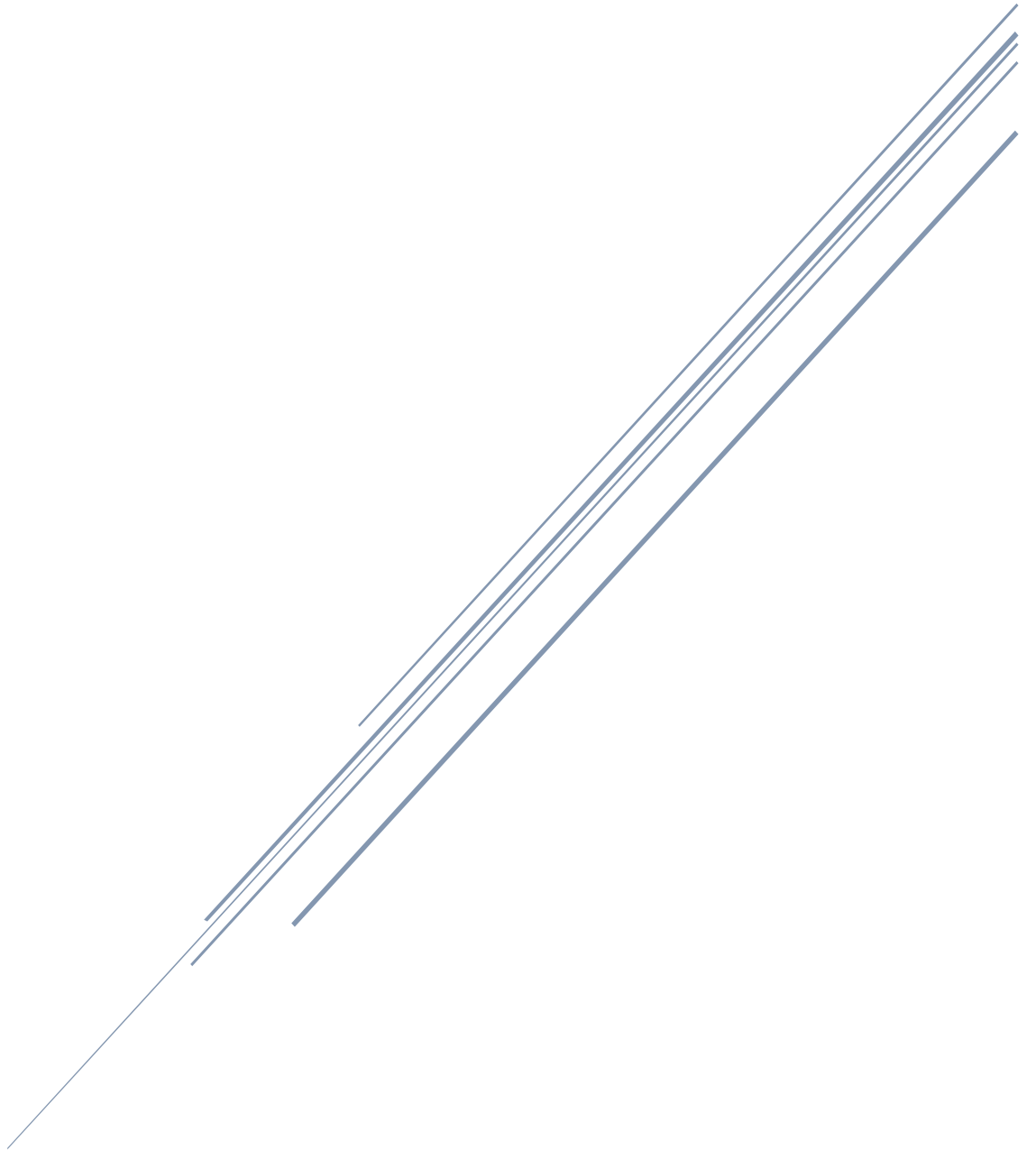


NESNE YÖNELİMLİ PROGRAMLAMA

Object-Oriented Programming

Son güncelleme

1.05.2023



Doç. Dr. Zafer CÖMERT

Ön Söz

Okumaya niyetlendiğiniz bu elektronik doküman bir akademisyenin ders notlarından oluşmaktadır.

Teknoloji hızla ilerlerken başta öğrencilerim ve daha sonra bilişim dünyasına ilgili duyan herkes için “**Nesne Yönelimli Programlama**” üst başlığı altında faydalı olabileceğini düşündüğüm pek çok içeriği bu doküman altında topluyorum.

Bu dokümanı benzerlerinden ayıran belirgin yönleri; sistematik bir içeriğe sahip olması ve hemen hemen tüm konuların video desteği ile belge içerisine yerleştirilmiş olmasıdır. Konuları inceyip, okudukça ve belge içerisinde ilerledikçe hemen her bölüm altında video içeriklerinin paylaşıldığını göreceksiniz. Tüm video içerikleri “**Virtual Campus**” Youtube kanalında paylaşıyor. İlgili konu altında paylaşılan bağlantılara tıklayarak videolara erişebilirsiniz.

Virtual Campus Youtube kanalı yazılım, teknoloji, tasarım ve modelleme üzerine eğitsel içerikler paylaşmak üzere kuruldu.

Bu dokümanın bir diğer önemli ve belirgin özelliği ise sürekli güncelleniyor olmasıdır. Örnek uygulamalar, konu başlıklarının organizasyonu ve içeriğin kapsamı sürekli güncellenerek geliştirilmektedir.

Söz konusu dokümanın en güncel haline her zaman <http://www.zafercomert.com/medya/ooop.pdf> bağlantısı üzerinden erişim sağlayabilirsiniz.

Doküman içerisinde karşılaştığınız hataları, önereceğiniz düzeltmeleri ve yeni bölüm taleplerini comertzafer@gmail.com eposta adresi üzerinden iletebilirsiniz.

Bu dokümanın sizlere, kariyerinize ve çalışmalarınıza katkı sunmasını umuyorum. Herkese başarılar dilerim.

Zafer CÖMERT

Nesne Yönelimli Programlamaya Genel Bir Bakış

Nesne Yönelimli Programlama (*Object-oriented Programming*) nesneleri odağa alan bir programlama paradigmasıdır. Bu paradigma gerçek hayattaki nesnelerin sınıflar (*class*) aracılığı ile modellenmesine olanak tanır. Sınıflar; alan (*field*), özellik (*property*), metot (*method*), olay (*event*), temsilci (*delegate*) gibi çok farklı üyeler içerebilir. Dersimiz kapsamında tüm bu teknik kavramlar detaylı olarak uygulamalar ile işlenir.

Nesne yönelimli programlamanın en temel ögesi sınıflardır (*class*). Sınıflar sahip olduğu üyeler ile nesnelerin modellenmesine olanak sağlar. Sınıf içerisindeki özellikler kapsüle (*encapsulation*) edilir. Burada soyutlama (*abstraction*) yapılır. Sınıf dışından sınıf üyelerine erişim için erişim düzenleyicileri (*access modifiers*) kullanılır.

Nesne yönelimli programlamanın en temel özelliklerinden biri kalıttır (*inheritance*). Kalıtım yaklaşımıyla hiyerarşik modellemelerin gerçekleştirilmesi sağlanır. Tekrar eden kod blokları engellenir ve genişleyebilir (*extendable*) sınıflar ortaya çıkarmak ya da modellemeler yapmak mümkün hale gelir.

Bir diğer önemli konu arayüz (*Interface*) kavramıdır. Arayüzler ile sınıflar üzerinde çeşitli kurallar ve kısıtlayıcılar kullanılabilir ve uygulama değişkenliklerinden bağımsız kodlamaların yapılması sağlanabilir. Arayüzler hem tasarım örüntülerinin (*design patterns*) uygulanması hem de belirli standartlara sahip projelerin gerçekleştirilebilmesi açısından oldukça önemlidir.

Nesne yönelimli programlamada çok-çeşitlilik (*polymorphism*) olarak ifade edilen ve farklı nesne türleri için tanımlı olan işlevlerin farklı şekillerde çalışmasına olanak tanıyan yapıların kurgulanmasında yine arayüzlerin önemi belirgindir. Gevşek bağlı sistemler (*loosely*

coupled systems) oluşturmak üzere; nesne yönelimli programlama da yine hem kalıtım hem de arayüzlerden faydalanılır.

Bu ders kapsamında nesne yönelimli programlama öğretilirken; yazılım mühendisliğinde daha esnek, daha anlaşılır ve daha sürdürülebilir yazılımlar üretmek üzere kullanılan ve beş temel ilkeyi ifade eden *SOLID* prensiplerini de incelenir ve bu prensiplere uygun şekilde geliştirme yapılır.

Nesne yönelimli programlamanın özelliklerinden faydalanabilmek için nesne yönelimli bir programlama dili tercihi yapılmalıdır. Nesne yönelimli başlıca diller arasında Java, C# ve C++ gösterilebilir. Bu eğitim kapsamında C# dili kullanılır. Geliştirme ortamı olarak *Visual Studio* ya da tercihe bağlı olarak *Visual Studio Code* programlarını kullanılabilir.

İçindekiler

Ön Söz	I
1. Hazırlıklar	1
1.1. Visual Studio	1
1.2. Blank Solution	1
1.3. CLI ile Proje Oluşturma	2
1.4. Visual Studio ile Proje Oluşturma	3
1.5. C# Dilinin Temelleri	4
Özet	4
2. Nesne Yönelimli Programlamaya Giriş	5
2.1. Class	5
2.2. Field ve Property Tanımı	6
2.3. Expression Bodied Property Accessors	6
2.4. Auto-implemented Properties	7
2.5. Constructor	7
2.6. Override	8
2.7. Alignment Component	8
2.8. Metot Tanımlama	8
2.9. params Anahtar Kelimesi	9
2.10. Expression-bodied-method	9
Özet	10
Kodlama Pratikleri	10

3. Kalıtım (Inheritance).....	12
3.1. virtual Anahtar Kelimesi	13
3.2. Çok-biçimlilik (Polymorphism)	13
3.3. Çok-seviyeli kalıtım (Multi-level Inheritance).....	14
3.4. Kalıtım ile Alınmış Metodu Gizleme (Hiding Methods).....	15
3.5. base Anahtar Sözcüğü.....	16
3.6. Mühürlenmiş (Sealed) Sınıflar ve Metotlar	16
3.6. Soyut (Abstract) Sınıflar ve Metotlar	17
3.7. as ve is Operatörleri.....	17
Özet	19
4. Arayüz (Interface).....	20
4.1. Interface ile abstract class karşılaştırması	20
4.2. Interface Tanımlama (Defination)	21
4.3. Interface Uygulama (Implementation).....	21
4.4. IFormattable Interface Implementation.....	21
4.5. ICloneable Interface Implementation.....	23
4.6. IComparable Interface Implementation	23
4.7. IComparable<T> Interface Implementation	24
4.8. IEnumerator Interface Implementation.....	25
4.9. IEnumerable Interface Implementation.....	25
4.10. Dependency Injection ve Loosely Coupled Applications	26
4.11. Interface Segregation Principle.....	26
5. Generic.....	28

5.1.	Genel Özellikler.....	28
5.2.	Performans.....	29
5.3.	Tip Güvenliği (Type Safety)	30
5.4.	Generic Tanımların Özellikleri	30
5.5.	Defaults.....	30
5.6.	Constraints	31
5.7.	Generic Methods.....	31
5.8.	Generic Inheritance	32
5.9.	Generic Interface	32
5.10.	Generic Repository Design Pattern	32
6.	ADO.NET.....	34
6.1.	SQL Console Application	35
6.2.	Data Access Layer	37
7.	Entity Framework Core.....	41
7.1.	Entity Framework Core Genel Bakış.....	41
7.2.	Kurulum	43
7.3.	Entity Tanımı	44
7.4.	Migrations	45
7.5.	EF Core Özellikleri.....	45
7.5.1.	Entity Property Constraints	45
7.5.2.	Entity Schema Attribute	46
7.6.	Entity Framework Core içerisinde ilişkiler	46
7.6.1.	EF Core ilişkin Terimlerin Açıklanması.....	47

7.6.2.	Bire-çok ilişki (one-to-many relations)	48
7.6.3.	Bire-bir ilişki (one-to-one Relation)	50
7.6.4.	Çoka-çok ilişki (Many-to-many relation)	53
7.7.	Scaffold-DbContext	56
7.8.	Uygulamalar (1)	57
7.9.	Uygulamalar (2)	59
8.	Dile Entegre Sorgular (LINQ)	62
8.1.	Giriş	62
8.2.	LINQ Sorguları.....	63
8.3.	Bir sorgu işleminin üç parçası.....	64
8.3.1.	Veri Kaynağı (Data source).....	65
8.3.2.	Sorgu (Query).....	67
8.3.3.	Sorgu ifadelerine genel bakış.....	67
8.3.4.	Sorgu Yürütme (Query execution)	68
8.4.	LINQ Operatörleri	70
8.5.	LINQ Sorgularına Karşılık Üretilen SQL Cümlelerinin Elde Edilmesi	70
8.6.	Select Operatörü	71
8.6.1.	Select ile sadece bir alanı seçme.....	71
8.6.2.	Select ile birden fazla alanı seçme	72
8.6.3.	Anonim nesne oluşturma	73
8.6.4.	Northwind Veri Tabanı için Seçme Sorguları	74
8.7.	SelectMany Operatörü.....	77
8.8.	Filtreleme Sorguları	78

8.8.1.	Filtreleme Fonksiyonları	78
8.8.2.	OfType Operatörü	80
8.8.3.	<i>Northwind veri tabanı için örnekler</i>	81
8.9.	Birleştirme (Join) Sorguları	83
8.10.	Set Operatörü	86
9.	Aspect Oriented Programming (AOP).....	88
9.1.	AOP Genel Bakış.....	88
9.2.	Proxy Tasarım Kalıbı.....	90
9.3.	DynamicProxy Generator	92
9.4.	Invocation	95
9.5.	Defensive Programming	98
9.6.	Aspect tanımlarının Attribute olarak yapılması	100
9.7.	MethodInterception.....	102
9.8.	IInterceptorSelector	105
9.9.	Using Aspect with Autofac Module	106

1. Hazırlıklar

Bu bölümde nesne yönelimli programlamaya giriş yapıyoruz. Eğitim boyunca ihtiyaç duyulan programları ve araçları hazırlıyoruz.



Nesne Yönelimli Programlama Giriş [1.19]

Nesne yönelimli programlama derslerimize bu video ile giriş yapıyoruz.

1.1. Visual Studio

Visual Studio Android, iOS, Mac, Windows, web ve bulut için uygulamalar geliştirmenizi sağlayan bir tümleşik geliştirme (**Integrated Development Environment, (IDE)**) ortamıdır. IDE'ler yazılım başlangıç aşamasından dağıtım ve bakım süreçlerine kadar ihtiyaç duyulan tüm araçlar ile birlikte gelir.

Visual Studio ile

- Akıllı kod tamamlama özelliği ile kodunuzu hızla yazabilirsiniz.
- Kolayca hata ayıklayıp ve tanımlamalar yapabilirsiniz.
- Yazılımınızı test edip, güvenle dağıtabilirsiniz.
- İstediğiniz gibi özelleştirebilirsiniz.
- Verimli bir şekilde işbirliği yapabilirsiniz.
- Git ya da GitHub gibi kaynak kod yönetim sistemleri ile proje kaynak kod yönetiminizi gerçekleştirebilirsiniz.



Visual Studio Kurulumu ve Yönetimi [4:34]

Visual Studio ve Visual Studio Installer hakkında daha fazla bilgi edinin.

1.2. Blank Solution

Solution kavramı Visual Studio programının bir yer tutucu (**container**) olarak düşünülebilir. Solution bir veya daha fazla ilgili projeyi düzenlemek için Visual Studio tarafından kullanılan

kapsayıcılarıdır. Visual Studio'da bir **Solution** açtığınızda, Solution içerisinde yer alan tüm projeler otomatik olarak yüklenir.

Benzer şekilde Boş bir Solution açıp projelerinizin organizasyonunu bu şablon altında gerçekleştirebilirsiniz.



Blank Solution [2:41]

Visual Studio'nun Start Window ekranını keşfedin ve boş bir Solution oluşturun.

1.3. CLI ile Proje Oluşturma

CLI aşağıdaki ifadelerin kısaltması olarak kullanılmaktadır:

- Command Line Interface
- Command Line Interpreter
- Command Line Input

Visual Studio geliştiriciler için iki komut satırı kabuğu içermektedir. Bunlardan biri **Developer Command Prompt** ve diğeri **Developer PowerShell**'dir. Şekil 1 ve Şekil 2'de sırasıyla ilgili komut satır ara yüzlerine yer verilmiştir.

```
C:\WINDOWS\system32\cmd.exe
*** Visual Studio 2019 Developer Command Prompt v16.11.2 ***
*** Copyright (c) 2021 Microsoft Corporation ***

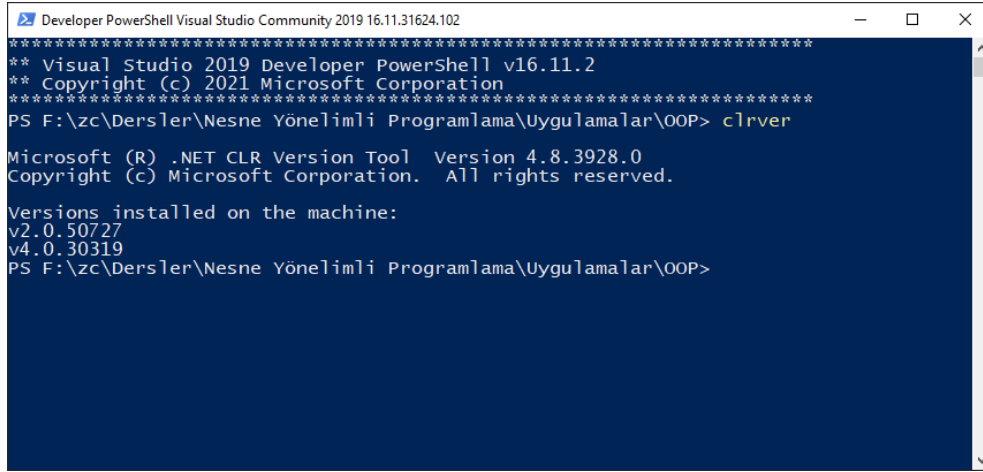
F:\zc\Dersler\Nesne Yönelimli Programlama\Uygulamalar\OOP>clrver

Microsoft (R) .NET CLR Version Tool Version 4.8.3928.0
Copyright (c) Microsoft Corporation. All rights reserved.

Versions installed on the machine:
v2.0.50727
v4.0.30319

F:\zc\Dersler\Nesne Yönelimli Programlama\Uygulamalar\OOP>
```

Şekil 1.1. Developer Command Prompt



```
Developer PowerShell Visual Studio Community 2019 16.11.31624.102
*****
** Visual Studio 2019 Developer PowerShell v16.11.2
** Copyright (c) 2021 Microsoft Corporation
*****
PS F:\zc\Dersler\Nesne Yönelimli Programlama\Uygulamalar\OOP> clrver

Microsoft (R) .NET CLR Version Tool  Version 4.8.3928.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Versions installed on the machine:
v2.0.50727
v4.0.30319
PS F:\zc\Dersler\Nesne Yönelimli Programlama\Uygulamalar\OOP>
```

Şekil 1.2. Developer PowerShell

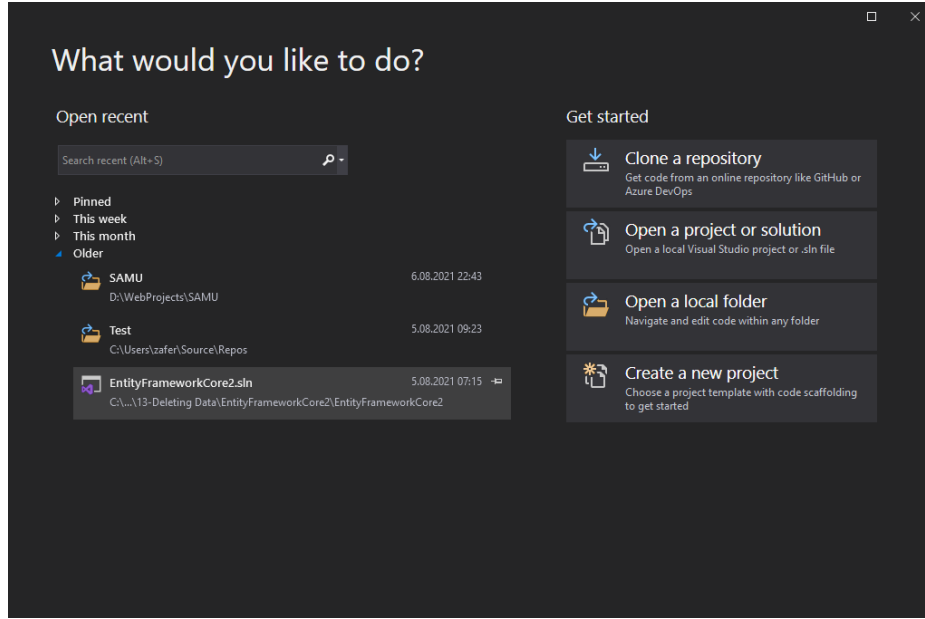


Command Line Interface (CLI) ile Proje Oluşturma [11:56]

Command Prompt üzerinden dotnet ile proje oluşturmak konusunda detaylı bilgi almak için tıklayınız.

1.4. Visual Studio ile Proje Oluşturma

Bir önceki bölümde **CLI** ile **dotnet** komutunu kullanarak proje oluşturmayı öğrendiniz. CLI ile işlem yapabilme genellikle Linux tabanlı sistemleri kullanan kullanıcılar tarafından çok sıcak karşılanırken; Windows işletim sistemi kullanıcıları tarafından gelen kullanıcılar için ise tersi bir durum söz konusudur. Bununla birlikte Visual Studio programının ara yüzü ile proje açmak, yeni proje oluşturmak ve daha fazlasını yapabilmek mümkündür. Şekil 1.3'de **Start Window** ekranına yer verilmiştir.



Şekil 1.3. Start Window

**Visual Studio ile Proje Oluşturma [3:53]**

Visual Studio ile farklı sistemler üzerinde çalışabilecek, farklı türlerde projeleri hali hazırda sunulan şablon aracılığı ile kolayca gerçekleştirebilirsiniz.

1.5. C# Dilinin Temelleri

Eğitim içeriğinde nesne yönelimli programlama konusu işlenirken C# dili kullanılır. Bu nedenle C# ile oluşan kod bloğunu anlamak, bağımlılıkları kavramak, proje referanslarını yönetebilmek, projeyi **Build** ya da **Debug** etmek gibi temel işlevlerin yerine getirilebilmesi önemli bir konudur. Bu kapsamda C# dilinin temellerinin kavranması bir ilerleyen bölümlerin anlaşılması açısından zorunlu bir adım olarak değerlendirilmelidir.

**C# Dilinin Temelleri [5:27]**

Visual Studio ile farklı sistemler üzerinde çalışabilecek, farklı türlerde projeleri hali hazırda sunulan şablon aracılığı ile kolayca gerçekleştirebilirsiniz.

Özet

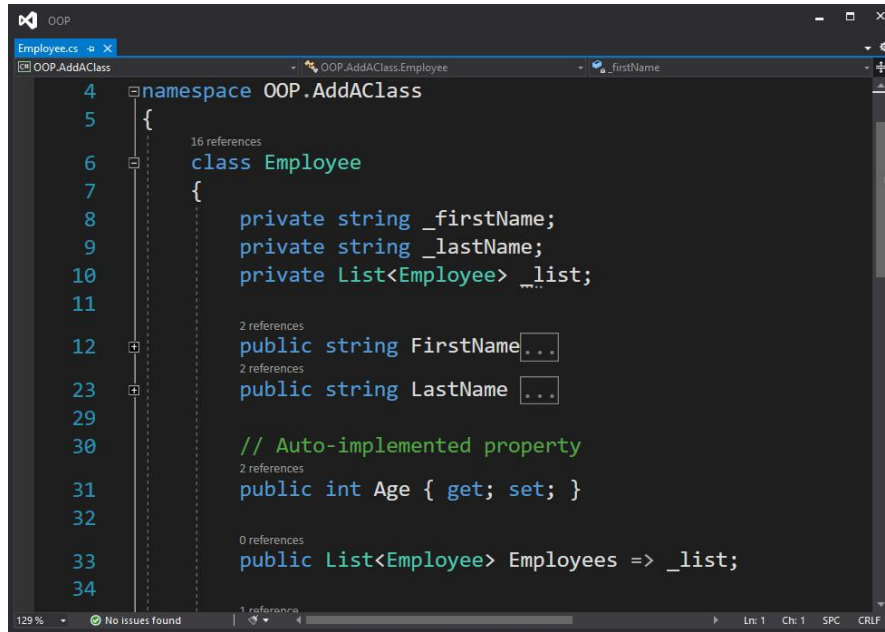
Bu bölümde Nesne Yönelimli Programlamayı Giriş yapabilmek için bir takım ön hazırlıklar yerine getirildi. İhtiyaç duyulan araçların kurulumu gerçekleştirildi ve ilgili programların ara yüzlerine ilişkin temel kazanımlar sağlandı.

2. Nesne Yönelimli Programlamaya Giriş

Bu bölümde nesne yönelimli programlamaya giriş yapılır. Sınıf (*class*) ekleme, sınıf içerisinde çeşitli üyeler tanımlama işlemleri gerçekleştirilir. Örneğin bir alan (*field*) ve özellik (*property*) tanımlama, yapılandırıcı metot (*constructor*) tasarımı, bir metot tanımlama, bir metodun geçersiz kılınması (*override*) ve formatlı çıktı oluşturma, gibi temel konular dikkate alınır.

2.1. Class

Nesne yönelimli programlamanın en temel birimi sınıflar (*class*)'dır. Sınıflar; alan (*field*), özellik (*property*), metot (*method*), temsilci (*delegate*), operatör (*operator*), olay (*event*) gibi pek çok farklı üye tanımları içerebilirler. Şekil 2.1'de örnek bir sınıf tanımına yer verilmiştir.



```
4 namespace OOP.AddAClass
5 {
6     16 references
7     class Employee
8     {
9         private string _firstName;
10        private string _lastName;
11        private List<Employee> _list;
12
13        2 references
14        public string FirstName...
15        2 references
16        public string LastName...
17
18        // Auto-implemented property
19        2 references
20        public int Age { get; set; }
21
22        0 references
23        public List<Employee> Employees => _list;
24    }
25 }
```

Şekil 2.1 Sınıf tanımı



Sınıf Tanımı [5:39]

Employee Sınıfının nasıl tasarlandığına bakınız. Alternatif sınıf tanımlama yöntemlerini keşfedin.

2.2. Field ve Property Tanımı

Alan (*field*), doğrudan bir sınıf veya yapı içinde bildirilen herhangi bir türdeki bir değişkendir. Alanlar, içerdikleri tipin üyeleridir. Bir sınıf (*class*) veya yapı (*struct*), örnek alanlara, statik alanlara veya her ikisine birden sahip olabilir. *private* erişimcisi ile tanımlanmış olan bir alanın depoladığı değer *public* olarak tanımlanmış bir özellik tarafından kullanılabilir.

Alan, doğrudan bir sınıf veya yapı içinde bildirilen bir değişkendir. Özellik (*property*), özel bir alanın değerini okumak, yazmak veya hesaplamak için esnek bir mekanizma sağlayan bir üyedir. Özellikler, genel veri üyeleriymiş (*public*) gibi kullanılabilir, ancak aslında erişimciler (*accessor*) adı verilen özel yöntemlere sahiptirler.

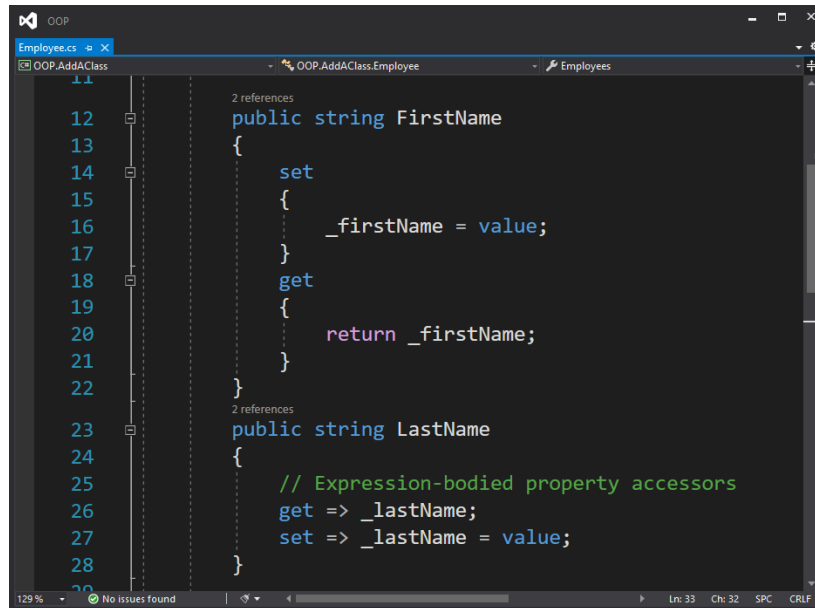


Field ve Property Tanımlama [11:09]

Alan ve özellik nesne yönelimli programlamanın temel üyeleridir.

2.3. Expression Bodied Property Accessors

get accessor ifadesi herhangi bir parametre almaz ve deklare edilmiş olan tip ile geri dönüş yapmalıdır. *set* erişimcisi için de herhangi bir açık parametre belirtilmez, ancak derleyici yine aynı tipte olan ve değer olarak adlandırılan bir parametre aldığını varsayar. Şekil 2.2'de örnek bir tanıma yer verilmiştir.



Şekil 2.2 Expression bodied property accessor

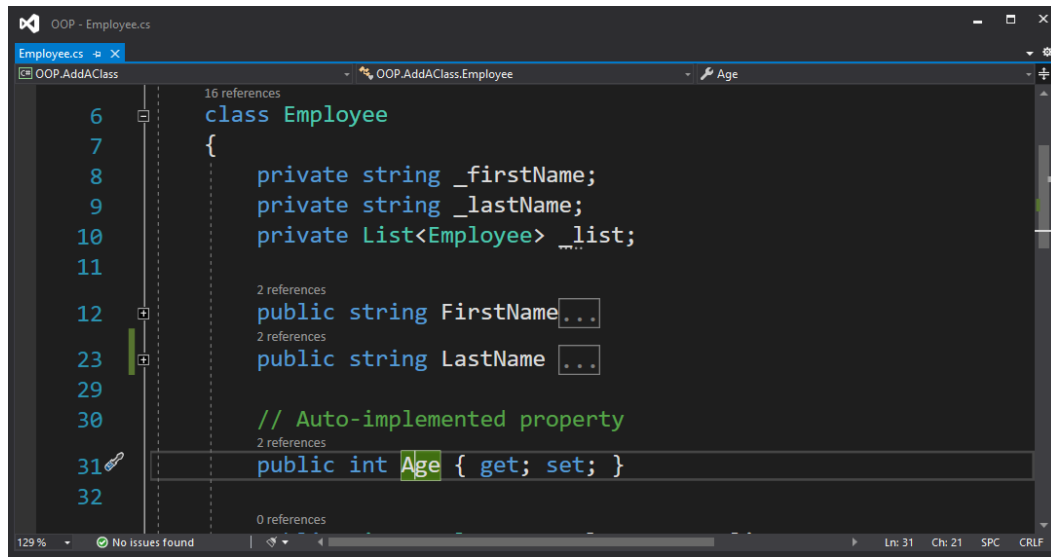


Expression bodied property accessors [03:22]

C# dilinin değişen esnek yapısını keşfedin.

2.4. Auto-implemented Properties

Eğer *set* ve *get* erişimcileri içerisinde herhangi bir *logic* ifadesi yer almayacak ise, *auto-implemented property* özelliği kullanılabilir. Bu özellikler herhangi bir alan tanımına ihtiyaç duymaksızın üyenin değerinin otomatik olarak geri dönecek şekilde uygulanmasını sağlar. Şekil 2.3'da örnek bir tanıma yer verilmiştir.



Şekil 2.3 Auto-implemented property



Auto-implemented property [02:35]

Özellik tanımlamanın alternatif bir yolunu keşfedin.

2.5. Constructor

Nesne yönelimli programlamada, bir yapıcı (*constructor*), bir nesne oluşturmak için çağrılan özel bir alt yordam türüdür. Yeni nesneyi kullanıma hazırlar, genellikle yapıcının gerekli üye değişkenlerini ayarlamak için kullandığı argümanları kabul eder.



Constructor [09:15]

Bir nesnenin kullanıma hazır hale getirilmesini sağlayan özel alt yordamdır.

2.6. Override

Geçersiz kılma (*override*), bir alt sınıfın, üst sınıfında veya üst sınıflarından birinde önceden tanımlanmış ve/veya uygulanmış bir yöntem için farklı uygulama sağlamasını olanak tanıyan nesne yönelimli bir programlama özelliğidir. *Override*, tek tip bir *interface* aracılığıyla farklı veri türlerinin işlenmesini sağlar. Kalıtım konusunda daha detaylı olarak bu anahtar sözcük incelenmektedir.



Override [10:03]

Override, nesne yönelimli programlamanın önemli bir anahtar kelimesidir.

2.7. Alignment Component

Hizalama bileşeni (*alignment component*), tercih edilen biçimlendirilmiş alan genişliğini gösteren işaretli bir tamsayıdır. Hizalama değeri, biçimlendirilmiş dizinin uzunluğundan küçükse, hizalama yok sayılır ve biçimlendirilmiş dizinin uzunluğu, alan genişliği olarak kullanılır. Alandaki biçimlendirilmiş veriler, hizalama pozitifse sağa hizalanır ve hizalama negatifse sola hizalanır. Dolgu gerekliyse beyaz boşluk kullanılır. Hizalama belirtilmişse virgül gereklidir.



Alignment Component [08:58]

Formatlı çıktı üretmek üzere hizalama bileşeni kullanımı hakkında tanımlanan örneği izlemek için tıklayınız.

Formatlı gösterimler hakkında daha fazla bilgi edinmek için <https://docs.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting> adresini ziyaret edebilirsiniz.

2.8. Metot Tanımlama

Metot ya da yöntem yalnızca çağrıldığında çalışan bir programlama bileşenidir. Parametre olarak bilinen veriler bir yönteme geçirilebilir. Yöntemler, belirli eylemleri gerçekleştirmek için kullanılır ve bunlara işlevler de denir.

Neden yöntemler kullanılır?

Kodu yeniden kullanmak için yöntemler kullanılır. Metot içerisinde kod bir kez tanımlanır ve birçok kez kullanılır.



Metot Tanımlama [10:38]

Metot söz dizilimi hakkında detaylı bilgi almak için videoyu izleyebilirsiniz.

2.9. params Anahtar Kelimesi

C#'daki bir yöntem, yöntem bildiriminde bir dizi bağımsız değişkeni ve türlerini tanımlar. Bir yöntem çağrıldığında, isteğe bağlı parametreler olmadığı sürece, çağırıcı kod tarafından iletilen argümanların (parametrelerin) aynı sayıda ve türde olmasını bekler.

Ya argüman sayısından emin değilseniz?

"**params**" anahtar sözcüğünün yararlı olduğu yer burasıdır. C#'daki "**params**" anahtar sözcüğü, bir yöntemin değişken sayıda argümanı kabul etmesine izin verir. C# parametreleri bir dizi nesne olarak çalışır. Bir yöntem argümanı tanımında **params** anahtar sözcüğünü kullanarak, bir dizi argüman yönteme iletilebilir.



params Anahtar Sözcüğü [04:07]

Bir metoda değişken sayıda argüman iletmek üzere **params** anahtar sözcüğü kullanılır.

2.10. Expression-bodied-method

İfade gövdeli bir yöntem (expression-bodied method) türü yöntemin dönüş türüyle eşleşen bir değer döndüren veya **void** döndüren yöntemler için bazı işlemler gerçekleştiren tek bir ifadeden oluşur. Örneğin, **ToString()** yöntemini geçersiz kılan türler, tipik olarak, geçerli nesnenin dize temsili döndüren tek bir ifade içerir.



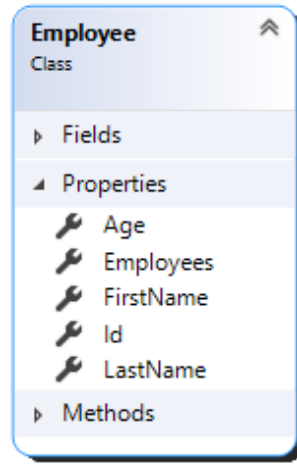
Expression-bodied method [05:31]

İfade-gövdeli bir yöntem örneğini bir video üzerinde birlikte inceleyelim.

Özet

Nesne yönelimli programlama paradigmasının en temel bileşeni sınıf kavramıdır. Sınıflar çok farklı üyeler içerirler. Bu bölümde bir sınıfı tanımlarken kullanılan alan (**field**), özellik (**property**), metot (**method**), yapılandırıcı metot (**constructor**) gibi temel kavramları inceledik.

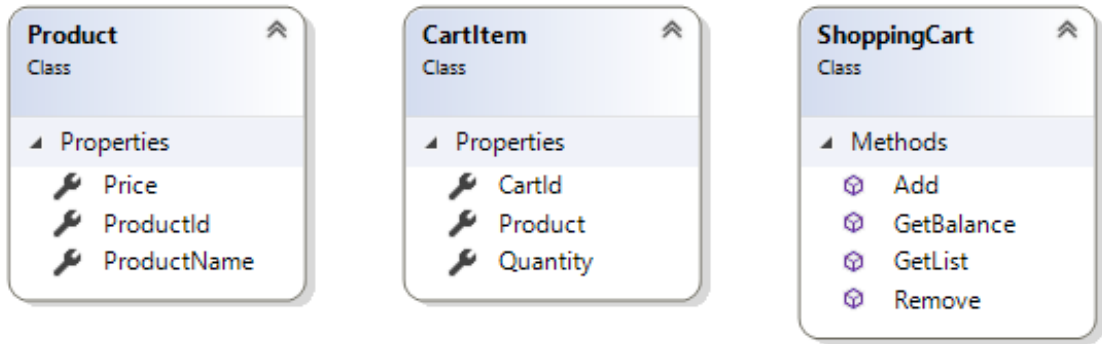
Kodlama Pratikleri



Şekil 2.4 Employee Sınıfı

1. *Employee* dizisi:
 - a. Şekil 2.4’de verilen *Employee* sınıfını tanımlayınız.
 - b. *Employee* sınıf tipinde bir dizi (*Array*) tanımlayınız.
 - c. İlgili diziye rastgele 10 adet eleman ekleyiniz.
 - d. Tüm elemanları property değerlerini de gösterecek şekilde ekrana yazdırınız.
 - e. Dizi üzerinde en yüksek yaşa sahip olan elemanı bulan bir metot tasarımı yapınız.
2. *Employee* listesi:
 - a. *List<Employee>* şeklinde bir liste tanımlayınız.
 - b. Listeye rastgele 10 eleman ekleyiniz.
 - c. Listedeki tüm elemanları ekrana yazdırınız.
 - d. Listedeki tüm elemanların yaşlarına göre büyükten küçüğe doğru sıralayan bir kod parçası yazınız.

- e. Listedeki tüm elemanları yaşlarına göre küçükten büyüğe doğru sıralayan bir kod parçasığı yazınız.
3. *Employee* koleksiyonu: *Employee* sınıfı içerisinde herhangi bir veri yapısını kullanarak aşağıdaki işlevleri gerçekleştirecek şekilde bir sınıf tasarımı yapınız:
 - a. *void Add(Employee)* sınıfa yeni *Employee* ekleyecek.
 - b. *void Remove(int)* Id değerini parametre olarak alıp; eğer ilgili *Id* sınıf içerisinde tanımlı ise onu sınıftaki listeden kaldıran; tanımlı değil ise hata fırlatan bir metot tasarımı yapmanız beklenmektedir.
4. Aşağıdaki sınıf diyagramını inceleyiniz:



Şekil 2.5 Shopping Cart uygulaması

Yukarıdaki projede *Product* sınıfı ile ürün tanımı yapılmaktadır. İlgili ürünlerden kaç adet alındığı bilgisi *CartItem* sınıfı ile tanımlanır. *ShoppingCart* ise *CartItem* nesnelerinin organize edilmesini sağlamaktadır. *GetList()* metodu alışveriş listesinde yer alan her bir *CartItem* ifadesinin listesini dönmelidir. *GetBalance()* metodu ise ürün adet (*Quantity*) ve ürün fiyat (*Price*) bilgilerine bağlı olarak; sepet tutarının toplamı yansıtacak şekilde hesaplama yapmalıdır. Böylelikle bir alışveriş sepeti uygulaması gerçekleştirilir.

Projeye ilişkin tüm eksiklerini giderip çalışır hale getiriniz ve örnek bir uygulama üzerinde (*Console* ya da *Windows Form*) çalıştırınız.

3. Kalıtım (Inheritance)

Kalıtım (*Inheritance*), nesne-yönelimli programlamanın önemli bir bileşenidir. Kalıtım ile bir sınıfı genişletmeden ve özelleştirmeden önce genel özelliklerin tanımı bir temel sınıf içerisinde yapılabilir. Böylelikle genel ifadelerin özelleştirilmiş sınıflar içerisinde tekrar etmesi engellenebilir. Bu şekilde bir sınıfın başka bir sınıfın özelliklerini (alanlarını, yöntemlerini ve diğer sınıf üyelerini) devralmasına izin verilen mekanizma kalıtım olarak tanımlanır.

Kalıtım ile devir edilme ya da kalıtma olarak ifade edilen işlev gerçekleştirilirken; özel olarak temel sınıftaki üyelerin geçersiz kılınabilmesi (*overriding*), gizlenmesi (*hidding*) ya da mühürlenmesi (*sealed*) gerçekleştirilebilir.

Kalıtım aynı zamanda çok-biçimliliğin (*polymorphism*) uygulanmasına olanak tanır.

Süper Sınıf (*Base class*): Özellikleri miras alınan sınıf, süper sınıf (veya temel sınıf (*base class*) veya üst sınıf) olarak bilinir.

Alt Sınıf (*Subclass*): Diğer sınıfı miras alan sınıf, alt sınıf (veya türetilmiş sınıf, genişletilmiş sınıf veya alt sınıf) olarak bilinir. Alt sınıf, üst sınıf alanlarına ve yöntemlerine ek olarak kendi alanlarını ve yöntemlerini ekleyebilir.

Yeniden Kullanılabilirlik (*Reusability*): Kalıtım, “yeniden kullanılabilirlik” kavramını destekler, yani yeni bir sınıf oluşturmak istediğimizde ve zaten istediğimiz kodun bir kısmını içeren bir sınıf varsa, yeni sınıfımızı mevcut sınıftan türetebilir. Bunu yapılarak, mevcut sınıfın alanlarını ve yöntemleri yeniden kullanılır.

C# dilinde birden fazla tipte kalıtım yapılabilir

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Interface inheritance

olarak özetlenebilir.

**Kalıtım (Inheritance) [11:16]**

Nesne yönelimli programlamanın önemli bir özelliği olan kalıtımı inceleyelim.

3.1. virtual Anahtar Kelimesi

virtual olarak bildirilmiş bir temel sınıf (*base class*) metodu, temel sınıftan türetilen sınıflarda *override* (ezilebilir, geçersiz kılınabilir) edilebilir, bu izin ilgili metod için verilir.

virtual anahtar kelimesi metotlar için kullanılabildiği gibi, özellikler için de kullanılabilir.

Temel sınıfta *virtual* anahtar kelimesiyle tanımlanmış bir metot, türetilen sınıfta ezildiği zaman (*override*) mutlaka metot imzasına, yani metodun sahip olduğu parametrelere, isime ve dönüş türüne, dikkat edilmeli ve bunlar tam anlamıyla birbiriyle örtüşmelidir.

Ne alanlar (*fields*) ne de *static* fonksiyonlar *virtual* olarak deklare edilemezler.

virtual anahtar sözcüğü, temel sınıfta bildirilen bir yöntemi, özelliği, dizin oluşturucuyu veya olayı değiştirmek ve türetilmiş sınıfta geçersiz kılınmasına izin vermek için kullanılır.

override anahtar sözcüğü, bir sanal/soyut yöntemi, özelliği, dizin oluşturucuyu veya temel sınıfın olayını türetilmiş sınıfa genişletmek veya değiştirmek için kullanılır.

new anahtar sözcük, bir yöntemi, özelliği, dizin oluşturucuyu veya temel sınıfın olayını türetilmiş sınıfa gizlemek için kullanılır.

**virtual Anahtar Kelimesi [07:22]**

Türetilmiş sınıfta geçersiz kılma.

3.2. Çok-biçimlilik (Polymorphism)

Polimorfizm, "bir isim birçok form" anlamına gelen Yunanca bir kelimedir. Başka bir deyişle, bir nesnenin birçok biçimi vardır veya birden çok işleve sahip bir adı vardır. "**Poly**" çok, "**morphism**" ise biçimler anlamına gelir. Polimorfizm, bir sınıfa aynı ada sahip birden çok uygulamaya sahip olma yeteneği sağlar. Kapsülleme ve kalıttan sonra Nesne Yönelimli Programlamanın temel ilkelerinden biridir.

Polimorfizmde, çağrılan yöntem derleme zamanı sırasında değil, dinamik olarak tanımlanır. Derleyici bir sanal metot tablosu (*vtable*) oluşturur ki bu tablo çalışma zamanı sırasında çağrılacak metotların bir listesini içerir ve derleyici çalışma zamanında *type* (tip) dikkate alarak ilgili metotları çağırır.



Polymorphism [05:16]

Polimorfizm, bir sınıfa aynı ada sahip birden çok uygulamaya sahip olma yeteneği sağlar.

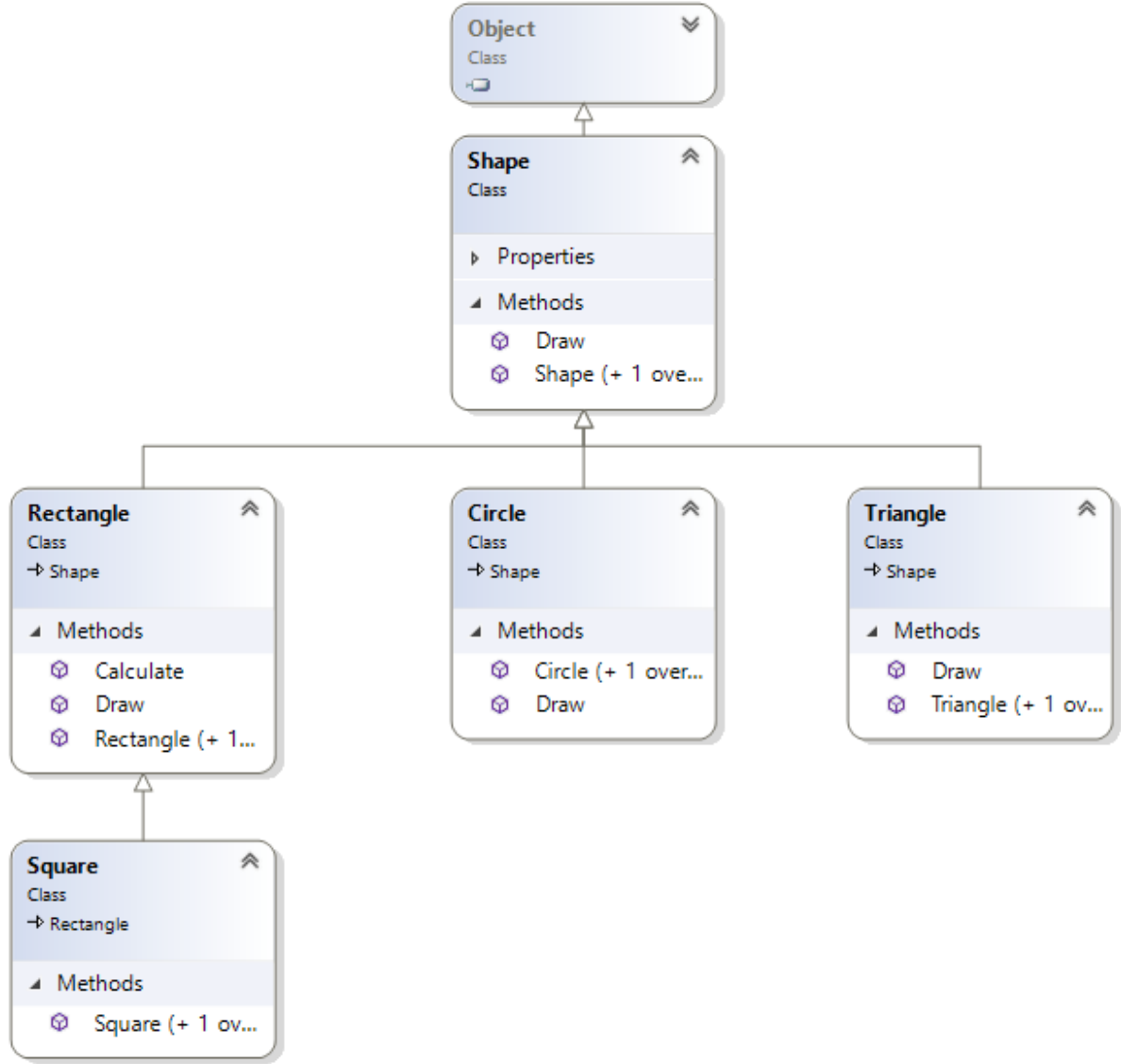
3.3. Çok-seviyeli kalıtım (Multi-level Inheritance)

Bir sınıf, başka bir sınıftan türetilmiş olan bir sınıftan da türeyebilir. Bu çok-seviyeli kalıtım (*multi-level inheritance*) olarak ifade edilir. Şekil 3.1'de *Rectangle* ifadesi *Shape* sınıfından türetilmiş iken; *Square* sınıfı ile *Rectangle* sınıfından türetilmiştir.



Multilevel Inheritance [05:46]

Polimorfizm, bir sınıfa aynı ada sahip birden çok uygulamaya sahip olma yeteneği sağlar.



Şekil 3.1. Multilevel inheritance örneği

3.4. Kalıtım ile Alınmış Metodu Gizleme (Hiding Methods)

C#'ta polimorfizm (*polymorphism*) ve metot geçersiz kılma (*overriding*) hakkında yeterli bilimiz var. Bu doğrultuda benzer bir işlev olarak C#, temel sınıf yöntemlerini türetilmiş sınıftan gizlemek için bir kavram sağlar, bu kavram **"yöntem gizleme (hiding method)"** ya da **"yöntem gölgeleme (method shadowing)"** olarak ifade edilir. Yöntem gizlemede, **new** anahtar sözcüğü kullanılarak bir temel sınıfın yöntemlerinin uygulanması türetilmiş sınıftan gizlenebilir. Alternatif olarak, yöntem gizlemede, türetilmiş sınıftaki temel sınıfın yöntemini **new** anahtar sözcüğü ile tanımlayarak ilgili metodun yeniden yazılması da sağlanabilir. Bu metodu geçersiz kılmanın bir başka yöntemidir.

**Hiding method [04:39]**

Temel sınıftaki bir metodun geçersiz kılınmasını sağlamanın alternatif bir yolu da **new** anahtar sözcüğü ile yöntemin gölgelemesidir.

3.5. base Anahtar Sözcüğü

base anahtar sözcüğü, türetilmiş bir sınıf içinden temel sınıfın üyelerine erişmek için kullanılır:

- Başka bir yöntemle geçersiz kılınan temel sınıfta bir yöntemin çağırması sağlamak üzere kullanılabilir.
- Türetilmiş sınıfın örneklerini oluştururken hangi temel sınıf oluşturucusunun (*constructor*) çağrılacağını da **base** anahtar kelimesi ile belirtilebilir.

Bir temel sınıf erişimine yalnızca bir kurucuda, yani yapılandırıcı metotta, bir örnek yönteminde veya bir örnek özellik erişimcisinde izin verilir.

base anahtar sözcüğü şayet *static* bir yöntem içinden kullanılmak istenirse hataya neden olacaktır.

**Passing Parameter with Constructor [09:47]**

Kod şişkinliğinin önüne geçen ve parametre ayarlanması için pratik bir yaklaşım olan **base** anahtar sözcüğünü keşfedin.

3.6. Mühürlenmiş (Sealed) Sınıflar ve Metotlar

Mühürlü sınıflar (*sealed classes*), nesne yönelimli programlamanın kalıtım özelliğini kısıtlamak için kullanılır. Bir sınıf bir kez mühürlü sınıf olarak tanımlandığında, bu sınıf miras alınamaz, yani bu sınıf kalıtım amaçlı kullanılamaz. Benzer bir durum metot tanımları içinde geçerlidir. Yani bir metot *sealed* olarak tanımlanır ise geçersiz kılınması (*overriding*) mümkün olmayacaktır.

**sealed classes and methods [05:20]**

Sınıf ya da metot mühürleme, kalıtımı kısıtlamak üzere kullanılan *sealed* anahtar kelimesini inceliyoruz.

3.6. Soyut (Abstract) Sınıflar ve Metotlar

C# hem metot hem de sınıfların soyut (*abstract*) olarak tanımlanmasına izin verir. Eğer bir sınıf *abstract* olarak tanımlanır ise o sınıftan *new* anahtar sözcüğü ile nesne türetmek mümkün olmaz. Ancak *abstract* sınıfı kalıtımla miras alan bir sınıf üzerinden türetme yapılabilir. Yine *abstract* sınıf içerisinde *abstract* metotlar tanımlanabilir. Bir metot *abstract* olarak tanımlanıyor ise sadece metot imzası belirtilir. *Abstract* metotların devir alınan sınıfta *override* anahtar sözcüğü ile geçersiz kılınıp gövdelerinin tanımlanması gerekir.

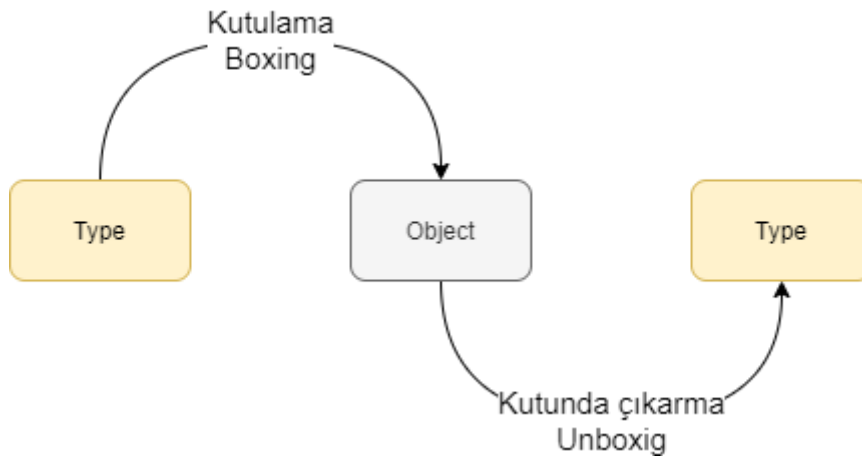


abstract classes and methods [14:28]

Soyut sınıf ya da metot tanımlama nesne yönelimli programlamanın işlevsel özelliklerinden biridir.

3.7. as ve is Operatörleri

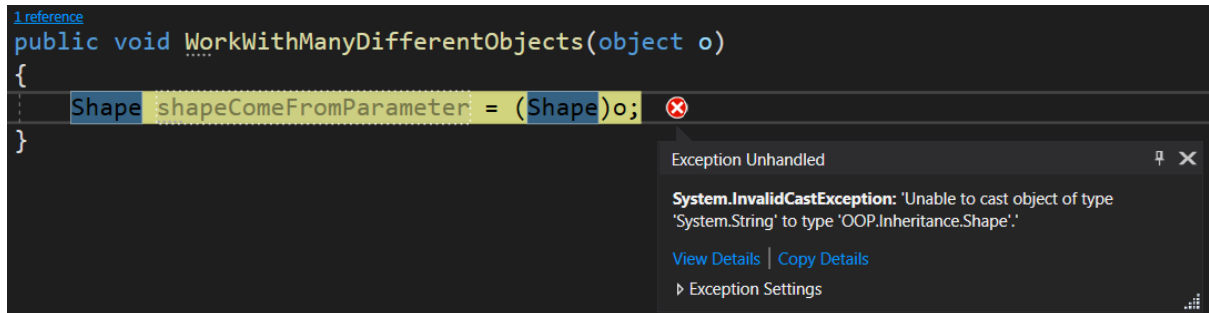
Kalıtım ile ilgili iki önemli operatör *as* ve *is*'dir. Daha önceki yaptığımız uygulamalardan biliyoruz ki herhangi bir tip kutularak *object* türüne çevrilebilir; benzer şekilde *object* türü içerisinde saklanan tip kutudan çıkarılarak beklenen tipe dönüştürülebilir. Burada yapılan işlemin teknik adı **Cast** olarak ifade edilir. Şekil 3.2'de kutulama ve kutudan çıkarma işlevleri gösterilmektedir.



Şekil 3.2 Kutulama ve kutunda çıkarma

```
public void WorkWithManyDifferentObjects(object o) {
    Shape shapeComeFromParameter = (Shape)o;
}
```

Yukarıdaki kod bloğunda **Cast** işlemi yapılmaktadır. Daha açık bir ifadeyle her şeyi saklayabilen **object** nesnesi içerisinde kutudan çıkarma yapılarak "o" nesnesi **Shape** tipine dönüştürülmektedir. Bu kod parçacığı **object** içerisinde **Shape** sakladığınız sürece sorunsuz çalışacaktır. Ancak, **object** içerisinde **Shape** dışında bir tip saklanması durumunda **InvalidCastException** hatası ile dönüş yapılır. Bu da programın kesintiye uğraması yani hata fırlatması anlamına gelir. Şekil 3.3'de istisna oluşturan örnek bir kod bloğuna yer verilmiştir.



Şekil 3.3 System.InvalidCastException hatası

Kod parçacığının daha güvenli hale getirilmesini sağlamak üzere doğrudan **Cast** işlemi uygulamadan **is** ya da **as** operatörleri kullanılarak daha güvenli kod yazılabilir. "**as**" operatörü sınıf hiyerarşini dikkate alarak **Cast** işlemine benzer bir işlevi yürütür. Nesneye bir referans döner. Ancak asla **InvalidCastException** fırlatmaz. Bunun yerine **null** değer ile dönüş yapar. Tip güvenliğini sağlamak üzere herhangi bir kod bloğu çalıştırmadan önce **null** kontrolü yapılarak devam edilebilir; aksi durumda **NullReferenceException** hatası fırlatılır.

```
public void WorkWithManyDifferentObjects(object o) {
    Shape shapeComeFromParameter = o as Shape;
    if (shapeComeFromParameter != null)
    {
        // work with shapeComeFormParameter
    }
}
```

Kod bloğu yukarıdaki gibi düzenlendiğinde tip dönüşümü başarısız olsa dahi, program kesilmez ve hata fırlatılmaz.

"**as**" operatörünün alternatifi olması amacıyla "**is**" operatörü de kullanılabilir. "**is**" operatörü herhangi bir referans dönmek yerine **true** ya da **false** şeklinde bir dönüş yapılmasını sağlar.

```
public void WorkWithManyDifferentObjects(object o){  
    if (o is Shape)  
    {  
        // work with parameter o  
    }  
}
```

Bu durumda yukarıdaki gibi bir kod bloğu ile güvenli kod yazılarak ilgili metodun gövdesi yazılabilir.



is and as operators [13:47]

Kalıtım ile ilgili iki önemli operatör **as** ve **is**'dir. Daha önceki yaptığımız uygulamalardan biliyoruz ki herhangi bir tip kutulanarak **object** türüne çevrilebilir; benzer şekilde **object** türü içerisinde saklanan tip kutudan çıkarılarak beklenen tipe dönüştürülebilir. Burada yapılan işlemin teknik adı **Cast** olarak ifade edilir. **as** ve **is** operatörler ile **cast** işlemini daha güvenli hale getiriyoruz.

Özet

Kalıtım nesne yönelimli programlama paradigmasıyla beraber gelen oldukça önemli bir özelliktir. Bu bölümde kalıtım ([inheritance](#)) kavramını birlikte inceledik. C# dili kalıtımın farklı şekillerde uygulanmasına olanak vermektedir. Bu kapsamda tek bir sınıftan miras alma ve çok-seviyeli kalıtım türlerini inceledik. Yapılandırıcı metod aracılığıyla [base](#) anahtar kelimesini kullanarak temel sınıfa parametre geçişi yaptık. Yine [virtual](#), [override](#), [new](#), [abstract](#), [sealed](#) gibi anahtar kelimeler ile kalıtımın özelliklerinden farklı şekillerde faydalandık.

4. Arayüz (Interface)

Arayüz (**Interface**), bir nesnenin yapabileceği eylemlerin bir açıklamasıdır. “Örneğin bir ışık düğmesini çevirdiğinizde, ışık yanar, elektriğin ilgili elektronik devreden nasıl geçtiği, anahtarlamanın nasıl olduğu umurunuzda olmaz, ışık düğmesine basarsınız ve ışık yanar.” Nesneye yönelimli programlamada, bir **Interface**, bir nesnenin sahip olması gereken tüm işlevlerin bir açıklamasıdır. Örnek olarak, bir ışık ya da lamba sınıfı için arayüz tanımı yapıyor ise bir `turn_on()` yöntemine ve bir `turn_off()` yöntemine sahip olabilir.

Interface, bilgisayarın bir nesne (sınıf) üzerinde belirli özellikleri zorlamasını sağlayan bir programlama yapısı/sözdizimidir. Bir **interface** yapısından sınıf türetmek, daha önce imzası tanımlanmış olan metotların gövdelerinin yazılması (**implementation**) anlamına gelir. Tüm nesne yönelimli diller **interface** yapısını desteklemez. Java ve C# dillerinde **interface** desteği vardır.

Genel olarak **interface** yapısı sadece metotlar, **properties**, **indexer** ve **events**'ların deklarasyonunu içerir. Bununla birlikte **class** kalıtımı ile **interface** kalıtımı tamamen farklı davranırlar.

4.1. Interface ile abstract class karşılaştırması

Bir **abstract** sınıf implementasyonlar veya implementasyonu olmayan üyeler içerir. Ancak **interface** yapısı, herhangi bir implementasyon içermez, yani tamamen soyuttur. Çünkü **interface** yapısının üyeleri her zaman **abstract** şeklinde tanımlanır. Bununla birlikte **abstract** deyiimi **interface** gerektirmez.

Abstract sınıflarda olduğu gibi **interface** üzerinden türetme, yani nesne üretme, **yapılamaz**. **Interface** yapısı sadece üyelerinin imzalarını taşır.

Interface yapısı ne yapılandırıcı (**constructor**) metoda ne de **field** üyelerine sahip değildir.

Interface yapısı operatör aşırı yükleme (**operator overloading**) yapısına da izin **vermez**. Tabi, bu durum zamanla dil değiştiğinde/evrildikçe değişkenlik gösterebilir.

4.2. Interface Tanımlama (Defination)

Interface tanımlamasında **modifiers** kullanımına da izin verilmez. **Interface** üyeleri her zaman **public** olarak tanımlanır ve **virtual** olarak deklare edilemezler.

Interface tanımlaması yapılırken genellikle **I** deyimi ile başlanır. Aşağıdaki **interface** tanımında bir banka hesabının tanımlanabilmesi için para yatırma (PayIn), para çekme (Withdraw) ve bakiye (Balance) üyelerinin olması gerektiği ifade edilmiştir.

```
public interface IBankAccount
{
    void PayIn(decimal amount);
    bool Withdraw(decimal amount);
    decimal Balance { get; }
}
```



Interface Defination [04:15]

Interface, bilgisayarın bir nesne (sınıf) üzerinde belirli özellikleri zorlamasını sağlayan bir programlama yapısı/sözdizimidir.

4.3. Interface Uygulama (Implementation)

Bir **interface** yapısının uygulanması (**implementation**), ilgili yapı içerisinde metot imzaları tanımlanmış olan metotlara ait gövdelerin yazılması anlamına gelir.



Interface Implementation [12:05]

Interface implementation anlamı Interface içinde imzası tanımlanmış olan property ya da method ifadelerinin gövdelerinin yazılmasına karşılık gelir.

4.4. IFormattable Interface Implementation

Bir nesnenin dize temsilimi biçimlendirmek üzere **IFormattable interface** kontratı kullanılabilir.

Türetildiği sınıflar arasında:

- System.Byte
- System.DateTime
- System.DateTimeOffset
- System.Decimal

gösterilebilir.

```
public override string ToString()
{
    return this.ToString("F", CultureInfo.CurrentCulture);
}

public string ToString(string format)
{
    return this.ToString(format, CultureInfo.CurrentCulture);
}

public string ToString(string format, IFormatProvider formatProvider)
{
    if (String.IsNullOrEmpty(format))
        format = "F";
    if (formatProvider == null)
        formatProvider = CultureInfo.CurrentCulture;
    switch (format.ToUpper())
    {
        case "F":
            return $"{Id,-5} {Title,-20} {FirstName,-15} {LastName,-15}";
        case "FL":
            return $"{FirstName} {LastName}";
        case "LF":
            return $"{LastName} {FirstName}";
        case "S":
            return $"{LastName} {FirstName[0]}. ";
        default:
            throw new FormatException(String.Format("The {0} is not supported.",format));
    }
}
```

Biçim dizesi genellikle bir nesnenin genel görünümünü tanımlar. Örneğin, .NET Framework aşağıdakileri destekler:

- Numaralandırma değerlerini biçimlendirmek için standart biçim dizeleri
- Sayısal değerleri biçimlendirmek için standart ve özel biçim dizeleri
- Tarih ve saat değerlerini biçimlendirmek için standart ve özel biçim dizeleri
- Zaman aralıklarını biçimlendirmek için standart ve özel biçim dizeleri

Uygulama tanımlı türlerinizi biçimlendirmeyi desteklemek için kendi biçim dizelerinizi de tanımlayabilirsiniz.



IFormattable Implementation [23:34]

Bir nesnenin dize temsilimi biçimlendirmek üzere **IFormattable interface** kontratı kullanılabilir.

4.5. ICloneable Interface Implementation

Var olan örnekle aynı değere sahip bir sınıfın yeni bir örneğini oluşturan kopyalamayı destekler.

```
public object Clone(){  
    //// deep copy  
    //return new Book()  
    //{  
    //    Id = this.Id,  
    //    Title = this.Title,  
    //    Price = this.Price  
    //};  
  
    return this.MemberwiseClone(); // shallow  
}
```

ICloneable Interface yapısı, var olan bir nesnenin bir kopyasını oluşturan özelleştirilmiş bir uygulama sağlamanıza olanak sağlar.

ICloneable Interface, *Clone* tarafından sağlananlar dışında kopyalama desteği sağlamaya yönelik yöntemi olan bir üye içerir **Object.MemberwiseClone**. Kopyalama, derinlemesine (**deep copy**) veya basit kopyalar (**shallow copy**) şeklinde gerçekleştirilebilir.



ICloneable Implementation [09:52]

Var olan örnekle aynı değere sahip bir sınıfın yeni bir örneğini oluşturan kopyalamayı destekler.

4.6. IComparable Interface Implementation

Genelleştirilmiş bir türe özgü (**a generalized type-specific comparison**) karşılaştırma yöntemi tanımlamak üzere kullanılır. Tip değer ya da referans tipli olabilir. *IComparable*

interface *implemente* edildiğinde *CompareTo(Object)* metodunun devir alınan sınıflarda gövdesinin yazılması zorunludur.

```
public int CompareTo(object obj){  
    var other = (Employee)obj;  
    if (this.Salary < other.Salary)  
        return -1;  
    else if (this.Salary.Equals(other.Salary))  
        return 0;  
    else  
        return 1;  
}
```



Comparable Implementation [12:42]

Genelleştirilmiş bir türe özgü karşılaştırma yöntemi tanımlamak üzere kullanılır.

4.7. Comparable<T> Interface Implementation

Genelleştirilmiş bir karşılaştırma metodu tanımlamak üzere kullanılır. Söz konusu metot bir değer ya da referans tipli veri yapısı için tanımlanabilir. Temel amaç örnekleri sıralamak üzere tipe özgü bir karşılaştırma metodu yazmaktır.

```
public int CompareTo(Employee other) {  
    // return this.Salary.CompareTo(other.Salary);  
    if (this.Salary < other.Salary)  
    {  
        return 1;  
    }  
    else if (this.Salary == other.Salary)  
        return 0;  
    else  
        return -1;  
}
```



Comparable<T> Implementation [08:19]

Temel amaç örnekleri sıralamak üzere tipe özgü bir karşılaştırma metodu yazmaktır.

4.8. IEnumerator Interface Implementation

Genel olmayan (**non-generic**) bir koleksiyon üzerinde basit bir yinelemeyi destekler.

IEnumerable ve **IEnumerator** interface yapıları beraber kullanılarak iteratif yapılar kurulabilir ya da ilgili veri yapısı üzerinde **foreach** döngülerinin çalıştırılması sağlanabilir.

```
var collection = new List<Product>()
{
    new Product(){ ProductId=1, ProductName="Buz dolabı", Price=8500 },
    new Product(){ ProductId=2, ProductName="Ocak", Price=5800 },
    new Product(){ ProductId=3, ProductName="Kahve Makinesi", Price=3000 },
    new Product(){ ProductId=4, ProductName="Bulaşık Makinesi", Price=4000 },
    new Product(){ ProductId=5, ProductName="Fırın", Price=8250 }
};

foreach (var product in collection)
{
    Console.WriteLine($"{product.ProductId} " +
        $"{product.ProductName} " +
        $"{product.Price}");
}

Console.WriteLine(new string('-',25));

IEnumerator<Product> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    Console.WriteLine($"{enumerator.Current.ProductId} " +
        $"{enumerator.Current.ProductName} " +
        $"{enumerator.Current.Price}");
}
```



IEnumerator Implementation [10:38]

Genel olmayan (non-generic) bir koleksiyon üzerinde basit bir yinelemeyi destekler.

4.9. IEnumerable Interface Implementation

Genel olmayan bir koleksiyon üzerinde basit bir yinelemeyi destekleyen bir

IEnumerator nesnesini kullanılır.

**IEnumerable Implementation [11:09]**

Bir IEnumerator ifadesini sömürerek iteratif bir kurgu yapılmasına olanak sağlar.

**IEnumerable Custom Design [15:11]**

İteratif bir yapısının istediğiniz şekilde, sizin belirlediğiniz şekilde, çalışmasını sağlayın.

4.10. Dependency Injection ve Loosely Coupled Applications

Dependency injection kaba tabir ile bir sınıfın/nesnenin bağımlılıklardan kurtulmasını amaçlayan ve o nesneyi olabildiğince bağımsızlaştıran bir programlama tekniği/prensibidir.

**Dependency Injection Nedir ? Nasıl Uygulanır ? (Kod Örneğiyle)**

Dependency Injection uygulayarak; bir sınıfının bağımlı olduğu nesneden bağımsız hareket edebilmesini sağlayabilirsiniz.

**Dependency Injection ve Loosely Coupled Applications [12:07]**

Basit bir örnekle Dependency Injection kullanıp, bir sınıfın bağımlılığını ortandan kaldırıyoruz.

4.11. Interface Segregation Principle

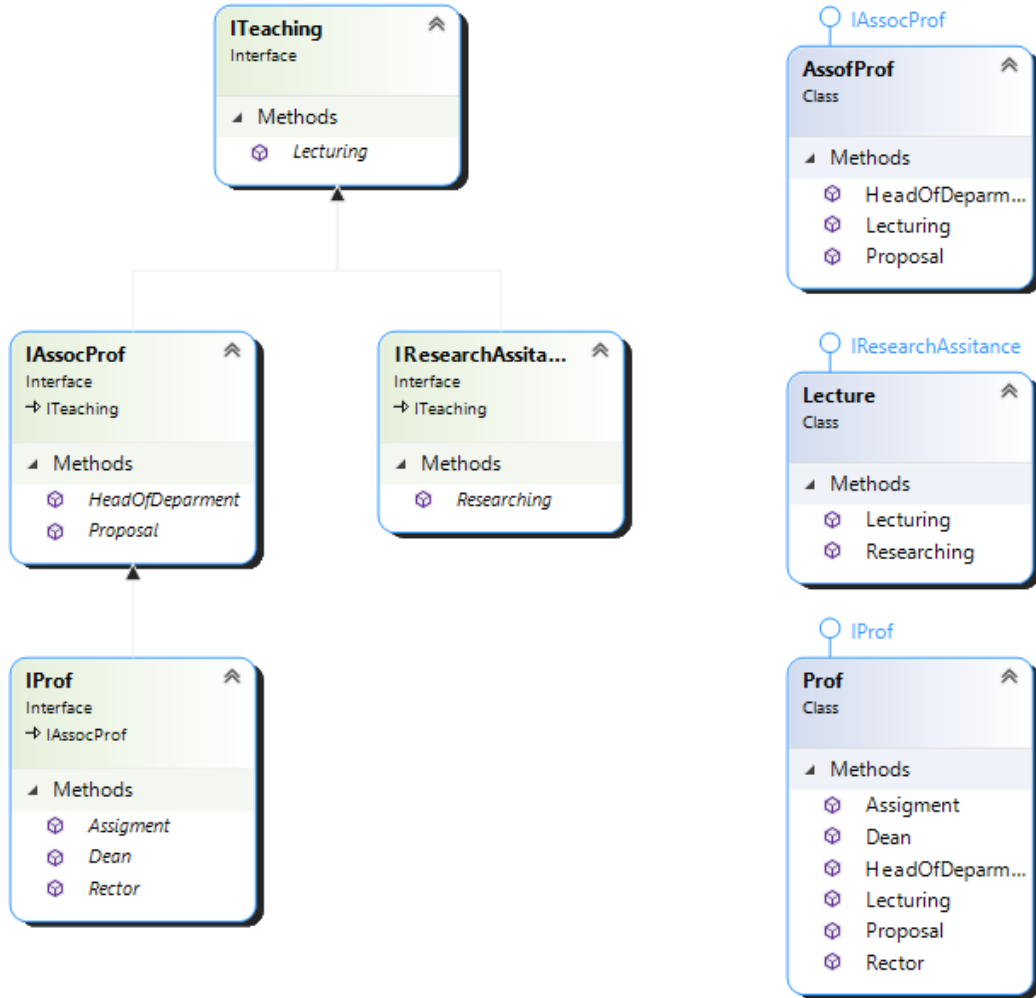
Sorumlulukların hepsini tek bir arayüze (**interface**) toplamak yerine daha özelleştirilmiş birden fazla **interface** oluşturmayı tercih etmenin gerekli olduğunu vurgulayan **SOLID** yazılım geliştirme prensibidir.

SOLID**I: Interface Segregation Principle**

**Interface Segregation Prensibi**

Sorumlulukların ayrılması ilkesi nedir?

Şekil 4.1'de Interface Segregation Principle için tasarlanmış olan bir örneğin sınıf diyagramına yer verilmiştir.



Şekil 4.1 Interface segregation prensibi için örnek uygulama



Interface Segregation Principle [12:58]

Basit bir örnekle Dependency Injection kullanıp, bir sınıfın bağımlılığını ortandan kaldırıyoruz.

5. Generic

Generic, spesifik değil, genel form anlamına gelir. C#'ta **generic**, belirli bir veri türüne özgü olmayan tanımlamalar gerçekleştirmek üzere kullanılır. C#, tür parametresini kullanarak ve belirli veri türü olmadan genel **class**, **interface**, **abstract class**, **field**, **methods**, **static methods**, **properties**, **events**, **delegates** ve **operators** tanımlamanıza olanak tanır.



Generic Class Defination [10:37]

`class<T>` şeklinde tanımladığımız **generic** sınıf tanımı hakkında basit bir uygulama yapıyoruz.

5.1. Genel Özellikler

- **Generic** kavramı hem C# hem de .NET için önemli bir konsepti ifade etmektedir.
- **Generic** spesifik olmayan genel form anlamına gelir. C# dilinde genel yani belirli bir türe ait olmayan anlamında kullanılır.
- **Generic** C# programlama dilinin bir parçasından çok **assembly** yapısında IL (**intermediate language**) ile tümleşiktir.
- **Generic** ile içeren tiplerden bağımsız olarak sınıflar, soyut sınıflar, alanlar, metotlar, statik metotlar, özellikler, olaylar, delegeler ve operatörler oluşturulabilir.
- **Generic** ifadeler genellikle **T** ile gösterilir. Burada **T** tip için bir yer tutucu olarak kullanılır ve yer tutucunun tipi örnek oluşturma aşamasında belirlenir.
- **Generic** pek çok metot ve sınıf yazmak yerine; aynı fonksiyonluğu diğer tipler içinde sağlar ve bunu yapmak için tek bir metot ve sınıf yeterlidir.
- Çok fazla kod yazmanın önüne geçmek için bir diğer seçenek ise **Object** sınıfını kullanmaktır. Fakat tip güvenliği açısından bu çok uygun değildir.
- **Generic** sınıflar, gerektiğinde belirli türlerle değiştirilen genel türlerden yararlanır. Bu tip güvenliği sağlar.
- **Generic** ifadeleri sınıflar ile sınırlı değildir; **interface** ve metotlar ile birlikte de kullanılabilirler. **Delegates**, **Lambdas** ve **Events** ile de kullanımları mevcuttur.
- **Generic** ifadeler **Just-In-Time (JIT)** compiler ile derlenirler.

- Sınıflara benzer şekilde **struct** yapısı da **generic** olabilir. Kalıtım özelliği dışında kullanımları son derece benzerdir.
- **Generic struct** yapısının .NET Framework yapısındaki en basit örneği **Nullable<T>** olarak ifade edilebilir.
- **Generic delegeler**, algoritmaları koleksiyonlardan ayırmayı mümkün kılar.
- **Generic**'ler daha iyi ikili kod kullanımı sağlarlar.
- **Generic** kullanımı kod şişmesinin (**code bloat**) de önüne geçmektedir.

5.2. Performans

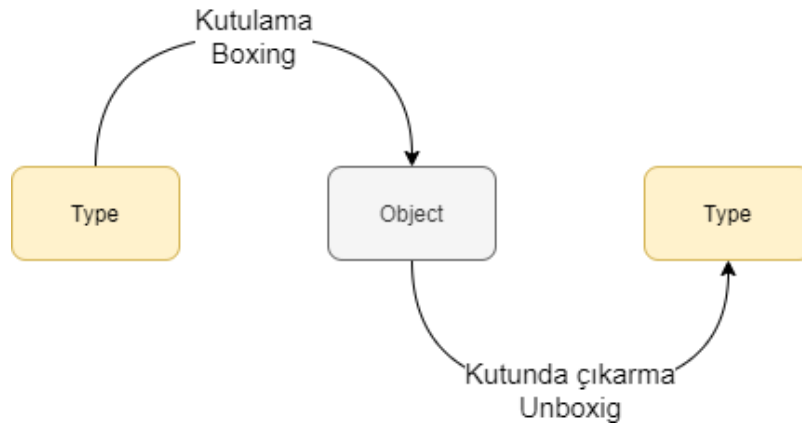
Generic kullanımının en büyük avantajı performanstır. Değer tipleri **stack** bölümünde referans tipleri ise **heap** bölümünde depolanır. C# sınıfları referans tiplidir; **struct** yapıları ise değer tiplidir. .NET ile değer ve referans tipleri arasındaki dönüşüm kolayca gerçekleştirilebilmektedir.



Generic ve non-generic [06:47]

Generic tanımlamalar kutulama ve kutudan çıkarma işlemlerini engeller ve performans artışı ile birlikte tip güvenliğinin sağlanmasına olanak tanır.

Örneğin **int** ile tanımlanmış olan bir değer tipli bir değişken, **object** türüne atanabilir. Buradaki dönüşüm değer tipinden referans tipine geçiştir ve bu kutulama (**boxing**) olarak ifade edilir. **Boxing** eğer bir metod **object** türüne ihtiyaç duyuyor ise otomatik olarak meydana gelir. Ters durumda, kutulanmış bir değer tipi **unboxing** ile değer tipine dönüştürülebilir. Bu durumda **cast** operatörü gereklidir. Bu süreç Şekil 5.1'de gösterilmiştir.



Şekil 5.1 Kutulama ve Kutudan çıkarma

5.3. Tip Güvenliği (Type Safety)

Generic ifadelerin bir diğer özelliği tip güvenliğidir. **ArrayList** sınıfı **object** ile kullanılmaktadır ve herhangi bir nesne listeye eklenebilmektedir. **Generic** kullanımda ise tip güvenliği **<T>** ile sağlanmaktadır.



Generic Collection Örneği [07:14]

Pek çok farklı Generic Collection yapısı olmakla birlikte; bu derste [LinkedList<T>](#) üzerinde bir örnek çalışma yapılmaktadır.

5.4. Generic Tanımların Özellikleri

Generic ifadeleri tanımlarken yaptığımız örneklerde şimdiye kadar hep referans tipli (**class**) sınıflardan faydalandık. Genel kullanım bu şekilde olmakla birlikte aynı zamanda değer tipli (**struct**) veri yapıları ile birlikte **generic** tanımlar gerçekleştirmek mümkündür.



Generic ve struct [04:49]

Bu örnekte **struct** ile tanımlanmış olan bir tipi, **generic** bir **collection** içerisinde kullanıyoruz.

Generic bir sınıf oluşturulduğunda, bazı ek C# anahtar kelimelerine ihtiyaç duyulabilir. Örneğin **generic** bir tipe **null** değer atamak olası değildir. Bu durumda **default** anahtar kelimesine ihtiyaç olacaktır. Eğer **generic** tip **Object** sınıfının özelliklerine ihtiyaç duymuyorsa ancak **generic** sınıf içinde bazı spesifik metotlar çağrılacaksa, **constraints** tanımlanabilir:

- **Default values**
- **Constraints**
- **Inheritance**
- **Static members**

5.5. Defaults

Generic tipler **null** değer alamazlar çünkü **generic** tipler **value** tipinde tanımlanırlar ve **null** değerler ancak referans tipli öğelerde kullanılabilir. Bu problemi aşmak için **default** anahtar kelimesi kullanılır.

5.6. Constraints

Eğer **generic** sınıfın **generic** tipten bazı metotları çağırması gerekiyorsa, **constraints** eklenmelidir. Tablo 5.1’de generic ifadeler için kullanılan kısıtlayıcılara ait örneklerle yer verilmiştir.

Tablo 5.1 Generic kısıtlayıcılar (constraints)

Constraint	Tanımı
<i>where T: struct</i>	<i>T</i> değer tipli olmalı.
<i>where T: class</i>	<i>T</i> referans tipli olmalı.
<i>where T: IFoo</i>	<i>T IFoo interface</i> implemente etmeli.
<i>where T: Foo</i>	<i>T Foo</i> temel sınıfından türetilmeli.
<i>where T: new()</i>	<i>T default</i> bir <i>constructor</i> sahip olmalı.
<i>where T1: T2</i>	T1, T2 tipinden türetilmeli.



Constraints [09:18]

Type temsil eden *T* ifadesi üzerinde kısıtlama yaparak, daha tutarlı tanımlamalar gerçekleştirilebilir.

5.7. Generic Methods

Bir sınıf **generic** olarak ya da yalın olarak (**non-generic**) tanımlanabilir. Sınıfın **generic** ya da **non-generic** olmasına bakmaksızın sınıflar **generic** metotlar içerebilir. Dolayısıyla **Type** temsil eden **T** ifadesi tanımı dikkate alınarak **generic** bir metot tanımı yapılabilir ve ilgili üye sınıfa eklenebilir.

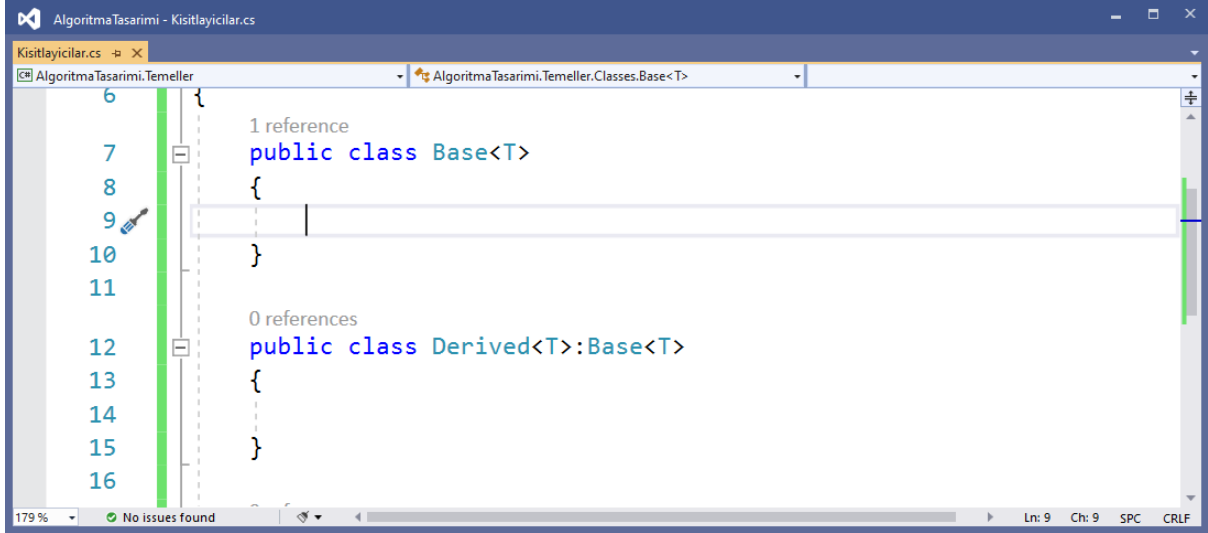


Generic Methods [06:10]

Type temsil eden *T* ifadesi dikkate alınarak sınıfın **generic** ya da **non-generic** olmasına bakmaksızın **generic method** tanımlaması yapılabilir.

5.8. Generic Inheritance

Generic bir sınıf **generic** bir **interface**'i implemente edebilir. **Generic** bir sınıf **generic** bir sınıftan türetilebilir. Şekil 5.2'de örnek bir kod parçasığına yer verilmiştir. Görüldüğü üzere generic bir temel sınıftan bir başka sınıfın türetilmesi sağlanmıştır.



Şekil 5.2 Generic sınıfta kalıtım

5.9. Generic Interface

Generic kullanılarak, **generic** parametreler içeren metotları tanımlayan **interface** tanımları gerçekleştirilebilir.

5.10. Generic Repository Desing Pattern

Programlarımızda veritabanı işlemleri yaparken bilinçsizce yapılan kodlama, bizleri müthiş bir kod tekrarına götürecektir. Özellikle bir uygulamada yapılan **CRUD(Create-Read-Update-Delete)** işlemleri, bu tekrarları hat safhaya çıkarmaktadır. İşte tam da bu noktada repository tasarım şablonu derde derman olmaktadır. Şekil 5.3'de ilgili tasarım deseninin örnek bir uygulamasına yer verilmiştir.



Generic Repository Design Pattern Nedir?

CRUD işlemlerini tekrar kod bloklarından kurtaran bir tasarım desendir.



Şekil 5.3 Generic Repository Design Pattern

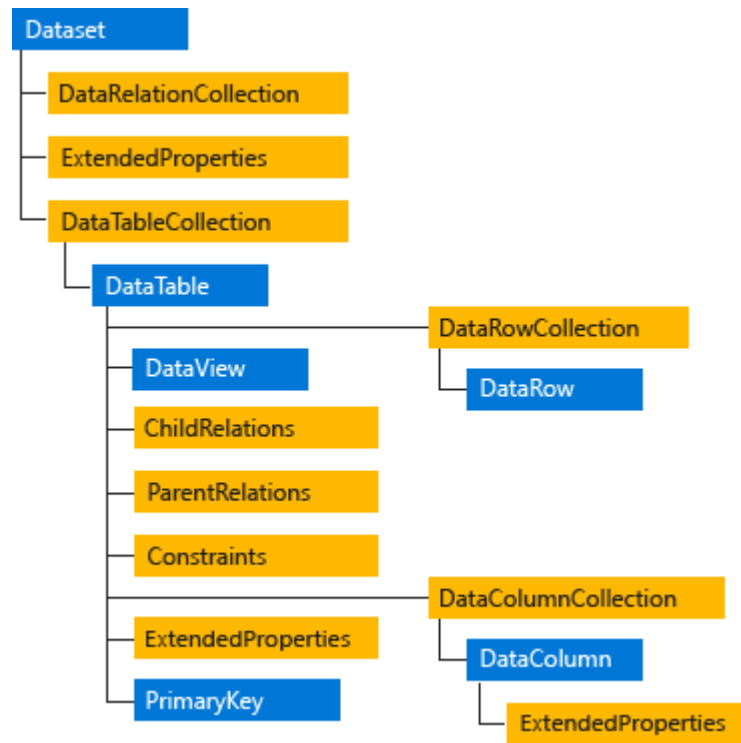


Generic Repository Design Pattern [29:39]

CRUD işlemlerini tekrarlı kod bloklarından kurtaran bir tasarım desenini genel bir bakış gerçekleştiriyoruz.

6. ADO.NET

ADO.NET, **ActiveX Data Object**'in kısaltmasıdır ve Microsoft tarafından .NET çerçevesinin bir parçası olarak oluşturulan ve her türlü veri kaynağına erişebilen bir veritabanı erişim teknolojisidir. İstemci-sunucu uygulamaları ve ayrıca İnternet ve intranetler üzerinden dağıtılmış ortamlar için yüksek performanslı, güvenilir ve ölçeklenebilir veritabanı uygulamaları oluşturmak için zengin bir veri bileşenleri kümesi sağlayan bir nesne yönelimli sınıflar kümesidir. Şekil 6.1'de ADO.NET mimarisine yer verilmiştir.



Şekil 6.1. ADONET Mimarisi

ADO.NET, SQL Server, Microsoft Access veya XML gibi veri kaynaklarına ve OLE DB ve ODBC aracılığıyla sunulan veri kaynaklarına tutarlı erişim sağlar. Veri kaynağı kullanan uygulamalar, bu veri kaynaklarına bağlanmak ve içerdikleri verileri almak, işlemek ve güncellemek için ADO.NET'i kullanabilir.

“ADO.NET bir Object Relational Mapping (ORM) değildir.”

ADO.NET modelinde, ADO (bağlı durumda) ve önceki veri erişim teknolojileri uygulamalarının aksine, verileri okurken veya güncellerken veri kaynaklarına bağlanır. Bundan

sonra bağlantı kapanır. Bu önemlidir çünkü istemci-sunucu veya dağıtılmış uygulamalarda bağlantı kaynaklarının her zaman açık olması en çok kaynak tüketen kısımlardan biridir. Her zaman bir veri kaynağına bağlanmanız gerekmez; bir veri kaynağına bağlanmanız gereken tek zaman, bir veri kaynağındaki son değişiklikleri okuyup yazdığınız zamandır.

ADO.NET bir veri kaynağı üzerinde yazma, güncelleme, okuma ve silme işlevleri için SQL sorgularını ve saklı yordamları kullanır. Verileri **DataReader** veya **DataSet** nesneleri biçiminde döndüren ADO.NET **Command** nesnesi aracılığıyla SQL sorguları kullanılır. Bu bağlantı kapandıktan sonra, verilerle çalışmak ve veri kaynağını güncellemeniz gerektiğinde veri kaynağına yeniden bağlanmak için **DataSet** nesnelerini kullanırsınız.

Başlıca avantajları

- Tek Nesneye Yönelik API
- Yönetilen Kod
- Dağıtım
- XML Desteği
- Görsel Veri Bileşenleri
- Performans ve Ölçeklenebilirlik
- Bağlantılar ve Bağlantısı Kesilen Veriler

6.1. SQL Console Application

Bu bölümde **Northwind** veri tabanına bağlanmak üzere SQL Server kullanıyoruz. Söz konusu teknolojinin kullanılmasını sağlamak üzere **ADO.NET** teknolojisinden faydalıyoruz.

Bu amaçla öncelikle bir **SqlConnection** nesnesi tanımlanmasına ihtiyaç duyulmaktadır. Veri tabanı bağlantısını sağlamak amacıyla öncelikle **System.Data.SqlClient** paketinin projeye eklenmesi gerekir. Paket yüklemesi tamamlandıktan sonra bir **ConnectionString** tanımı yapılır ve nihai olarak **SqlConnection** sınıfı üzerinden bir bağlantı nesnesi oluşturulur.

```
<ItemGroup>  
  <PackageReference Include="System.Data.SqlClient" Version="4.8.3" />  
</ItemGroup>
```

Bir .NET Core projesi üzerinde yüklenmiş olan paketlerin listesini almak üzere projeye sağ tıklanarak **“Edit Project File”** komutu verilerek yüklü olan paketlerin listesine erişim sağlanabilir. Örneğin **OOP.ADONET.csproj** dosyası içerisinde yüklü olan paket yukarıda belirtildiği gibi görülecektir.



ADO.NET SqlConnection and SqlCommand [15:26]

Bu ders kapsamında *Northwind* veri tabanına bağlanmak üzere gerekli paket yüklemesini gerçekleştiriyor ve gerekli tanımları yapıyoruz.

Veri tabanına erişim sağlamak üzere tanımlanan kod bloklarının nesne yönelimli programlama teknikleri ile modellenmesi, ilgili kod bloklarının tekrarlanmasını engelleyecektir.



ADO.NET SQL Console Application Refactoring [08:12]

Veri tabanı kodları üzerinde nesne yönelimli programlama tekniklerini uyguluyoruz.

Veri tabanından okuma yapılırken bağlantının sürekli açık olması gerekir. Bu nedenle okuma yapıldıktan sonra okunan verilerin bir **DataTable** ya da **DataSet** gibi bir nesne üzerine aktarılması gerekir. **DataTable** ya da **DataSet** gibi ifadelerin doldurulmasını sağlamak üzere **SqlDataAdapter** sınıfının kullanılması gerekir.



ADO.NET SQL Console Application Reading From Database [08:02]

SqlDataAdapter ve *DataSet* nesnesi ile okuma işlemi gerçekleştiriyoruz.

CRUD işlevlerini gerçekleştirmek üzere **Create** ve **Read/Retrieve** şeklindeki tanımlamalar gerçekleştirildi. Bu noktadan sonra **Update** ve **Delete** işlevlerinin yapılması gerçekleştirilir.



ADO.NET SQL Console Application Update ve Delete [06:59]

Update ve Delete cümleleri ile veri tabanına erişimin yapılmasını gerçekleştirilelim.

CRUD işlevlerinin tamamı gerçekleştirilmiş durumda ancak nesne yönelimli programlama tekniklerinden uzak durumdayız. Nesne yönelimli programlama tekniklerini uygulamak üzere **Relational Database Management System (RDMS)** şeklinden bir sınıf tanımlı yapılarak SQL cümlelerinin bu sınıf aracılığı ile çalıştırılması hedeflenir. Bu kapsamda sırasıyla **SqlNonQuery(SqlCommand)** ve **SqlReader(SqlCommand)** isiminde iki metot tasarımı gerçekleştirilmiştir.



ADO.NET SQL Console Application RDMS SqlNonQuery [12:22]

RDMS.SqlNonQuery(SqlCommand) üyesi tanımlanır.

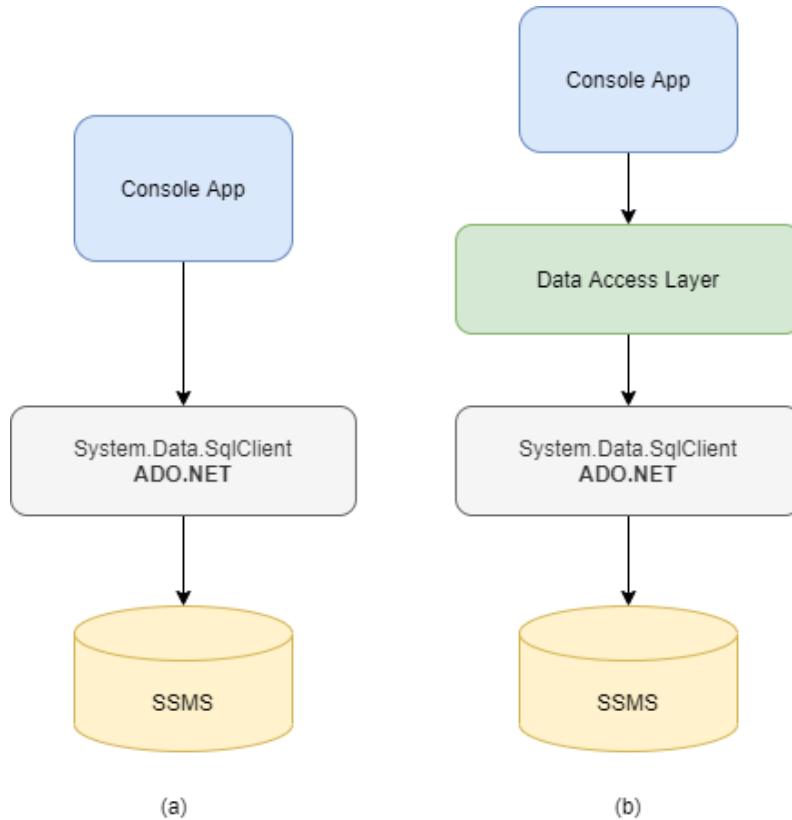


ADO.NET SQL Console Application RDMS SqlDataReader [12:37]

RDMS.SqlReader(SqlCommand) üyesi tanımlanır.

6.2. Data Access Layer

Projenin mevcut durumunda, veriye erişim katmanı ayrılmadığından **SqlCommand** nesneleri doğrudan tanımlanmakta ve kullanılmaktadır. Çalışan kodların tekrar kullanılabilirliğini (**reusability**) sağlamak üzere veri ile ilgili işlevlerin **Data Access Layer** (DAL)'da yapılması sağlanır.



Şekil 6.2 (a) Projenin mevcut hali (b) Data Access Layer ile veriye erişimin düzenlenmesi

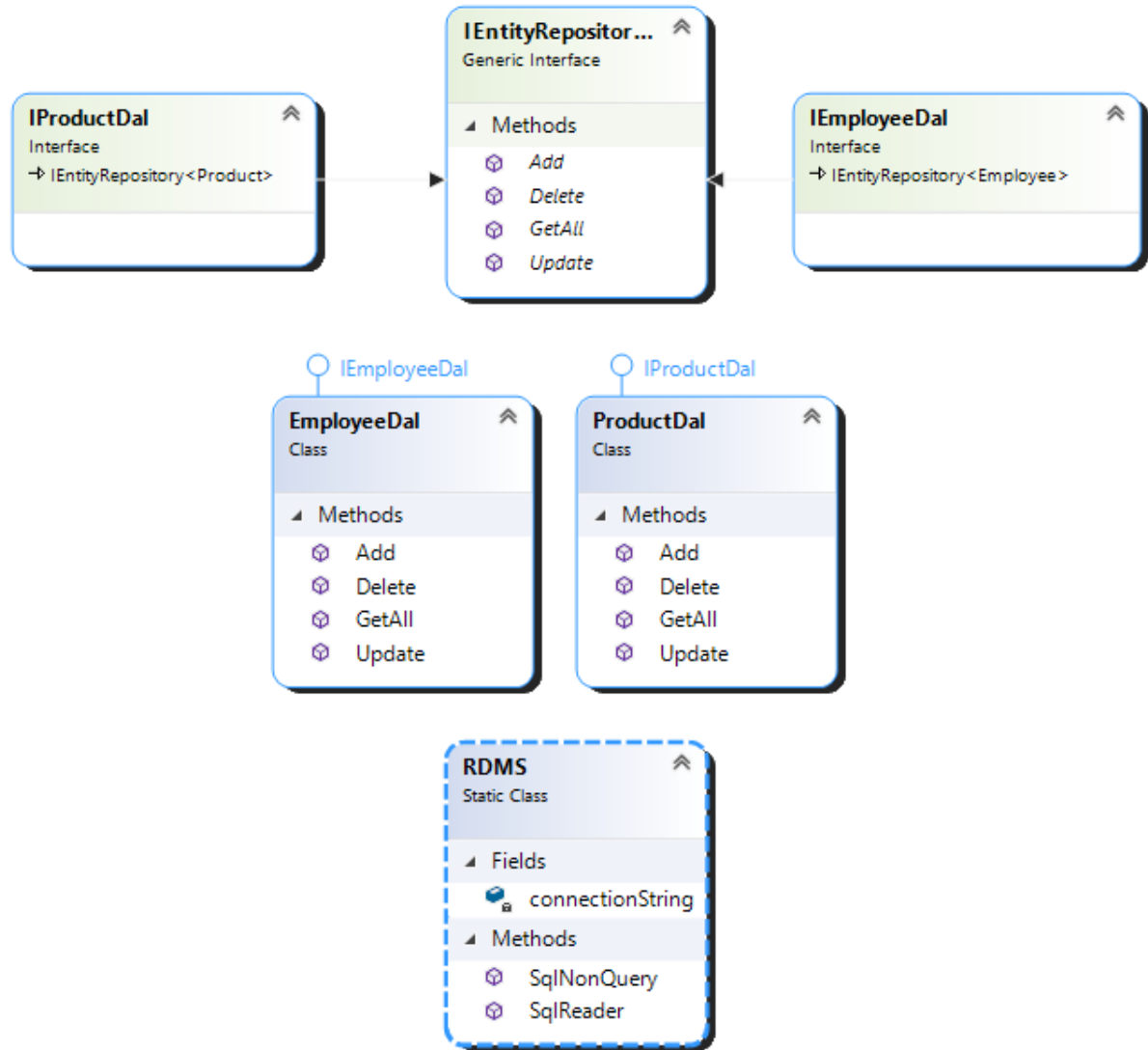
Böylelikle **SqlCommand** nesnelerinin de tekrar tekrar kullanılması engellenerek ilgili varlıklar (**Employee**, **Product**, **Order** vb.) için **CRUD** işlevleri bir sınıf içerisinde gerçekleştirilir.



ADO.NET SQL Console Application Data Access Layer [20:09]

Veriye erişimin ve veri işleme yöntemlerinin daha sistematik hale getirilmesini sağlıyoruz.

Şekil 6.3'de **DAL** eklenmesine bağlı olarak projenin sınıf diyagramına yer verilmiştir. Böylelikle **RDMS** sınıfının SQL cümlelerini her bir nesne için ilgili nesnenin **DAL** sınıfında tanımlayan yapı oluşturulmuştur.



Şekil 6.3 Data Access Layer ile Projenin Modellenmesi

DAL üzerinde **Abstract** ve **Concrete** ifadelerin tanımlanması gerçekleştirilir. En temel soyut sınıf **IEntityRepository<T>** şeklinde tanımlanır. Bu kontrol üzerinde tanımlı olan metotlar devir alınan somut sınıflar üzerinde mutlak suretle uygulanır (**implementation**).

```
public interface IEntityRepository<T>
    where T:class, new() {
        void Add(T entity);
        void Update(T entity);
        List<T> GetAll();
        void Delete(T entity);
    }
```

Soyut olarak tanımlanan **IEntityRepository<T>** içerisindeki metotlar temel **CRUD** işlevlerine karşılık gelir. Eğer nesnelere özel yazılması gereken metotlar var ise söz konusu metotların ilgili varlıklara karşılık gelen **interface** yapıları üzerinde tanımlanması gerekir.

```
public interface IEmployeeDal : IEntityRepository<Employee> {
}
```

```
public interface IProductDal : IEntityRepository<Product>{
}
```

Yukarıda tanımlanmış olan **IProductDal** ve **IEmployeeDal interface** yapıları, ilgili varlıklara ait özel metot tanımları içerebilecek şekilde tasarlanmıştır.

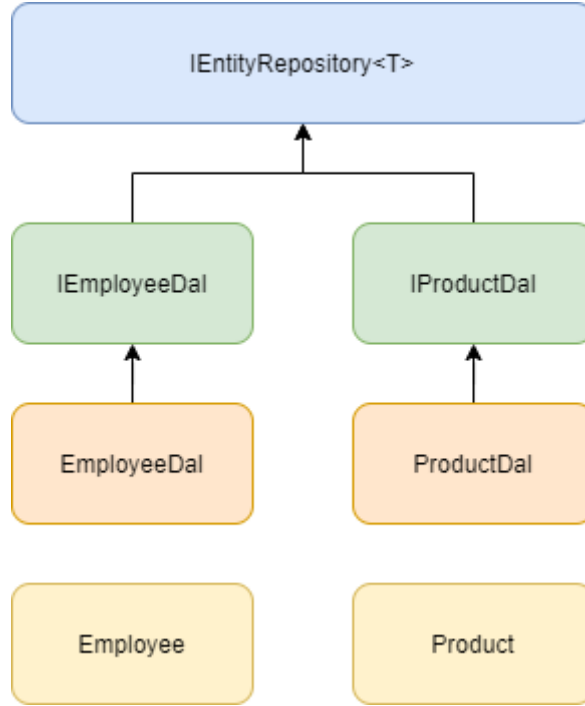
ADO.NET için somut olan temel bir sınıf yazılmadığından her nesne için **CRUD** işlevlerine ait **SQL** sorguları **ProductDal** ve **EmployeeDal** altında yazılmak durumundadır. Örneğin **ProductDal.Add(Product)** işlevi için tanımlanması gereken kod bloğu aşağıdaki gibidir:

```
public void Add(Product entity) {
    var cmd = new SqlCommand("INSERT INTO Products(ProductName,
UnitPrice,UnitsInStock) VALUES(@ProductName,@UnitPrice,@UnitsInStock)");
    cmd.Parameters.AddWithValue("ProductName", entity.ProductName);
    cmd.Parameters.AddWithValue("UnitPrice", entity.UnitPrice);
    cmd.Parameters.AddWithValue("UnitsInStock", entity.UnitsInStock);
    RDMS.SqlNonQuery(cmd);
}
```



ADO.NET SQL Console Application ProductDal [09:30]

Veriye erişimin ve veri işleme yöntemlerinin daha sistematik hale getirilmesini sağlıyoruz. Bu kapsamda yapılan işlemlerin pekişmesini sağlamak üzere **Product** nesnesi için ilgili tanımlamaları tekrar ediyoruz.



Şekil 6.4 Data Access ve Entity ilişkisi

Şekil 6.4’de **DAL** içerisinde tanımlı olan somut ve soyut sınıflar ile ilgili **Entity** ilişkisini ifade eden bir şekle yer verilmiştir.

İlgili tanımlamalar yapıldı ancak modellerin çalışmasını sağlamak üzere henüz herhangi bir test yapılmadı. Bu bölümde ilgili metotların çalışıp çalışmadığı test edilecektir.



ADO.NET SQL Console Application ProductDal Practice [09:30]

Geliştirdiğimiz DAL sınıflarını kullanıyoruz.

7. Entity Framework Core

7.1. Entity Framework Core Genel Bakış

Entity Framework (EF) Core, popüler Entity Framework veri erişim teknolojisinin hafif, genişletilebilir, açık kaynaklı ve platformlar arası bir sürümüdür.

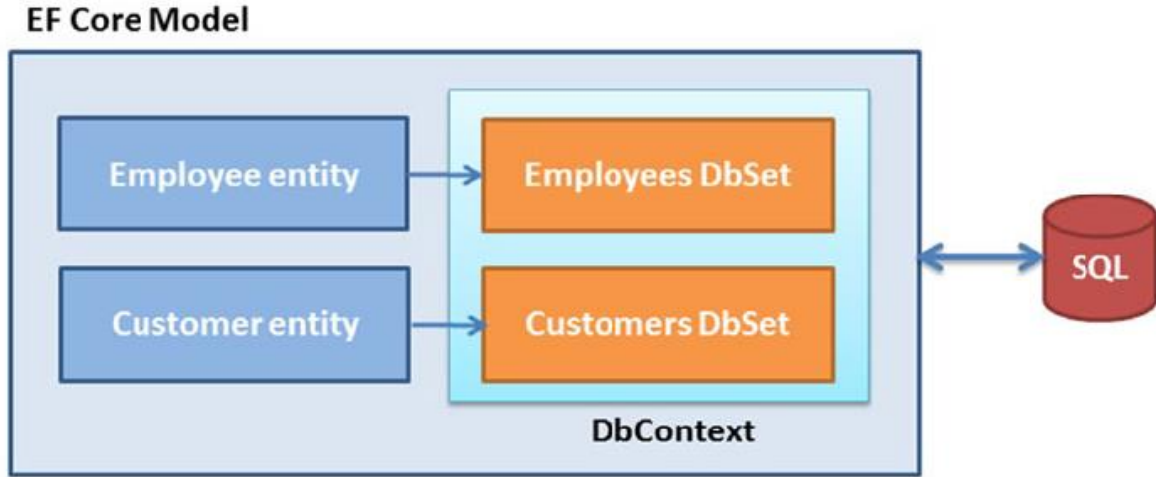
EF Core, nesne-ilişkisel eşleyici (O/RM, **Object/Relational Mapping**) işlevi görebilir:

- NET geliştiricilerinin. NET nesneleri kullanarak bir veri tabanıyla çalışmasına olanak tanır.
- Genellikle yazılması gereken veri erişim kodunun çoğuna olan ihtiyacı ortadan kaldırır.

Bu noktada *Object-Relational Mapping* (**ORM**) teknolojisine öncelikle değinmek gerekir. Veri tabanında yer alan tabloların **class**, kolonların **property** olarak adlandırıldığı, tablodaki verilerin nesnelere dönüştüğü bu yapıda sınıf ve nesneler üzerinde veri tabanı işlemleri kolaylıkla yapılabilmektedir. ORM çatısı sayesinde veri tabanı işlemleri kolaylıkla yapılabildiğinden; daha kısa sürede ürün geliştirmek mümkün olabilmektedir.

- *Entity Framework* Microsoft tarafından geliştirilmiş **ORM** çatılarından biridir.
- Varlıklar, *Entity Framework Core*'nin odak noktasıdır.
- Varlık, bir veri tabanındaki bir veya daha fazla tabloyla eşleşen bir nesnedir.

Şekil 7.1'de EF Core yapısına yer verilmiştir.



Şekil 7.1 Entity sınıfları, DbSet ve DbContext

Belirli bir veri modeli için *Entity Framework* işlevselliğini koordine eden *database context* sınıfıdır. Bu sınıf *System.Data.Entity.DbContext* sınıfından türetilerek oluşturulur. Aşağıdaki örnekte SQLite3 veri tabanı için tanımlanan *DbContext* sınıfına yer verilmiştir. *DbContext* içerisinde tablolar *DbSet<T>* üzerinde tanımlanmaktadır.

```

public class NorthwindDbContext : DbContext
{
    public string DbPath { get; set; }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Product> Products { get; set; }

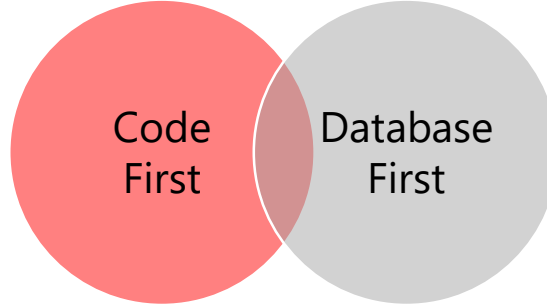
    public NorthwindDbContext()
    {
        DbPath = "D:\\Northwind.db";
    }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Data Source={DbPath}");
    }
}
  
```

EF ile veriye erişim sağlarken kod-öncelikli (*code first*) ya da veri tabanı-öncelikli (*database-first*) yaklaşımları kullanabilir.

Code First yaklaşımında, öncelikle bir uygulamanın etki alanına odaklanılarak varlıklar veya sınıflar oluşturulur. Varlıklarla eşleşen veritabanını tasarlamadan sınıfları ve gerekli özellikleri oluşturmaya başlanabilir. Daha sonra *Entity Framework* tabloları ve veritabanını buna göre oluşturur ve kod çalıştırıldığında veritabanı oluşturulur.

Database First yaklaşımında önce veritabanı ve ilgili tablolar oluşturulur. Bundan sonra, veritabanını kullanarak bir varlık veri modelleri oluşturabilirsiniz. Grafik kullanıcı arabirimleri kullanarak birden fazla seçenek mevcut olduğundan, bir veritabanı oluşturmak daha kolaydır.



Şekil 7.2 EF ile veriye erişim yaklaşımları



Entity Framework Core Giriş [07:32]

Entity Framework Core teknolojisine giriş yapıyoruz.

7.2. Kurulum

EF Core Microsoft SQL Server, SQLite, Cosmos ya da in-memory veri tabanları için provider'lara sahiptir.

Visual Studio içerisinde kurulum için *NuGet Package Manager* kullanılabilir. Aşağıda EF Core işlemleri için yaygın olarak kullanılan paketlere yer verilmiştir:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools
- Microsoft.Extensions.Logging.Console

paketleri kullanılabilir. Bu paketlerin dışında tercih edilen teknolojilere bağlı olarak kurulması gereken paketler farklılık gösterebilir.

EF Core teknolojisi kendi içinde detayları sahip olan oldukça geniş bir konu olarak değerlendirilmelidir.



Entity Framework Core Paket Kurulumları ve DbContext [15:46]

Entity Framework Core için gerekli paketlerin kurulumunu gerçekleştiriyoruz.

7.3. Entity Tanımı

Entity Framework Core temelinde varlıklar yani *Entity* şeklinde ifade edilen sınıflar vardır. Sınıfların bir başka ifadeyle *Entity*'lerin ilişkilerini EF Core teknolojisini kullanarak yapılandırıyoruz. Aşağıda örneğin *Book* varlığının içeriğine yer verilmiştir.

```
public class Book
{
    public int BookId { get; set; }

    public string Title { get; set; }

    public DateTime CreatedDate { get; set; }

    public decimal Price { get; set; }

    // foreign key
    public int? CategoryId { get; set; }

    // simple navigation property
    public Category Category { get; set; }

    // navigation property
    public BookDetail BookDetail { get; set; }

    // collection navigation property
    public ICollection<BookAuthor> BookAuthors { get; set; }
}
```



Entity Framework Core Entity Tanımı [04:35]

Entity Framework Core için varlık yani Entity tanımı gerçekleştiriyoruz.

7.4. Migrations

Nesne tarafında gerçekleştirilen tanımların ve ilişkilerin veri tabanına yansımaları sağlamak üzere *Migration* komut seti kullanılır. Migration komut seti *dotnet* üzerinden ya da *Package Manager Console* (PM Console) üzerinden verilebilir.

PM Console üzerinde *Migration* tanımlamak üzere aşağıdaki komut setleri kullanılır.

- Add-Migration MigrationName
- Update-Database
- Remove-Migration

dotnet üzerinde de benzer şekilde aşağıdaki komut setleri Migration işlemleri için kullanılır.

- dotnet ef migrations add MigrationName
- dotnet ef database update
- dotnet ef migrations remove



Entity Framework Core Migrations [12:48]

Nesneler üzerinde tanımladığımız değişiklikleri veri tabanına yansıtmak üzere Migration komut setlerini kullanıyoruz.

7.5. EF Core Özellikleri

7.5.1. Entity Property Constraints

Required, *MaxLength(255)* gibi ifadeler *Entity* olarak kullanılan sınıfların özellikleri üzerinde *Data Annotation* olarak tanımlanarak özellikler üzerinde kısıtlayıcılar tanımlamak (*Entity Property Constraints*) mümkündür. *Annotations* dışında *FluentAPI* yaklaşımı ile de kısıtlayıcılar kullanılabilir ¹.

¹ <https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/fluent/types-and-properties>

**Entity Framework Core Property Constraints Data Annotation [06:18]**

Bir varlığın özellikleri üzerinde çeşitli sınırlandırma tanımlamalarını gerçekleştirmek üzere **Data Annotation** yaklaşımına değiniyoruz.

**Entity Framework Core Property Constraints Fluent Api [09:42]**

Bir varlığın özellikleri üzerinde çeşitli sınırlandırma tanımlamalarını gerçekleştirmek üzere **Fluent API** yaklaşımına değiniyoruz.

**Entity Framework Core IEntityConfiguration [11:14]**

Her bir varlığın (**Entity**) konfigürasyonu ayrı bir dosyada gerçekleştirmek üzere **IEntityTypeConfiguration** interface yapısını uyguluyoruz.

7.5.2. Entity Schema Attribute

- Table("Persons", Schema = "dbo")
- NotMapped
- [Column("Person_Id")]

gibi tanımlar yapılarak şema özelliklerinin tanımlanması EF Core ile mümkündür.

7.6. Entity Framework Core içerisinde ilişkiler

İlişkisel veri tabanlarında, tablolar arasındaki ilişkiler (*relations*) yabancı anahtarlar (*foreign keys, FK*) aracılığıyla tanımlanır. Yabancı anahtar (FK), iki tablodaki veriler arasında bir bağlantı kurmak ve zorlamak için kullanılan bir sütun veya sütunların birleşimidir.

Genellikle üç tür ilişki vardır: bire bir (*one-to-one*), bire çok (*one-to-many*) ve çok-a-çok (*many-to-many*). Bire-çok ilişkide, yabancı anahtar (FK), ilişkinin çok ucunu temsil eden tabloda tanımlanır. Çok-a-çok ilişki, birincil anahtarı (*primary key, PK*) her iki ilgili tablodaki yabancı anahtarlardan oluşan üçüncü bir tablonun (birleşim (*junction* veya *join*) tablosu olarak adlandırılır) tanımlanmasını içerir. Bire-bir ilişkide, birincil anahtar (PK) ayrıca bir yabancı anahtar (FK) görevi görür ve her iki tablo için de ayrı bir yabancı anahtar sütunu yoktur.

EF içerisinde ilişki, ilişkideki hangi ucun ana rol (*principal role*) ve hangisinin bağımlı rol (*dependent role*) olduğunu tanımlayan bir referans kısıtlaması tarafından yönetilebilir ².

7.6.1. EF Core ilişkin Terimlerin Açıklanması

Bir ilişki, iki varlığın birbiriyle nasıl ilişkili olduğunu tanımlar. İlişkisel bir veri tabanında bu, bir yabancı anahtar (*FK*) kısıtlaması ile temsil edilir. İlişkileri tanımlamak için kullanılan birkaç terim vardır.

- **Dependent entity (Bağımlı varlık):** Yabancı anahtar özelliklerini içeren varlıktır. Bazen ilişkinin 'çocuğu (child)' olarak anılır.
- **Principal entity (Asıl varlık):** Bu, birincil/alternatif anahtar özelliklerini içeren varlıktır. Bazen ilişkinin 'ebeveyn (parent)' olarak anılır.
- **Primary key (Ana anahtar):** Asıl varlığı benzersiz şekilde tanımlayan özellikler. Bu, birincil anahtar veya alternatif anahtar olabilir.
- **Foreign key (Yabancı anahtar):** İlgili varlık için temel anahtar değerlerini depolamak için kullanılan bağımlı varlıktaki özellikler.
- **Navigation Property (Gezinme özelliği):** Asıl ve/veya bağımlı varlık üzerinde tanımlanan ve ilgili varlığa referansta bulunan bir özellik.
 - **Collection navigation property (Koleksiyon gezinme özelliği):** Birçok ilgili varlığa referanslar içeren bir gezinme özelliği.
 - **Reference navigation property (Referans navigasyon özelliği):** Tek bir ilgili varlığa referansı tutan bir navigasyon özelliği.
- **Inverse navigation property (Ters gezinme özelliği):** Belirli bir gezinme özelliği tartışılırken bu terim, ilişkinin diğer ucundaki gezinme özelliğine atıfta bulunur.
- **Kendinden referanslı ilişki (self-referencing relationship):** Bağımlı ve asıl varlık türlerinin aynı olduğu bir ilişki.

² <https://docs.microsoft.com/en-us/ef/ef6/fundamentals/relationships>


```

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}

```

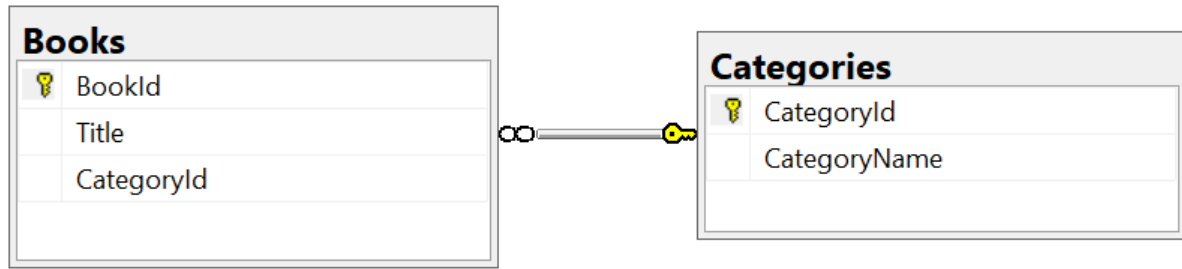
- *Post*, bağımlı (dependent) varlıktır.
- *Blog*, ana (principal) varlıktır.
- *Blog.BlogId* birincil anahtardır.
- *Post.BlogId* yabancı anahtardır.
- *Post.Blog* reference navigation property'dir.
- *Blog.Posts* collection navigation property'dir.

Varsayılan olarak bir navigation property ile karşılaşıldığında ilişki oluşturulur. Bir özellik, işaret ettiği tür geçerli veritabanı sağlayıcısı tarafından skaler bir tür olarak eşlenemiyorsa, gezinme özelliği olarak kabul edilir ³.

7.6.2. Bire-çok ilişki (one-to-many relations)

Bire-çok ilişkide, yabancı anahtar (FK), ilişkinin çok ucunu temsil eden tabloda tanımlanır. Şekil 7.3'de bire-çok ilişkiye sahip olan iki varlığın diyagramına yer verilmiştir. Bire-çok ilişki, ilişkisel veri tabanlarında en fazla kullanılan ilişki türü olarak ifade edilebilir.

³<https://docs.microsoft.com/en-us/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key>



Şekil 7.3 Bire-çok ilişki (One-to-many relation)

```

public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }

    public int? CategoryId { get; set; } // foreign key
    public Category Category { get; set; } // simple navigation property

    public override string ToString()
    {
        return $"{BookId,-5} {Title,-50}";
    }
}
  
```

```

public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }

    // Collection navigation property
    public ICollection<Book> Books { get; set; }
}
  
```

Bu örnekte bağımlı varlık *Book*, temel varlık ise *Category* olarak tanımlanabilir. *Category.CategoryId* alanı *Book.CategoryId* alanında çok defa tekrar edebilecektir. Bir başka ifadeyle her kitaba ait olarak bir kategori tanımı yapılır.

Fluent API yaklaşımı ile *Book* ve *Category* varlıkları arasındaki ilişki aşağıdaki gibi tanımlanabilir. *Context* üzerinde *OnModelCreating* metodu içinde bu özellikler

tanımlanabileceği gibi *IEntityTypeConfiguration<T>* interface yapısını kabul eden bir sınıf içerisinde de ilgili tanımlamalar gerçekleştirilebilir.

```
public class BookMap : IEntityTypeConfiguration<Book>
{
    public void Configure(EntityTypeBuilder<Book> builder)
    {
        builder.HasKey(b => b.BookId);
        builder.Property(b => b.Title).IsRequired();
        builder.HasData(
            new Book { BookId = 1, Title="Devlet", CategoryId=1 },
            new Book { BookId = 2, Title = "Beyaz Zambaklar Ülkesi", CategoryId=1 },
            new Book { BookId = 3, Title = "Yoldaki İşaretler", CategoryId=1 }
        );
        // one - to - many relation
        builder.HasOne(b => b.Category)
            .WithMany(c => c.Books)
            .HasForeignKey(b => b.CategoryId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```



Entity Framework Core One-To-Many Relation [24:36]

Book ve **Category** nesneleri arasında bire-çok (**one-to-many**) ilişki kuruyoruz.

7.6.3. Bire-bir ilişki (one-to-one Relation)

Bire bir ilişkide, birincil anahtar (PK) ayrıca bir yabancı anahtar (FK) görevi görür ve her iki tablo için de ayrı bir yabancı anahtar sütunu yoktur. *Principal entity* içinde tanımlı olan *primary key* alanı, *dependent entity* içinde *foreign key* olarak tanımlanır ve aynı zamanda *Unique* yani benzersiz özelliğine sahip olur.

Bire-bir ilişki için bir kitaba ait olan detayların ayrı bir tabloda tutulması üzerine bir senaryoyu dikkate alalım. Bu kapsamda öncelikle *Detail* ifadeleri tutan bir varlık tanımı yapılır. Bu varlık içerisinde *BookId* alanı *foreign key* olarak yerleştirilir ve bir de *Book* tipinde bir alan *navigation property* olarak tanımlanır.

```
public class BookDetail
{
    public int BookDetailId { get; set; }

    // foreign key - unique
    public int BookId { get; set; }
    // navigation property
    public Book Book { get; set; }

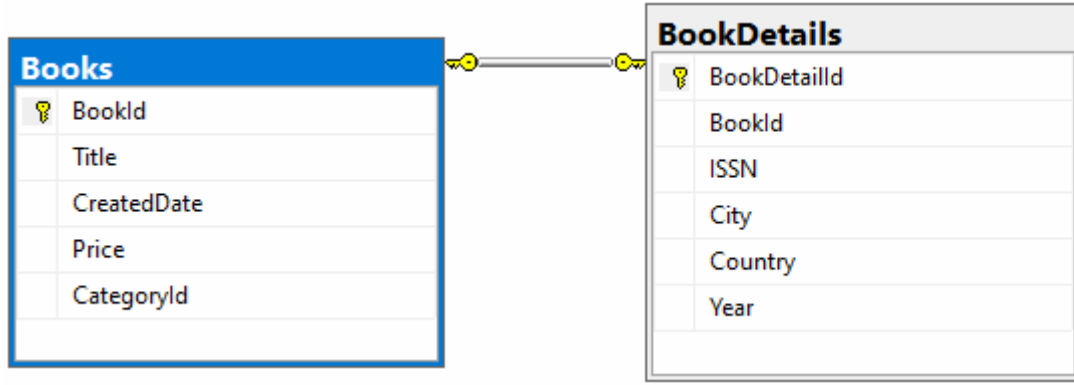
    public string ISSN { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public int Year { get; set; }
}
```

Buna karşılık *Book* varlığı içerisinde aradaki bağlantıyı kurmak üzere *simple navigation property* tanımı yapılır. Bu tanım *Details* tablosunda *Details.BookId* alanının *Foreign Key* olmasını ve aynı zamanda *Unique* yani benzersiz olmasını sağlar.

```
public class Book
{
    // simple navigation property
    public BookDetail BookDetail { get; set; }
}
```

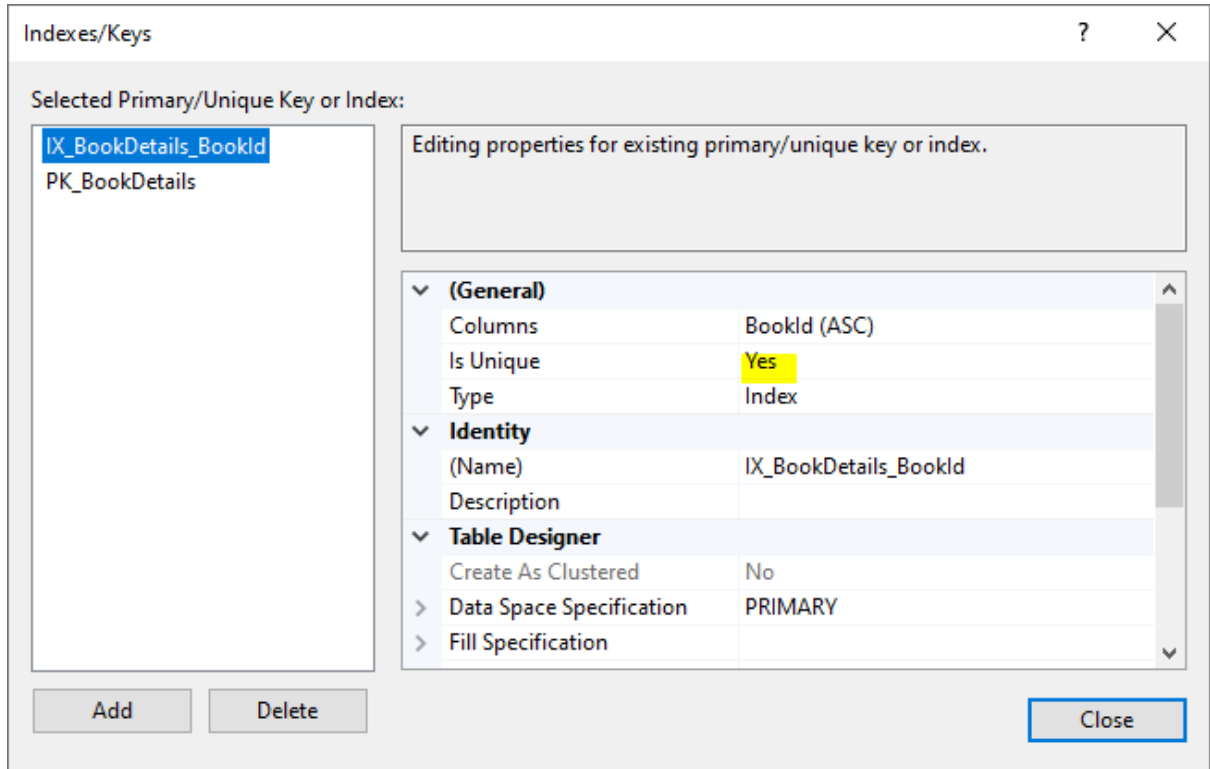
Aradaki ilişkinin *FluentAPI* ile tanımlanmasını sağlamak üzere aşağıdaki gibi bir tanım yapılır.

```
public class BookDetailMap : IEntityTypeConfiguration<BookDetail>
{
    public void Configure(EntityTypeBuilder<BookDetail> builder)
    {
        builder.HasOne(bd => bd.Book)
            .WithOne(b => b.BookDetail)
            .HasForeignKey<BookDetail>(bd => bd.BookId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```



Şekil 7.4 Bire-bir ilişki

Şekil 7.5’de *BookDetails* tablosunda tanımlanmış olan *Unique key* görülmektedir. Bir başka ifadeyle *BookId* alanı *BookDetails* tablosunda yalnızca bir kez kullanılabilir.



Şekil 7.5 Unique key

Bire-bir ilişki tanımı için *principal entity* alanında tanımlanmış olan *primary key*, *dependent entity* içinde *foreign key* olarak kullanılır ve aynı zamanda *unique* yani benzersizdir. Bu nedenle *principal entity*’nin bir örneği, *dependent entity* içinde yalnızca bir kez kullanılabilir.



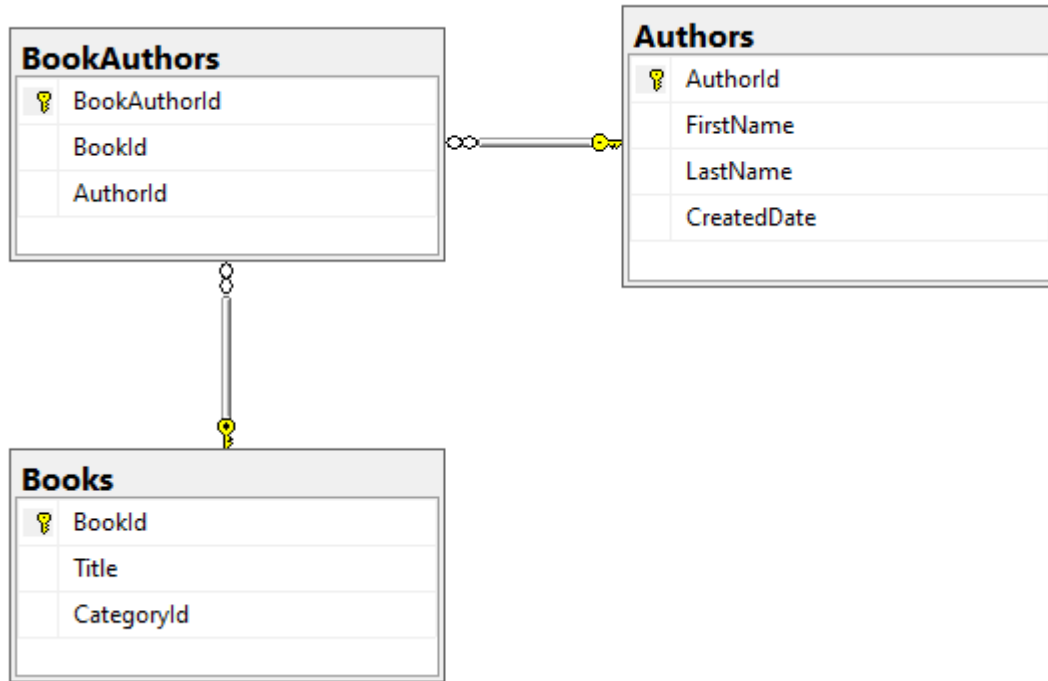
Entity Framework Core One-To-One Relation [24:36]

Book ve **BookDetail** nesneleri arasında bire-bir (**one-to-one**) ilişki kuruyoruz.

7.6.4. Çokla-çok ilişki (Many-to-many relation)

Çoka çok ilişki, birincil anahtarı her iki ilgili tablodaki yabancı anahtarlardan (FK) oluşan üçüncü bir tablonun (birleşim *junction* veya *join*) tablosu olarak adlandırılır) tanımlanmasını içerir.

Şekil 7.6'da bir çoka-çok ilişki örneğine yer verilmiştir. *Author* yani yazar bir ya da daha fazla kitap yazabilir; benzer şekilde bir *Book* yani kitap bir ya da daha fazla yazar ile ilişkili olabilir. Böyle bir ilişkiyi tanımlamak üzere bir *BookAuthors* isimli yeni bir birleşim tablosu kullanılır. Bu tablonun anahtarları bileşik (*composite*) olarak da tanımlanabilir.



Şekil 7.6 Çokla-çok ilişki

Böyle bir ilişkiyi tanımlamak üzere *Author* varlığı öncelikle tanımlanır. *Author* varlığı içerisinde *Collection navigation property* (*ICollection List<BookAuthor> BookAuthors*) tanımı yapılır. Benzer şekilde *Book* varlığı içinde de *Collection navigation property* (*ICollection List<BookAuthor> BookAuthors*) tanımı yapılır.

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName => string.Concat(FirstName, " ", LastName);
    public DateTime CreatedDate { get; set; }

    public ICollection<BookAuthor> BookAuthors { get; set; }
}
```

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }

    public BookDetail Detail { get; set; }

    public ICollection<BookAuthor> BookAuthors { get; set; }
}
```

Birleşim tablosu içinde *BookAuthor* isminde bir varlık tanımı yapılır. Bu varlık tanımı *Book* ve *Author* alanlarından gelen *primary key* ifadelerini *foreign key* olarak kullanabilir. Yine aradaki bağlantının doğru bir şekilde yapılmasını sağlamak üzere *simple navigation property* tanımları gerçekleştirilir.

```
public class BookAuthor
{
    public int BookAuthorId { get; set; }
    public int BookId { get; set; }
    public Book Book { get; set; }
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}
```

BookAuthor varlığı içerisinde tanımlanmış olan özellikler ve özelliklerin açıklamasında Tablo 7.1’de yer verilmiştir.

BookAuthor varlığı içerisinde:

Tablo 7.1 BookAuthor varlığı içinde tanımlı olan özellikler

Özellik	Açıklama
BookAuthorId	Primary key
BookId	Foreign key
Book	Simple Navigation Property
AuthorId	Foreign key
Author	Simple Navigation Property

Aradaki ilişkinin istenilen şekilde kurulmasını sağlamak üzere *BookAuthor* varlığı için *OnModelCreating* içerisinde bir *Mapping* tanımı kullanılabilir. *BookAuthorMap* konfigürasyon dosyasının içeriği aşağıdaki gibidir.

```
public class BookAuthorMap : IEntityTypeConfiguration<BookAuthor>
{
    public void Configure(EntityTypeBuilder<BookAuthor> builder)
    {
        builder.HasKey(ba => ba.BookAuthorId);

        builder
            .HasOne(ba => ba.Book)
            .WithMany(b => b.BookAuthors)
            .HasForeignKey(ba => ba.BookId);

        builder
            .HasOne(ba => ba.Author)
            .WithMany(a => a.BookAuthors)
            .HasForeignKey(ba => ba.AuthorId);
    }
}
```

Bu tanımlamada *FluentApi* yaklaşımı ile *BookAuthor* varlığının *Author* ve *Book* ile olan ilişkileri istenilen şekilde tanımlanmıştır.



Entity Framework Core Many-To-Many Relation [24:36]

Book ve **Author** varlıkları arasında çok-açok (**many-to-many**) ilişki kuruyoruz.



Entity Framework Core Projesi GitHub [03:38]

OOP.EFCore projesini GitHub üzerinden nasıl indirip kullanabilirsiniz? İnceliyoruz.



OOP.EFCore

OOP.EFCore Projesini indirerek kaynak kodları düzenleyebilirsiniz.

7.7. Scaffold-DbContext

Bir veritabanı için bir *DbContext* ve varlık türleri (*Entity Types*) için kod üretir. *Scaffold-DbContext*'in bir varlık türü oluşturması için veritabanı tablosunun bir birincil anahtarı olması gerekir.

Tablo 7.2. Scaffold-DbContext için parametreler

-Connection <String>	Veritabanına bağlantı dizesi.
-Provider <String>	Kullanılacak sağlayıcı. Genellikle bu, NuGet paketinin adıdır, örneğin: Microsoft.EntityFrameworkCore.SqlServer.
-OutputDir <String>	Varlık sınıfı dosyalarının yerleştirileceği dizin. Yollar proje dizinine göreler.
-Namespace <String>	Oluşturulan tüm sınıflar için kullanılacak ad alanı. Kök ad alanından ve çıktı dizininden oluşturulacak varsayılanlar.
-Context <String>	Oluşturulan DbContext sınıfı için kullanılacak ad alanı.



Scaffold-DbContext [19:03]

Veri tabanında tanımlı olan nesneler için varlıklar üretmek üzere Scaffold-DbContext kullanılabilir.

7.8. Uygulamalar (1)

- 1) GitHub üzerinden <https://github.com/zcomert/OOP.EFCore> projesini indiriniz. Uygulamayı çalışır hale getiriniz.
- 2) *Console* uygulaması içerisinde bir *Book* nesnesi oluşturup bu nesneyi veri tabanına kayıt ediniz.
 - a. Yazdığınız kodları Refactore edip yeni bir metot olarak, *AddBook()* adı ile *Program* sınıfına ekleyiniz.
- 3) *Console* uygulaması içerisinde *list* adında ve *List<Book>* tipinde bir değişken tanımlayınız. Tanımlama aşamasında beş adet kitap tanımı yapınız.
 - a. Oluşturduğunuz *list* değişkenini *context* açıp *_context.Books.AddRange(list)* gibi bir tanım ile veri tabanına kayıt ediniz. Döngü kullanmayınız.
 - b. Yazdığınız kodları Refactor edip *AddBooks* isimli bir metot olarak *Program* sınıfına yerleştiriniz.
- 4) *Books* tablosunda tüm kayıtların listesini *Console* uygulaması içerisinde listeleyen bir program yazınız. Yazdığınız kodları *Refactore* edip *ListOfBook()* adı ile *Program* sınıfına ayrı bir metot olarak ekleyiniz.
- 5) *Books* tablosundaki ilk kaydı seçip, güncelleyen bir program yazınız.
 - a. Yazdığınız programı Refacotre edip *UpdateBook()* olarak *Program* sınıfına ekleyiniz.
- 6) *Books* tablosundaki son kaydı seçip, silen bir program yazınız.
 - a. Yazdığınız programı Refactore edip *DeleteBook()* olarak *Program* sınıfına ekleyiniz.
- 7) *Book* sınıfından yeni bir nesne üretiniz. *Book* sınıfındaki *Category* özelliği içinde yine *Cageteory* sınıfından bir nesne üretiniz. Tanımladığınız *Book* nesnesini veri tabanına kayıt ediniz.
 - a. *Books* ve *Categories* tablolarındaki kayıtları listeleyiniz.
 - b. Tanımladığınız yeni *Category* örneğinin *Categories* tablosunda yer alıp almadığını kontrol ediniz.
 - c. Yazdığınız programı Refactore edip ayrı bir metot *AddBookWithCategory()* adı ile Program sınıfına ekleyiniz.

- 8) *Book* sınıfından yeni bir örnek üretiniz.
- Ürettiğiniz örneğin *Book.Category* alanı için *Categories* tablosundaki ilk örneği seçiniz ve ilgili property atayınız.
 - Book.BookDetail* için ise ilgili sınıftan (*BookDetail*) bir örnek oluşturup *Book.BookDetail* alanına gerekli atamayı yapınız.
 - Oluşturduğunuz *Book* nesnesini veri tabanına kayıt ediniz.
 - Book*, *Categories* ve *BookDetails* tablolarında yer alan kayıtları listeleyiniz. *Categories* ve *BookDetails* tablolarında yeni kayıtların eklenip eklenmediğini kontrol ediniz.



Entity Framework Core Uygulamalar [26:53]

Bu dersimizde **Entity Framework Core** kullanarak veri manipülasyonu için çeşitli uygulamalar yazıyoruz.

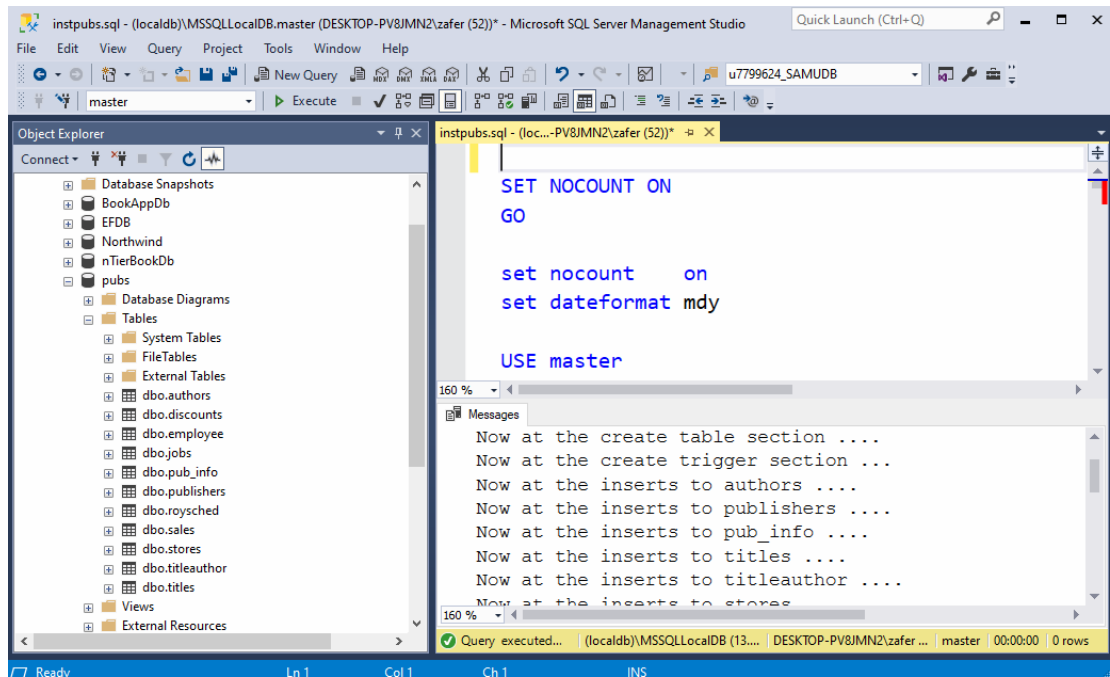
7.9. Uygulamalar (2)

- 1) Microsoft'un GitHub adresini ziyaret ediniz.

<https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs>

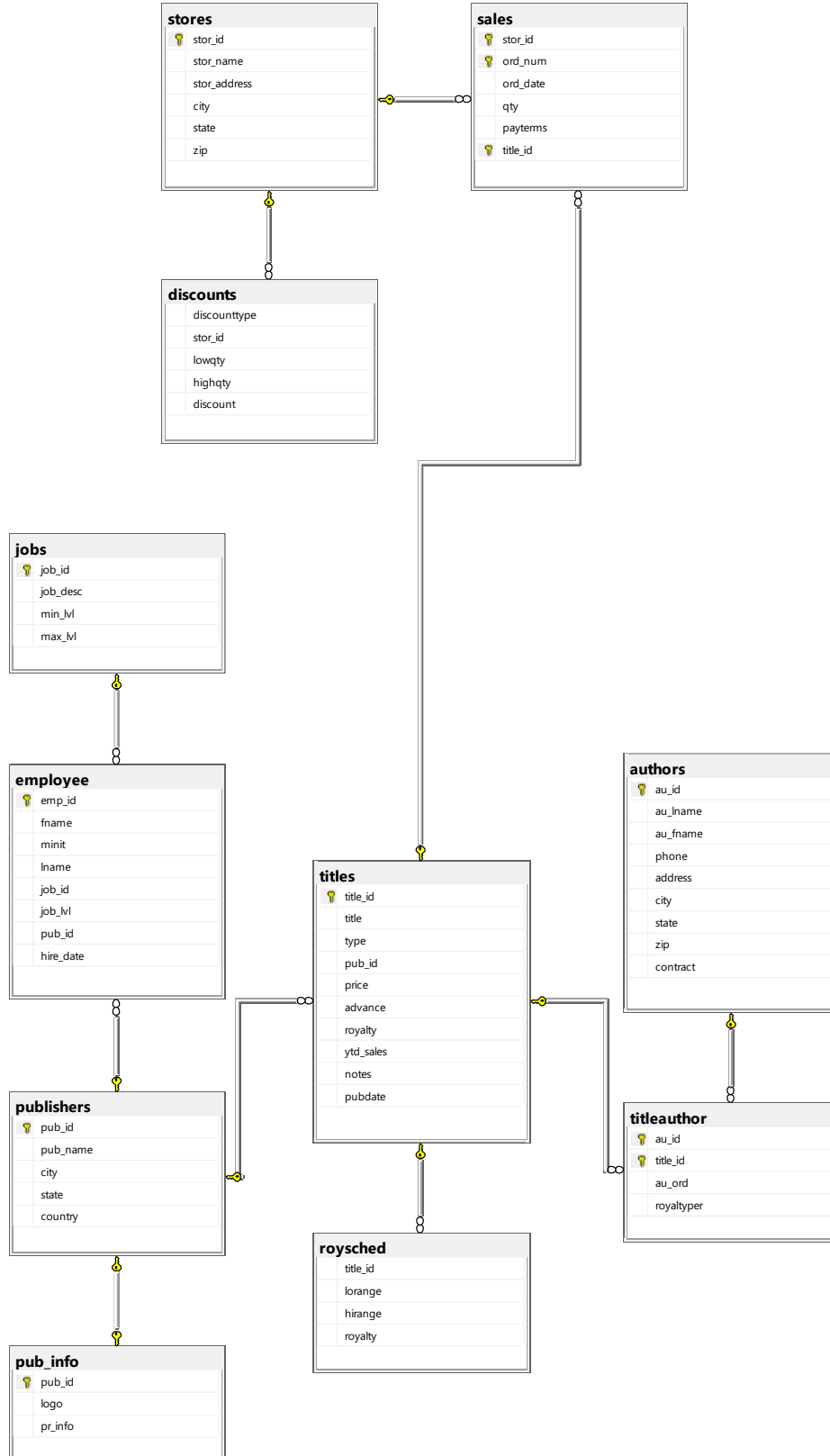
Bu adres üzerinden **instpubs.sql** dosyasını indiriniz.

- 2) İndirdiğiniz dosyayı SQL Server Management Studio üzerinden açarak ilgili SQL ifadelerinin çalışmasını sağlayınız. Bu işlem sonunda Şekil 7.7'de gösterildiği gibi **pubs** veri tabanının SQL Server eklenmesi gerekir.



Şekil 7.7 pubs veri tabanı

- 3) *Pubs* veri tabanı için bir Diyagram oluşturunuz ve ilgili diyagramı inceleyerek veri tabanındaki ilişkilerini gözlemleyiniz.



Şekil 7.8 pubs veri tabanı diyagramı

4) Aşağıda verilen ifadeleri Doğru (+) /Yanlış (-) şeklinde ayırınız.

<i>sales</i> tablosu ve <i>stores</i> arasında bire-bir ilişki vardır.	
<i>publishers</i> ve <i>pub_info</i> arasında bire-bir ilişki vardır.	
<i>employee</i> ve <i>jobs</i> arasında bire-çok ilişki vardır.	
<i>titles</i> tablosu ve <i>authors</i> tablosu arasında çok-çok ilişkisi vardır.	
<i>titleauthor</i> tablosu bir birleşim tablosudur.	
<i>jobs</i> tablosundaki <i>job_id</i> alanı employee tablosunda çok defa tekrar edebilir.	
<i>discounts</i> tablosunun birincil anahtarı (<i>primary key</i>) [<i>discounttype</i>] alanıdır.	

5) *pubsDbContext* şeklinde bir *DbContext* oluşturup aşağıdaki tanımlamaları gerçekleştiriniz.

- Title* ve *Publisher* varlıklarını uygun şekilde tanımlayınız.
- Title* ve *Publisher* için *pubsDbContext* içerisinde *DbSet<T>* tanımlarını gerçekleştiriniz.

6) *Title* ve *Publisher* varlıkları için *Console* uygulaması üzerinde CRUD işlevleri için birer örnek gerçekleştiriniz.

7) Yeni bir proje açınız ve **Scaffold-DbContext** ile *pubs* veri tabanına ait modelin otomatik olarak oluşturulmasını sağlayınız.

- İlgili kod bloklarına ait varlıklar *DAL\Entities* klasörü altında organize edilecektir.
- pubsContext* *DAL* klasörü altında yer alacaktır.

8. Dile Entegre Sorgular (LINQ)

8.1. Giriş

Language integrated query (LINQ), dile entegre edilmiş sorgu, *Haskell*, *XML* ve *SQL* gibi **bildirimsel** (*declarative*) yazım şeklini kullanan bir programlama bileşeni olarak ifade edilebilir.

LINQ, sorgu yeteneklerinin doğrudan C# diline entegrasyonuna dayalı bir dizi teknolojinin adıdır. Geleneksel olarak, verilere yönelik sorgular, derleme zamanında veya *IntelliSense* desteğinde tür denetimi olmaksızın basit dizeler olarak ifade edilir. Ayrıca, normalde her veri kaynağı türü için farklı bir sorgu dilinin öğrenilmesi gerekebilir. SQL veritabanları, XML, JSON belgeleri, vb. veri kaynakları için LINQ ile sorgular yazılabilir. Böylelikle farklı veri kaynakları için yeni sorgulama dilleri öğrenme zorunluluğu ortadan kaldırılır.

Sorgu yazan bir geliştirici için, LINQ teknolojisinin görünür "**dille tümleşik**" kısmı sorgu ifadesidir. Sorgu ifadeleri, bildirim dayalı bir sorgu sözdiziminde yazılır. Sorgu söz dizilimini kullanarak minimum kod ile veri kaynakları üzerinde filtreleme, sıralama ve gruplama işlemleri yapılabilir. SQL veri tabanlarında, ADO.NET Veri Kümelerinde, XML belgelerinde ve akışlarında ve .NET koleksiyonlarında verileri sorgulamak ve dönüştürmek için aynı temel sorgu ifadesi kalıplarını kullanılır.

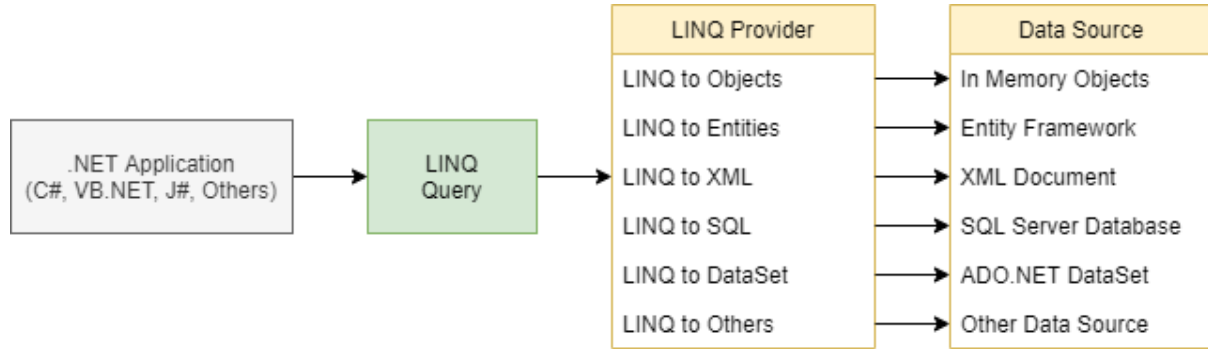
LINQ kullanılan programa, biçimsel veri sorgulama yeteneği kazandırır. Bu teknoloji ile veri tabanları, XML belgeleri, ADO.NET veri kümeleri ve hafızada bulunan koleksiyonlar türündeki veriler üzerinde sorgulama içeren deyimler yazılabilir. Bir başka ifadeyle, LINQ deyimleri kullanılarak söz konusu veri türleri üzerinden seçme, sıralama ve sorgulama işlemleri gerçekleştirilebilir.

Başlıca LINQ Provider'ları aşağıdaki gibi sıralanabilir:

- LINQ to Object
- LINQ to XML (XLINQ)
- LINQ to SQL (DLINQ)
- LINQ to Datasets

- LINQ to Entity

LINQ ifadeleri genellikle bildirimsel olarak tanımlanır. Ancak bu ifadelere karşılık gelen metotların tamamı derleme zamanında yöntem (fonksiyon) çağrılarına dönüştürülür. Şekil 8.1’de LINQ mimarisine yer verilmiştir.



Şekil 8.1 LINQ Mimarisi



LINQ Giriş [15:49]

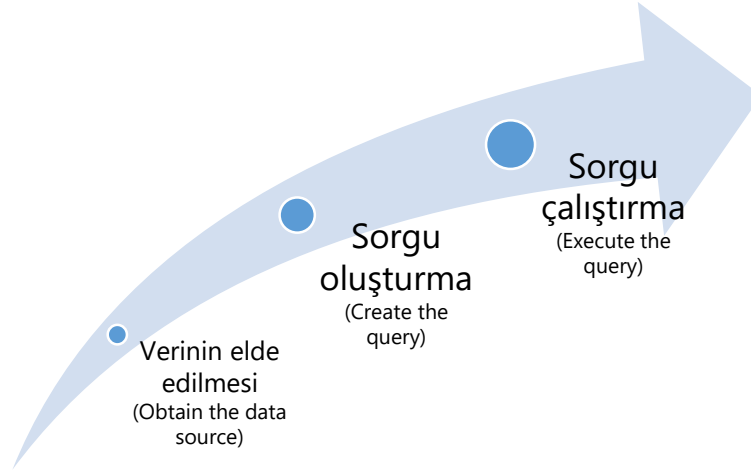
Bu dersimizde LINQ üzerinde genel bilgiler verip basit bir sorgu tasarımı yapıyoruz.

8.2. LINQ Sorguları

Sorgu, bir veri kaynağından veri alan bir ifadedir. Sorgular genellikle özel bir sorgu dilinde ifade edilir. Çeşitli veri kaynakları türleri için zaman içinde farklı diller geliştirilmiştir, örneğin ilişkisel veritabanları için [SQL](#) ve [XML](#) için [XQuery](#). Bu nedenle geliştiriciler, desteklemeleri gereken her tür veri kaynağı veya veri biçimi için yeni bir sorgu dili öğrenmek zorunda kaldılar. LINQ, çeşitli veri kaynakları ve biçimleri arasında verilerle çalışmak için tutarlı bir model sunarak bu durumu basitleştirir. Bir LINQ sorgusunda her zaman nesnelerle çalışılır. XML belgelerinde, SQL veri tabanlarında, ADO.NET veri kümelerinde, .NET koleksiyonlarında ve bir LINQ sağlayıcısının kullanılabildiği diğer tüm biçimlerde verileri sorgulamak ve dönüştürmek için aynı temel kodlama modelleri kullanılır.

8.3. Bir sorgu işleminin üç parçası

Tüm LINQ sorguları üç ayrı bağımsız işlemten oluşur. Şekil 8.2’de bu adımlar özetlenmiştir.



Şekil 8.2 Bir LINQ sorgusunun çalıştırılması için gerçekleştirilen üç adım

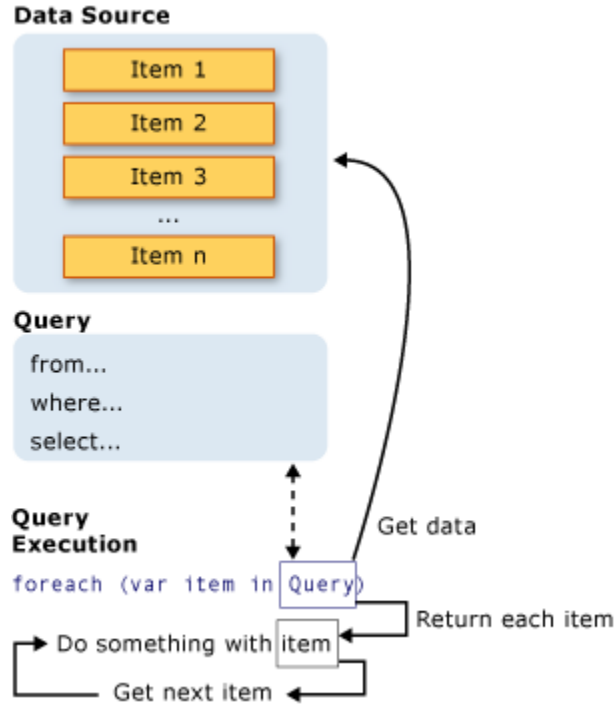
```
static void Main(string[] args)
{
    // 1. data source
    int[] numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7 };
    Console.ReadKey();

    // 2. query creation
    var numQuery = from num in numbers
                   where (num % 2) == 0
                   select num;

    // 3. Query execution
    foreach (int num in numQuery)
    {
        Console.WriteLine("{0,1}", num);
    }

    Console.ReadKey();
}
```

Şekil 8.3’deki çizim, tam sorgu işlemini gösterir. LINQ’da sorgunun yürütülmesi sorgunun kendisinden farklıdır. Başka bir deyişle, yalnızca bir sorgu değişkeni oluşturarak herhangi bir veri almadınız.

Şekil 8.3 Sorgu tasarımı⁴

8.3.1. Veri Kaynağı (Data source)

Önceki örnekte, veri kaynağı bir dizi olduğundan, genel *IEnumerable<T>* arabirimini örtük olarak destekler. Bu gerçek, LINQ ile sorgulanabileceği anlamına gelir. Bir *foreach* ifadesinde bir sorgu yürütülür ve *foreach IEnumerable* veya *IEnumerable<T>* gerektirir. *IEnumerable<T>*'yi veya genel *IQueryable<T>* gibi türetilmiş bir arabirimi destekleyen türlere sorgulanabilir türler denir. *IEnumerable<T>* ve *IQueryable<T>* arasındaki temel fark kısaca şu şekilde özetlenebilir: *IEnumerable<T>* veri tabanına gider, tüm veriyi istemciye getirir ve daha sonra veri üzerinden süzme işlemini gerçekleştirir. *IQueryable<T>* ise sorguyu çözerek sadece ilgili veriyi veri kaynağından getirecek şekilde çalışır. *IEnumerable<T>* **temel yükleme** (lazy loading) desteklemez⁵.

Sorgulanabilir bir tür, LINQ veri kaynağı olarak hizmet vermek için herhangi bir değişiklik veya özel işlem **gerektirmez**. Kaynak veriler sorgulanabilir bir tür olarak bellekte

⁴<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries>

⁵ Tembel yükleme, performansı artırmak ve sistem kaynaklarını kurtarmak için gerçekten ihtiyaç duyulana kadar kaynakların veya nesnelerin yüklenmesini veya başlatılmasını geciktirme uygulamasıdır.

zaten değilse, LINQ sağlayıcısı bunu bu şekilde sunmalıdır. Örneğin, *LINQ to XML*, bir *XML* belgesini sorgulanabilir bir *XElement* türüne yükler:

```
static void Main(string[] args)
{
    // Create a data source from an XML document
    string _path = Path.Combine("D:", "DataSources", "Books.xml");
    XElement books = XElement.Load(_path);

    IEnumerable<XElement> bookList =
        books.Elements();

    foreach (XElement book in bookList)
    {
        Console.WriteLine($"{book.Element("title").Value,-50}" +
            $"{book.Element("price").Value,-5}");
    }

    Console.ReadKey();
}
```

Yukarıdaki kod bloğu örnek programa karşılık gelir. *Book.xml* dosyasını Microsoft'un web sitesinden indirebilirsiniz⁶.

LINQ to SQL ile, önce tasarım zamanında ya manuel olarak ya da Visual Studio'da *LINQ to SQL* Araçlarını kullanarak bir nesne-ilişkisel eşleme yaratırsınız. Sorgularınızı nesnelere karşı yazarsınız ve çalışma zamanında *LINQ to SQL*, veri tabanı ile iletişimi yönetir. Aşağıdaki örnekte, *Employee Northwind* veri tabanındaki belirli bir tabloyu temsil eder ve sorgu sonucunun türü olan *IQueryable<T>*, *IEnumerable<T>* ögesinden türetilir.

⁶ [https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms762271\(v=vs.85\)](https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v=vs.85))

```
static void Main(string[] args)
{
    var db = new NorthwindDbContext();
    IQueryable<Employee> employeeQuery =
        from employee in db.Employees
        where employee.City == "London"
        select employee;

    foreach (var employee in employeeQuery)
    {
        Console.WriteLine(employee);
    }

    Console.ReadKey();
}
```

8.3.2. Sorgu (Query)

Sorgu, veri kaynağından veya kaynaklarından hangi bilgilerin alınacağını belirtir. İsteğe bağlı olarak, bir sorgu ayrıca bu bilgilerin döndürülmeden önce nasıl sıralanacağını, gruplandırılacağını ve şekillendirileceğini de belirtebilir. Bir sorgu, bir sorgu değişkeninde depolanır ve bir sorgu ifadesi ile başlatılır. Sorgu yazmayı kolaylaştırmak için C#, yeni sorgu sözdizimi sunar.

Temelde LINQ sorgu ifadesi üç yan tümce içerir: *from*, *where* ve *select*. (SQL'e aşina iseniz, cümleciklerin sırasının SQL'deki sıranın tersine olduğunu fark etmişsinizdir.) *From* cümlesi veri kaynağını belirtir, *where* cümlesi filtreyi uygular ve *select* cümlesi, cümlemin türünü belirtir. Burada önemli olan nokta, LINQ'da sorgu değişkeninin kendisinin hiçbir işlem yapmaması ve hiçbir veri döndürmemesidir. Yalnızca, sorgu daha sonraki bir noktada yürütüldüğünde sonuçları üretmek için gereken bilgileri depolar.

8.3.3. Sorgu ifadelerine genel bakış

- Sorgu ifadeleri, herhangi bir *LINQ* etkin veri kaynağından verileri sorgulamak ve dönüştürmek için kullanılabilir. Örneğin, tek bir sorgu bir *SQL* veri tabanından veri alabilir ve çıktı olarak bir *XML* akışı üretebilir.
- Sorgu ifadelerinde ustalaşmak kolaydır çünkü birçok tanıdık C# dil yapısı kullanılır.
- Bir sorgu ifadesindeki değişkenlerin tümü güçlü bir şekilde yazılır, ancak çoğu durumda derleyici bunu çıkarabileceği için türü açıkça belirtmeniz **gerekmez**.
- Bir sorgu *foreach* gibi iteratif bir sorgu değişkeni ile karşılaşmadığı sürece çalıştırılmaz.

- Derleme zamanında, sorgu ifadeleri, C# belirtiminde belirtilen kurallara göre Standart Sorgu Operatörü yöntem çağrılarına dönüştürülür. **Sorgu sözdizimi** (*query syntax*) kullanılarak ifade edilebilen herhangi bir sorgu, yöntem sözdizimi (*method syntax*) kullanılarak da ifade edilebilir. Ancak, çoğu durumda sorgu sözdizimi daha okunaklı ve özlüdür.
- Kural olarak, LINQ sorguları yazarken, mümkün olduğunda sorgu sözdizimini ve gerektiğinde yöntem sözdiziminin kullanılması tavsiye edilir. İki farklı form arasında anlamsal veya performans farkı yoktur. Sorgu ifadeleri genellikle yöntem sözdiziminde yazılmış eşdeğer ifadelerden daha okunabilir.
- *Count* veya *Max* gibi bazı sorgu işlemlerinin eşdeğer sorgu ifadesi yan tümcesi yoktur ve bu nedenle bir yöntem çağrısı olarak ifade edilmelidir. Yöntem sözdizimi, sorgu sözdizimi ile çeşitli şekillerde birleştirilebilir.
- Sorgu ifadeleri, sorgunun uygulandığı türe bağlı olarak ifade ağaçlarına (*expression trees*) veya temsilcilere derlenebilir. *IEnumerable<T>* sorguları, temsilcilere (**delegate**) derlenir. *IQueryable* ve *IQueryable<T>* sorguları, ifade ağaçlarına (**expression tree**) derlenir.



LINQ Query Syntaxes [12:20]

Bu dersimizde LINQ üzerinde sorgu tasarımı yaparken Query Syntax, Method Syntax ve Mix Syntax gibi söz dizimlerini inceliyoruz.

8.3.4. Sorgu Yürütme (Query execution)

8.3.4.1. Ertelenmiş Yürütme (Deferred Execution)

Daha önce belirtildiği gibi, sorgu değişkeninin kendisi yalnızca sorgu komutlarını depolar. Bir *foreach* ifadesinde sorgu değişkeni üzerinde yinelenene kadar sorgunun gerçek yürütülmesi ertelenir. Bu kavram ertelenmiş yürütme (**deferred execution**) olarak adlandırılır ve aşağıdaki örnekte gösterilmiştir:

```
foreach (var employee in employeeQuery)
{
    Console.WriteLine(employee);
}
```

Bir başka ifadeyle *employeeQuery* sorgusu *iteratif* bir yapı ya da *foreach* gibi bir deyim ile karşılaşmadığı sürece çalışması ertelenecektir.

Sorgu değişkeninin kendisi hiçbir zaman sorgu sonuçlarını tutmadığından, istediğiniz sıklıkta çalıştırabilirsiniz. Örneğin, ayrı bir uygulama tarafından sürekli olarak güncellenen bir veri tabanınız olabilir. Uygulamanızda, en son verileri alan bir sorgu oluşturabilir ve her seferinde farklı sonuçlar almak için belirli aralıklarla tekrar tekrar çalıştırabilirsiniz.

8.3.4.2. Derhal Yürütmeyi Zorlamak (Forcing Immediate Execution)

Bir dizi kaynak öge üzerinde toplama işlevleri gerçekleştiren sorgular, önce bu öğeler üzerinde yinelenmelidir. Bu tür sorgulara örnek olarak *Count*, *Max*, *Average* ve *First* verilebilir. Bunlar, bir sonuç döndürmek için sorgunun kendisinin *foreach* kullanması gerektiğinden, açık bir *foreach* ifadesi olmadan yürütülür. Ayrıca, bu tür sorguların bir *IEnumerable* koleksiyonu değil, tek bir değer döndürdüğünü unutmayın.

Aşağıdaki sorgu, kaynak dizideki çift sayıların sayısını döndürür:

```
static void Main(string[] args)
{
    var numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7 };
    var evenNumQuery = from num in numbers
                       where num % 2 == 0
                       select num;

    int evenNumCount = evenNumQuery.Count();
    Console.WriteLine(evenNumCount);

    Console.ReadKey();
}
```

Herhangi bir sorgunun hemen yürütülmesini zorlamak ve sonuçlarını önbelleğe almak için *ToList* veya *ToArray* yöntemlerini çağırabilirsiniz.

```
static void Main(string[] args)
{
    var numbers = new List<int> { 0,1, 2, 3, 4, 5, 6, 7 };

    List<int> numQuery2 =
        (from num in numbers
         where (num % 2) == 0
         select num).ToList();

    numQuery2.ForEach(x => Console.WriteLine(x));
    Console.ReadKey();
}
```

8.4. LINQ Operatörleri

- Projection Operators
- Ordering Operators
- Filtering Operators
- Set Operators
- Quantifier Operators
- Grouping Operators
- Partitioning Operators
- Equality Operators
- Element Operators
- Conversion Operators
- Concatenation Operators
- Aggregation Operators
- Generation Operators
- Join Operators
- Custom Sequence Operators
- Miscellaneous Operators

8.5. LINQ Sorgularına Karşılık Üretilen SQL Cümlelerinin Elde Edilmesi

LINQ sorguları veri tabanından kayıtları alırken tasarlanan sorgulara karşılık SQL cümleleri elde edilir. Bu cümleleri elde etmek üzere Console uygulamasına [Microsoft.Extensions.Logging.Console](#) isimli yeni bir paket eklenebilir. İlgili konfigürasyonu için Microsoft dokümanları takip edilebilir.

EF Core ile veri erişimi bir model kullanılarak gerçekleştirilir. Bir model, varlık (*entity*) sınıflarından ve veritabanı ile bir oturumu temsil eden bir bağlam (*context*) nesnesinden oluşur. *Context* nesnesi, verileri sorgulamaya ve kaydetmeye izin verir.



Microsoft.Extensions.Logging in EF Core

İlgili paketin konfigürasyonunu sağlamak üzere Microsoft dokümanları takip edilebilir.

Log ifadelerini görebilmek için *Context* sınıfına aşağıdaki kod bloklarının eklenmesi gerekir.

```
public static readonly ILoggerFactory MyLoggerFactory =
    LoggerFactory.Create(builder => { builder.AddConsole(); });
```

Bu ifade ile birlikte *OnConfiguring* metodu içerisinde *UseLoggerFactory()* metodu kullanılmalıdır.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){
    optionsBuilder
        .UseLoggerFactory(MyLoggerFactory)
        .UseSqlite("Data Source = D:\\Northwind.db");
}
```



LINQ Logging Console [10:30]

Bu dersimizde LINQ üzerinde sorgu tasarımlarını gerçek zamanlı bir veri tabanı üzerinde test etmek amacıyla EF Core projemize dahil ediyoruz. LINQ sorgularının SQL karşılıklarını da almak üzere Microsoft.Extensions.Logging ve Microsoft.Extensions.Logging.Console paketlerinin kurulumunu gerçekleştiriyoruz.

8.6. Select Operatörü

8.6.1. Select ile sadece bir alanı seçme

Select operatörü ile bir sorgu sonucunda sadece istenilen alanlar alınabilir. Buna göre sonuç setinin veri tipi belirlenebilir.

```
var context = new NorthwindDbContext();
// query syntax
IEnumerable<int> basicSelectQuery =
    (from emp in context.Employees
     select emp.EmployeeId)
     .ToList();

// method syntax
IEnumerable<int> basicPropterySelection =
    context.Employees.Select(emp => emp.EmployeeId);
```


Bu LINQ sözcüğü *Employees* tablosunda tüm alanları seçmek yerine sadece *Employees.EmployeeId* alanını getirir.

SQL Karşılığı

```
SELECT [e].[EmployeeId] FROM [Employees] AS [e]
```

8.6.2. Select ile birden fazla alanı seçme

Benzer şekilde bu yöntem ile bir varlık ya da nesnenin sadece istenilen özellikleri de seçilebilir.

```
var context = new NorthwindDbContext();

var selectOperatorQS = (from emp in context.Employees
                        select new Employee
                        {
                            EmployeeId = emp.EmployeeId,
                            FirstName = emp.FirstName,
                            LastName = emp.LastName
                        }
                        );

var selectOperatorMS = context.Employees
    .Select(emp => new Employee()
    {
        EmployeeId = emp.EmployeeId,
        FirstName = emp.FirstName,
        LastName = emp.LastName
    })
    .ToList();

foreach (var emp in selectOperatorMS)
{
    Console.WriteLine($"{emp.FirstName} {emp.LastName}");
}
```

Bu durumda Employee varlığı için oluşturulacak yeni IEnumerable<Employee> tipinde veri içerisinde Employee varlığının sadece ilgili alanlarına karşılık veriler atanır.

SQL Karşılığı

```
SELECT [e].[EmployeeId], [e].[FirstName], [e].[LastName] FROM [Employees] AS [e]
```

8.6.3. Anonim nesne oluşturma

Bir LINQ ifadesi sonrası dönen sorgu sonucuna bağlı olarak anonim tipler de oluşturulabilir.

```
var context = new NorthwindDbContext();

var selectOperatorQS =
(from emp in context.Employees
select new
{
    Id = emp.EmployeeId,
    FullName = string.Concat(emp.FirstName, " ", emp.LastName)
});

var selectOperatorMS = context.Employees
    .Select(emp => new {
        Id = emp.EmployeeId,
        FullName = string.Concat(emp.FirstName, " ", emp.LastName)
    }).ToList();

foreach (var item in selectOperatorMS)
{
    Console.WriteLine(item);
}
```

Sorgu sonucu oluşan ekran çıktısına Şekil 8.4'de yer verilmiştir.

```

F:\zc\Dersler\Nesne Yönelimli Programlama\Uygulamalar\OOP\OOP.LinqApp\bin\Debug\net5.0\OOP.LinqApp.exe
Command[20101]
    Executed DbCommand (19ms) [Parameters=[],
    CommandType='Text', CommandTimeout='30']
    SELECT [e].[EmployeeId], [e].[FirstName],
    [e].[LastName]
    FROM [Employees] AS [e]
{ Id = 1, FullName = Nancy Davolio }
{ Id = 2, FullName = Andrew Fuller }
{ Id = 3, FullName = Janet Leverling }
{ Id = 4, FullName = Margaret Peacock }
{ Id = 5, FullName = Steven Buchanan }
{ Id = 6, FullName = Michael Suyama }
{ Id = 7, FullName = Robert King }
{ Id = 8, FullName = Laura Callahan }
{ Id = 9, FullName = Anne Dodsworth }
{ Id = 12, FullName = Mehmet Demir }
{ Id = 15, FullName = Ahmet Türkoglu }
{ Id = 1012, FullName = Zeynep Yılmaz }
{ Id = 1016, FullName = Melek Subaşı }

```

Şekil 8.4 Anonim sorgu örneklerinin Console uygulamasında yazdırılması



LINQ Select Clauses [15:53]

Bu dersimizde LINQ ile Select işlevlerini yerine getiriyoruz. Bu kapsamda ilgili Select ile farklı projeksiyon ifadelerinin nasıl elde edildiğini inceliyoruz.

8.6.4. Northwind Veri Tabanı için Seçme Sorguları

Sorgu tasarımı gerçekleştirirken *query syntax* ya da *method syntax* yaklaşımı tercih edilebilir. Bu noktada her iki yaklaşım sonucu elde edilecek veriler aynı olacaktır.

Employees tablosundaki tüm kayıtların listelenmesini sağlayan LINQ ifadesi

Query Syntax

```
var q = from emp in _context.Employees
        select emp;
```

Method Syntax

```
_context.Employees.ToList()
    .ForEach(e => Console.WriteLine(e));
```

SQL Karşılığı

```
SELECT "e"."EmployeeId", "e"."City", "e"."FirstName", "e"."LastName" FROM "Employees" AS "e"
```

Employee tablosundaki ilk beş kaydın seçilmesi

Method Syntax

```
_context.Employees  
    .Take(5)  
    .ToList()  
    .ForEach(e => Console.WriteLine(e));
```

SQL Karşılığı

```
SELECT "e"."EmployeeId", "e"."City", "e"."FirstName", "e"."LastName" FROM "Employees" AS "e" LIMIT @_p_0
```

Employee tablosundaki kayıtlar *FirstName* alanına göre Artan şekilde sıralanması

Query Syntax

```
var query = from emp in _context.Employees  
             orderby emp.FirstName  
             select emp;
```

SQL Karşılığı

```
SELECT "e"."EmployeeId", "e"."City", "e"."FirstName", "e"."LastName" FROM "Employees" AS "e" ORDER BY "e"."FirstName"
```

Employee tablosundaki kayıtların *LastName* alanına göre azalan ve daha sonra *FirstName* alanına göre artan şekilde sıralanması

Method Syntax

```
_context
    .Employees
    .OrderByDescending(e => e.LastName)
    .ThenByDescending(e => e.FirstName)
    .ToList()
    .ForEach(e => Console.WriteLine(e));
```

SQL Karşılığı

```
SELECT "e"."EmployeeId", "e"."City", "e"."FirstName", "e"."LastName" FROM "Employees" AS
"e" ORDER BY "e"."LastName" DESC, "e"."FirstName"
```

Modele göre seçme ifadesinin düzenlenmesi

Query Syntax

```
var context = new NorthwindDbContext();

// query syntax
var employeeQuery =
    from emp in context.Employees
    orderby emp.FirstName
    select new EmployeeModel
    {
        FirstName = emp.FirstName,
        LastName = emp.LastName
    };

employeeQuery.ToList().ForEach(e =>
    Console.WriteLine(e));
```

Method Syntax

```
// method syntax
context.Employees
    .OrderBy(e => e.FirstName)
    .Select(e => new EmployeeModel
    {
        FirstName = e.FirstName,
        LastName = e.LastName
    })
    .ToList()
    .ForEach(e =>
        Console.WriteLine(e));
```

Select ifadesiyle sadece istenilen alanların seçilmesi

Query and Method Syntax

```
var query = from emp in _context.Employees
             select new Employee
             {
                 EmployeeId = emp.EmployeeId,
                 FirstName = emp.FirstName
             };

query.ToList().ForEach(e => Console.WriteLine(e));

Console.WriteLine();

_context
    .Employees
    .Select(emp => new Employee()
    {
        EmployeeId = emp.EmployeeId,
        FirstName = emp.FirstName
    })
    .ToList()
    .ForEach(e => Console.WriteLine(e));
```

8.7. SelectMany Operatörü

SelectMany operatörü hazırlanan sorgu sonucu oluşan veriyi `IEnumerable<T>` şeklindeki bir dizye aktarır.

```
var list = new List<string>() { "Ahmet", "Mehmet", "Eren" };
IEnumerable<char> querySyntax = from str in list
                                from ch in str
                                select ch;

foreach (var c in querySyntax)
{
    Console.Write(c + " ");
}

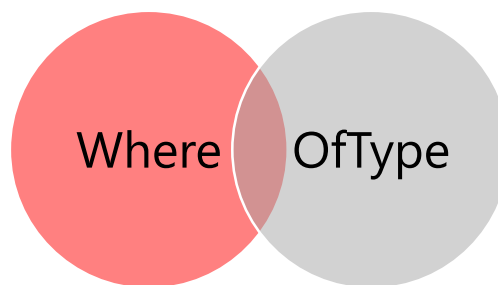
Console.WriteLine();
var methodSyntax = list.SelectMany(n => n);
foreach (var c in methodSyntax)
{
    Console.Write(c + " ");
}
```

Böyle bir örnek sonucunda isim listesinde yer alan bir ismin karakterlerinin arasına bir boşluk karakterinin eklendiği gözlemlenebilir.

8.8. Filtreleme Sorguları

8.8.1. Filtreleme Fonksiyonları

LINQ ifadeleriyle birlikte filtreleme için iki metot kullanılır. İlgili metotlara Şekil 8.5’de yer verilmiştir.

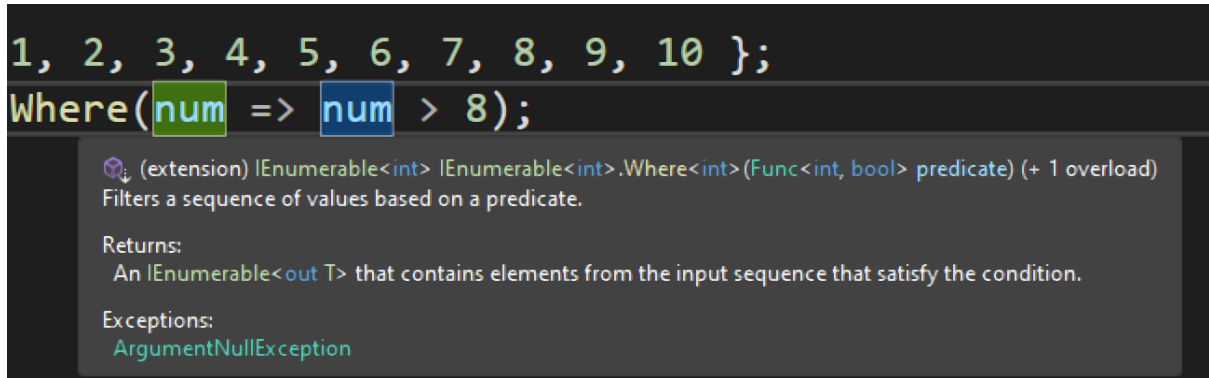


Şekil 8.5 Filtreleme fonksiyonları

Standart olarak kullanılan *where* sözcüğü *LINQ* içerisinde *Filtering Operators* kategorisi altında yer almaktadır.

```
var list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var filteredNumbers = list.Where(num => num > 8);
```

Yukarıdaki kod parçası içerisinde *filteredNumbers* değişkeni *IEnumerable<int>* şeklinde bir tipe sahip olacaktır. *Where* metodu incelendiğinde ise ilgili metodun bir *Extension* metod olduğu görülecektir. Şekil 8.6'da ilgili metodun detaylarına yer verilmiştir.



Şekil 8.6 Extension metod

Görüldüğü üzere bu metod **Func<int,bool> predicate** şeklinde bir ifadeyi metod olarak kabul etmektedir. Bu *generic delegate* tanıımıdır ki *int* veri tipini parametre olarak kabul edip, *true* ya da *false* şeklinde yani *bool* tipinde bir dönüşe sahiptir. Bu tanımda giriş parametresi zorunlu olmasa da çıkış parametresi zorunludur.

```
static void Main(string[] args)
{
    var list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    var filteredNumbers = list.Where(num => CheckNumber(num));
    Console.ReadKey();
}

private static bool CheckNumber(int number) =>
    number > 8 ? true : false;
```

Yukarıdaki *CheckNumber* fonksiyonunu **Func<int,bool>** predicate tanımı ile de yazmak mümkündür.

Herhangi bir lambda ifadesi bir temsilci türüne (*delegate type*) dönüştürülebilir. Bir lambda ifadesinin dönüştürülebileceği *delegate type*, parametrelerinin türleri ve dönüş değeri ile tanımlanır. Bir lambda ifadesi bir değer döndürmezse, *Action* delegate türlerinden birine dönüştürülebilir; aksi takdirde, *Func delegate types* türlerinden birine dönüştürülebilir. Örneğin,

iki parametresi olan ve hiçbir değer döndürmeyen bir lambda ifadesi, bir *Action<T1,T2>* temsilcisine dönüştürülebilir ⁷.

Bu durumda yukarıdaki kod parçasığı aşağıdaki gibi yazılabilir.

```
var list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

Func<int, bool> predicate = i => i > 8;
var filteredNumbers = list.Where(predicate);
foreach (var item in filteredNumbers)
{
    Console.WriteLine(item);
}
```



LINQ Where Clauses [10:08]

Bu dersimizde *where* sözcüğünün çalışma prensibini inceliyoruz. *Func<Tin,Tout>* predicate ifadesi için örnek bir tanım gerçekleştiriyoruz.

8.8.2. OfType Operatörü

Bir veri kaynağındaki spesifik verileri filtreleme üzere *OfType* operatörü kullanılır. Aşağıdaki örnekte liste içerisinde bir birinden farklı veri tipine sahip örnekler *object* olarak tutulmaktadır. İstenen ise sadece tam sayı (*int*) verilerinin alınmasıdır.

⁷ <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions>

```
var list = new ArrayList
{
    "Ahmet",
    "Mehmet",
    18,
    28,
    "Hatice",
    true,
    false,
    40,
    50,
    60,
    DateTime.Now
};

var intData = list.OfType<int>().ToList();
foreach (var item in intData)
{
    Console.WriteLine(item);
}
```

Daha önce Generic programlama konseptini inceledik. Buna göre LINQ *OfType<T>* işlevini yerine getirebilecek bir metod tasarlamayı deneyiniz.



LINQ OfType<T> Clauses [11:46]

Bu dersimizde LINQ ile süzme işlevlerini yerine getiriyoruz. Bu kapsamda *OfType<T>()* metodu inceliyoruz.

8.8.3. Northwind veri tabanı için örnekler

Products tablosunda *UnitPrice* alanı 50'den büyük olan kayıtların listelenmesi

Query Syntax

```
var query = from prd in _context.Products
             where prd.UnitPrice > 50
             select prd;
```

SQL Karşılığı

```
SELECT "p"."ProductId", "p"."ProductName", "p"."UnitPrice", "p"."UnitsInStock" FROM
"Products" AS "p" WHERE ef_compare("p"."UnitPrice", '50.0') > 0
```

Products tablosunda *UnitsInStock* alanı 10'dan büyük ve 15'den küçük olan kayıtların yine *UnitsInStock* alanına göre artan şekilde sıralanmasını sağlayıp, ilk beş kaydı alacak şekilde bir sorgunun tasarlanması

Method Syntax

```
var query = _context
    .Products
    .Where(p => p.UnitsInStock > 10 && p.UnitsInStock < 15)
    .OrderBy(p => p.UnitsInStock)
    .Take(5)
    .ToList();
```

SQL Karşılığı

```
SELECT "p"."ProductId", "p"."ProductName", "p"."UnitPrice", "p"."UnitsInStock" FROM
"Products" AS "p" WHERE ("p"."UnitsInStock" > 10) AND ("p"."UnitsInStock" < 15) ORDER BY
"p"."UnitsInStock" LIMIT @__p_0
```

Products tablosunda *ProductName* alanı içerisinde <rt> ifadesini barındıran ürünlerin listesinin alınması

Query Syntax

```
var query = from prd in _context.Products
             where prd.ProductName.Contains("<rt>")
             select prd;
```

SQL Karşılığı

```
SELECT "p"."ProductId", "p"."ProductName", "p"."UnitPrice", "p"."UnitsInStock" FROM
"Products" AS "p" WHERE ('rt' = '') OR (instr("p"."ProductName", 'rt') > 0)
```



LINQ Clauses on Northwind <T> [11:46]

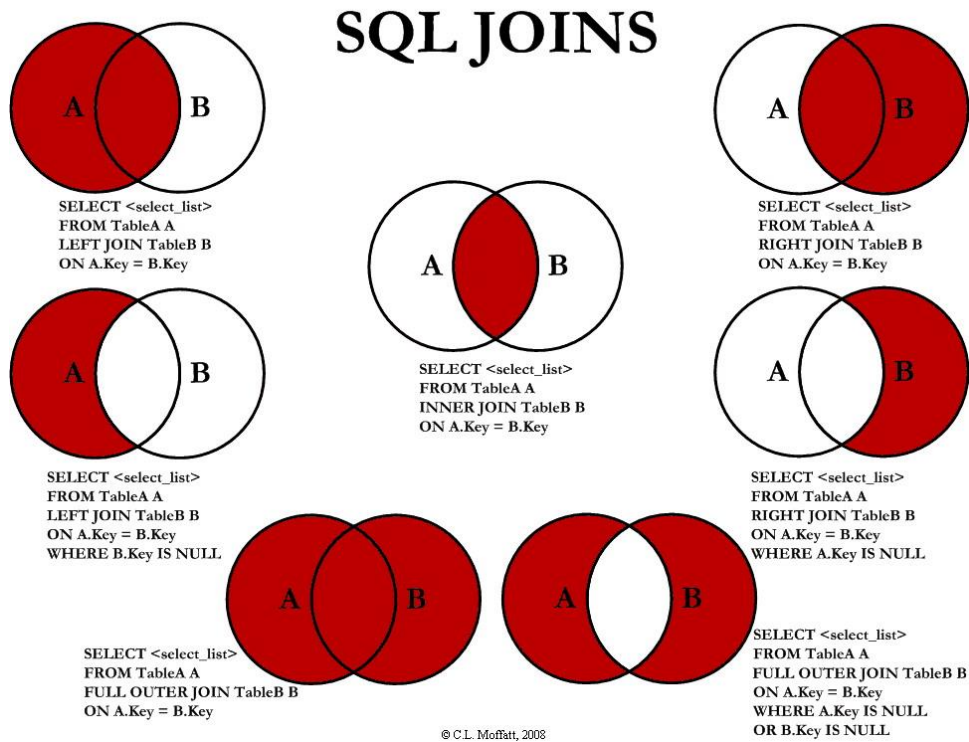
Bu dersimizde LINQ ile süzme işlevlerini yerine getiriyoruz. Bu kapsamda uygulamalarımızı Northwind veri tabanı üzerinde gerçekleştiriyoruz.

8.9. Birleştirme (Join) Sorguları

Farklı birleşim ifadeleri kullanıldığında bir ya da daha fazla tablonun birleştirilmesi sonucu elde edilen sonuç seti değişmektedir. Bu durum Şekil 8.7'de gösterilmiştir.

SQL Server üzerinde dört farklı birleşim ifadesi (join types) vardır:

- Inner join
- Outer join
 - Left outer join
 - Right outer join
 - Full outer join
- Cross join
- Group join



Şekil 8.7 SQL Join

Product ve *Category* alanlarını birleştirmek üzere sorgu tasarımı yapıyoruz.

Sorgu incelendiğinde *context* üzerinden hem *Products* hem de *Categories* tablolarından kayıtların seçildiği görülmektedir. Bu kapsamda söz konusu kayıtların tutarlı bir şekilde verileri getirmesi için *Products.CategoryId* alanı ile *Categories.CategoryId* alanlarının *join*, *on* ve *equals* deyimleri ile birleştirildiği görülmektedir.

```
var context = new NorthwindDbContext();
var query = from p in context.Products
             join c in context.Categories on p.CategoryId equals c.CategoryId
             select new ProductCategory
             {
                 Product = p,
                 Category = c
             };

foreach (var item in query)
{
    Console.WriteLine($"{item.Product.ProductId,-5}" +
                      $"{item.Product.ProductName,-35} " +
                      $"{item.Category.CategoryName,-15}");
}
```

SQL İfadesi

```
SELECT [p].[ProductId], [p].[CategoryId], [p].[ProductName], [p].[UnitPrice], [p].[UnitsInStock],
[c].[CategoryId], [c].[CategoryName] FROM [Products] AS [p] INNER JOIN [Categories] AS
[c] ON [p].[CategoryId] = [c].[CategoryId]
```

Categories ve *Products* tablolarının bağlanması

Benzer şekilde ilgili *Join* işleminin yapılabilmesini sağlamak amacıyla *Method Syntax* yaklaşımı da kullanılabilir. Bu durumda *Products* ifadesi *context* üzerinden çağrıldıktan sonra *Include()* deyimini ile *Category Navigation Property* ifadesi kullanılarak ilgili alanın bilgilerinin de sorgu sonucuna dâhil edilmesi sağlanır.

```
var context = new NorthwindDbContext();
context.Products
    .Include(p => p.Category)
    .ToList()
    .ForEach(p => Console.WriteLine($"{p.ProductId,-5} " +
    $"{p.ProductName,-45}" +
    $"{p.Category.CategoryName,-15}"));
```

SQL Karşılığı

```
SELECT [p].[ProductId], [p].[CategoryId], [p].[ProductName], [p].[UnitPrice], [p].[UnitsInStock],
[c].[CategoryId], [c].[CategoryName] FROM [Products] AS [p] LEFT JOIN [Categories] AS [c]
ON [p].[CategoryId] = [c].[CategoryId]
```

Yalnız dikkate edilirse sorgu ifadesinde *Inner/outer Join* ifadesi ile sorgu tasarımı gerçekleşirken; *method syntax* yaklaşımında *Left Join* ifadesi yer almıştır. Şekil 8.7'de SQL bağlantı ifadelerine yer verilmiştir. İlgili şekilden de anlaşılacağı üzere *left join*, *right join*, *inner join* ve *full outer join* sorgularının kapsadığı kayıtlar referans ve ilişkili tablolar açısından farklılıklar taşımaktadır.

Categories ve *Products* tablolarını birleştiren sorgu tasarımı

Categories.Products (*Collection Navigation Property*) yardımı ile *Products* tablosundaki kayıtlara erişim sağlanabilir.

```
var context = new NorthwindDbContext();
var query = context.Categories
    .Include(p => p.Products);

foreach (var c in query)
{
    Console.WriteLine($"{c.CategoryName}");
    foreach (var p in c.Products)
    {
        Console.WriteLine($"{p.ProductName,-15}");
    }
}
```

Yukarıda *method syntax* yaklaşımı ile tasarlanan sorguda öncelikle *Categories* tablosundaki kayıtlar elde edilmektedir. Daha sonra *Categories.Products* alanı ile *Product* tablosundaki

kayıtlara erişim sağlanmaktadır. İlgili sorgu incelendiğinde *Category.CategoryId* alanı ile *Product.CategoryId* alanlarının eşleştirildiği görülmektedir.

SQL İfadesi

```
SELECT [c].[CategoryId], [c].[CategoryName], [p].[ProductId], [p].[CategoryId],  
[p].[ProductName], [p].[UnitPrice], [p].[UnitsInStock] FROM [Categories] AS [c] LEFT JOIN  
[Products] AS [p] ON [c].[CategoryId] = [p].[CategoryId] ORDER BY [c].[CategoryId],  
[p].[ProductId]
```



LINQ Join Clauses [19:35]

Bu dersimizde birden fazla tablo üzerinden gelen kayıtların Inner Join, Outer Join, Cross Join ve Group join gibi farklı birleşim türleri ile nasıl ifade edildiğini inceliyoruz. Bu kapsamda, birleşim ifadelerinin hem SQL cümlelerini hem de LINQ cümlelerini inceliyoruz.

8.10. Set Operatörü

LINQ'deki *Set* Operatörleri, aynı veya farklı veri kaynakları içindeki öğelerin varlığına ve yokluğuna dayalı olarak sonuç kümesini üretmek için kullanılır. Bu, bu işlemlerin tek bir veri kaynağında veya birden fazla veri kaynağında gerçekleştirildiği ve çıktıda verilerin bir kısmının mevcut olduğu ve verilerin bir kısmının bulunmadığı anlamına gelir.

LINQ içinde kullanılan *Set* operatörleri:

- Distinct
- Except
- Intersec
- Union

Distinct ile veri kümesi içerisinde tekrar eden öğelerin ortadan kaldırılması sağlanır.

```
var l1 = new List<int> { 1, 3, 3, 5, 7, 9, 10 };
    var l2 = new List<int> { 2, 4, 4, 6, 8, 9, 10 };

    var l3 = l1.Distinct();

    foreach (var item in l3)
    {
        Console.WriteLine($"{item,-3}");
    }
```

Except ile harici tutulacak olan öğeler belirtilir.

```
var l1 = new List<int> { 1, 3, 3, 5, 7, 9, 10 };
    var l2 = new List<int> { 2, 4, 4, 6, 8, 9, 10 };

    var l3 = l1.Except(l2);

    foreach (var item in l3)
    {
        Console.WriteLine($"{item,-3}");
    }
```

Intersect ile iki veri setinin kesişim kümesi elde edilebilir.

```
var l1 = new List<int> { 1, 3, 3, 5, 7, 9, 10 };
    var l2 = new List<int> { 2, 4, 4, 6, 8, 9, 10 };

    var l3 = l1.Intersect(l2);

    foreach (var item in l3)
    {
        Console.WriteLine($"{item,-3}");
    }
```

Union ile iki veri kümesinin birleştirilmesi sağlanabilir.

```
var l1 = new List<int> { 1, 3, 3, 5, 7, 9, 10 };
    var l2 = new List<int> { 2, 4, 4, 6, 8, 9, 10 };

    var l3 = l1.Union(l2);

    foreach (var item in l3)
    {
        Console.WriteLine($"{item,-3}");
    }
```


9. Aspect Oriented Programming (AOP)

9.1. AOP Genel Bakış

Aspect-oriented programming (AOP), bir başka ifadeyle cepheye-yönelik programlama kulağa oldukça karmaşık gelen teknik bir kavram, fakat algılandığı kadar karmaşık değil. AOP tekniği yazılımcılara basmakalıp ve tekrar eden kod bloklarının kopyalayıp, yapıştırmaktan daha fazla zaman kazandıran ve bu konuda geliştiricileri destekleyen bir programlama tekniğidir. AOP bunu yaparken hem kod bloklarının okunabilirliğini artırır hem de projelerinize değer katar.

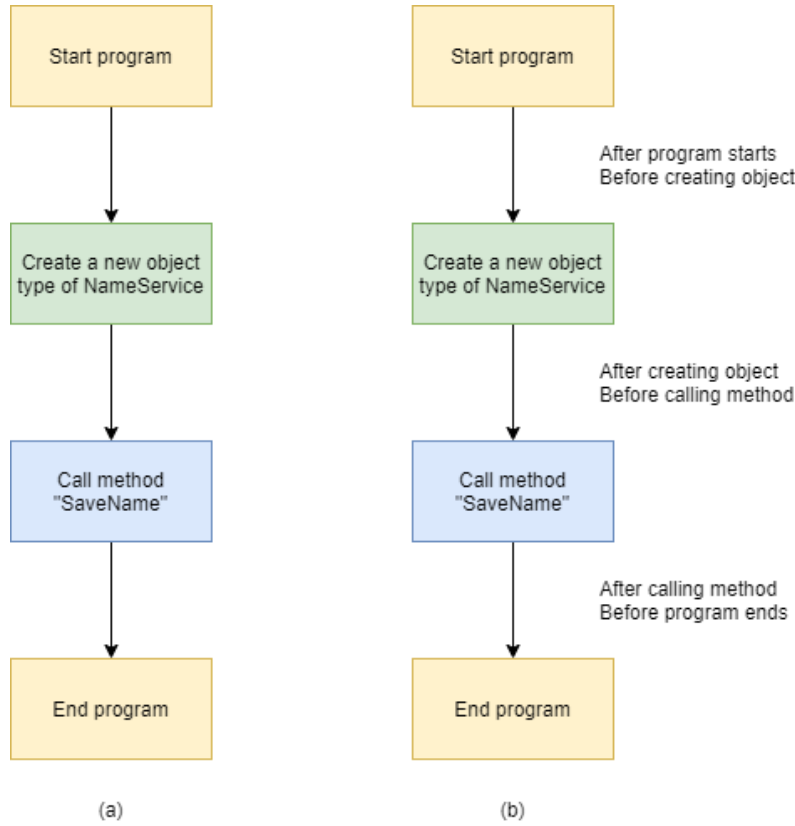
Temelde AOP ile kod bloklarına ilişkin olarak:

- Temiz kod blokları elde edilir (**clean code**)
- Kod bloklarını okumak kolaylaşır (**easier to read**)
- Hatalara daha az açıktır (**less prone to bugs**)
- Sürdürülmesi daha kolaydır (**easier to maintain**)
- Tekrarları azaltır (**reduce repetetion**)
- Test süreçlerini iyileştirir

AOP kullanmanın temel niyeti *logging*, *caching* ve *transacting* gibi kesişen kaygıların (*cross-cutting concerns*) veya fonksiyonel olmayan gereksinimlerin yönetimidir. Kes, kopyala ve yapıştırın ötesinde, *Proxy pattern* ya da *decorator pattern* gibi yapıları kullanarak kesişen endişeler güvenle yönetilebilir. Bu çerçevede, nesne yönelimli programlama dilleriyle geliştirilen ve yaygın olarak kullanılan *logging*, *caching*, ve *transacting* gibi basmakalıp kod bloklarının daha kolay bir şekilde AOP yaklaşımı ile yönetilmesi hedeflenmiştir. Bu çalışmalar sonucunda "[Aspect-Oriented Programming](https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf)" isimli bir araştırma makalesi ortaya çıkmıştır⁸.

⁸ <https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf>

AOP gerçekleştirmek üzere; *PostSharp* ya da *Castle DynamicProxy* gibi araçlar kullanılabilir. Bu kapsamda bizim tercihimiz *Castle DynamicProxy* olacaktır. Temelde bir metod çalıştırılırken araya girilerek bir dizi fonksiyonel olmayan gereksinimin karşılanması bu tekniklik ile gerçekleştirilmektedir. Şekil 9.1’de AOP’nin program akışına etkisini gösterilmiştir.



Şekil 9.1 AOP’nin program akışına etkisi

Bir metod çalışırken araya girme fikri kulağa kötü gelebilir ancak pratik uygulamalarda bu durum önemli bir amaca hizmet eder.



AOP Overview [07:52]

Aspect-oriented programming (AOP), bir başka ifadeyle cepheye-yönelik programlama kulağa oldukça karmaşık gelen teknik bir kavram, fakat algılandığı kadar karmaşık değil.

9.2. Proxy Tasarım Kalıbı

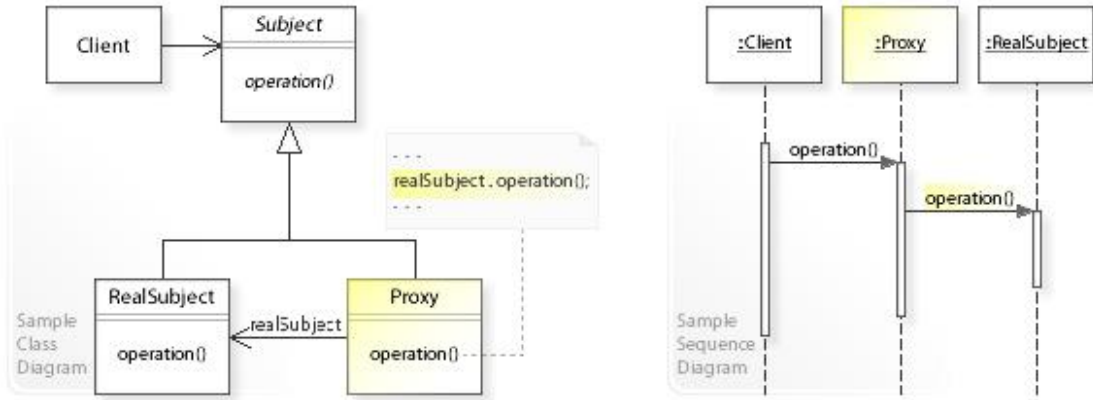
Proxy tasarım modeli, iyi bilinen *Design Patterns: Elements of Reusable Object-Oriented Software*⁹ (GoF) tasarım modellerinden biridir. Bu kalıp esnek, yeniden kullanılabilir nesneler tasarlamak üzere yinelenen tasarım sorunlarını çözmek üzere kullanılır. Bu kalıp ile üretilen nesnelerin kolayca implemente edilmesi, değiştirilmesi, test edilmesi ve yeniden kullanılabilmesi hedeflenir.



AOP Proxy Design Pattern [14:23]

Proxy tasarım modeli, iyi bilinen Design Patterns: Elements of Reusable Object-Oriented Software (GoF) tasarım modellerinden biridir.

DynamicProxy yapısından önce, *Proxy* kalıbının nasıl çalıştığını incelemek faydalı olabilir. Şekil 9.2'de Proxy deseninin UML diyagramına yer verilmiştir. Şekilden de anlaşılacağı üzere Proxy gerçek nesneyi temsil eden bir sınıftır. Dolayısıyla bir istemci Proxy üzerinden talepte bulunduğunda Proxy gerçek nesnenin bir örneğini alıp ya da oluşturup istemciye dönüş yapmaktadır.



Şekil 9.2 Proxy tasarım kalıbı¹⁰

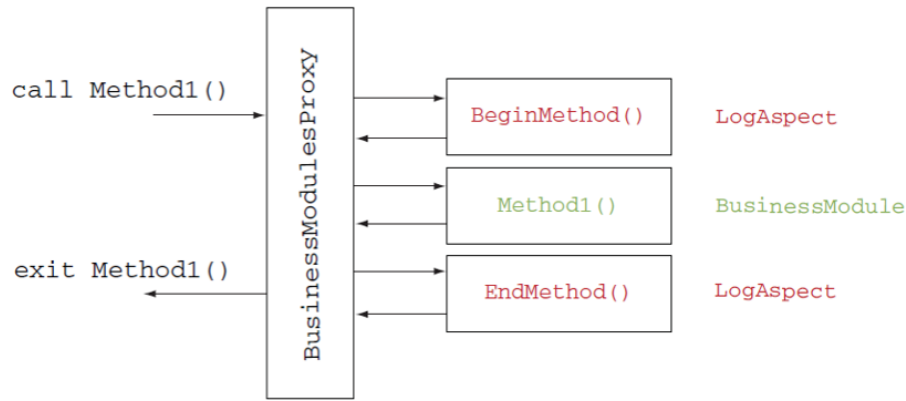
Proxy kalıbı ve ona son derece benzeyen *decorater* kalıbı, biraz farklı amaçları ve uygulamaları olan desenlerdir, ancak AOP perspektifinden pratik olarak aynıdır. Her iki yaklaşımda bir sınıfı değiştirmeden sınıfa fonksiyonellik eklenmesine olanak tanırırlar.

⁹ https://en.wikipedia.org/wiki/Design_Patterns

¹⁰ https://en.wikipedia.org/wiki/Proxy_pattern

Tipik olarak Proxy bir başka objenin yerine kullanılır. Genellikle gerçek nesneyi başlatmaktan sorumludur, gerçek nesneyle aynı *interface* yapısına sahiptir ve bu gerçek nesneye erişimi kontrol edebilir veya ek işlevsellik ve denetim sağlayabilir.

Şekil 9.3'de Proxy kalıbına yer verilmiştir. Tüm program bir nesne üzerinden tanımlanmış olan *interface* yapısı nedeniyle *Method1()* çağırmaı bilir. Bu nesne, kendi kodunu çalıştırma şansı olan bir proxy'dir. Sonra gerçek *Method1()*'e çağrı yapar. *Method1()* tamamlandıktan sonra, yürütmeyi orijinal programa döndürmeden önce kendi kodunu çalıştırmak için başka bir fırsatı vardır. Proxy modeli genellikle programın dışındaki nesneleri veya hizmetleri temsil etmek için kullanılır.



Şekil 9.3 Proxy kalıbı ¹¹

Bir *decorator* nesnesi, gerçek nesneyle aynı *interface* sahip olması açısından Proxy kalıbına benzerdir; ancak genellikle nesnenin **somutlaştırılmasından sorumlu değildir**. Bu nedenle gerçek nesnenin üzerine birden çok dekoratör katmanının olması mümkündür.

Örnek bir senaryo tanımı ile devam edelim:

¹¹ Practical Aspect-Oriented Programming AOP in NET, Matthe D. Groves, Manning, 2013

```
public interface IBusinessModule
{
    void Method1();
}

public class BusinessModule : IBusinessModule
{
    public override void Method1()
    {
        Console.WriteLine("Method1");
    }
}
```

Yukarıdaki kod bloğunda bir *interface* ve bir de *class* tanımı yapılmıştır. Bir alttaki kod bloğu ile de bu yapı için bir *Proxy* tanımlanmıştır.

```
public class BusinessModuleProxy : IBusinessModule
{
    BusinessModule _realObject;

    public BusinessModuleProxy()
    {
        _realObject = new BusinessModule();
    }

    public void Method1()
    {
        Console.WriteLine("BusinessModuleProxy before");
        _realObject.Method1();
        Console.WriteLine("BusinessModuleProxy after");
    }
}
```

BusinessModuleProxy aynı *interface* yapısını kullanmaktadır ve talep edildiğinde somut sınıfı oluşturmaktadır. Bununla birlikte *BusinessModuleProxy Method1()*'in öncesine ve sonrasına istenilen kod bloklarını yazma imkânını vermektedir. Üstelik bu noktada *BusinessModule.Method1()* üzerinden yapılacak değişikliklerin Proxy üzerinde bir etkisi söz konusu değildir.

9.3. DynamicProxy Generator

Castle DynamicProxy, çalışma zamanında hafif .NET proxy'leri (*lightweight .NET proxy*) oluşturmaya yönelik bir kitaplıktır. Proxy nesneleri, sınıfın kodunu değiştirmeden bir nesnenin

üyelerine yapılan çağrının durdurulmasına izin verir. Hem *class* hem de *interface* yapıları proxy'lenebilir, ancak yalnızca ***virtual* üyeler için kesme (*interception*) gerçekleştirebilir**. Bir örnek çalışmada bu kurala yer verilmiştir. Bu örneğin başlıca amaçları aşağıdaki gibi sıralanmıştır:

- *DynamicProxy* kurulumu gerçekleştirmek.
- *Proxy* tanımı yapmak.
- En yalın şekilde *Aspect* yazmak.



AOP DynamicProxy [15:02]

Castle DynamicProxy, çalışma zamanında hafif .NET proxy'leri (lightweight .NET proxy) oluşturmaya yönelik bir kütüphanedir.

Söz konusu amaçlar çerçevesinde; öncelikle *DynamicProxy* için *Castle.Core* paketinin yüklenmesi gerekir.

```
Install-Package Castle.Core
```

DynamicProxy tüm nesneyi çerçeveleyecek şekilde çalışır. Bu durumu ifade etmek üzere öncelikle bir *class* tanımı gerçekleştirilir.

```
public class MyClass
{
    public virtual void MyMethod()
    {
        Console.WriteLine("MyMethod body.");
        Console.ReadKey();
    }
}
```

Mevcut durumda bir sınıf tanımı gerçekleştirildi ve söz konusu sınıf içerisine metod türünde bir üye eklendi. Lütfen metodun *virtual* olarak tanımlandığını dikkat ediniz. *ProxyGenerator* ifadesi araya girilecek (*Interception*) olan metotların *virtual* olarak tanımlanması gerekmektedir.

Proxy oluşturmadan önce bir de basit bir *Aspect* tanımının yapılmasını sağlayalım. Böylelikle kodun nasıl dokunduğunu (*weaving*) gözlemleyebiliriz.

```
public class MyInterceptionAspect : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Before method.");
        invocation.Proceed();
        Console.WriteLine("After method.");
    }
}
```

MyInterceptionAspect ifadesi bir metodun çalışmasının kesilerek araya girme (*interception*) işleminin yapıldığı bloklarının en basit hali ile bir örneğini temsil etmektedir. *Aspect* sınıfı *IInterceptor* *interface* yapısını kalıtım ile devir almaktadır. İlgili *interface* yapısı *Intercept* metodunun implemente edilmesini zorunlu kılmaktadır.

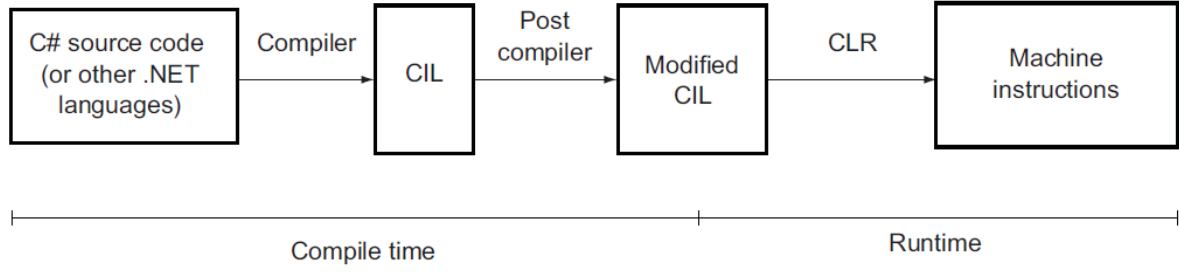
Son adımda yapılması gereken bir *Proxy* tanımı yapmak ve söz konusu kod bloklarının çalışmasını sağlamaktır.

```
internal class Program
{
    static void Main(string[] args)
    {
        var proxyGenerator = new ProxyGenerator();
        var svc = proxyGenerator
            .CreateClassProxy<MyClass>(new MyInterceptionAspect());

        svc.MyMethod();
    }
}
```

Main yordamı içerisinde bir *ProxyGenerator* nesnesi oluşturulmuştur. *ProxyGenerator* nesnesi çalışma zamanında (*runtime*) *MyClass* sınıfının *MyInterceptionAspect* ile sarmallanmasına, dokunması (*weaving*) olarak sağlamak üzere kullanılmaktadır. Temel bir *Proxy* inşası gerçekleştirilirken *Type*, *ProxyGenerationOptions* ve *Interceptors* ifadelerine ihtiyaç duyulur. En basit hali ile bir *IInterceptor* kullanılması gerekir. Mevcut örnekte *MyInterceptionAspect* *IInterceptor* *interface* yapısını kabul etmektedir.

Şekil 9.4'de derleme ve çalışma zamanına yer verilmiştir. Şekilden anlaşılacağı üzere *DynamicProxy runtime* anında devreye girmektedir.



Şekil 9.4 Derleme (compiler) zamanı ve çalışma (runtime) zamanı ¹²

9.4. Invocation

Invocation aspect kesilecek yani araya girilecek yapıyı ifade eder. Buna karşılık *invocation* ifadesinin anlaşılmasını sağlamak üzere bir örnek sunulmuştur.

Bu örneğin temel amaçları aşağıdaki gibi sıralanmıştır:

- *Invocation* parametresinin keşfedilmesi
- Metot parametrelerin tip ve değerlerinin alınması

Söz konusu amaçlara erişmek üzere öncelikle *Employee* isimli bir sınıf oluşturulur ve bu sınıfa ilgili *property* tanımları ile birlikte bir adet *Add()* metodu eklenir.



AOP Invocation [15:43]

Invocation aspect kesilecek yani araya girilecek yapıyı ifade eder. Buna karşılık *invocation* ifadesinin anlaşılmasını sağlamak üzere bir örnek sunulmuştur.

¹² Practical Aspect-Oriented Programming AOP in NET, Matthe D. Groves, Manning, 2013


```

public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public virtual void Add(int id, string firstName, string lastName)
    {
        Console.WriteLine("Method body start!");
        Console.WriteLine($"\\t{\\\"Id\\\",-15} : {id}\\");
        Console.WriteLine($"\\t{\\\"FirstName\\\",-15} : {firstName}\\");
        Console.WriteLine($"\\t{\\\"LastName\\\",-15} : {lastName}\\");
        Console.WriteLine("Method body end!");
    }
}

```

Bir önceki örnekte olduğu gibi [Castle.Core](#) paket kurulumu yapılır ve [InterceptorAspect](#) isimli bir [aspect](#) tanımı gerçekleştirilir. Bu [aspect](#) üzerinden [invocation](#) parametresinin sahip olduğu bazı özellikler [Aspect](#) içerisinde verilir. Bu özellikler Tablo 9.1'de sunulmuştur.

```

public void Intercept(IInvocation invocation)
{
    Console.WriteLine("Before invocation.");
    Console.WriteLine();
    Console.WriteLine($"\\\"MethodName\\\",-20} : {invocation.Method.Name}\\");
    Console.WriteLine($"\\\"TargetType\\\",-20} : {invocation.TargetType}\\");
    Console.WriteLine($"\\\"InvocationTarget\\\",-20} : {invocation.InvocationTarget}\\");

    Console.WriteLine($"\\\"Proxy\\\",-20} : {invocation.Proxy}\\");
    Console.WriteLine($"\\\"Arguments\\\",-20}\\");
    foreach (var p in invocation.Arguments)
    {
        Console.WriteLine($"\\t{\\\"Type\\\",-12} : {p.GetType(),-15} | {p}\\");
    }
    Console.WriteLine();
    invocation.Proceed();
    Console.WriteLine();

    Console.WriteLine("After invocation.");
}

```

Tablo 9.1 Invocation parametresi

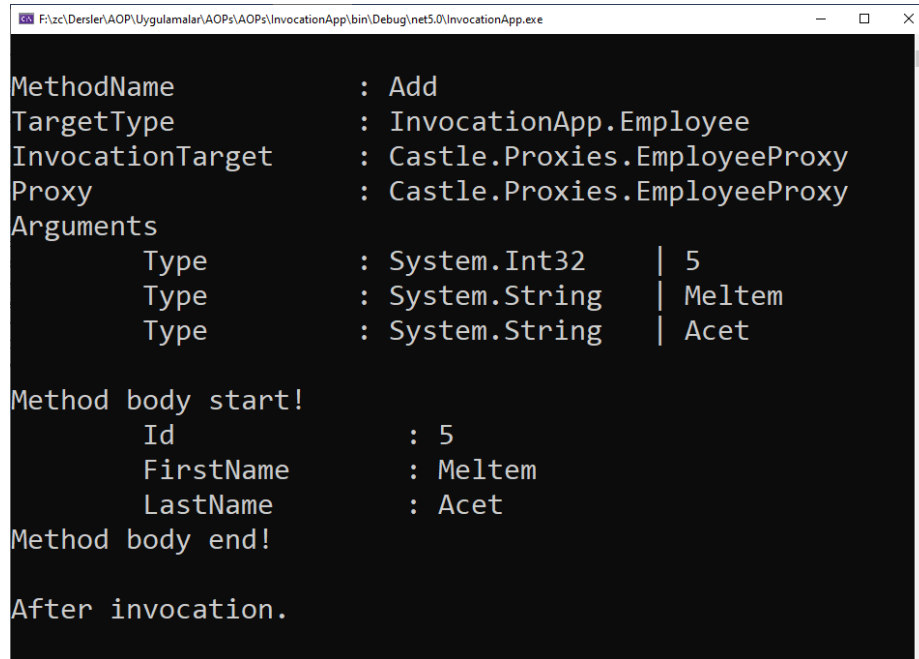
invocation.Method.Name	Metot adını verir.
invocation.TargetType	Metodun hedef tipini verir.
invocation.Arguments	Metodun içerdiği parametrelerin bir dizisini döner.

Sonraki adımda *Proxy* oluşturulur. *Proxy* ile sarmallanan sınıf için *Add* metodu çağrılır.

```
public class Program
{
    static void Main(string[] args)
    {
        var proxyGenerator = new ProxyGenerator();
        var svc = proxyGenerator
            .CreateClassProxy<Employee>(new InterceptionAspect());

        svc.Add(5, "Meltem", "Acet");
        Console.ReadKey();
    }
}
```

Yapılan bu işlem sonucunda aşağıdaki gibi bir ekran çıktısı elde edilir. Şekil 9.5’de *invocation* parametresinden elde edilen bilgiler ile oluşturulan ekran çıktısına yer verilmiştir.



```

F:\zc\dersler\AOP\Uygulamalar\AOPs\AOPs\InvocationApp\bin\Debug\net5.0\InvocationApp.exe

MethodName           : Add
TargetType           : InvocationApp.Employee
InvocationTarget      : Castle.Proxies.EmployeeProxy
Proxy                 : Castle.Proxies.EmployeeProxy
Arguments
    Type              : System.Int32      | 5
    Type              : System.String    | Meltem
    Type              : System.String    | Acet

Method body start!
    Id                : 5
    FirstName         : Meltem
    LastName          : Acet
Method body end!

After invocation.
```

Şekil 9.5 invocation parametresinden elde edilen bazı bilgiler

9.5. Defensive Programming

Bu bölümde *Employee* sınıfına *Update* isimli yeni bir metot daha ekliyoruz ve bir önceki bölümde eklediğimiz *Add* metodunun içeriğini de güncelliyoruz. Bu durumda sınıf içeriği aşağıdaki tanımlanmaktadır.

```
public virtual void Add(int id, string firstName, string lastName)
{
    if (id == null || firstName == null || lastName == null)
        throw new ArgumentNullException();
    Console.WriteLine("Added.");
}

public virtual void Update(int id, string firstName, string lastName)
{
    if (id == null || firstName == null || lastName == null)
        throw new ArgumentNullException();
    Console.WriteLine("Updated.");
}
```

Dikkate edilirse hem *Update* hem de *Add* metodu içerisinde parametre olarak alınan değerlerin *null* olma durumları kontrol edilmekte ve herhangi bir şekilde *null* ifadeyle karşılaşılması durumunda ilgili metotların *ArgumentNullException* ile dönüş yapması sağlanmaktadır. Bu durumda *Add* metoduna hem nesne ekleme görevi hem de parametrelerin kontrol edilmesi görev yüklenmektedir. Aynı durum *Update* metodu içinde söz konusudur. Benzer kontrollerin diğer modüller içerisinde de yapılacağı varsayımına bağlı olarak söz konusu kontrol ifadesinin uygulamaya boyunca yayılıp, saçılacağı öngörülebilir. Aynı zamanda *SOLID* yazılım geliştirme prensiplerinden *Single Responsibility* ilkesi bu durumda ihlal edilir.



AOP Defensive Programming [15:43]

Bu bölümde *Employee* sınıfına *Update* isimli yeni bir metot daha ekliyoruz ve bir önceki bölümde eklediğimiz *Add* metodunun içeriğini de güncelliyoruz. Bir örnek üzerinden devam ediyoruz.

Parametre değerlerinin *null* olup olmadığını kontrol edecek bir *Aspect* yazılarak; bu kod karmaşasının önüne geçilebilir, tekrarlar engellenebilir ve daha sürdürülebilir bir kod yapısı oluşturulabilir.

```
public class DefensiveProgrammingAspect : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        var parameters = invocation.Arguments;
        foreach (var p in parameters)
        {
            if (p.Equals(null))
                throw new ArgumentNullException();
        }
        Console.WriteLine("Null checked for {0}", invocation.Method);
    }
}
```

Söz konusu *Aspect* ifadesinin kod dokunurken, bir başka ifadeyle *Proxy* oluşturulurken dikkate alınmasını sağlamak amacıyla, ilgili *Aspect* yapısının da *Proxy* ifadesine bildirilmesi gerekir.

```
static void Main(string[] args)
{
    var proxyGenerator = new ProxyGenerator();
    var svc = proxyGenerator
        .CreateClassProxy<Employee>(
            new InterceptionAspect(),
            new DefensiveProgrammingAspect()
        );

    var emp = new Employee { Id = 1, FirstName = "Ömer" };
    svc.Add(5, "Meltem", "Acet");
    svc.Add(emp.Id, emp.FirstName, emp.LastName);

    Console.ReadKey();
}
```

Bu *aspect* ifadesi yazıldıktan sonra artık *Employee* sınıfı içerisinde metot üyeleri üzerinden parametrelerin değerini kontrol eden kod bloklarının kaldırılması sağlanabilir. Böylelikle daha temiz, sürdürülebilir ve okunabilir bir kod bloğu inşa edilir.

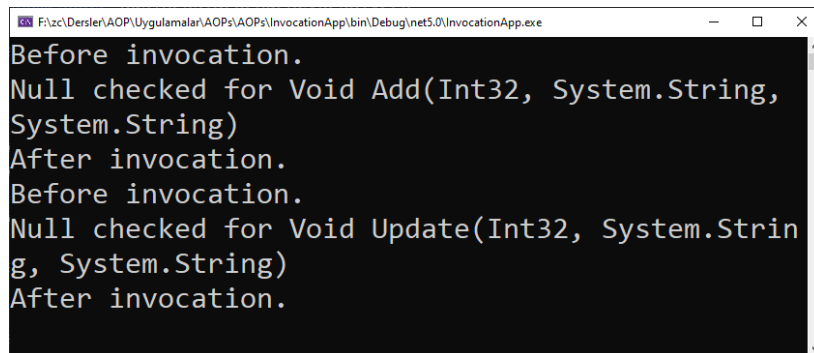
```

public virtual void Add(int id, string firstName, string lastName)
{
    Console.WriteLine("Added.");
}

public virtual void Update(int id, string firstName, string lastName)
{
    Console.WriteLine("Updated.");
}

```

Programın çalıştırılmasının sonrasında Şekil 9.6'de gösterilen ekran çıktısı elde edilir.



```

F:\zc\dersler\AOP\Uygulamalar\AOPs\InvocationApp\bin\Debug\net5.0\InvocationApp.exe
Before invocation.
Null checked for Void Add(Int32, System.String,
System.String)
After invocation.
Before invocation.
Null checked for Void Update(Int32, System.Strin
g, System.String)
After invocation.

```

Şekil 9.6 Defensive programming aspect

9.6. Aspect tanımlarının Attribute olarak yapılması

Bir *Aspect* ifadesinin *class*, *method* ya da *assembly* üzerinde çalışmasını sağlamak üzere *System.Attribute* özelliği kullanılmalıdır. İlaveten söz konusu yapının *aspect* olduğunu bildirmek üzere *Castle DynamicProxy* için *IInterceptor* interface yapısı da uygulanmalıdır. Bu tanımlamalara bağlı olarak en yalın hali ile bir *aspect* aşağıdaki gibi yazılabilir.

```

public class MyInterceptionAspect : Attribute, IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Before {0}", invocation.Method);
        invocation.Proceed();
        Console.WriteLine("After {0}", invocation.Method);
    }
}

```

Bu adımdan sonra *Aspect* kullanıldığı yerde doğrudan metot üzerinde ya da sınıf üzerinde deklare edilebilir.

```
public class MyClass
{
    [MyInterceptionAspect]
    public virtual void MyMethod()
    {
        Console.WriteLine("Method body run.");
    }
}
```

Söz konusu tanımın daha sistematik bir şekilde gerçekleştirilmesini sağlamak üzere bir *abstract class* tanımı yapılabilir. Bu amaçla *Interceptors* adı altında açılan bir klasöre *MethodInterceptionBaseAttribute* adı altında bir *abstract* sınıfı tanımı yapılmıştır.

```
namespace AspectAttribute.Interceptors
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Method |
        AttributeTargets.Assembly,
        AllowMultiple = true,
        Inherited = true)]
    public abstract class MethodInterceptionBaseAttribute :
        Attribute, IInterceptor
    {
        public int Priority { get; set; }
        public virtual void Intercept(IInvocation invocation)
        {
        }
    }
}
```

Tanımlanan bu soyut sınıf, söz konusu *Attribute* niteliğinin hedeflerini sınıf, metod ve assembly olarak bildirmektedir. Aynı zamanda hem çoklu kullanıma hem de kalıtımı izin vermektedir.

Abstract sınıflar içerisinde hem soyut hem de somut üyeler içerebilirler.



AOP MethodInterceptionBaseAttribute [07:25]

Bir Aspect ifadesinin class, method ya da assembly üzerinde çalışmasını sağlamak üzere System.Attribute özelliği kullanılmalıdır.

Bu durumda artık `MyInterceptionAspect` sınıfı `MethodInterceptionBaseAttribute` sınıfından kalıtım ile devir alınabilir. Bu durumda `Attribute` adımı için nihai `aspect` tanımı aşağıda gösterildiği gibi gerçekleştirilebilir. `Intercept` metodunun `override` edildiğine dikkat ediniz. Daha açık bir ifade ile `abstract` sınıfta `virtual` olarak tanımlanmış olan bu metot ilgili `aspect` içinde geçersiz kılınmaktadır.

```
public class MyInterceptionAspect : MethodInterceptionBaseAttribute
{
    public override void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Before {0}", invocation.Method);
        invocation.Proceed();
        Console.WriteLine("After {0}", invocation.Method);
    }
}
```

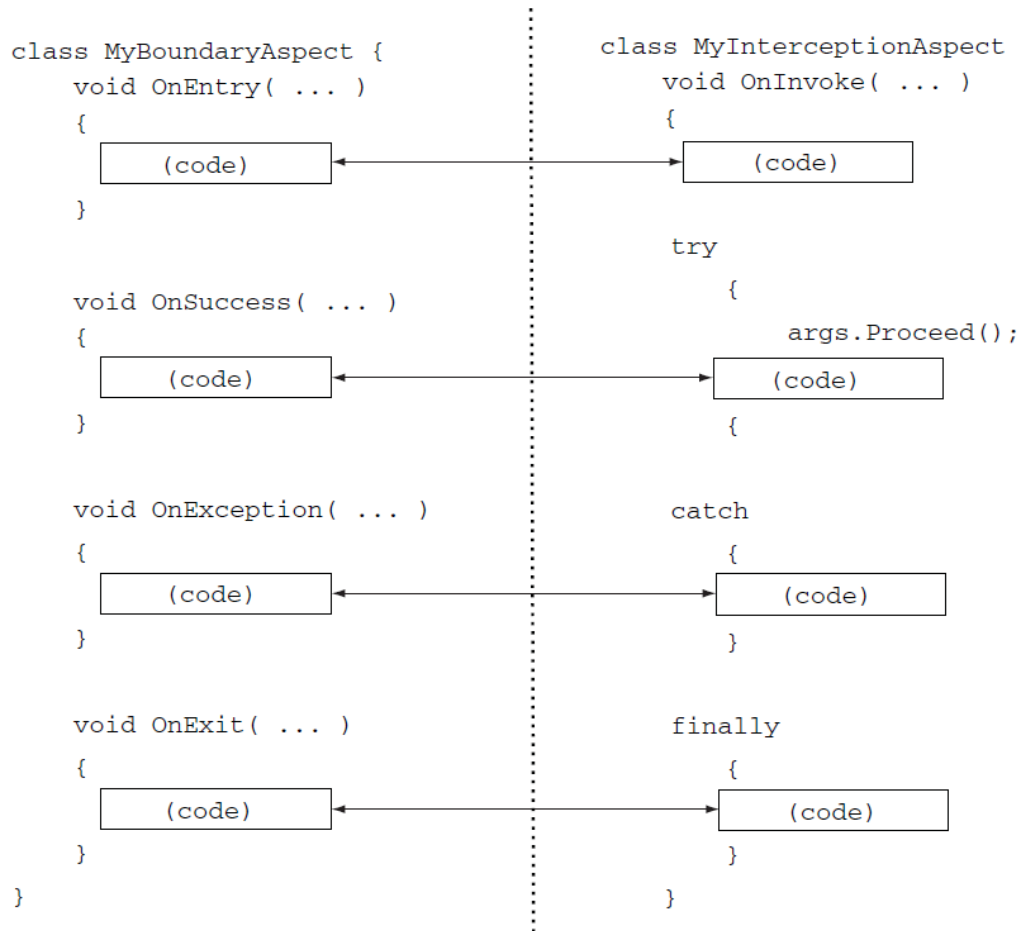
9.7. MethodInterception

Bir metot çalışırken `IInterceptor` aracılığı ile metodun çalışmasının kesilip araya girilebileceğini AOP ile tekniği ile öğrendik. Bu noktada, söz konusu metodun çalışmasına ilişkin olarak metodun öncesinde (`OnBefore`), metodun sonrasında (`OnAfter`), hata durumunda (`OnException`), metodun çalışması tamamlandığında (`OnSuccess`) neler yapılacağı ayrı ayrı metotlar içerisinde belirtilebilir. Şekil 9.7'de `BoundayAspect` ve `InterceptionAspect` için bu senaryo dikkate alınmıştır.



AOP MethodInterception [13:17]

Bir metot çalışırken `IInterceptor` aracılığı ile metodun çalışmasının kesilip araya girilebileceğini AOP ile tekniği ile öğrendik. Bu noktada, söz konusu metodun çalışmasına ilişkin olarak metodun öncesinde (`OnBefore`), metodun sonrasında (`OnAfter`), hata durumunda (`OnException`), metodun çalışması tamamlandığında (`OnSuccess`) neler yapılacağı ayrı ayrı metotlar içerisinde belirtilebilir.

Şekil 9.7 MethodInterception ¹³

MethodInterception içerisinde yukarıda ifade edilen olaylar bir kez tanımlanır. Daha sonra her bir *Aspect* tanımı; *MethodInterception* sınıfının kalıtım ile devir alınıp ilgili metotların geçersiz kılınması sağlanarak gerçekleştirilebilir.

Buna göre *MethodInterception* ifadesi aşağıdaki gibi tanımlanabilir.

```

public class MethodInterception : MethodInterceptionBaseAttribute
{
    public override void Intercept(IInvocation invocation) { }
    protected virtual void OnBefore(IInvocation invocation) { }
    protected virtual void OnAfter(IInvocation invocation) { }
    protected virtual void OnException(IInvocation invocation, Exception e) { }
    protected virtual void OnSuccess(IInvocation invocation) { }
}

```

¹³ Practical Aspect-Oriented Programming AOP in NET, Matthe D. Groves, Manning, 2013

Söz konusu yapıda *OnBefore*, *OnAfter*, *OnException*, *OnSuccess* ifadeleri *Custom method* olarak tanımlanmış olup esas işleyişe yön veren metot *Intercept* metodudur. *Intercept* metodunun içeriği ise aşağıdaki gibi tanımlanır:

```
public override void Intercept(IInvocation invocation) {
    bool successFlag = true;
    OnBefore(invocation);
    try
    {
        invocation.Proceed();
    }
    catch (Exception ex)
    {
        successFlag = false;
        OnException(invocation, ex);
        throw;
    }
    finally
    {
        if (successFlag)
            OnSuccess(invocation);
    }
    OnAfter(invocation);
}
```

Bu durumda daha önce yazdığımız *DefensiveProgrammingAspect* tanımını *MethodInterception* ile yeniden düzenleyebiliriz. Görüldüğü üzere parametre kontrolü metot çalışmadan önce gerçekleştiği için sadece ilgili bölümde tanımlamalar gerçekleştirilir.

Eski

```
public class DefensiveProgrammingAspect : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        var parameters = invocation.Arguments;
        foreach (var p in parameters)
        {
            if (p.Equals(null))
                throw new ArgumentNullException();
        }
        Console.WriteLine("Null checked for {0}", invocation.Method);
    }
}
```

Yeni

```
public class DefensiveProgrammingAspect : MethodInterception
{
    protected override void OnBefore(IInvocation invocation)
    {
        var parameters = invocation.Arguments;
        foreach (var p in parameters)
        {
            if (p.Equals(null))
                throw new ArgumentNullException();
        }
        Debug.WriteLine("Null checked for {0}", invocation.Method);
    }
}
```

9.8. IInterceptorSelector

IInterceptorSelector interface yapısı bir uzantı/genişleme (extension) sağlar. Bu uzantı ifadesi *Proxy* yapılarının belirli ya da spesifik olarak tanımlanmış olan *interceptors* ifadelerini dikkate almasına olanak tanır. Böylelikle bir sınıfın ya da bir metodun üzerinde tanımlı olan *Attribute* ifadelerine bağlı olarak kesme (*interception*) işlevi gerçekleştirilebilir.

```
public class AspectInterceptorSelector : IInterceptorSelector
{
    public IInterceptor[] SelectInterceptors(Type type,
        MethodInfo method,
        IInterceptor[] interceptors)
    {
        var classAttributes =
            type.GetCustomAttributes<MethodInterceptionBaseAttribute>(true).ToList();
        var methodAttributes =
            type.GetMethod(method.Name)?.GetCustomAttributes<MethodInterceptionBaseAttribute>(true);
        if (methodAttributes != null)
        {
            classAttributes.AddRange(methodAttributes);
        }
        return classAttributes.OrderBy(x => x.Priority).ToArray();
    }
}
```



AOP AspectInterceptorSelector [14:08]

`IInterceptorSelector` interface yapısı bir uzantı/genişleme (extension) sağlar. Bu uzantı ifadesi `Proxy` yapılarının belirli ya da spesifik olarak tanımlanmış olan `interceptors` ifadelerini dikkate almasına olanak tanır.

9.9. Using Aspect with Autofac Module

`DynamicProxy` ile bir `Aspect` ifadesinin kullanılabilmesi için `Proxy` tanımının yapılması gereklidir. Daha önceki örneklerde aşağıdaki gibi ifadeler tanımlandık:

```
var proxy = new ProxyGenerator();
var aspect = proxy.CreateClassProxy<MyClass>(
    new MyInterceptorAspect());
aspect.MyMethod();
```

`Main` yordamı içerisine yazılmış olan bu `Proxy` tanımına bağlı olarak `MyClass.MyMethod()` ifadesi için tanımlı olan `MyInterceptorAspect()` dokuması gerçekleştirilir.

Bağımlılıkların yönetimi ve otomatik olarak çözümlenmesini sağlamak üzere `Autofac` kullanılabilir. Bu durumda bir `ContainerBuilder` ifadesi yardımıyla temel `IoC` işlevleri yani `Register`, `Resolve` ve `Dispose` yerine getirilebilir.

`MyClass.MyMethod` ifadesi için söz konusu bağımlılığın çözümlenmesini sağlamak üzere ilgili ifadenin `ContainerBuilder` aracılığı ile öncelikle `Register` edilmesi sağlanır.

```
builder.RegisterType<MyClass>()
    .As<IMyClass>()
    .EnableInterfaceInterceptors(proxyOptions)
    .InstancePerDependency();
```

Bu arada `MyClass` ve `IMyClass` tanımları aşağıdaki gibidir:

```
public interface IMyClass
{
    void MyMethod();
}
public class MyClass : IMyClass
{
    [MyInterceptorAspect]
    public virtual void MyMethod()
    {
        Console.WriteLine("MyMethod body.");
    }
}
```

`EnableInterfaceInterceptors(proxyOptions)` ifadesinin tanımlanabilmesi için `Autofac.Extras.DynamicProxy` paketinin de kurulması gerekir. Ayrıca bu yöntem parametre olarak geçen `proxyOptions` ifadesi aşağıdaki gibi tanımlanmalıdır.

```
var proxyOptions = new ProxyGenerationOptions()
{
    Selector = new AspectInterceptorSelector()
};
```

Bu durumda aşağıdaki adımlar sırasıyla takip edilir.

- 1) `Main` yordamı içerisinde ya da harici olarak tanımlanan bir `ContainerBuilder` ifadesi arzu edilen şekilde yapılandırılır.
- 2) `ContainerBuilder` ifadesi üzerinden gerekli kayıtlanma işlevi yani `Register` olma işlevi her nesne için tek tek ya da `assembly` üzerinden toplu olarak gerçekleştirilir.
- 3) `Aspect` ifadelerinin çalıştırılabilmesi için `Register` aşamasında `ProxyGenerationOptions` için `Selector` olarak `AspectInterceptorSelector` tanımının yapılması gerekir. Aksi durumda `Aspect` ifadeleri **çalışmaz**.
- 4) `ContainerBuilder` ifadesi `Build` edilerek bir `container` elde edilir.
- 5) İlgili `container` üzerinden çözme yani `Resolve` işlemi gerçekleştirilerek hem söz konusu `aspect` ifadelerinin çalıştırılması hem de aynı zamanda bağımlılıkların çözülmesi sağlanır.

```
{
    var proxy = new ProxyGenerator();
    var aspect = proxy.CreateClassProxy<MyClass>(new MyInterceptorAspect());
    aspect.MyMethod();

    Console.WriteLine(new String('-', 50));

    var builder = new ContainerBuilder();
    builder.RegisterModule(new DefaultModule());

    var container = builder.Build();
    var willBeIntercepted = container.Resolve<IMyClass>();
    willBeIntercepted.MyMethod();
}
```

**AOP Autofac Module [14:08]**

DynamicProxy ile bir Aspect ifadesinin kullanılabilmesi için Proxy tanımının yapılması gereklidir.