Net Framework içerisindeki yapıların aliaslarıdır. Tamsayı türdeki bir sayının gerçek değeri stack bölgesinde saklanır.

Sınıflar ise referans-tiplerdir. string veri tipi, Syste.String sınıfının bir aliasıdır. Yani bir sınıftır. Sınıf da referans-tiptir. Bir referans tip, heap bölgesinde bir alanı gösteren bir referans olarak stack bölgesinde saklanır.

Yapılar değer-tiptirler. Sınıflar referans-tiptirler. Bir yöntemin aşağıdaki tanımlamayı içerdiğini düşünün şimdi.

string Ad;

Yöntem çalışmaya başladığında, stack bölgesinden 4 byte’lık yer ayrılır. Bu yerde, heap bölgesinden bir alana referans barındırılabilir. Ama Ad değişkenine bir değer ataması yapılmadığı müddetçe heap bölgesinden bir alan tahsisi yapılmaz.

Şimdi bir değer atayalım.

Ad = "Ali";

Heap bölgesinde string ifade sığacak kadar yer ayrıldı ve artık stack bölgesindeki yerimiz heap bölgesinde string ifadeyi barındıran alanın adresini tutuyor, yani o bölgeye referans tutuyor.

Bir string değişkene aşağıdaki gibi hiçbir şey atamak istemezsek ne olur, inceleyelim.

Ad = " ";

Şu anda hiçbir karakter içermiyor ama bir atama işlemi söz konusu olduğu için heap bölgesinden yer tahsisi yapılır.

Ad.Length ifadesinin değerini ekrana yazdırırsak 0 göreceksiniz. bu da heap bölgesinde yer içeriği boş olan bir yer ayrıldığını gösteriyor.

## Çöp Toplama (Garbage Collection)

Bir string türü değişkenin değerini değiştirip heap bölgesinde bir yere referans içermemesini sağlayabiliriz. Bunu da

Ad = null;

atamasıya yaparız. Biz bunu yaptığımız zaman ya da

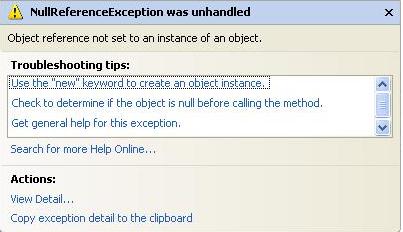
Ad = "Veli";

gibi başka bir değer ataması yaptığımız zaman, ilk atadığımız “Ali” ifadesi hala heap bölgesinde mevcuttur ama hiçbir referans onu göstermediği için ulaşılamaz haldedir. Bu şekilde artık kullanımı biten verileri heap bölgesinden temizleyen ve o bölgeleri kullanılabilir hale getiren çok güzel bir yapı barındırır C# ve CLR (Ortak Dil Çalışma Zamanı) : Çöp toplama (Garbage Collection). Programcı işi biten bölgeleri temizleme derdinden kurtulmuştur. Çöp Toplayıcı bunu programcının yerine yapar.

Heap bölgesinde hiçbir yere referans içermeyen yani null bir string değişkenin uzunluğunu bulmak için

Ad.Length

ifadesinin değerini öğrenmeye çalışırsak aşağıdaki istisna oluşur. 

Yukarıda anlatıldığı gibi, null string ile boş bir string aynı gibi görünse de aynı değildir. Bir string null’a eşitse, stack bölgesinde saklanan değer 0’a eşittir ve heap bölgesinde hiçbir yere referans içermez. Eğer bir string, boş bir string’e eşitse stack bölgesinde saklanan değer heap bölgesinde bir bölgeye referans ifade eder. Heap bölgesinde de uzunluğu 0 olan bir string için yer ayırma işlemi yapılmıştır.

**New Keyword’u ve Instance Mantığı**

Nasıl oluyor da bir class’ tan bir nesne türetebiliyoruz? Bu “türetme” işine kod tarafında baktığımızda en sık karşılaştığımız yöntem, “**new**” anahtar kelimesi kullanmaktır. Örneğin;

Urun yeniUrun = new Urun();

Örnekteki “**new Urun()**” kısmı, **Urun** class’ı içinde bulunan tüm üyeleri, **yeniUrun** isimli nesneye aktarır. Biz bu duruma Urun class’ından nesne türetmek ya da daha teknik bir ifadeyle **instance** almak diyoruz. Şimdi, işi biraz daha basit almak istiyorum. Aslında, nesne üretmenizi sağlayan şey class’ ınızın içinde bulunan oluşturucu metot (constructor) dediğimiz üyedir. **Constructor**, kesinlikle bir metottur. Ama herhangi bir değer döndüren metot değildir. Ya da void metot olarak da düşünülemez. Constructor, yalnızca üyesi bulunduğu class’dan nesne üretimi sırasında çalışacak olan metodudur. Bu durumda şunu söylemek çok yanlış olmayacaktır, classımdan nesne üretilirken; üyelerin varsayılan değerlerini ayarlamak için constructor kullanırım. Örneğin, Windows uygulaması geliştirirken hep gözümüzün önünde bulunan bir constructor’u inceleyelim şimdi… Bildiğiniz gibi Windows Form’u aslında bir class. Bakalım bu class’ ın bir constructor’u var mı?

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

}

Public Form1() ifadesi ile başlayan üye bizim constructor’ ımız oluyor. Şimdi gelin bu tanımı biraz inceleyelim. Gördüğünüz gibi constructor’ un erişim düzenleyicisi (Access Modifier) public. Çoğu constructor’un erişim düzenleyicisi public’ tir. Bunun sebebi, elbette constructor’a class dışından erişilebilir olması gerektiğidir. ( Aksi halde, Form1 frm = newForm1 (); diye bir ifade yazamazdık.) Constructor tanımının bir diğer özelliği ise isminin class ismiyle aynı olmasıdır (Form1()).