

```

//Ado.net (C# Dilinde SQL yapılarını kullanmamızı sağlayan çerçeve)

Console.WriteLine("***** C# Veri Tabanlı Ürün Katagori Bilgi Sistemi **");
Console.WriteLine();
Console.WriteLine();

string tableNumber;

Console.WriteLine("-----");
Console.WriteLine("1-Katagoriler");
Console.WriteLine("2-Ürünler");
Console.WriteLine("3-Siparişler");
Console.WriteLine("4-Çıkış Yap");
Console.WriteLine("Lütfen getirmek istediğiniz tablo numarasını giriniz:");

tableNumber = Console.ReadLine();
Console.WriteLine("-----");

SqlConnection connection = new SqlConnection("Data Source = YIĞIT\\SQLEX");
connection.Open();
SqlCommand command = new SqlCommand("Select * From TblCategory",connection);
SqlDataAdapter adapter = new SqlDataAdapter(command);
DataTable dataTable = new DataTable();
adapter.Fill(dataTable);
connection.Close();

foreach (DataRow row in dataTable.Rows)
{
    foreach (var item in row.ItemArray)
    {
        Console.Write(item);
    }
    Console.WriteLine();
}

```

Code 2:

```

//Crud --> Create-Read-Update-Delete

Console.WriteLine("***** Menü Sipariş İşlem Paneli *****");
Console.WriteLine();

```

```

Console.WriteLine("-----");

#region Katagori Ekleme İşlemleri
Console.WriteLine("Eklemek istediğiniz Kategori Adı : ");
string CategoryName = Console.ReadLine();

SqlConnection connection = new SqlConnection("Data Source = YIĞIT\\SQLEXPRESS; initi
connection.Open();
SqlCommand command = new SqlCommand("insert into TblCategory(CategoryName) values (@
command.Parameters.AddWithValue("@p1", CategoryName);
command.ExecuteNonQuery();
connection.Close();
Console.Write("Kategori Başarılı bir şekilde eklendi");
#endregion

#region Ürün Ekleme İşlemi

string productName;
decimal productPrice;

Console.Write("Ürün adı: ");
productName = Console.ReadLine();
Console.Write("Ürün Fiyatı: ");
productPrice = decimal.Parse(Console.ReadLine());

SqlConnection connection = new SqlConnection("Data Source = YIĞIT\\SQLEXPRESS; initi
connection.Open();
SqlCommand command = new SqlCommand("insert into TblProduct (ProductName, ProductPri
command.Parameters.AddWithValue("@productName", productName);
command.Parameters.AddWithValue("@productPrice", productPrice);
command.Parameters.AddWithValue("@productStatus", true);
command.ExecuteNonQuery();
connection.Close();
Console.WriteLine("Ürün eklemesi başarılı.");
#endregion

#region Ürün Listeleme İşlemi

SqlConnection connection = new SqlConnection("Data Source = YIĞIT\\SQLEXPRESS; initi
connection.Open();
SqlCommand command = new SqlCommand("Select * From TblProduct", connection);
SqlDataAdapter adapter = new SqlDataAdapter(command);
DataTable dataTable = new DataTable();
adapter.Fill(dataTable);

```

```

foreach (DataRow row in dataTable.Rows)
{
    foreach (var item in row.ItemArray)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine();
}

#endregion

#region Ürün Silme İşlemi

int ProductId;

Console.WriteLine("Silinecek ürün Id: ");
ProductId = int.Parse(Console.ReadLine());

SqlConnection connection = new SqlConnection("Data Source = YIĞIT\\SQLEXPRESS; initi
connection.Open();
SqlCommand command = new SqlCommand("Delete from TblProduct Where ProductId=@Product
command.Parameters.AddWithValue("ProductId", ProductId);
command.ExecuteNonQuery();
connection.Close();

Console.WriteLine("İşlem tamamlandı...");

#endregion

```

OOP Modülü Entity Layer

- EntityLayer
- DataAccessLayer
- BusinessLayer
- Presentation/UI Layer

Entity Layer için ilk olarak blank proje açtık daha sonra içine add project diyerek Class Library(.Net) kütüphanesi ekledik. Adını da CsharpEgitim301.EntityLayer yaptık. Daha sonra sınıflar için Concrete adında bir klasör oluşturduk.

Bu klasöre sağ tık yapıp yeni bir öge ekledik ve buna Category ismini verdik SQL deki tablomuzu aslında C# dilinde yapıyoruz.

Access Modifiers:

Public --> Her yerden erişim

Private --> Sadece bulunduğu class

Internale --> Sadece bulunduğu katman

Protected

Oluşturduğumuz kısımdaki Class'ı public yapıyoruz.

Bir değişken direkt sınıfın içine tanımlanıyorsa --> Field

Eğer o değişken sonuna get ve set alıyorsa --> Property

Eğer bir değer metod içinde tanımlanıyorsa --> Variable

Classları uygun bir şekilde yapıyoruz.

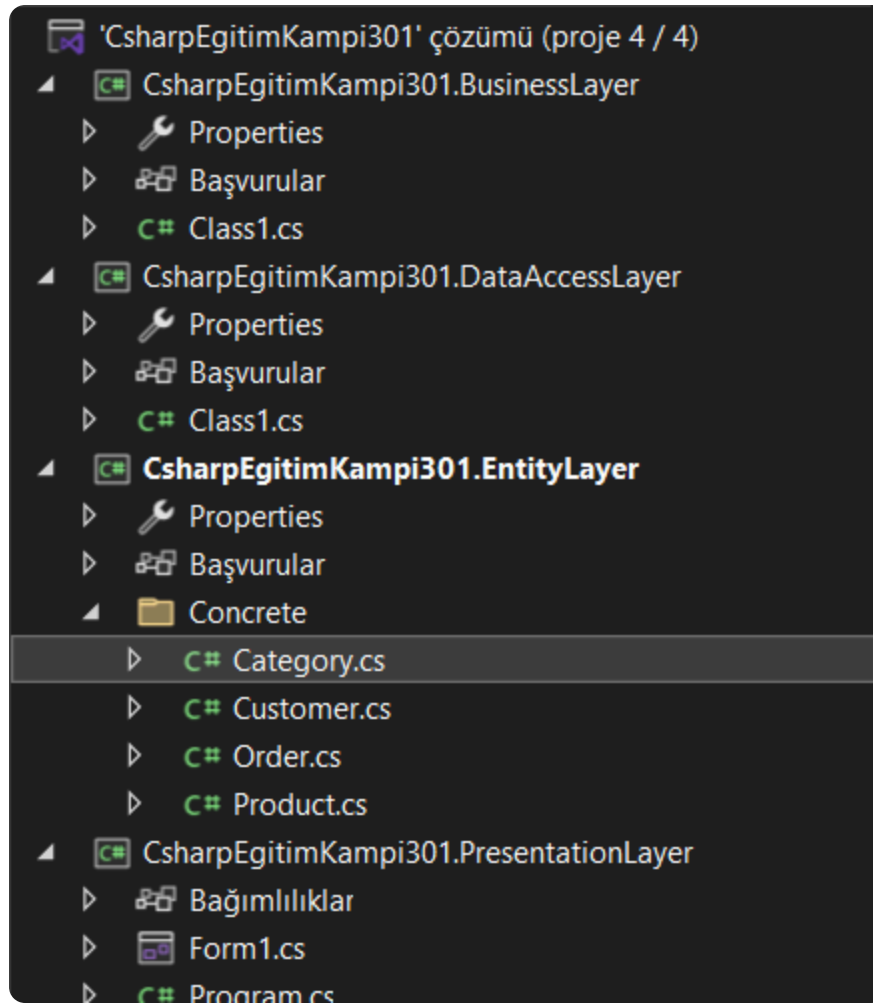
```
public class Product
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public int ProductStock { get; set; }
    public decimal ProductPrice { get; set; }
    public string ProductDescription { get; set; }
}
```

```
public class Customer
{
    public int CustomerId { get; set; }
    public string CustomerName { get; set; }
    public string CustomerSurname { get; set; }
}
```

```
public class Category
{
    public int CategoryId { get; set; }

    public string CategoryName { get; set; }
    public bool CategoryStatus { get; set; }
}
```

Daha sonra gerekli katmanların hepsini oluşturuyoruz. En son presentation Layerı Windows uygulaması olarak oluşturuyoruz.



Böyle bir görüntü oluyor.

Her ürünün bir kategorisi olmalı onun için ürün tablosu ile kategori tablosunu ilişkilendirmemiz gerekiyor bunu yapıyoruz: (Product.cs)

```
public int CategoryId { get; set; }  
public virtual Category Category { get; set; }
```

Category.cs:

```
public List<Product> Products { get; set; }
```

Şimdi Order ile Product'ı bağlayacağız (Order.cs):

```
public int ProductId { get; set; }
public Product Product { get; set; }
```

Product.cs:

```
public List<Order> Orders { get; set; }
```

Şimdi Order kısmına yeni özellikler ekleyip Customer ile bağlayacağız.

Order.cs:

```
public int Quantity { get; set; }
public decimal UnitPrice { get; set; }
public decimal TotalPrice { get; set; }
public int CustomerId { get; set; }
public Customer Customer { get; set; }
```

Customer.cs

```
public List<Order> Orders { get; set; }
```

Daha sonra entity layerı komple oluşturmak için projeye sağ tıklayıp paketleri yönete basıp 'EntityFramework' ü yüklüyoruz ve bunu DataAccess katmanına referans atıyoruz bunun için DataAccess e sağ tıklayıp Ekle kısmından Referansı seçip daha sonra EntityLayer'ı ekliyoruz.

DataAccess te yeni bir klasör oluşturduk Context diye daha sonra bu klasöre yeni bir sınıf ekledik KampContext.cs diye

```
public class KampContext:DbContext
{
}
```

bunu yazdıktan sonra Ctrl + '.' tuşlarına bastık ve EntityFramework localden yükledik. Veri tabanına yansiyacak olan tüm sınıfları bunun içerisine koyucaz.

```
public DbSet<Category> Categories { get; set; }
public DbSet<Product> Products { get; set; }
```

```
public DbSet<Order> Orders { get; set; }
public DbSet<Customer> Customers { get; set; }
public DbSet<Admin> Admins { get; set; }
```

Daha sonra tüm dosyaların bağlantılarını yapıyoruz ve Presentation projesine EntityFramework ü kuruyoruz.

Presentation projesinde app.config dosyasına

```
<connectionStrings>
  <add name = "KampContext" connectionString="Data Source= YIĞIT\\SQLEXPRESS;
</connectionStrings>
</configuration>
```

bu bağlantı kodunu yazdık

DataAccessLayer projesine "EntityFramework","Abstract" ve "Repositories" klasörlerini ekledik.

Migration İşlemleri ve Abstract Interfaceler

Görünümünden > Diğer pencerelerden > Paket yönetme konsolunu açıyoruz.

Varsayılan projeyi DataAccessLayerı seçiyoruz.

enable-migrations bu kodu yazıyoruz

```
AutomaticMigrationsEnabled = true;
```

açılan ekranda bu kodu true yapıyoruz.

update-database bu kodu yazıyoruz

SORU:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) { modelBuilder.
```

Bu kod nedir nasıl çalışır?

Bu kod, Entity Framework (EF) Code First yaklaşımında veritabanı modelinin nasıl oluşturulacağını özelleştiren `OnModelCreating` metodunu override eder. İşlevleri ve çalışma mantığı:

☞

1.

```
modelBuilder.Configurations.Add(new Urun());
modelBuilder.Configurations.Add(new Sepet());
```

- **Amacı:** `Urun` ve `Sepet` entity sınıfları için özel Fluent API konfigürasyonlarını uygular.
- **Nasıl Çalışır?**
 - `Urun` ve `Sepet` sınıfları, `EntityTypeConfiguration<T>` 'dan türeyen ayrı konfigürasyon sınıflarıdır (örneğin `UrunMap` , `SepetMap`).
 - Bu sınıflar, ilgili tabloların veritabanı eşlemesini (primary key, ilişkiler, kolon tipleri vb.) **Fluent API** ile tanımlar.
 - Örnek konfigürasyon:

```
public class UrunMap : EntityTypeConfiguration<Urun>
{
    public UrunMap()
    {
        ToTable("Urunler");
        HasKey(u => u.Id);
        Property(u => u.Ad).HasMaxLength(100).IsRequired();
        // Diğer özelleştirmeler...
    }
}
```

- **Neden Kullanılır?**
DataAnnotations (attribute'lar) yerine daha esnek ve karmaşık konfigürasyonlar için Fluent API tercih edilir.


```
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
```

- **Amacı:** Tüm "bire-çok" (one-to-many) ilişkilerde **cascade delete** özelliğini devre dışı bırakır.
- **Nasıl Çalışır?**
 - Varsayılan olarak EF, "bire-çok" ilişkilerde üst kayıt silinince alt kayıtların otomatik silinmesini sağlar (cascade delete).
 - Bu kural kaldırılınca:
 - Üst kayıt (örneğin **Kategori**) silinirse, ona bağlı alt kayıtlar (örneğin **Urun**) **silinmez**.
 - Veritabanı **referans bütünlüğü hatası** (foreign key constraint) verir.
- **Neden Kullanılır?**
 - Yanlışlıkla veri silinmesini önlemek.
 - İlişkili verilerin manuel kontrolünü sağlamak.



Kodun Tam İşleyişi

1. **DbContext Başlatıldığında:**
 - EF, **OnModelCreating** metodunu otomatik çağırır.
 - Önce **Urun** ve **Sepet** için özel konfigürasyonlar yüklenir.
 - Sonra **OneToManyCascadeDeleteConvention** kuralı kaldırılır.
2. **Model Oluşturma:**
 - Konfigürasyonlar ve kalan kurallar kullanılarak veritabanı şeması oluşturulur.
 - Tüm "bire-çok" ilişkilerde cascade delete **kapalı** olur.



Örnek Senaryo

- **Cascade Delete Kapalı İken:**
 - `Kategori` tablosunda bir kayıt silmeye çalışırsanız:

```
DELETE FROM Kategoriler WHERE Id = 1 -- Hata!  
-- Hatayı tetikler: "The DELETE statement conflicted with the REFERENCE constraint..
```

- • çözüm: Önce ilgili `Urun` kayıtlarını silmek veya foreign key'i `NULL` yapmak.



Önemli Not

- **Code First Migrations:** Bu değişiklikler migration oluşturulurken dikkate alınır (cascade delete kısıtlaması veritabanından kaldırılır).
- **Alternatif:** Tek bir ilişki için cascade delete kapatmak isterseniz:

```
modelBuilder.Entity<Kategori>()  
    .HasMany(k => k.Urunler)  
    .WithRequired(u => u.Kategori)  
    .WillCascadeOnDelete(false); // Sadece bu ilişkide kapalı
```

Bu yapı, veri bütünlüğü ve güvenliği için kritik özelleştirmeler sağlar.

SORU2:

```
public virtual ICollection<Marka> Markas { get; set; }  
public virtual Kategori Kategori { get; set; }
```

Bu kodların çalışma mantığı nedir

Bu kodlar, Entity Framework'te **navigation property** olarak adlandırılan ve ilişkileri yönetmek için kullanılan özelliklerdir. İşlevlerini ve çalışma mantığını ayrıntılı açıklayayım:

1. `public virtual Kategori Kategori { get; set; }`

- **İlişki Türü:** "Çoktan-bire" (Many-to-One) ilişkinin "bire" tarafını temsil eder.
- **Anlamı:**
Bulunduğu sınıfın (örneğin `Urun` sınıfı) **tek bir** nesnesine **bağlı olduğunu** gösterir.
- **Veritabanı Karşılığı:**
`Urun` tablosunda `KategoriId` adında bir **foreign key** sütunu oluşturulur.
- **Örnek Kullanım:**

```
public class Urun
{
    public int Id { get; set; }
    public string Ad { get; set; }

    // Foreign key (genellikle otomatik tanımlanır)
    public int KategoriId { get; set; }

    // Navigation property
    public virtual Kategori Kategori { get; set; }
}
```

Nasıl Çalışır?

Bir ürün nesnesi üzerinden ilişkili kategoriye erişim sağlar:

```
var urun = context.Urunler.Find(1);
Console.WriteLine(urun.Kategori.Ad); // İlişkili kategori adı
```

2. `public virtual ICollection<Marka> Markas { get; set; }`

- **İlişki Türü:** "Bire-çok" (One-to-Many) ilişkinin "çok" tarafını temsil eder.
- **Anlamı:**
Bulunduğu sınıfın (örneğin `Kategori` sınıfı) **birden fazla** nesne **içerebileceğini** gösterir.

- **Veritabanı Karşılığı:**
`Marka` tablosunda `KategoriId` adında bir **foreign key** sütunu oluşturulur.
- **Örnek Kullanım:**

```
public class Kategori
{
    public int Id { get; set; }
    public string Ad { get; set; }

    // Navigation property (çok tarafı)
    public virtual ICollection<Marka> Markas { get; set; }
}
```

Nasıl Çalışır?

Bir kategori nesnesi üzerinden bağlı markalara erişim sağlar:

```
var kategori = context.Kategoriler.Find(1);
foreach (var marka in kategori.Markas)
{
    Console.WriteLine(marka.Ad); // İlişkili markalar
}
```

İlişkinin Tam Hali

Bu iki property birlikte **iki yönlü bir ilişki (bidirectional navigation)** oluşturur:

```
// Kategori Sınıfı
public class Kategori
{
    public int Id { get; set; }
    public string Ad { get; set; }
    public virtual ICollection<Marka> Markas { get; set; } // 1-N ilişki
}

// Marka Sınıfı
public class Marka
{
    public int Id { get; set; }
    public string Ad { get; set; }
```

```
public int KategoriId { get; set; } // Foreign key
public virtual Kategori Kategori { get; set; } // N-1 ilişki
}
```

Önemli Özellikler

1. **virtual Anahtar Kelimesi:**

- **Lazy Loading** için gereklidir. İlk erişimde ilişkili veriler otomatik yüklenir:

```
var marka = context.Markalar.Find(1);
var kategoriAdi = marka.Kategori.Ad; // Kategori verisi bu satırda DB'den yüklenir
```

Foreign Key Oluşumu:

- Entity Framework otomatik olarak **KategoriId** gibi foreign key alanları oluşturur.
- İsterseniz manuel tanımlayabilirsiniz:

```
public int KategorId { get; set; }
```

İlişki Yapılandırması:

- Fluent API ile özelleştirme örneği:

```
modelBuilder.Entity<Marka>()
    .HasRequired(m => m.Kategori)
    .WithMany(k => k.Markas)
    .HasForeignKey(m => m.KategoriId);
```

Kullanım Senaryosu

```
// Yeni kategori oluştur
var elektronik = new Kategori { Ad = "Elektronik" };

// Marka ekle
elektronik.Markas = new List<Marka>
{
    new Marka { Ad = "Apple" },
}
```

```
new Marka { Ad = "Samsung" }  
};  
  
// Veritabanına kaydet  
context.Kategoriler.Add(elektronik);  
context.SaveChanges();
```

Çıktı:

- **Kategori** tablosuna 1 kayıt
- **Marka** tablosuna 2 kayıt (otomatik olarak **KategoriId** = elektronik.Id ile)

SORU3:

API nedir FluentAPI ne anlama geliyor

API (Application Programming Interface) Nedir?

API, uygulamaların birbiriyle iletişim kurmasını sağlayan bir **arayüz ve kurallar bütünüdür**.

Temel özellikleri:

- **Ne İşe Yarar?**

Farklı yazılım bileşenlerinin (web servisleri, kütüphaneler, işletim sistemleri) birbiriyle konuşmasını standartlaştırır.

Örnek: Bir hava durumu uygulaması, meteoroloji servisinin API'sini kullanarak veri çeker.

- **Nasıl Çalışır?**

- **İstek (Request)**: "Şu koordinatların hava durumunu ver" diye sorar.
- **Yanıt (Response)**: API, JSON/XML formatında veri döndürür.

GET <https://api.havadurumu.com/istanbul>

Türleri:

- **Web API'leri**: REST, SOAP (HTTP üzerinden).
- **Sistem API'leri**: Windows API (dosya işlemleri).
- **Kütüphane API'leri**: .NET Framework'teki **List<T>** sınıfı.

Fluent API Nedir? (Entity Framework Bağlamında)

Fluent API, kod yazımını **doğal dile benzer şekilde** zincirleme metodlarla ifade eden bir tasarım modelidir. Entity Framework'te veritabanı modeli konfigürasyonu için kullanılır.

🔑 Temel Özellikleri:

1. Zincirleme Metodlar (Method Chaining):

Her metod bir sonraki metodla bağlanır, tek satırda birden fazla işlem yapılır.

Örnek:

```
modelBuilder.Entity<Urun>()  
    .Property(u => u.Ad)           // Ürünün "Ad" özelliği  
    .IsRequired()                 // Zorunlu alan  
    .HasMaxLength(100);          // Maksimum 100 karakter
```

1. Neden "Fluent"?

- Okunabilirliği artırır ("akıcı" bir anlatım sağlar).
- Data Annotations (attribute'lar) ile yapılamayan karmaşık konfigürasyonları destekler.

Entity Framework Fluent API vs Data Annotations

Özellik	Fluent API	Data Annotations (Attribute'lar)
Konfigürasyon Yeri	<code>OnModelCreating</code> içinde	Sınıf/özellik üstünde (e.g., <code>[Required]</code>)
Esneklik	Yüksek (ilişkiler, composite key, inheritance)	Sınırlı (basit kurallar)
Okunabilirlik	Zincirleme yapı okumayı kolaylaştırır	Sınıfı dağınık hale getirebilir
Örnek Kullanım	Aşağıda örneklerle açıklandı	<pre>public class Urun { [Key] public int Id { get; set; } }</pre>

Neden Fluent API Tercih Edilir?

1. Karmaşık İhtiyaçlara Çözüm:

- TPC (Table Per Concrete Type) gibi kalıtım stratejileri.
- Çoktan-çoğa (many-to-many) ilişkilerde ara tablo özelleştirme.

2. Temiz Sınıf Yapısı:

Domain sınıflarınızı veritabanı konfigürasyonundan ayırır.

3. Merkezi Yönetim:

Tüm konfigürasyon `OnModelCreating` 'de toplanır.



Örnek: Tam Bir Fluent API Konfigürasyonu

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Ürün konfigürasyonu
    modelBuilder.Entity<Urun>()
        .ToTable("Products")
        .HasKey(u => u.Id)
        .Property(u => u.Name)
        .IsRequired()
        .HasMaxLength(100);

    // Kategori-Ürün ilişkisi
    modelBuilder.Entity<Urun>()
        .HasRequired(u => u.Kategori)
        .WithMany(k => k.Urunler)
        .HasForeignKey(u => u.KategoriId);

    // Cascade delete kapatma
    modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
}
```

Nasıl Çalışır?

- Entity Framework, `DbContext` başlatıldığında `OnModelCreating` 'i çağırır.
- Fluent API konfigürasyonları veritabanı şemasına dönüştürülür.
- Migration oluşturulurken bu kurallar otomatik uygulanır.

Fluent API, Entity Framework'te **güçlü ve esnek modelleme** için vazgeçilmez bir araçtır

SORU4:

`u=>u.Kategori` kodundaki `'=>'` işareti ne anlama geliyor örnekle açıkla

`u => u.Kategori` ifadesindeki `=>` işareti, C#'ta **lambda operatörü** (lambda operator) olarak adlandırılır. Bu operatör, bir **lambda ifadesi** (lambda expression) oluşturur. Lambda ifadeleri, anonim (isimsiz) fonksiyonlar yazmak için kullanılır ve özellikle LINQ, Entity Framework Fluent API gibi yapılarda sıkça karşınıza çıkar.



Lambda İfadesinin Temel Yapısı:

parametre => ifade


- **Sol taraf** (`parametre`): Fonksiyona giren parametre(ler).
- **Sağ taraf** (`ifade`): Parametre üzerinde yapılacak işlem veya dönüş değeri.



Örnekle Açıklama:

Entity Framework Fluent API'deki `u => u.Kategori` ifadesini parçalayalım:

1. (**Parametre**)

- `u` ,  **entity sınıfından bir nesneyi** temsil eder (tıpkı bir fonksiyon parametresi gibi).
- Derleyici otomatik olarak tipini çıkarır (`Urun` tipinde).

2. (**Lambda Operatörü**)

- "Şu parametre şu işlemi yapar" anlamına gelir.

3. (İfade)

- u nesnesinin Kategori property'sine erişir.
- Bu, Urun sınıfındaki navigation property'yi seçer:

```
public class Urun
{
    public int Id { get; set; }
    public string Ad { get; set; }
    public virtual Kategori Kategori { get; set; } // <-- İşte bu!
}
```

Neden Lambda İfadesi Kullanılır?

Fluent API'da lambda ifadeleri, derleme zamanında tip güvenliği (compile-time safety) sağlar. Örneğin:

```
modelBuilder.Entity<Urun>()
    .HasRequired(u => u.Kategori) // <-- Lambda ifadesi
    .WithMany(k => k.Urunler);
```

- Eğer Urun sınıfında Kategori diye bir property olmasaydı, derleyici hemen hata verirdi.
- Refactoring (yeniden yapılandırma) sırasında property adı değişirse, otomatik olarak hata alırsınız.

☞

Lambda vs. Geleneksel Yöntem Karşılaştırması:

Lambda olmasaydı ne yapardık?

String bazlı çözümler kullanmamız gerekirdi (tip güvenliği olmadan):

```
// ESKİ YÖNTEM (String ile - Riskli!)
modelBuilder.Entity<Urun>()
    .HasRequired("Kategori") // Property adı string olarak
    .WithMany("Urunler");
```

- **Büyük risk:** "Kategori" yazım hatası yaparsanız, hata **ancak runtime'da** (çalışma zamanında) fark edilir.
- **Lambda ifadesi** bu riski ortadan kaldırır.



✚ Diğer Lambda İfadesi Örnekleri:

Örnek 1: Basit bir lambda

```
// x parametresini alıp x*2 döndüren lambda
Func<int, int> kare = x => x * 2;
Console.WriteLine(kare(5)); // Çıktı: 10
```

Örnek 2: LINQ'da lambda

```
var ucuzUrunler = urunler.Where(u => u.Fiyat < 100).ToList();
// "u" parametresi (her bir ürün), "u.Fiyat < 100" koşulu
```

Örnek 3: Fluent API'de sütun özelleştirme

```
modelBuilder.Entity<Urun>()
    .Property(u => u.Ad) // <-- Lambda!
```

```
.IsRequired()  
.HasMaxLength(100);
```

💡 Lambda İfadesinin Çalışma Mantığı:

Entity Framework, `u => u.Kategori` ifadesini şu şekilde yorumlar:

1. `u` parametresi üzerinden `Urun` sınıfının property'lerine erişir.
2. `Kategori` property'sinin metadata'sını (veri tipi, ilişki türü vb.) analiz eder.
3. Bu bilgileri kullanarak veritabanı şemasını oluşturur.



⚠️ Önemli Not:

- `u` ifadesi **sadece bir isimlendirmedir**. İstedığınız kelimeyi kullanabilirsiniz:
- `modelBuilder.Entity()
.HasRequired(urunNesnesi => urunNesnesi.Kategori)`

`=>` sonrasında sadece property seçimi değil, daha karmaşık mantıklar da yazılabilir:
`modelBuilder.Entity()
.Property(u => u.Ad.ToUpper())` // Geçerli değil! (EF çeviremez)

- Ancak EF, veritabanı işlemlerine dönüştüremeyeceği ifadelere izin vermez.

❓ Sık Sorulan Soru:

S: "`u => u.Kategori`" yazınca sanki bir fonksiyon çağırıyormuş gibi, ama parantez yok. Nasıl oluyor?"

C: Bu, lambda ifadelerinin söz dizimsel (syntactic sugar) bir özelliğidir. Derleyici bunu arka planda bir **delegate**'e dönüştürür. Örneğin:

```
Func<Urun, Kategori> ifade = u => u.Kategori;  
// Arka planda: Kategori func(Urun u) { return u.Kategori; }
```

Lambda ifadeleri, C#'ın modern özelliklerinden biri olup kodun daha okunabilir ve güvenli olmasını sağlar. Entity Framework Fluent API'daki kullanımı da bu güvenliği veritabanı modellemesine taşır.