WPF Data Binding İle Veritabanı Uygulaması

WPF ile basit bir veritabanı uygulaması yapmak için şunlar gereklidir:
1. Veritabanına erişecek bir sınıf
2. Veritabanındaki tabloyu temsil edecek bir sınıf
3. Bir tane XAML sayfası

Örnek:

Veritabanına erişen **StoreDb** isimli bir sınıf

```
1   public class StoreDB
2   {
3        private string connectionString = Properties.Settings.Default.StoreDatabase;
4        public Product GetProduct(int ID)
5        {
6            SqlConnection con = new SqlConnection(connectionString);
7            SqlCommand cmd = new SqlCommand("GetProductByID", con);
8            cmd.CommandType= CommandType.StoredProcedure;
9            cmd.Parameters.AddWithValue("@ProductID", ID);
10           try
11           {
12               con.Open();
13               SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.SingleRow);
14               if (reader.Read())
15               {
16                   // Create a Product object that wraps the
17                   // current record.
18                   Product product = new Product((string)reader["ModelNumber"],
19                   (string)reader["ModelName"], (decimal)reader["UnitCost"],
20                   (string)reader["Description"] ,
21                   (string)reader["ProductImage"]);
22                   return(product);
23               }
24               else
25               {
26                   return null;
27               }
28           }
29           finally
30           {
31               con.Close();
32           }
33       }
34  }
```

Veritabanındaki bir tabloyu temsil eden **Product** sınıfı

```
1   public class Product
2   {
3        private string modelNumber;
4        public string ModelNumber
5        {
6            get { return modelNumber; }
7            set { modelNumber = value; }
8        }
9        private string modelName;
10       public string ModelName
11       {
12           get { return modelName; }
13           set { modelName = value; }
14       }
15       private decimal unitCost;
16       public decimal UnitCost
```

```
        {
            get { return unitCost; }
            set { unitCost = value; }
        }
        private string description;
        public string Description
        {
            get { return description; }
            set { description = value; }
        }
        public Product(string modelNumber, string modelName,
        decimal unitCost, string description)
        {
            ModelNumber = modelNumber;
            ModelName = modelName;
            UnitCost = unitCost;
            Description = description;
        }
}
```

Product sınıfı gösteren bir XAML sayfası

```
<Grid Name="gridProductDetails">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto">
        </ColumnDefinition>
        <ColumnDefinition>
        </ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto">
        </RowDefinition>
        <RowDefinition Height="Auto">
        </RowDefinition>
        <RowDefinition Height="Auto">
        </RowDefinition>
        <RowDefinition Height="Auto">
        </RowDefinition>
        <RowDefinition Height="*">
        </RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Margin="7">
        Model Number:
    </TextBlock>
    <TextBox Margin="5" Grid.Column="1"
             Text="{Binding Path=ModelNumber}">
    </TextBox>
    <TextBlock Margin="7" Grid.Row="1">
        Model Name:
    </TextBlock>
    <TextBox Margin="5" Grid.Row="1" Grid.Column="1"
             Text="{Binding Path=ModelName}">
    </TextBox>
    <TextBlock Margin="7" Grid.Row="2">
        Unit Cost:
    </TextBlock>
    <TextBox Margin="5" Grid.Row="2" Grid.Column="1"
             Text="{Binding Path=UnitCost}">
    </TextBox>
    <TextBlock Margin="7,7,7,0" Grid.Row="3">
        Description:
    </TextBlock>
    <TextBox Margin="7" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
             TextWrapping="Wrap" Text="{Binding Path=Description}">
    </TextBox>
</Grid>
```

Şimdi de **GetProduct** butonunun **click** eventini handle edelim:

```
1   private void cmdGetProduct_Click(object sender, RoutedEventArgs e)
2   {
3        int ID;
4        if (Int32.TryParse(txtID.Text, out ID))
5        {
6            try
7            {
8                gridProductDetails.DataContext= App.StoreDB.GetProduct(ID);
9            }
10           catch
11           {
12               MessageBox.Show("Error contacting database.");
13           }
14       }
15       else
16       {
17           MessageBox.Show("Invalid ID.");
18       }
19  }
```

Ekran çıktısı

Dikkat edersek **cmdGetProduct_Click()** metodunda **DataContext** property'e veritabanından çekilen veri atanıyor.

Not: **Product** nesnesini **DataContext** property'den şu şekilde alabiliriz: `Product product = (Product)gridProductDetails.DataContext;`

Not: Veritabanından veri çekildiğinde bazı sütunların değeri **null** olabilmektedir. Bundan dolayı **Product** sınıfında **nullable data tipler** kullanmak mantıklı olacaktır. Örneğin, **decimal** türünün **nullable** hali `decimal?` şeklindedir. Sonuna soru işarti konulduğu zaman veri tipi **nullable** olmaktadır. Eğer **nullable** tip kullanmazsak, WPF nümerik olan yerleri 0 ile gösterecektir. Bu problemi gidermek için kullanıcıya verinin var olmadığını **TargetNullValue** property'i kullanarak söylebiliriz: `Text="{Binding Path=Description,`

`TargetNullValue=[Veri bulunamadı]}"`. **Description** kısmı **null** olduğu zaman **TextBox** içerisinde **Veri bulunamadı** yazacaktır. Bu arada köşeli parantezleri kullanmak zorunlu değildir.

Product Nesnesindeki Değişikliği TextBox İçinde Göstermek

Product nesnesinin herhangi bir property'sinde meydana gelen değişikliği göstermek için üç farklı yol vardır:

1. Her **Product** sınıfında bulunan property'i dependency property olarak tanımlarsak, bu property'lerde meydana gelen bir değişiklik **TextBox**'a direkt yansıtılır.
2. Her property için bir event meydana getirilerek yapılabilir. Event yaratırken şu sentaksa uymak zorunludur: PropertyNameChanged (örneğin, **UnitCostChanged).** Bu tarz bir event tanımlayarak ilgili property değiştiğinde tanımlamış olduğumuz eventi tetikleriz.
3. Product sınıfının **System.ComponentModel.INotifyPropertyChanged** interface'ini implement etmesini sağlayarak değişikliğin **TextBox**'a yansıtılmasını sağlayabiliriz. Bu interface içerisinde **PropertyChanged** isimli bir event bulunmaktadır. Bu yöntem en kullanışlı yöntemdir. Örnek olarak aşağıdaki gibi kullanılır:

```
1   public class Product : INotifyPropertyChanged
2   {
3        public event PropertyChangedEventHandler PropertyChanged;
4        public void OnPropertyChanged(PropertyChangedEventArgs e)
5        {
6            if (PropertyChanged != null)
7            PropertyChanged(this, e);
8        }
9   }
```

Şimdi **UnitCost** property içinde bu eventi tetikleyelim:

```
1   private decimal unitCost;
2   public decimal UnitCost
3   {
4        get { return unitCost; }
5        set {
6            unitCost = value;
7            OnPropertyChanged(new PropertyChangedEventArgs("UnitCost"));
8        }
9   }
```

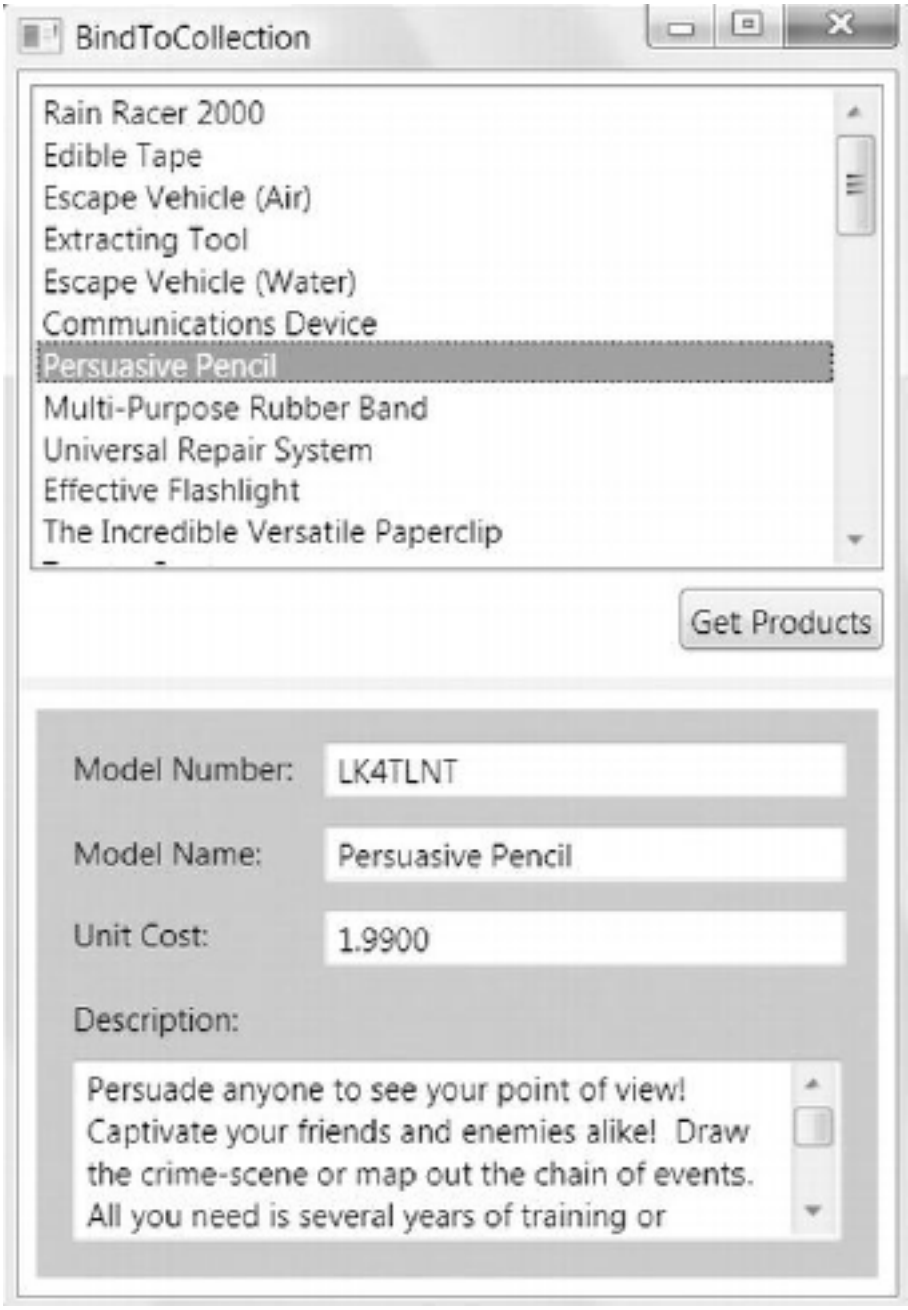
Collection Nesnelerini Listelemek

WPF'te **ItemsControl** sınıfından türüyen tüm sınıflar collection nesnesini handle edebilir. Bu sınıflardan bazıları şunlardır: **ListBox**, **ComboBox**, **ListView** ve **DataGrid** sınıflarıdır. **ItemsControl** sınıfının üç önemli property'si vardır:

ItemsSource	Collection nesnesine point eder.
DisplayMemberPath	Her bir item'ın gösterilmesi için kullanılacak property'i ifade eder.
ItemTemplate	Data template nesnesi alır. Template nesne, collection nesnesinin elemanlarının formatlı şekilde listelenmesini sağlar.

Not: Data binding işleminde kullanabileceğimiz collection sınıfları **IEnumerable** interface'ini implement etmeleri gerekmektedir.
Not: IEnumerable interface'i sadece read-only binding sağlamaktadır. Düzenleme ve silme işlemlerinin nasıl sağlanacağına birazdan değineceğiz.

Örnek:



Bu örneği yapmak için **StoreDB** sınıfına **GetProducts()** isimli bir metod eklenir. Bu metod bize collection olarak product listesi döndermesi sağlanır:

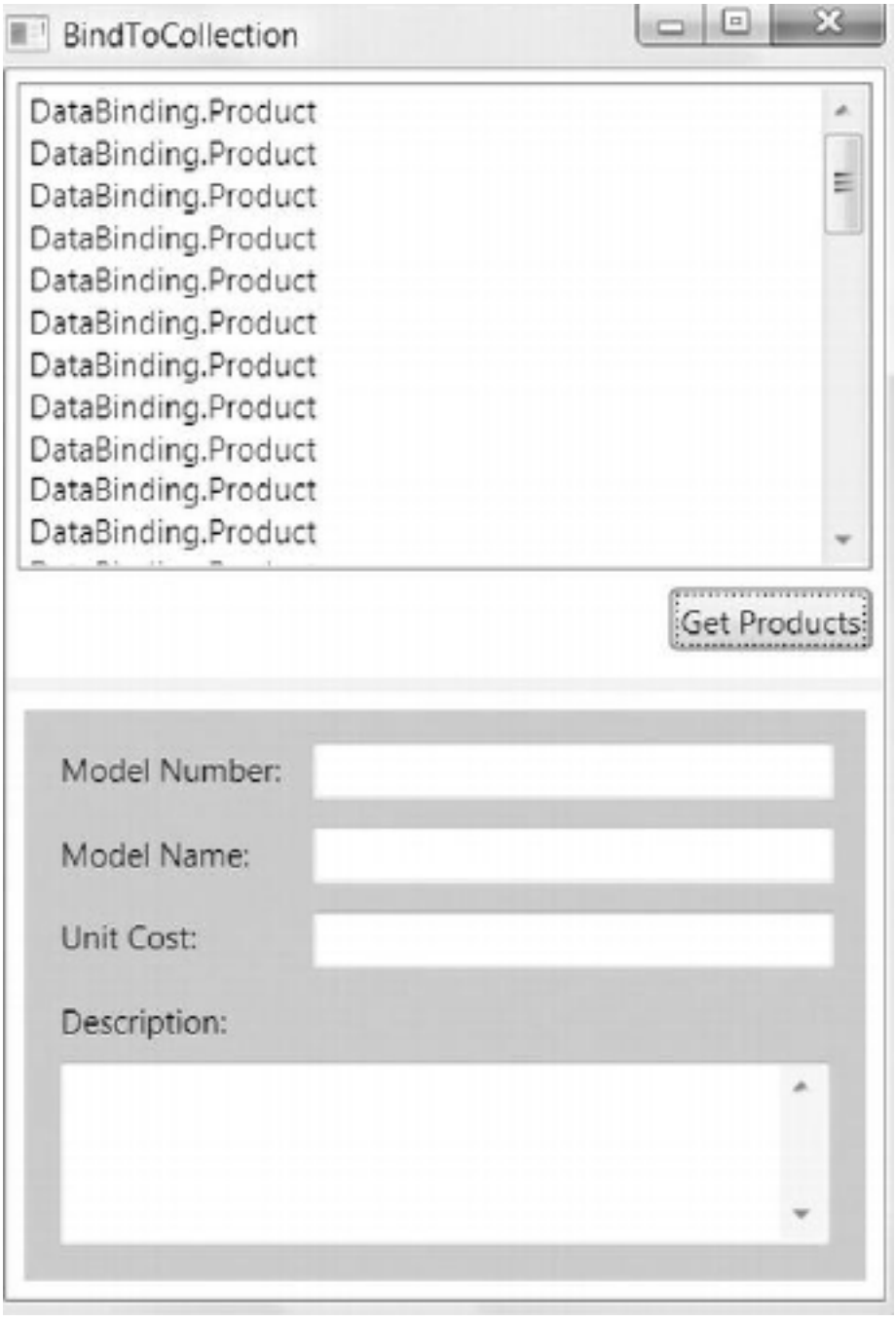
```
1   public List<Product> GetProducts()
2   {
3        SqlConnection con = new SqlConnection(connectionString);
4        SqlCommand cmd = new SqlCommand("GetProducts", con);
5        cmd.CommandType= CommandType.StoredProcedure;
6        List<Product> products = new List<Product>();
7        try
8        {
9            con.Open();
10           SqlDataReader reader = cmd.ExecuteReader();
11           while (reader.Read())
12           {
13               // Create a Product object that wraps the
14               // current record.
15               Product product = new Product((string)reader["ModelNumber"],
16               (string)reader["ModelName"], (decimal)reader["UnitCost"],
17               (string)reader["Description"], (string)reader["CategoryName"],
18               (string)reader["ProductImage"]);
19               // Add to collection
20               products.Add(product);
```

```
            }
        }
        finally
        {
            con.Close();
        }
        return products;
    }
```

Get **Products** butonununun click event'i aşağıdaki gibi implement edilir:

```
    private List<Product> products;
    private void cmdGetProducts_Click(object sender, RoutedEventArgs e)
    {
         products = App.StoreDB.GetProducts();
         lstProducts.ItemsSource= products;
    }
```

Bu implementasyon, başarılı bir şekilde **Product** nesnelerini listemize ekler. Fakat, **ListView** elementi, collection item'larını nasıl göstereceğini bilemez, bundan dolayı item'ların **ToString()** metodunu çağırır. Eğer collection nesnesinde bulunan item'ların **ToString()** metodu override edilmemişse, **ListView** elementi bu item'ların tam sınıf adlarını yazar:



Bu problemi üç farklı yolla çözebiliriz:

1. **LisDisplayMemberPath** property'e collection item nesnesinin property değerini atayabiliriz. Örneğin **ModelName** property değerini verdiğimiz zaman listelenen item'ların **ModelName**'leri gözükecektir.

```
<ComboBox Grid.Row="1" Grid.Column="2"
          DisplayMemberPath="{ModelName}" />
```

Eğer birden fazla property'nin gösterilmesini istiyorsak, örneğin **ModelName** ve **UnitCost**, yeni bir property yaratılır ve bu property bu iki property'i dönderir:

```
string ModelNameUnitCost { get { return ModelName + ": " + UnitCost; }}
```

2. **ToString()** metodunu override edebiliriz.

3. Data template kullanabiliriz. Bu yöntem en kullanışlı yöntemdir.

```
<ListBox ItemsSource="{Binding Products}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}: {1}>
                        <Binding Path="ModelName"/>
                        <Binding Path="UnitCost"/>
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Listede seçili bir elemanın bilgilerini göstermek için **Grid** elementi kullanabiliriz:

```
<Grid DataContext="{Binding ElementName=lstProducts, Path=SelectedItem}">
    ...
</Grid>
```

Buradaki **lstProducts** nesnesi **ListView** elementinin adıdır. **Path** property'nin aldığı değer seçilen item'dır.

Item Ekleme ve Silme İşlemleri

Item silmek için şu tarz bir kod yazabiliriz:

```
private void cmdDeleteProduct_Click(object sender, RoutedEventArgs e)
{
     products.Remove((Product)lstProducts.SelectedItem);
}
```

Bu kod çalıştığı zaman silinen item hala ekranda gösterilecektir. Yapılan değişikliğin yansıtılması için **collection change tracking** aktif olması gerekir. Bu özelliğin aktif olduğu collection sayısı çok azdır. Örneğin **List** collection'u bu özelliğe sahip değildir; çünkü bir collection'nun bu özelliğe sahip olabilmesi için **INotifyCollectionChanged** interface'ini implement etmiş olması gerekir. WPF'te bu interface'i implement eden tek bir collection sınıfı vardır: **ObservableCollection** sınıfı.

Collection nesnesinde yapılan değişikliklerin yansıtılması için **ObservableCollection** sınıfından türeyen bir custom sınıf yaratabiliriz veya aşağıdaki gibi kod yazarak custom collection oluşturmaktan kurtulabiliriz:

```
public List<Product> GetProducts()
{
     SqlConnection con = new SqlConnection(connectionString);
     SqlCommand cmd = new SqlCommand("GetProducts", con);
     cmd.CommandType= CommandType.StoredProcedure;
     ObservableCollection<Product> products = new ObservableCollection<Product>();
```

```
        ...
    }
```

Uygulamamızı çalıştırdıktan sonra listeden bir eleman silindiğinde otomatik olarak liste güncellenecektir.

ADO.NET Nesnelerini Bağlamak

GetProducts() metodunu aşağıdaki gibi düzenleyerek **DataSet** sınıfını kullanabiliriz.

```
public DataTable GetProducts()
{
     SqlConnection con = new SqlConnection(connectionString);
     SqlCommand cmd = new SqlCommand("GetProducts", con);
     cmd.CommandType= CommandType.StoredProcedure;
     SqlDataAdapter adapter = new SqlDataAdapter(cmd);
     DataSet ds = new DataSet();
     adapter.Fill(ds, "Products");
     return ds.Tables[0];
}
```

DataTable nesnesini direkt olarak binding işleminde kullanamadığımız için aşağıdaki kodta görüldüğü gibi **DefaultView** property'i kullanmalıyız:

```
private DataTable products;
private void cmdGetProducts_Click(object sender, RoutedEventArgs e)
{
     products = App.StoreDB.GetProducts();
     lstProducts.ItemsSource= products.DefaultView;
}
```

DataTable kullanıldığı zaman, **DataView** sınıfı **IBindingList** interface'ini implement ettiği için, yapılan değişiklikler otomatik olarak yansıtılmaktadır. Bundan dolayı custom nesneleri direkt kullanmak yerine **DataTable** ile kullanırsak işimiz daha da kolaylaşmış olur.

DataTable'da silme işlemi biraz farklıdır. Örneğin aşağıdaki gibi bir silme işlemi yanlış olacaktır:

```
products.Rows.Remove((DataRow)lstProducts.SelectedItem);
```

Bu ifadenin yanlış olmasının nedeni seçilen item'ın türü **DataRow** değil **DataRowView** sınıfıdır. Bundan dolayı type-casting'te **DataRow** yerine **DataRowView** yazılmalıdır. Ayrıca **DataRow**'u collection'dan silmek yerine **silinecek** şeklinde işaretlemek gerekir. Silinecek şeklinde işaretlenen bu item'ların silinmesi işlemi yapılan değişikliklerin veritabanına kaydedilmesinden sonra gerçekleşmesini sağlamak daha mantıklı olacaktır. Bu yüzden yazılması gereken kod aşağıdaki gibi olmalıdır:

```
((DataRowView)lstProducts.SelectedItem).Row.Delete();
```

Silinecek olarak işaretlenen item'lar teknik olarak listeden silinmese de liste içerisinde gösterilmez. Çünkü **DataView**'in varsayılan filtrasyonunda silinecek olarak işaretlenen item'lar gösterilmez. Örnek:

```
SqlConnection conn = new SqlConnection(

System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionString"].ConnectionStr
ing);
conn.Open();
SqlDataAdapter sqlDa = new SqlDataAdapter();
sqlDa.SelectCommand= new SqlCommand(selectStatement, conn);
SqlCommandBuilder cb = new SqlCommandBuilder(sqlDa);
sqlDa.Fill(dt);
dt.Rows[0]["Name"] = "Some new data here";
sqlDa.UpdateCommand= cb.GetUpdateCommand();
sqlDa.Update(products);
```

LINQ İfadelerini Bağlamak

WPF, tüm özellikleri ile **Language Integrated Query (LINQ)**'yu desteklemektedir. LINQ memory'deki bir collection'dan, bir xml

dosyasından veya veritabanından veri çekmek için kullanılır. Örnek olarak **Product** nesnelerinden oluşan bir collection nesnesi olsun. 100 dolardan daha pahalı olan product nesnelerini ayrı bir collection nesnesinde tutmak için LINQ kullanabiliriz:

```
// Get the full list of products.
List<Product> products = App.StoreDB.GetProducts();
// Create a second collection with matching products.
List<Product> matches = new List<Product>();
foreach (Product product in products)
{
     if (product.UnitCost >= 100)
     {
         matches.Add(product);
     }
}
```

Yukarıdaki gibi kod yazmak yerine LINQ ile aşağıdaki gibi daha verimli kod yazabiliriz:

```
// Get the full list of products.
List<Product> products = App.StoreDB.GetProducts();
// Create a second collection with matching products.
IEnumerable<Product> matches = from product in products
where product.UnitCost >= 100
select product;
```

LINQ, **IEnumerable**<T> interface'ini kullanır. Hangi veri kaynağını kullanırsak kullanalım, LINQ, **IEnumerable**<T> interface'ini implement eden bir nesne dönderir. Bundan dolayı `lstProducts.ItemsSource = matches;` şeklinde kod yazabiliriz.

IEnumerable<T> interface'i **ekleme** ve **silme** gibi işlemleri yapma özelliğine sahip değidlir. Bundan dolayı bu türden bir nesneyi **ToArray()** veya **ToList()** metodları ile array'e veya **List** nesnesine dönüştürmeliyiz.

```
List<Product> productMatches = matches.ToList();
```

ObservableCollection nesnesine dönüştürerek yapılan değişikliğin hemen yansımasını da sağlayabiliriz:

```
ObservableCollection<Product> productMatchesTracked =
     new ObservableCollection<Product>(productMatches);
```

Sonuç

Bu makalede geniş bir şekilde **data binding** konusu ele alınmıştır. **Custom** nesnelerin gösterilmesi, collection nesnelerinin verimli bir şekilde kullanılması konuları anlatılmıştır.